AI Literacy: Ada, AI, and Verification

A self-study sequence about ordered algorithms, language-model generation, and why human verification matters.

Frank C. Langbein (Human author & director)
Made with AI assistance (Claude Fable 5 and Claude Opus 4.8 (Anthropic), Gemini 3.1 Pro / 3.6 Flash (Google), GPT-5.6 Sol / Terra (OpenAI); orchestrated by Baba Yaga V10)

24 July 2026

Abstract

Course Brief:

The first program ever written computed Bernoulli numbers. Let's use it to learn how to work with — and check — the newest kind of machine. Ada wrote an exact, ordered procedure; a language model predicts plausible code. Ask the AI to reconstruct her computation, then do what her era couldn't do cheaply: verify it. The arc **deterministic algorithm → statistical generation → human verification** is the whole literacy story in one example.

Contents

1 AI Literacy: Course Handbook

This handbook contains the complete core and optional reading and activities for the AI Literacy module.

2 Unit 0 — Human introduction and how to use this module

Reading time: about 10 minutes. Activity: about 10 minutes.

2.1 What this module is

This is a self-study module about working with artificial intelligence. It is built around one example, and it stays with that example from the first unit to the last.

In 1843 Ada Lovelace published a set of notes describing how a machine designed by Charles Babbage — the Analytical Engine, which was never built — could be made to compute a sequence of numbers called the Bernoulli numbers. Her account is an ordered procedure: a fixed sequence of operations, with every quantity named and every step written down.

You are going to ask a modern AI system to reconstruct that computation. Then you are going to find out whether what it produced is correct, using a reference you built yourself.

The arc of the module is three steps long:

1. An exact ordered procedure, which either follows its specification or does not.
2. Generated text, which is fluent whether or not it is correct.
3. Verification against suitable evidence, which is how you establish which claims the output supports.

Most of the module is stage three. That is deliberate. Generating output is now the easy part of most tasks; establishing that the output is right is the part that still needs you.

2.2 The disclosure model

Almost everything you are about to read was drafted with AI assistance from a plan written by a human instructor, and reviewed by that instructor before release. The module says so once, at module level, rather than repeating it on every file.

There are two reasons for that, and neither is legal caution.

The first is that a reader of any piece of work is entitled to know how it was made. That is a low bar and this module clears it.

The second is that the module will ask you to write the same kind of disclosure about your own work in Unit 8. Material that demanded a disclosure it was unwilling to make itself would not deserve to be believed.

Note what the disclosure does **not** say. It does not say the material is correct because a human reviewed it. It says a human reviewed it, which is a statement about process. Correctness is a separate claim, and it needs separate evidence — which is what the checkpoints, the reference implementation, and the verification lab are for. And here is the part worth saying plainly at the start: **disclosure is the easy half, and using AI is not something to apologise for; the important, harder half is being able to stand behind the work — to say you understand it and have evidence it is right.** Keeping “how this was made” and “why this is right” apart is a skill this module returns to repeatedly, and it spends most of its effort on the second one.

2.3 The learning contract

One sentence, and it does not have exceptions:

> Using an AI system does not remove your responsibility for how you use and submit its output.

If something you submit is wrong, you remain answerable for using it. Other people and organisations may also have responsibilities for design, deployment, policy, review, or harm. The learning contract does not settle those wider duties; it prevents you from treating the tool as a substitute for your own checking. Disclosure is a statement about provenance, not evidence of correctness.

This has a practical consequence that runs through every unit. After checking whether the use is permitted, ask:

> What would I have to check before I could stand behind this?

Pause rather than approve the result when you can no longer do any one of four things: predict relevant behaviour, explain the transformation from input to output, identify a suitable independent check, or interpret a failed check. Confidence is not evidence and does not repair any of those gaps. Narrow the task, rebuild the missing foundation, or seek qualified review before relying on the result.

That question has different answers for different kinds of output. A calculation can be run against values you already know. A claim about the world has to be looked up. A low-risk first draft that you will rewrite may need less checking, but claims, personal data, confidential material, and consequential decisions still need appropriate handling. Unit 2 turns that into a working triage you can apply in seconds; the rest of the module makes you do it on a real example, where it is possible to be wrong.

2.3.1 Real-world and industry context

The same habit matters outside assessment. In software, data analysis, documentation, or workplace decision-making, the person who approves an AI-assisted result remains answerable for it. An unchecked generated snippet can introduce a security bug; an unchecked generated summary can misstate a policy; an unchecked calculation can be numerically wrong. Verification is not only an academic-integrity rule. It is a way to make work defensible.

2.4 How the module is organised

Ten short units: nine core, taken in order, plus one optional historical unit (Unit 7). The middle units run about 25 to 45 minutes each; the opening units are shorter. Set aside about four and three-quarter hours for the measured core route; a shorter route of about three hours ten minutes exists — see `../learning-paths.md`. Spread it over as many sittings as you like.

Each unit gives you:

- a short **video** with slides (and a transcript, if you would rather read than watch);
- `page.md` — the **reading**. This page is one of them.
- `activity.md` — the **task**. This is where the learning happens; skipping the activities leaves you with a story you have heard rather than a skill you have. Its marking guidance is shown to you once you have done the work, so you can assess yourself honestly.
- a **checkpoint** you can pass or fail, in the browser or a terminal.

You also accumulate an evidence pack as you go: your prompts, the outputs you got, what you checked, what you found, what you changed, and a short statement of why your final answer is correct. Unit 8 assembles it. Start collecting from this unit.

2.5 Choose your learning route

No programming is required. Choose a route now, and switch later if useful.

	Table-based route	Python route
Best for	Learners who do not program, or want to see the comparison by hand.	Learners who know some Python, or want automated testing practice.
In Unit 4	Ask for Bernoulli values or use the supplied outputs.	Ask for a Python function or use a supplied implementation.

	Table-based route	Python route
In Unit 5	Compare exact values row by row and record the first divergence.	Run `check.py` against the candidate routine and inspect its diagnostics.
Evidence produced	The rows compared, first divergence, convention, and limitations.	The command, tested range, properties, diagnostics, and limitations.
Standing	Complete route.	Complete route, with additional CS practice.

Both routes teach the same core outcomes: specify what correct means, compare against evidence independent of the candidate, diagnose disagreement, and state the limits of the check. They do not produce identical evidence. The Python route uses the code-only `cp05-bernoulli` checkpoint and the shared `cp05-diagnose`; the table route uses `cp05-table-evidence` and the same shared diagnostic. The first establishes executable behaviour over the checker's stated range. The second establishes only what was manually compared.

2.6 The self-check framework

Every unit has at least one **checkpoint**: a question you can pass or fail, tied to the unit's content. Checkpoints are **formative** — practice that gives feedback, not marks (see the glossary) — unless your instructor says otherwise. They exist so that you find out you have misunderstood something in the unit where it is explained, rather than three units later.

The **browser and terminal interfaces** are of equal standing. For every browser-compatible checkpoint they ask the same question, accept the same answer, and give the same feedback because both are generated from one shared definition. The additional Unit 5 code checkpoint runs only in Python.

```
python3 selfcheck.py run <id>          # or use the browser self-check
```

The Python route lives in `course/lab/` and needs nothing beyond a Python 3 installation — no packages to install. The browser route is a single page that works from a local file, makes no network requests, and keeps your answers in the page; use the module website's self-check page if you were given a link, open `dist/site/selfcheck.html` from a downloaded release, or — if you are working from source — `build/site/selfcheck.html`, which exists only after `make selfcheck` has been run. Use whichever you prefer, and switch whenever you like.

Three things worth knowing before you start:

- **Feedback is diagnostic, not binary.** A wrong answer that matches a well-known misunderstanding will say which one you have hit and point you at the unit that deals with it.
- **Some checkpoints will not show you the answer**, because deriving it is the point of the exercise. You get a hint aimed at your next step instead.
- **The framework is explorable.** `python3 selfcheck.py explore` lets you query the trusted reference implementation with your own inputs and compare two implementations against each other. Use it to poke at things, not only to be tested.

The answers to the checkpoints are, of course, sitting in a source file in this repository. Nobody is stopping you. A learner who goes looking has spent their own time to avoid learning something, which is a poor trade in a module whose entire subject is the difference between having an answer and being able to defend one.

2.7 Where this sits in the wider course

This module is a case study inside a wider eight-item AI literacy course. It leads strongly on some items, touches others where the Ada and Bernoulli example genuinely supports them, and only signposts the rest.

See `../course-map.md` for the item numbers, the item names, and this module's coverage level for each. That file is the single source of truth for those names, which is why they are not repeated here.

Item	What this module does with it
1	Taught directly, by contrasting an ordered procedure with generated text.
2	The core of the module: the oracle, the checker, and the verification lab.
3	Touched through disclosure, transparency, and human accountability only.
4	Taught directly, by making you derive a reference before you trust a generated one.
5	Signposted, with one privacy reminder. The wider course does this properly.
6	Taught directly, through disclosure and the "generated" versus "defensible" distinction.
7	Touched through human-in-the-loop accountability and a review-style workflow.
8	Optional extensions for learners who write code, clearly labelled as such.

A learner who takes only this module should still finish with the story, the concept, the verification method, and the disclosure habit. A learner taking the full course should treat this as the practical half of the evaluation item.

2.8 Before you go on

Do the activity in `activity.md`. It takes about ten minutes and produces the first line of your evidence pack. Then take checkpoint `cp00-contract`, and start Unit 1.

3 Unit 0 activity — your verification sentence

Time: about 10 minutes. Working alone is fine.

You need somewhere to keep written work for the rest of the module — a document, a notebook, a file. This activity produces its first entry.

3.1 Why this activity exists

The learning contract says responsibility for a piece of work stays with you. That is easy to agree with and easy to forget the moment an output arrives looking finished. Writing down, in advance, one specific thing you intend to check makes the commitment concrete enough to be kept — and specific enough to notice when you have broken it.

3.2 Part 1 — write the sentence (5 minutes)

Think about a real piece of work you have coming up, or one you finished recently, where you have used or might use an AI system. Coursework, a summary, a piece of code, an email, a set of references, a revision plan — anything real.

Write one sentence in this shape:

> In [the piece of work], I will verify [the specific thing] rather than trust it, by [the specific check].

Three rules for the sentence:

1. **Name a specific artefact**, not a category. “The three references it gave me” beats “the sources”.
2. **Name a check that could actually fail.** If there is no result that would make you say “that is wrong, I need to fix it”, it is not a check.
3. **Keep yourself as the subject.** The sentence describes something you will do, not something you hope the tool will get right.

Two examples of the shape, so you can see the difference the rules make:

- Too vague to act on: *In my essay, I will make sure the AI output is accurate.*
- Specific enough to keep: *In my lab report, I will verify the unit conversion it produced by recomputing it myself on paper and comparing the two numbers to three significant figures.*

Write your own. One sentence is enough; two is fine if the check needs a clause of its own.

3.3 Part 2 — name what you are not checking (2 minutes)

Underneath your sentence, add one line naming something in the same piece of work that you have decided **not** to check, and why that is a reasonable decision.

This is not a trick. Checking everything is not possible and pretending otherwise produces a policy nobody follows. The skill is deciding deliberately, and being able to say afterwards what your decision was. Unit 2 gives you a way to make that decision systematically.

3.4 Part 3 — get your self-check route working (3 minutes)

Pick a route now, before you need it under time pressure, and confirm it runs.

Browser route. How you open the self-check page depends on how you got the module. If you were given a link to the module website, use its self-check page. If you downloaded a release, open `dist/site/selfcheck.html` directly — it works straight from a file, with no internet connection. If you are working from the source repository, `build/site/selfcheck.html` exists only after `make selfcheck` has been run; if it is missing, use the release version instead.

Python route. From `course/lab/`, run:

```
python3 selfcheck.py list
```

You should see every checkpoint in the module, grouped by unit, with your status against each.

If neither route runs, say so to your instructor now rather than at Unit 5, where the checkpoints start doing real work.

3.5 Now take the checkpoint

This unit’s checkpoint is **`cp00-contract`**. It asks one question about the learning contract. Either route:

```
python3 selfcheck.py run cp00-contract          # or use the browser sel
```

Run it from `course/lab/` if you are using the command line. If you get it wrong, read the feedback rather than just retrying — it names the specific confusion involved, and that confusion is a common one worth having named.

Then keep your verification sentence somewhere you will find it again. Unit 8 asks you to look back at it and say honestly whether you kept it.

3.5.1 Activity Answers

4 Unit 0 answers and marking guidance

There is no single correct answer to this activity. There is, however, a clear difference between a sentence that could be kept and one that could not, and that difference is what to look for.

4.1 For a learner marking their own work

Read your sentence back and ask three questions.

1. Could someone else tell whether I did it?

If a classmate read your sentence and then looked at your finished work, could they say whether the check happened? “I will make sure it is accurate” fails this. “I will recompute the conversion on paper and compare to three significant figures” passes it.

2. Is there an outcome that would count as failure?

Name it. If your check cannot come out badly, it is a reassurance, not a check. A useful test: finish the sentence “…and if that check fails, I will ___“. If you cannot finish it, tighten the check.

3. Am I the one doing the work?

Sentences of the form “I will use a better prompt so the output is reliable” describe a hope about the tool. Sentences of the form “I will look up each reference in the library catalogue” describe an action of yours. The second kind is what the contract asks for.

If your sentence fails one of these, rewrite it now. It takes a minute and the rewritten version is the one Unit 8 will ask you about.

For Part 2 — the thing you decided *not* to check — a good answer names something specific and gives a reason connected to consequence: the error would be obvious, or cheap to fix, or the passage is going to be rewritten anyway. A weak answer is “I will check everything”, which is not a decision, or “the rest is probably fine”, which is a guess dressed as a policy.

4.2 For an instructor

Expected time: 10 minutes. This activity is formative and is best not graded; its value is as the first entry in the evidence pack and as a reference point for Unit 8’s reflection.

Strong responses typically show three features:

- a named artefact at the level of a single claim, number, function, or citation;
- a check whose method is stated, not just its intention;
- a failure condition that is at least implicit.

Common weak patterns, and the intervention for each:

Pattern	Example	What to say
Category, not artefact	“I will check the facts.”	Ask which fact, in which paragraph. Specificity is the whole exercise.
Check with no possible failure	“I will read it through carefully.”	Ask what a bad outcome would look like. Reading is attention, not verification.

Pattern	Example	What to say
Responsibility displaced onto the tool	“I will ask it to double-check itself.”	A system’s second output about its first is generated the same way as the first. Unit 2 explains why this is not independent evidence.
Disclosure offered in place of checking	“I will declare that I used AI.”	Both are required and they answer different questions. This is precisely the confusion `cp00-contract` names.
Whole-output verification	“I will verify everything it produces.”	Not a policy anyone keeps. Part 2 exists to make the trade-off explicit rather than silent.

Cohort signal worth watching for. If a large share of the group produces sentences of the last two kinds, spend a moment on the distinction between provenance and correctness before Unit 2, because the triage activity there assumes it.

4.3 The checkpoint

This unit’s checkpoint is `cp00-contract`. Its question, options, correct answer and diagnostic feedback are defined once, in the `checkpoint` block at the end of this file; `../lab/checkpoints.py` and the browser self-check both read that block, so there is a single source and the routes cannot drift apart.

What the checkpoint is testing is the distinction between disclosing how work was produced and being answerable for whether it is right. If a learner reports getting it wrong, do not supply the answer: the feedback the framework gives on the most common wrong option is more useful than a correction, because it names the confusion rather than replacing it.

Learners run it with either route:

```
python3 selfcheck.py run cp00-contract          # or use the browser sel
```

4.4 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.

```
{
  "id": "cp00-contract",
  "unit": 0,
  "title": "The learning contract",
  "kind": "choice",
  "question": "You use an AI assistant to produce part of work that you
  "answer": 0,
  "options": [
    "You remain answerable for using the output and must be able to supp
    "Only the AI vendor is answerable, since the tool produced the error
    "No one is answerable, provided you disclosed that you used AI.",
    "Only your instructor is answerable if they permitted AI use on the
  ],
  "diagnostics": {
    "1": "A vendor may have duties for design, safety, or claims about i
    "2": "Disclosure and responsibility are different things. Disclosure
```

```
    "3": "Permission to use a tool does not make the instructor solely a
  },
  "explanation": "AI assistance changes how you produce work but does no
}
```

5 Unit 1 — Ada's ordered procedure

Reading time: about 7 minutes. Activity: about 12 minutes.

5.1 The document

In 1843 an English scientific journal published a translation of a French-language account of Charles Babbage's Analytical Engine. The account had been written by the Italian engineer Luigi Menabrea; the translator was Ada Lovelace, who signed her work with her initials. She appended seven notes of her own, labelled A to G, and they run substantially longer than the memoir they accompany.

Note G — the last of them — sets out how the engine could be made to compute a sequence called the Bernoulli numbers. Unit 3 explains what those numbers are and makes you work one out by hand. This unit is about the *form* of what Lovelace wrote, because that form is one half of the comparison the whole module rests on.

The full bibliographic citation for the memoir, the translation and the notes is held in `course/references.md`. This page cites it as (Lovelace 1843, Note G).

5.2 What Note G actually is

It is a table.

Not a description of a method, and not an argument that a method would work in principle. A table, laid out in order: one line per operation, running from the first thing the machine does to the last. Each line identifies the operation to be performed, the storage columns it takes its inputs from, and the column that receives the result (Lovelace 1843, Note G).

Babbage's design kept two functions physically separate: a **store**, holding numbers in columns, and a **mill**, performing arithmetic on numbers brought to it from the store. Memory in one place, operations in another, and a written procedure saying which values move between them and in what order.

The engine was never built. Nothing in Note G was ever executed on the machine it was written for. That is worth sitting with: the procedure had to be exact without the safety net of being run, which is the opposite of how most of us write instructions.

5.3 Three properties worth naming

These three words come back in every later unit, so it is worth being precise about them now.

5.3.1 Determinism

Given the same starting values and the same sequence of operations, the procedure produces the same result. Every time. Not usually, not with high probability — every time.

This is a strong property and it is easy to underrate because it feels obvious. Its practical consequence is that a disagreement between two runs is not variation, it is a fault. Something is broken and can be found. That is a completely different world from one in which two runs are simply expected to differ, which is where Unit 2 takes you.

5.3.2 Explicit state

The **state** of the machine is the collection of values currently held in its storage columns. Every value in that state got there because some earlier line put it there.

There is no quantity that is merely understood to be available. If a value is needed, you can point at the line that produced it. When you write a procedure by hand, this is the property that fails first: human instructions

are full of objects and quantities that were never introduced, and we do not notice because we supply them ourselves as we read.

5.3.3 Control flow

The order of the lines is part of the meaning. Swap two of them and you have a different procedure, not the same procedure written differently.

The order is also not always a straight run from top to bottom. Note G's diagram marks a group of operations to be repeated, with the number of repetitions depending on which Bernoulli number is wanted (Lovelace 1843, Note G). The teaching point needed here is visible in the table itself: a procedure with a controlled repetition computes a whole family of answers, not one answer. That is the difference between a recipe and an algorithm.

5.4 "The first program"

You will have met the claim that Note G is the first computer program and that Ada Lovelace was the first programmer. The claim is useful — it points at something real, and it is why this module starts here. It is also more complicated than a slogan, and the module would be a poor advertisement for verification if it repeated a contested claim without saying so.

What is on firm ground. The notes were published in 1843 with the translation, signed with her initials. Note G contains an ordered table of operations for computing Bernoulli numbers [P2]. Those facts do not by themselves establish a priority claim.

What is genuinely contested.

- Babbage had written sequences of operations for the engine in his own unpublished notebooks before 1843, so whether Note G is "first" depends on whether publication is the criterion [S1], [S4].
- How the work on Note G was divided between Lovelace and Babbage is debated by historians, and the correspondence has been read in more than one way [S2], [S3], [S5].
- Nobody in 1843 had the concept of a program. Applying the word backwards is a choice we make, and a defensible one, but it is our choice and not a fact about 1843.

One more established fact, handled carefully. The published table in Note G contains an error. Where it is — operation 4, an inverted division — is established from a facsimile; only who introduced it stays contested. This module does not work through the detail here; Unit 7 treats the whole episode as its case study in historical honesty. See `course/historical-notes.md` for what the sources support.

If that feels unsatisfying, notice what it is modelling. "I know this much, this much is disputed, and here is where you can check" is a stronger position than a confident sentence, and it is exactly the position you will need to take about your own AI-assisted work in Unit 8.

5.5 The sentence that sets up the module

Writing about the limits of what the engine could do, Lovelace put it this way:

> The Analytical Engine has no pretensions whatever to originate anything.
>
> — (Lovelace 1843, Note G)

Read in context, this is a claim about following orders: the machine can carry out a procedure, and carrying out a procedure is not the same as producing something new. Whether the sentence still applies to the systems we have now is a genuine question that Unit 8 returns to, with the comparison made carefully rather than rhetorically.

For now, notice what the sentence takes for granted. It assumes you can say precisely what the machine was ordered to do, and then check whether it did that. Note G is fully checkable in this sense: every line is written down, and any line can be found to be wrong. Somebody did find one.

That is the property Unit 2 removes.

5.6 Why this module looks the way it does

This is background rather than argument, and nothing in the unit depends on it.

Ada Lovelace was the daughter of the poet Lord Byron, who left England in 1816, when she was a few months old, and never saw her again [S5] (keys resolve in `../references.md`) [G1]. In the summer of that year Byron presided over a gathering at the Villa Diodati on Lake Geneva, with Percy Shelley, Mary Godwin — later Mary Shelley — and Byron's physician John Polidori; out of it came *Frankenstein; or, The Modern Prometheus* (1818) and Polidori's *The Vampyre* (1819) [G2] [G3] [G4]; *Frankenstein* is the book usually credited with bringing the anxiety about a made thing that reasons into English literature. A module about a machine that produces plausible speech, and about whether it originates anything, is therefore working in a register it inherited rather than one it borrowed, which is why the module's visual design leans dark, with a single red accent, instead of the usual palette of an introductory course. What Lovelace herself made of any of that, and whether it touched her mathematical work at all, is not something this module claims. The connection is a visual motif, not a historical inference.

5.7 Summary

- Note G is an ordered table of operations for computing Bernoulli numbers, published in 1843 with Lovelace's translation of Menabrea's memoir.
- Each line is a state change: an operation, its inputs, and where the result goes.
- Ordered procedures are deterministic, hold explicit state, and depend on control flow — including controlled repetition.
- "The first program" is a useful claim with real historical nuance. State what is established, mark what is contested.
- The published table contains an error. Its location is established; only who made it is contested. Unit 7 deals with it.

Now do `activity.md`, then take checkpoint `cp01-ordered-steps`.

6 Unit 1 activity — write it down the way Ada had to

Required work: about 12 minutes. Pen and paper is fine; so is a text file. An optional extension at the end adds more if you want it.

Ada Lovelace had to write a procedure for a machine that could not be relied on to fill in anything she left out. You are going to do the same thing with a task you already know how to do, and find out how much of it you have never actually said out loud.

6.1 Part 1 — choose a task (1 minute)

Pick something ordinary that you can do without thinking, and that takes six to ten steps: posting a parcel, changing an inner tube, putting a duvet into its cover, returning an overdue library book, making a cheese sandwich for someone else.

Do not pick something you already think of as a procedure — a recipe you follow from a card, or anything with an instruction manual. The exercise only works on something you do from memory.

6.2 Part 2 — write the procedure (5 minutes)

Number your steps from 1, and hold to these six constraints. They are what turn a description into a procedure.

1. **One action per step.** If a step contains “and”, consider splitting it.
2. **Name every object before you use it.** If step 6 refers to the tray, some earlier step must have introduced the tray. This is explicit state.
3. **Give every quantity a number.** How much, how many, how far, how hot. “Enough” and “a little” are not quantities.
4. **Give every wait a duration or a condition.** Either “wait 4 minutes” or “wait until X happens”, where X is something observable.
5. **No pronouns.** No “it”, “them”, “the other one”. Write the noun again. This feels absurd and it is where most of the gaps get exposed.
6. **State the starting state and the finishing condition.** What is true before step 1, and what is true when you are done?

To see the difference the constraints make — bad: “Add some water.” Good: “Pour 250 ml of cold water from the tap into the mug introduced in step 1.”

Write the whole thing before you edit any of it. You want the first draft to be honest about what you would actually have written.

6.3 Part 3 — the literal reader pass (4 minutes)

Read your own procedure the way a machine would: refuse to supply anything that is not written down. Take each step in turn and ask “could I do only what this says, knowing nothing else?”

Mark every gap you find with a letter:

Code	Gap
A	**Unnamed object** — a noun used that no earlier step introduced.
B	**Missing quantity** — how much or how many is not stated.
C	**Missing duration or timing** — a wait with no length and no condition.
D	**Missing precondition** — the step only works if something was already true, and nothing says it is.
E	**Missing repetition or stopping rule** — something has to happen more than once and the procedure does not say how many times or when to stop.
F	**Ambiguous reference** — “it”, “them”, “the other one”.
G	**Hidden decision** — the step needs judgement, and the criterion for that judgement is not written down.

Count them. Most people find between five and fifteen in a first draft of a task they have done hundreds of times. That is the normal result and it is the point of the exercise, not a sign that you wrote it badly.

6.4 Part 4 — fix one of them properly (2 minutes)

You do not have time to repair the whole procedure, and you do not need to.

Choose the single gap that would cause the worst outcome if a literal reader hit it, rewrite that step so the gap is closed, and write one sentence saying why you chose that gap rather than another.

That sentence is doing real work. Deciding which unstated assumption matters most is the same judgement you will need in Unit 2, when you decide which parts of a generated output have to be checked and which do not.

6.5 Now take the checkpoint

This unit's checkpoint is **`cp01-ordered-steps`**. It gives you somebody else's procedure and asks what is missing from it — the same literal-reader pass you have just done on your own work, which is considerably easier on a stranger's writing than on your own. Parts 1 to 4 are all you need for it.

```
python3 selfcheck.py run cp01-ordered-steps          # or use the browse
```

Run the Python route from `course/lab/`. If you get it wrong, read the feedback before retrying: it distinguishes between things genuinely missing from a procedure and background knowledge that a procedure is not obliged to supply, and that distinction is the one this unit exists to teach.

Keep your annotated procedure. Unit 6 is about writing specifications precise enough to be tested, and it will ask you to look at this again.

6.6 Optional extensions

Optional. These sit outside the core study time, are not required for the checkpoint, and nothing later in the module depends on them.

Repair the rest of the procedure (about 5 minutes). Part 4 asked for one repair. Close the remaining gaps as well, then reread the result: it will be longer, flatter and duller than your first draft, and it will be the version a literal reader could follow. That trade is the one Note G makes on every page.

Swap with someone else (about 5 minutes). Run the literal-reader pass on another learner's procedure and have them run it on yours. Gaps in a stranger's writing are obvious; gaps in your own are invisible by construction, which is the whole difficulty this unit is about.

For learners who write code (about 15 minutes). Rewrite your procedure as a variable table in the style of Note G. Give each object a name — `V1`, `V2`, and so on — and for each step record: the operation, the variables read, the variable written, and the state of every variable afterwards. Then answer two questions. Which steps turned out to read a variable that had never been written? And where did you need a loop, and what is its termination condition?

6.6.1 Activity Answers

7 Unit 1 answers and marking guidance

The activity has no single correct output — the tasks differ from learner to learner. What can be marked is whether the gaps were found, and whether they were classified for the right reason.

7.1 A worked example

Here is a first draft of the kind learners typically produce, for posting a parcel. The gap codes are those from the activity.

1. Put the item in a box.
2. Seal the box.
3. Write the address on it.
4. Take it to the post office.
5. Queue.
6. Tell them what service you want.

7. Pay.
8. Keep the receipt.

Annotated:

Step	Code	What a literal reader cannot do
1	A, B	No box has been introduced, and no size is given. Which item, and which box?
2	A	Sealing requires tape or a seal. Nothing has introduced any.
3	F, A	"It" — the box or the item? And no address has been supplied to the procedure at all.
3	D	Writing on a sealed box assumes a writable surface and a pen; neither is stated.
4	A, D	Which post office? The procedure never establishes that one is open, or reachable.
5	E	Queue until when? The stopping condition is the whole content of the step.
6	A, G	"Them" is undefined, and *which* service is a decision with no stated criterion.
7	B	How much? The amount is produced by step 6 and never captured anywhere.
8	D	The receipt was never introduced; step 7 has to produce it for step 8 to keep it.

Two features of that annotation are worth pointing out to learners.

The gaps cluster around values that are produced and then needed later. The address, the price, the receipt: each is a value that some step must generate and store before a later step can use it. That is precisely the explicit-state property from the reading. In Note G, every such value has a named column and a line that puts it there.

Step 5 is the interesting one. "Queue" is not a single action; it is a repetition with a termination condition, and the condition is the only part that matters. Learners who spot repetition and stopping rules have understood control flow, which is the hardest of the three properties to see in everyday language.

7.2 For a learner marking their own work

Count the gaps you found, then check your classification against these two questions.

Did I mark anything that is not really a gap? A procedure has to state every action, object and quantity it depends on. It does **not** have to explain what the result is for, why anyone would want it, or supply general background knowledge about the world. "It doesn't say what a parcel is" is not a gap in the procedure. The distinction matters and the checkpoint tests it.

Did I miss a whole category? Look at the seven codes and see which ones you never used. Most people find A, B and F easily, because they are visible on the page. The three that get missed are:

- **D, missing preconditions** — the things that have to be true before step 1 for the procedure to work at all.
- **E, missing repetition and stopping rules** — steps that are secretly loops.
- **G, hidden decisions** — steps that require judgement, where the criterion for the judgement is doing all the work and is nowhere written down.

If you used none of D, E or G, go back through your procedure once more looking only for those. There is almost certainly at least one.

A good answer to Part 4 chooses its gap by **consequence**, not by how obvious it is: which gap would produce the worst outcome if a literal reader hit it? A learner who repairs the easiest gap has done the exercise; a learner who repairs the most dangerous one has done the thinking that Unit 2 needs.

Repairing the remaining gaps, swapping procedures with someone else, and the Note G variable table are all **optional extensions**. They are not part of the required work, they are not needed for the checkpoint, and the guidance below assumes they have not been done.

7.3 For an instructor

Expected time: 12 minutes for the required work. Formative; not suited to a grade. The optional extensions at the end of the activity add roughly 5, 5 and 15 minutes and are outside the core budget.

What a strong response looks like:

- six to ten numbered steps, one action each;
- five or more gaps found, spread across at least four of the seven codes;
- at least one gap of type D, E or G;
- one repair chosen for consequence, with the reason stated.

What to intervene on:

Observation	Likely cause	Response
Very few gaps found	The learner is reading co-operatively, supplying meaning as they go.	Have them swap procedures with someone else. Gaps in a stranger's writing are obvious; gaps in your own are invisible by construction.
Gaps found only of types A, B, F	Surface reading.	Ask directly: "which step happens more than once, and how does it stop?"
Procedure written as prose	Instruction not followed, or the task chosen was too abstract.	Reassign to a physical task with objects that can be pointed at.
Learner marks background knowledge as a gap	Over-application of the rule.	Distinguish "the procedure never introduces the tape" from "the procedure never explains what tape is". Only the first is a defect.

The point to make in feedback, whatever the cohort produces. Unstated assumptions are invisible to the person holding them. That is not carelessness; it is what an assumption *is*. The remedy is not to try harder, it is to have a procedure for finding them — which is why the activity gives seven codes instead of telling learners to check their work carefully.

This also sets up the module's central problem. In Unit 4 you will read a generated routine that looks complete, and the assumptions it makes will be just as invisible as the ones in your own procedure — with the added difficulty that they are somebody else's assumptions, made by a process that has no obligation to be consistent about them.

7.4 On the historical material

If learners ask which parts of the “first program” story they are supposed to believe, the honest answer is the one on the student page: publication date and the existence of the ordered table are established; priority relative to Babbage’s unpublished notebooks and the division of labour on Note G are contested and cited to the relevant historical sources in the reading. Direct them to `course/historical-notes.md` and to Unit 7 rather than resolving it in discussion.

Do not let the contested points be settled by an AI system’s confident summary. That is a live and slightly awkward demonstration of the module’s own thesis, and it is worth making explicitly if it comes up.

7.5 The checkpoint

This unit’s checkpoint is `cp01-ordered-steps`. Its question, options, correct answer and diagnostics are defined once, in the `checkpoint` block at the end of this file; `../lab/checkpoints.py` and the browser self-check both read that block, so there is a single source and the routes cannot drift apart.

It tests the same literal-reader pass as the activity, and in particular the distinction between a genuine omission and background knowledge a procedure is not required to supply. If a learner reports failing it, ask them to say which nouns in the given procedure were never introduced before use — that is the question the framework’s own feedback pushes them towards.

```
python3 selfcheck.py run cp01-ordered-steps          # or use the browse
```

7.6 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.

```
{
  "id": "cp01-ordered-steps",
  "unit": 1,
  "title": "What the procedure left unsaid",
  "kind": "multi",
  "question": "A learner writes this procedure for making tea: (1) Boil
  "answer": [
    0,
    1,
    2
  ],
  "options": [
    "How long to wait in step 4.",
    "What to add the teabag to -- no container is ever named.",
    "How much water to boil.",
    "The fact that tea is a hot drink."
  ],
  "diagnostics": {
    "0,1,2,3": "Three of your four are right, but 'tea is a hot drink' i
    "0,1": "You have the timing and the missing container. One more is g
    "0": "The timing is missing, yes -- but look again for the step that
  },
  "hint": "Read it as a machine would: refuse to supply anything not wri
  "explanation": "Unstated assumptions are invisible to the person who h
}
```

8 Unit 2 — What a language model does differently

Media for this unit:

- **Interactive Animation**: This unit features an interactive animation (`next-token.html`). Please view the web version of the course to interact with it.
- **Interactive Animation**: This unit features an interactive animation (`transformer-system-layers.html`). Please view the web version of the course to interact with it.
- **Interactive Animation**: This unit features an interactive animation (`agentic-effort.html`). Please view the web version of the course to interact with it.

Reading time: about 9 minutes. Activity: about 16 minutes.

8.1 Start from what changes

Unit 1 gave you three useful properties of an ordered procedure: a specified sequence, inspectable state, and reproducible execution under fixed conditions. A language-model service is also software with algorithms, state, and control flow. The important difference is what its generated answer exposes to you: the answer is not an execution trace that proves each claim. This unit explains why that changes how you check it.

8.2 The mechanism, at the level this module will defend

Most current large language models use a neural-network architecture called a **Transformer** [L1]. Text is divided into **tokens**, which are words, parts of words, punctuation, or other text fragments. The model turns those tokens into numerical representations. Through many layers, **attention** lets the representation at each position combine information from relevant positions in the available context. The final layer assigns a score, converted to a probability, to each possible next token.

During generation, one token is selected and appended. The model then processes the longer context and produces a new distribution for the next position. A paragraph is built by repeating this step. Attention is a way of combining context; it is not a database lookup, a citation, or a truth test.

The base model's **pre-training** adjusts many numerical parameters to improve next-token prediction over a large training corpus. A deployed assistant is usually more than this base model. It may be **post-trained** on demonstrations and preference judgements to follow instructions [L2], and a product may add search, document retrieval [L3], calculators, code execution, memory, or other tools. Some systems also produce intermediate plans or reasoning text. These layers can improve the result, but none makes a generated claim correct by definition. Visible reasoning is material to inspect, not a guaranteed trace of every internal computation.

Optional detail: Computational effort versus formal reasoning

State-of-the-art models increasingly use "agentic" loops or generate hidden "thinking" tokens before returning an answer. What they mean by "thinking" is that the system uses the standard next-token generation process as a scratchpad. Instead of directly outputting the final answer, it generates intermediate text—breaking the problem down, planning steps, calling tools like search or a calculator, critiquing its own drafts, and trying alternative paths. These intermediate tokens are often hidden from the user, giving the illusion of a silent cognitive process.

This extra effort often produces significantly better and more reliable answers. However, it is a hazard for verification if you confuse this *computational effort* with *formal reasoning*. An agentic loop is still generating text and choosing actions probabilistically. It is not building a mathematically guaranteed proof trace. The basic principle remains: no matter how much intermediate "thinking" the system did, the final output must still be independently verified.

This is the level of mechanism the module needs: transformer processing over a finite context, next-token generation, and optional system layers around the model. Detailed claims about internal representations or

any particular product need separate evidence.

8.3 Three consequences

8.3.1 Selection can involve sampling

When generation samples from the token distribution rather than always choosing the highest-probability token, the same prompt and context can produce different text on different occasions. A service can also change its hidden instructions, model version, tools, or settings between runs.

Compare that with Unit 1. There, a deterministic procedure should repeat its output when its implementation, input, environment, and settings are fixed. Here, variation may reflect sampling or a changed service. Deterministic decoding can also repeat the same answer. Repetition can reveal stability or change, but neither outcome establishes correctness, and re-asking is not an independent check.

8.3.2 Correctness is not computed at the generation step

The base generation objective produces a continuation that fits its context. In a system with no search, database, or tool attached, the generation step does not consult an external record of what is true.

Some deployed systems do attach retrieval or search. That is a genuine difference and it changes *where* you aim your checking: whether the retrieved material is appropriate, whether it says what the answer claims, and whether the answer preserves its qualifications. It does not remove the need to check.

8.3.3 Fluency does not establish correctness

This is the one to keep.

A true sentence and a false sentence can both be well formed, specific, confident in register, and correctly punctuated. A false answer need not look visibly worse.

Now list the cues you actually use when judging whether a piece of writing is reliable: it reads fluently, it is specific, it is not hedging, it is properly formatted, the details sound plausible. A text-generating system can produce every one whether or not the content is correct. Surface cues may help you notice a poor answer, but they cannot substitute for checking the underlying claim.

The established term for fluent false output is **hallucination**; it is defined in `course/glossary.md` and used consistently across this module. Treat the word with some caution. It suggests a malfunction, an unusual event. It is the same process that produced the correct output, on an occasion when the plausible continuation and the true one happened to differ.

8.4 Two things this unit is not saying

It is not saying the output is random. It is not: token probabilities constrain which continuations are likely. That still does not make a generated claim self-verifying.

It is not saying avoid these systems. In Unit 4 you will use one deliberately. The claim here is narrower and more awkward than “do not trust AI”: inspection alone cannot establish which case you are in, so confidence must be proportionate to an appropriate check.

8.5 The two systems, side by side

	Ordered procedure (Unit 1)	Text generation (this unit)
Same input, same output?	Yes, when implementation and environment are fixed.	Depends on decoding settings and the surrounding service.

	Ordered procedure (Unit 1)	Text generation (this unit)
Repeating the run	Tests reproducibility; disagreement signals a changed condition or fault.	May show stability or variation; neither verifies correctness.
What you can inspect	Named state and operations, if the implementation exposes them.	Prompt, output, and any exposed sources or tool records; not a proof trace for every claim.
What "wrong" means	Diverges from the intended specification, even if the procedure ran exactly as written.	Diverges from the task, evidence, or intended specification.
How you find an error	Trace the steps and test against the specification.	Use evidence appropriate to each claim: tests, sources, calculations, or measurements.
What good output looks like	Requires comparison with the specification	A bad output can look as polished as a good one

The last row is why the rest of the module exists.

8.6 The triage

If you cannot judge output by looking at it, you need a habit that runs before you look. The question is not "does this look right?" It is:

> How would this be wrong, and what check catches that?

That question has three broad answers, and sorting outputs into them takes a couple of seconds once you are used to it.

Needs exact checking. The output has a right answer that you can compare against: a calculation, a conversion, a piece of code, a data transformation. The check is to run it against values you already know, or to execute it against an automated test suite. Examples include unit conversion, budget formulas, recipe scaling, a data-parsing function, or a small mathematical algorithm. This is the category where a check can be decisive about the values it covers — and it is the category Units 3 to 5 live in.

Needs source checking. The output makes a claim about the world: a date, a citation, an attribution, a legal or institutional requirement, an API specification, or a summary of a document you have not read. Nothing can be executed here. The check is to look it up in the official record: the source document, the handbook, the API documentation, the licence, or another authoritative record. Running the output a second time, or asking the system whether it is sure, may produce a useful revision, but it is not **independent verification**: the second output can preserve the first output's unsupported assumptions. Check against an appropriate source, calculation, test, or independently derived method.

Low-risk drafting. An error would be cheap and visible, and you were going to rework the text anyway: a first draft of a paragraph you will rewrite, candidate titles, a docstring you will edit, a meeting-summary outline, or a rephrasing of your own notes. This category is real, and pretending otherwise produces a policy nobody follows.

Two warnings about using the triage.

Sort by **how an error would reach you**, not by how important the task feels. Important work can contain low-risk drafting; a throwaway task can contain a number that has to be exactly right.

And notice that a single output usually spans categories. Generated code with an explanatory comment about who invented the method contains both an exactly checkable part (the code) and a source-checkable one (the historical comment), and they fail independently. That combination is precisely what Unit 4 puts in front of you.

8.7 A note on the general case

The operational lesson is broader than text generation: an answer's own assurance is not independent evidence for that answer. Re-prompting and self-critique can improve an output [T9], but they can also preserve a shared error; a stronger check uses evidence appropriate to the claim, such as an authoritative source, an exact calculation, or an independently derived test. The closing unit sets this lesson through the older, published idea of second-order reasoning; it does not present that as a theorem about language models. See `course/second-order.md`; the closing Unit 9 teaches from it, and nothing on this page, in the activity, or in either Unit 2 checkpoint depends on either.

8.8 Summary

- A transformer uses attention to combine token context and produce next-token probabilities; generation selects and appends tokens repeatedly.
- Modern assistants may add post-training, retrieval, tools, and intermediate reasoning around that model. These layers change capability, not the need for evidence.
- Selection is often sampled, so repetition can vary an answer without independently verifying it.
- Correctness is not a quantity computed at the generation step.
- Fluent presentation can accompany correct or incorrect content, so surface cues alone are insufficient.
- Triage by failure mode: exact checking, source checking, low-risk drafting.

Now do `activity.md`, then take checkpoints `cp02-risk-triage` and `cp02-system-layers`.

9 Unit 2 activity — triage ten outputs

Required work: about 16 minutes. Optional extensions at the end add more if you want them.

Ten things a text-generating system has produced. For each one, decide which kind of checking it needs before you could rely on it, and name the check you would actually run.

The three categories, from the reading:

- **E — needs exact checking.** There is a right answer you can compare against, by running it or by recomputing it against values you already know.
- **S — needs source checking.** It is a claim about the world. It cannot be executed, only looked up in the record.
- **L — low-risk drafting.** An error would be cheap and visible, and you were going to rework the text anyway.

9.1 Part 1 — sort them (7 minutes)

Copy the table and fill in the last two columns. One or two words for the category; one clause for the check.

#	The output	E / S / L	The check I would run
1	A regular expression it wrote to validate UK postcodes in a form — a regular expression is a pattern a computer uses to test whether text has the right shape.		

#	The output	E / S / L	The check I would run
2	A statement that a named professional body has required a particular qualification since 2019.		
3	A rewrite of your own lecture notes as a bulleted revision list.		
4	A unit conversion in a lab report: 2.3 kg expressed in pounds.		
5	A summary of a journal article you have not read.		
6	Five candidate titles for your dissertation.		
7	A database query that deletes every row older than a given date — an instruction to a database that permanently deletes records.		
8	A claim that a piece of software behaves a certain way unless you change a setting — checkable in its manual or by trying it.		
9	A rewritten opening paragraph for an essay you are still drafting.		
10	A list of six references at the end of a generated literature review.		

Do not agonise. If an item feels like it sits in two categories, write both letters and move on — several of them genuinely do, and the answer key says which and why.

9.2 Part 2 — two follow-up questions (3 minutes)

A. What would stop a low-risk item being low-risk?

Take one item you marked **L** and write one condition that would move it out of that category — "if I were not going to reread it", or "if this went straight to a client". Low risk is a property of your situation, not of the text.

B. Which action is not independent verification?

For one item, write down something a person might do that *feels* like independent verification but is not. Examples: asking the system whether it is sure; regenerating and seeing whether you get the same thing; noticing that the output is very specific. Say what the action can reveal, and what it cannot establish.

9.3 Part 3 — one real item of your own (2 minutes)

Add an eleventh row: something you have personally used or are about to use an AI system for. Sort it, and name the check.

If your honest answer is “I would not have checked that”, write that down too. That is more useful than an aspirational answer, and Unit 8 asks you to look back at it.

9.4 Part 4 — locate the system layer (4 minutes)

Write one layer beside each description: **attention**, **post-training**, **retrieval**, **next-token scoring**, or **pre-training**.

Description	Layer
Combines information from relevant token positions in the available context.	
Shapes the base model into an assistant that follows instructions.	
Supplies external documents, whose relevance and support still need checking.	
Ranks possible continuations; its scores are not a truth predicate.	
Learns statistical patterns for next-token prediction from a large corpus.	

Check that your labels describe different functions. In particular, attention does not fetch a source, retrieval does not certify a source, and pre-training is not the complete deployed product.

9.5 Now take the checkpoints

`cp02-risk-triage` gives you three outputs and asks you to match each to the kind of checking it needs — the same judgement you made in Parts 1 to 3. **`cp02-system-layers`** checks the five distinctions from Part 4.

```
python3 selfcheck.py run cp02-risk-triage
python3 selfcheck.py run cp02-system-layers
```

Run the Python route from `course/lab/`, or use the browser self-check. Each checkpoint names the specific mix-up behind the most likely wrong answers.

Keep your table. Units 4 and 5 hand you an output that spans two of these categories at once, and you will want to have made the distinction on easy cases first.

9.6 Optional extensions

Optional. These sit outside the core study time, are not required for the checkpoint, and nothing later in the module depends on them.

Finish Part 2A across the whole table (about 3 minutes). Part 2A asked for one **L** item. Do the rest: for every item you marked **L**, name the condition that would move it out of that category. The pattern that emerges is the useful part — the conditions are all about where the output goes next, never about the text.

Build the full list of false verification (about 4 minutes). Part 2B asked for one item. Write down as many actions as you can that feel like independent verification and are not, and for each say what it can reveal and what it cannot establish. The answer key lists six; getting to four unaided is a good result.

Rewrite the three weakest checks (about 5 minutes). Go back through your Part 1 table and find the three entries where you wrote something closer to an intention than a method — “check it carefully”, “look at it again”. Replace each with a check somebody else could run and get the same answer from. If nothing about a check could fail, it is not a check.

9.6.1 Activity Answers

10 Unit 2 answers and marking guidance

Mark the reason, not the letter. A learner who writes **S** for item 1 and justifies it by “the postcode rules are published, so I would check the specification before testing anything” has understood more than one who writes **E** because code is usually **E**.

10.1 Part 1 — the ten items

#	Intended	Why	The check
1	E (with S first)	A regular expression has a right answer relative to a specification — but the specification itself is a published claim about the world.	Look up the published postcode format, then test the expression against a list of addresses you know to be valid and a list you know to be invalid.
2	S	A claim about what an organisation requires, and since when. Nothing here can be executed.	Find the requirement on the body’s own published material. A secondary summary is not the record.
3	L, conditionally	Low risk *because you hold the original*. The failure mode is silent omission: material dropped in the rewrite, then revised from.	Read the rewrite against your own notes once. If you would not do that, it is not low-risk.
4	E	A conversion with one correct value.	Recompute it yourself and compare. Check the direction of the conversion as well as the number.
5	S	A summary is a set of claims about a document.	Read the article. Note the awkward consequence: the only real check costs the labour the summary was meant to save.
6	L	Candidate titles are options you will choose between; a bad one is visible and free to discard.	None needed beyond reading them.

#	Intended	Why	The check
7	E, with the stakes raised	Correctness is exactly checkable, but the error is not recoverable once it has run.	Run it against a copy, or run the equivalent selection first and inspect what would have been deleted. Never test destructive operations on the live data.
8	S or E	Both routes are legitimate. The manual for the version you are using is the record; trying the software is a direct observation.	Whichever is cheaper — but check the version. A default setting that changed between versions is a common and quiet failure.
9	L	You are still drafting; an error is yours to fix on the next pass.	None, provided you actually reread it.
10	S, and the highest-value check here	Two separate failures: a reference may not exist, and an existing reference may not support the claim attached to it.	Look up each reference in a catalogue or database, then confirm that each source says what the review claims it says.

Items 1, 3 and 8 are the productive ones to discuss. Item 8 in particular demonstrates that the categories are about how an error would reach you rather than about what kind of object the output is: the same claim about a default setting can be checked by reading the manual or by trying the software, and both are real verification.

10.2 Part 2A — what stops something being low-risk

The required work asks for **one** item marked **L**; covering the rest of them is an optional extension, so do not expect to see it. Good answers name a change in **the learner's situation**, not in the text:

- the output goes somewhere you will not reread it;
- it is passed to someone else who will assume it was checked;
- it becomes the basis of later work, so an error propagates instead of sitting still;
- the original you would have compared against is no longer available;
- the cost of being wrong shifts from your marks to someone else's decision.

Weak answers say "if it were more important". Importance is not the axis. Push for the mechanism by which the error would survive undetected.

10.3 Part 2B — actions that are not independent verification

The required work asks for **one** item, so expect one row rather than a list; compiling the full set is an optional extension. All six are worth naming aloud in feedback whichever way round it was done.

Action	What it cannot establish
Asking the system whether it is sure	The revision may be useful, but it is generated by the same system and is not independent evidence.

Action	What it cannot establish
Regenerating and comparing	Agreement or variation can be diagnostic, but neither establishes correspondence with the world.
The output being very specific	Specificity is a property of the text, produced whether or not the content is right. A fabricated page number looks exactly like a real one.
The output being long and well organised	Same argument. Formatting is generated, not earned.
The code running without error	Running is not the same as being correct. A routine can execute cleanly and return the wrong number at every index — Unit 5 gives you exactly that.
Recognising the answer as familiar	Familiarity tracks how often something is said, which is close to what the generation step optimises. It is the least reliable cue on this list.

10.4 Why plausibility is not correctness

This is the point the activity exists to establish, and it is worth stating explicitly in feedback.

Plausibility and correctness are different properties. Plausibility is a property of the text: whether it reads like the sort of thing that follows. That is what the generation step is doing, and it does it well and uniformly. Correctness is a relation between the text and something outside it — a specification, a calculation, a document, a fact about the world. Nothing in the generation step evaluates that relation.

Two consequences follow, and they are the ones learners need.

First, **surface quality is insufficient evidence**. Fluency, specificity, and confident formatting can accompany correct or incorrect content. They may help a reader notice a poor answer, but they do not establish the underlying claim.

Second, **verification needs evidence separable from the assertion**. A revision can improve an answer [T9], but it does not provide the independence of a value you derived, a document you opened, or a test you ran. This is why the next three units build an independent reference before any generated code is looked at, rather than after.

10.5 For an instructor

Expected time: 16 minutes for the required work. Formative. The three optional extensions at the end of the activity add roughly 3, 4 and 5 minutes and are outside the core budget.

Strong responses get at least eight of the ten broadly right, name a check that could fail for each, and flag at least one item as legitimately spanning two categories.

What to watch for:

Observation	Diagnosis	Response
Everything marked E or S; nothing L	The learner has heard “check everything” and produced a policy they will not keep.	The **L** category is not permission to be careless; it is what makes the other two affordable. Ask which items they would actually check next Tuesday.
Everything sorted by stakes	The most common error, and the one `cp02-risk-triage` targets.	Item 7 versus item 5: both matter, and they need completely different checks.

Observation	Diagnosis	Response
“I would check it carefully” as the check	Intention offered as method.	Ask what a failing result would look like. If nothing could fail, it is not a check.
Item 5 marked E	A summary treated as something with a computable right answer.	It is a set of claims about a document. The only check is the document.
Anthropomorphic reasoning in the justifications (“it wouldn’t lie about a date”)	Intent attributed to the system.	Return to the mechanism: there is no step at which truth is evaluated, so there is nothing to be honest or dishonest about.

Part 3 — the learner’s own item — is the part worth reading if you read only one. It shows what they actually do rather than what they can classify, and a learner who honestly writes “I would not have checked that” has given you the most useful sentence in the exercise. Do not penalise it.

10.6 The checkpoint

This unit’s checkpoints are `cp02-risk-triage` and `cp02-system-layers`. Their prompts, options, correct matching and diagnostics are defined once, in the `checkpoint` blocks at the end of this file; `../lab/checkpoints.py` and the browser self-check both read those blocks, so there is a single source and the routes cannot drift apart.

It uses three items distinct from the ten above and tests the same judgement. The expected failure is a swap driven by sorting on apparent seriousness rather than on failure mode; the framework’s feedback names that directly, so let learners read it rather than correcting them.

```
python3 selfcheck.py run cp02-risk-triage          # or use the browser
python3 selfcheck.py run cp02-system-layers
```

10.7 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.

```
{
  "id": "cp02-risk-triage",
  "unit": 2,
  "title": "Triage by how it can be wrong",
  "kind": "match",
  "question": "A language model produces each of these. Match each outpu
  "answer": [
    0,
    1,
    2
  ],
  "options": [
    "Needs exact checking -- run it against known values.",
    "Needs source checking -- verify against the record.",
    "Low-risk drafting -- errors are cheap and visible."
  ],
  "prompts": [
```

```
    "A function that claims to compute compound interest.",
    "A statement that a named researcher published a particular paper in
    "A first draft of a covering email you will rewrite anyway."
  ],
  "diagnostics": {
    "1,0,2": "You have the first two swapped. Code is checked by *executi
    "0,2,1": "You have treated the citation as low-risk drafting and the
    "2,1,0": "This looks like sorting by how serious each item sounds, n
  },
  "explanation": "Plausibility is uniform across all three -- the model
}

{
  "id": "cp02-system-layers",
  "unit": 2,
  "title": "Separate the assistant's system layers",
  "kind": "match",
  "question": "Match each function to the layer that performs it. Use ea
  "answer": [
    0,
    1,
    2,
    3,
    4
  ],
  "options": [
    "Attention",
    "Post-training",
    "Retrieval",
    "Next-token scoring",
    "Pre-training"
  ],
  "prompts": [
    "Combines information from relevant token positions in the available
    "Shapes instruction-following behaviour after base-model training.",
    "Supplies external documents that still need relevance and support c
    "Ranks possible continuations; the scores are not a truth predicate.
    "Learns statistical patterns for next-token prediction from a large
  ],
  "diagnostics": {
    "2,1,0,3,4": "Attention and retrieval are swapped. Attention combine
    "0,1,3,2,4": "Retrieval and next-token scoring are swapped. Retrieva
    "0,4,2,3,1": "Pre-training and post-training are swapped. Pre-traini
  },
  "hint": "Ask whether the function combines existing context, shapes be
  "explanation": "A deployed assistant is a system of distinguishable la
}
```

11 Unit 3 — Meet the Bernoulli numbers

Media for this unit:

- **Interactive Animation**: This unit features an interactive animation (`first-divergence.html`). Please view the web version of the course to interact with it.
- **Interactive Animation**: This unit features an interactive animation (`ada-indexing.html`). Please view the web version of the course to interact with it.

Reading time: about 11 minutes. Activity: about 20 minutes. Course items: see `course/course-map.md` for items 2 and 4.

11.1 What you should be able to do afterwards

1. Say, in plain terms, what a Bernoulli number is and where the sequence comes from.
2. Derive B4 from the recurrence, by hand, and show your working.
3. Explain why two correct-looking programs can disagree about B1, and say which convention this module uses.
4. Translate between Lovelace's Note G labels and modern indices.

11.2 Why this unit comes before the AI unit

In Unit 4 you will ask an AI system to produce a routine that returns Bernoulli numbers. It will produce something. It will look competent. Generated code almost always does.

The question that decides whether that output is useful to you is not "does this look right" but "what should the answer be". If you cannot answer the second question independently, you have no way of answering the first. So this unit gives you the independent answer first, deliberately, before anything is generated.

This ordering is the whole point of the module's approach to critical thinking: build enough understanding to judge, then use the tool.

11.3 What a Bernoulli number is

The Bernoulli numbers are a fixed sequence of exact rational numbers — ordinary fractions — with one value for each whole-number index from 0 upwards. They are written `B0`, `B1`, `B2`, and so on.

They are not approximations, not estimates, and not a matter of taste. Given the index, the value is completely determined. That property is what makes them a good subject for this module: there is a right answer, it can be written down exactly, and disagreement can always be resolved.

Here is the sequence this module works with.

n	B_n	n	B_n
0	1	10	5/66
1	-1/2	12	-691/2730
2	1/6	14	7/6
3	0	16	-3617/510
4	derive in the activity	18	43867/798
6	1/42	20	-174611/330
8	shown after your derivation	30	8615841276005/14322

All odd `B_n` for `n > 1` are exactly zero. That is not a rounding artefact or a convenient approximation; those values are zero, precisely.

Two things are worth noticing before you read on. First, the values do not shrink tidily — `B14` is larger than `B12`, and `B30` is enormous. Second, the denominators are already awkward by `B12`. Both facts matter later: a routine that returns decimals will look plausible for small indices and drift once the numbers grow.

11.4 Where the sequence comes from

You do not need this section to complete the unit. It is here so the numbers do not feel as though they were invented for the exercise.

Ask a simple question: what is $1^2 + 2^2 + \ldots + 9^2$? The answer is 285, and you can get it by adding nine numbers. But there is also a closed formula that gives the answer for any upper limit, without adding anything up. There is another for cubes, another for fourth powers, and so on.

Write those formulas in a single general form covering every power at once, and the same sequence of fractions appears in the coefficients each time. That sequence is the Bernoulli numbers. They were studied in the seventeenth and early eighteenth centuries and were long-established mathematics by the time Ada Lovelace selected them as the subject of Note G. For sources on Note G itself, see `course/references.md`.

11.5 The rule that generates them

Every Bernoulli number is fixed by the ones before it, through one identity:

```
sum_{j=0}^{n} C(n+1, j) * B_j = 0
```

`C(n+1, j)` is the binomial coefficient — "n plus one choose j" — the numbers from Pascal's triangle. For a given `n`, the identity gives you a sum of terms, all of them known except the last, which contains `B_n`. Substitute what you know, and solve.

Starting from `B0 = 1`, this generates the whole sequence, one value at a time, in a fixed order, with no judgement required at any step. It is an ordered procedure in exactly the sense of Unit 1.

11.5.1 Worked example: B2

Take `n = 2`. The identity gives three terms:

```
C(3,0)·B0 + C(3,1)·B1 + C(3,2)·B2 = 0
```

The binomial coefficients for `n + 1 = 3` are 1, 3, 3. Substituting `B0 = 1` and `B1 = -1/2`:

```
(1)(1) + (3)(-1/2) + (3)(B2) = 0
```

Work the known part first:

```
1 - 3/2 = -1/2
```

So:

```
-1/2 + 3·B2 = 0
       3·B2 = 1/2
         B2 = 1/6
```

Which is what the table says. Notice what you did not need: a calculator, a computer, or anyone's assurance. You needed the rule, the earlier values, and care with fractions.

In the activity you will do the same for `B4`, where `n = 4` gives five terms instead of three. That is the derivation checkpoint `cp03-b4-by-hand`, and it is worth doing before you read further into this page. Stop here, open the activity, and attempt Task 1 from your own working. The rest of this reading is placed behind a labelled disclosure on the course site because it contains the result.

11.5.2 Post-attempt Reading

11.6 Trap 1 — there are two conventions

Here is a scenario you will meet in Unit 5. Two programs both claim to compute Bernoulli numbers. They agree on every value except one. One says `B1 = -1/2`, the other says `B1 = +1/2`. Everything else — `B2 = 1/6`, `B4 = -1/30`, all of it — is identical.

Neither program has a bug.

There are two established conventions for this sequence:

	B0	B1	B2	B4	B6
First Bernoulli numbers	1	**-1/2**	1/6	-1/30	1/42
Second Bernoulli numbers	1	**+1/2**	1/6	-1/30	1/42

Both are standard, both are published, and they differ in exactly one value. Which one you want depends on which formula you are feeding it into; neither is more correct than the other in the abstract.

This module fixes the first convention: `B1 = -1/2`. That is what the reference implementation in `course/lab/reference_bernoulli.py` returns, it is what the checker enforces, and it is a clause you will write into a prompt in Unit 6.

The lesson generalises well beyond Bernoulli numbers. When two competent sources disagree, the first thing to check is not who made a mistake but whether they were answering the same question. A specification that omits a convention has not made the convention go away; it has only made the disagreement harder to diagnose. In Unit 5 this failure mode has a name: `b1-sign`.

11.7 Trap 2 — Ada's indexing is not ours

The second trap is historical, and it is easier to walk into.

Since every odd Bernoulli number above `B1` is zero, Lovelace does not give those zeros a label. She numbers only the values that are actually present. Her label numbers are therefore one smaller: her `Bk` is the modern `B(k+1)`:

Ada's label	modern index	value
B1	B2	1/6
B3	B4	-1/30
B5	B6	1/42
B7	B8	-1/30

Note G's worked example computes Ada's **B7**, which in modern notation is **B8 = -1/30**.

This mapping is shown only after the derivation task because two rows contain the same numerical value as `B4`. The checkpoint tests whether you can produce that value from the recurrence, not whether you can find it in a later table.

Nothing here is an error on Lovelace's part. It is a different notation that happens to use identical symbols, which is exactly what makes it hazardous. A generated routine can be a faithful reading of Note G and a broken modern component at the same time. Historical fidelity and specification compliance are not the same requirement, and a routine can fail by being right about the wrong century. In Unit 5 this failure mode is named `ada-indexing`.

The mapping is also available in code, in the oracle: `ada_to_modern_index` and `modern_to_ada_index`.

11.8 Two routes to verification

The activity offers two routes. They carry equal weight, cover the same checkpoints, and are marked the same way, which is to say not at all.

Table-based verification. Derive `B4` on paper from the recurrence, then compare a claimed set of values against the table above, index by index. This is what a reviewer does when they have a document rather than a program: line up the claim against the reference and look for the first place they part company. Reading a specification against a known-good table is a professional technique in its own right, not a reduced version of running the code.

Python with exact rationals. Do the same comparison in code, using `fractions.Fraction` from the standard library. Nothing to install.

```
from fractions import Fraction

b0 = Fraction(1)
b1 = Fraction(-1, 2)
b2 = Fraction(1, 6)

print(b0 + 3 * b1)          # the known part of the n = 2 identity
```

Use `Fraction`, not `float`. `float` would give you `0.1666666666666666` for `B2`, which is not `1/6` and will not compare equal to it. For small indices the difference is invisible; by `B30` it is not. Exact arithmetic is a requirement of this module's interface contract, not a stylistic preference.

11.9 Terms used here

- **Convention** — a choice that is fixed by agreement rather than derived, such as `B1 = -1/2`. Two people following different conventions can both be right and still disagree.
- **Exact rational arithmetic** — computing with fractions as numerator and denominator, so no rounding occurs. Python's `fractions.Fraction` does this.
- **Oracle** — a trusted reference you compare an answer against. Here, `course/lab/reference_bernoulli`

These also appear in `course/glossary.md`.

11.10 Check yourself

This unit has three checkpoints:

- `cp03-b4-by-hand` — derive B4 from the recurrence
- `cp03-convention` — identify which convention a program is using
- `cp03-ada-index` — translate a Note G label into a modern index

Run them by either route:

```
python3 selfcheck.py run cp03-b4-by-hand    # or use the browser self-ch
```

For `cp03-b4-by-hand`, a wrong answer will not show you the right one. It will name the specific slip your answer looks like and give you the next step. That is deliberate: deriving the value is the objective, so handing it over would remove the exercise.

12 Unit 3 activity — derive it before you trust it

Time: about 20 minutes. Read `page.md` first, at least as far as the worked example for B2.

Two routes run through this activity: **table-based verification** and **Python with exact rationals**. They cover the same ground and end at the same checkpoints. Pick whichever suits how you work. If you want to do

both, do the paper working first — the code route is more useful as a check on your own arithmetic than as a substitute for it.

Keep your working. You will refer back to it in Unit 5, and it belongs in your evidence pack for Unit 8.

12.1 Task 1 — Work out B4 by hand (about 8 minutes)

This is the central task of the unit. Do not look it up, do not compute it with a library, and do not ask an AI system. The value of this exercise is entirely in having produced the number yourself, because in the next unit you will need a reference that did not come from a machine.

The recurrence, again:

```
sum_{j=0}^{n} C(n+1, j) * B_j = 0
```

Take `n = 4`.

Step 1. Write out the identity with all five terms, from `j = 0` to `j = 4`. Every binomial coefficient uses `n + 1 = 5` on top.

```
C(5,0)·B0 + C(5,1)·B1 + C(5,2)·B2 + C(5,3)·B3 + C(5,4)·B4 = 0
```

Step 2. Fill in the five binomial coefficients. Row 5 of Pascal's triangle is 1, 5, 10, 10, 5, 1 — you need the first five of those. Write them into your copy of the identity.

Step 3. Substitute the values you already have for `B0`, `B1`, `B2` and `B3`. All four are in the table on the student page, and `B2` is the one you watched being derived. `B3` is odd and above 1, so you know what it is without looking.

Step 4. Add up everything that does not contain `B4`. Keep it as a single fraction — do not convert to decimals at any point. You should end up with one small fraction plus a whole-number multiple of `B4`, and the two together equal zero.

Step 5. Solve for `B4`. Watch the sign: the whole sum is zero, so the term containing `B4` must be the negative of everything else.

Write the result as a fully reduced fraction. Keep every line of working — the intermediate steps are what you will check, not just the final number.

12.1.1 If you are taking the Python route

Do steps 1 to 5 on paper first. Then use Python to check your own arithmetic line by line, rather than to produce the answer.

```python
from fractions import Fraction
from math import comb

# Step 2 - the coefficients you wrote down
print([comb(5, j) for j in range(5)])

# Step 3 and 4 - the part of the sum that does not contain B4.
# Fill in the values you substituted; do not import the oracle.
b0 = Fraction(1)
b1 = Fraction(-1, 2)
b2 = Fraction(1, 6)
b3 = Fraction(0)
```

```
known = comb(5, 0) * b0 + comb(5, 1) * b1 + comb(5, 2) * b2 + comb(5, 3)
print(known)            # compare against your Step 4 fraction
```

If the printed value matches the fraction you got by hand, your substitution and addition are sound and any remaining error is in Step 5. If it does not match, you have found your slip without being told the answer, which is the more useful outcome.

Do not finish this task by calling `reference_bernoulli.bernoulli(4)`. That routine is the thing you are learning to be able to check. Using it here would invert the whole exercise.

12.1.2 If you are taking the table-based route

Your working on paper is the deliverable. Before moving on, do one extra pass: re-derive Step 4 by putting all four known terms over a common denominator of 6, independently of how you did it the first time. Two routes to the same intermediate value is a much stronger check than reading your own arithmetic twice.

12.2 Task 2 — Diagnose a disagreement (about 5 minutes)

A colleague sends you a Python file. Its output begins:

```
B0 = 1
B1 = 1/2
B2 = 1/6
B3 = 0
B4 = -1/30
B5 = 0
B6 = 1/42
B10 = 5/66
B12 = -691/2730
```

Compare it against the table in `page.md`, index by index, and write down:

1. Every index where it disagrees with this module's table.
2. Whether that pattern of disagreement looks like a mistake in the code, or like something else.
3. The one sentence you would add to a specification so that this disagreement could not arise again. Write it as a requirement, not as a complaint — something a reader could test against.

Keep sentence 3. You will compare it with the clause that appears in Unit 6's precise prompt, and with the failure-mode name it corresponds to in Unit 5.

12.3 Task 3 — Read an index from Note G (about 4 minutes)

Lovelace labels only the Bernoulli numbers that are not zero. The zeros never receive a label at all.

Fill in this table from that rule alone. Work out the modern index for each of her labels rather than copying the mapping from the student page, then check your answers against it.

Ada's label	modern index	value
B1		
B3		
B5		
B7		

Then answer in one sentence each:

1. What is the general rule connecting one of Lovelace's labels to its modern index?
2. Note G's worked example computes the number Lovelace calls B7. If a modern routine were asked to reproduce Note G's result and returned its own `B7` instead, what would go wrong, and would the returned value be a genuine Bernoulli number?

Question 2 is the one to sit with. A routine can return real Bernoulli numbers, correctly computed, and still be wrong — because the values are filed under the wrong labels. That is a failure that survives being read carefully, and it is why Unit 5 compares index by index against a reference instead of asking whether output "looks like" Bernoulli numbers.

12.4 Task 4 — Optional CS extension (about 5 minutes, outside the core budget)

Clearly optional. Nothing later in the module depends on it.

Write a short function that takes a list of claimed values and reports the **first** index at which it disagrees with the table on the student page — not every disagreement, just the first, with the index, the claimed value and the expected one.

```
from fractions import Fraction

TABLE = {
    0: Fraction(1),
    1: Fraction(-1, 2),
    2: Fraction(1, 6),
    3: Fraction(0),
    # ... complete this from the student page, as fractions, not floats
}

def first_divergence(claimed: dict[int, Fraction]) -> str:
    """Return a description of the first index where claimed differs from
    ...
```

Two constraints worth honouring, because they are the same ones the module's own checker works under:

- Compare `Fraction` against `Fraction`. Do not compare against `float`, and do not use a tolerance. These values are exact and an approximate comparison would hide precisely the errors you are looking for.
- Report the first divergence rather than all of them. A single specific index is a reproducible debugging boundary; do not assume it causes every later mismatch.

You have now written, in miniature, the tool that Unit 5 hands you.

12.5 Now check yourself

This unit has three checkpoints. Attempt them from your own working, not from the student page.

- `cp03-b4-by-hand` — the value you derived in Task 1
- `cp03-convention` — the disagreement you diagnosed in Task 2
- `cp03-ada-index` — the mapping you worked out in Task 3

Either route:

```
python3 selfcheck.py run cp03-b4-by-hand    # or use the browser self-ch
```

Run this from the `course/lab` folder (setup: `course/lab/README.md`).

To attempt all three at once:

```
python3 selfcheck.py run --unit 3
```

The browser route at `build/site/selfcheck.html` has the same three checkpoints, the same answers and the same feedback, and needs nothing installed.

A wrong answer on `cp03-b4-by-hand` will not reveal the correct value. It will name the specific slip your answer resembles and point you at the step to redo. Deriving the number is the objective, so being given it would defeat the exercise. If you are stuck, ask for the hint and go back to Step 2.

12.5.1 Activity Answers

13 Unit 3 answers and guidance

For learners after attempting the activity, and for instructors reviewing it. Working through this page before attempting Task 1 removes the only part of the module that has to happen before an AI system is involved.

Checkpoint answers are not restated here. Checkpoints are defined once in `course/lab/checkpoints.py`, and both self-check routes grade against that file. This page explains the reasoning; the checkpoints do the marking.

13.1 Task 1 — B4 by hand

13.1.1 The full working

With `n = 4`, the identity has five terms and every binomial coefficient is taken from row 5:

```
C(5,0)·B0 + C(5,1)·B1 + C(5,2)·B2 + C(5,3)·B3 + C(5,4)·B4 = 0
```

The coefficients are 1, 5, 10, 10, 5. Substituting `B0 = 1, B1 = −1/2, B2 = 1/6` and `B3 = 0`:

```
(1)(1) + (5)(−1/2) + (10)(1/6) + (10)(0) + (5)(B4) = 0
```

Term by term:

```
1 − 5/2 + 5/3 + 0 + 5·B4 = 0
```

Putting the known part over a denominator of 6:

```
6/6 − 15/6 + 10/6 = 1/6
```

So:

```
1/6 + 5·B4 = 0
      5·B4 = −1/6
        B4 = −1/30
```

Which agrees with the table on the student page, and with `course/lab/reference_bernoulli.py`.

13.1.2 Where the working usually goes wrong

Four slips account for nearly all incorrect answers, and each of them is worth recognising rather than merely correcting.

- **Sign lost at the last step.** The learner reaches `1/6 + 5·B4 = 0` and reads off `1/30` instead of `−1/30`. The sum equals zero, so the term containing `B4` is the negative of everything else. This is

the single most common error, which is worth saying aloud: the same slip, in code, produces a routine that disagrees with the reference at every even index.
- **Stopping at n = 2.** Answering `1/6` means the recurrence was run for `B2` rather than `B4`. Usually a misread of which `n` to take, rather than an arithmetic error.
- **Wrong binomial row.** Using row 4 (1, 4, 6, 4, 1) instead of row 5. The identity uses `n + 1` on top, not `n`, and it is easy to substitute `n` reflexively.
- **Decimals.** Converting to decimals partway through gives `-0.0333...`, which is not exact and cannot be compared for equality with `-1/30`. This is the same error that the module later names `not-fraction` when it appears in generated code, and it is worth pointing out that it began here, on paper.

13.1.3 Why this task exists

The value matters less than its provenance. In Unit 4 an AI system will produce a Bernoulli routine, and you will need a value that did not come from that system to judge it by. A number you derived is such a value. A number you looked up in the same conversation is not.

The course site withholds the numerical value in the initial reading and animations, then places the later tasks, this guidance, and the revealing mapping table behind labelled disclosures. This is an instructional boundary, not an access-control claim: a learner can still open the source or disclosure. The direction remains to attempt the derivation before doing so.

13.2 Task 2 — Diagnosing the disagreement

1. Where it disagrees. At exactly one index: `B1`. The file gives `+1/2` where this module's table gives `-1/2`. Every other value shown — `B0`, `B2`, `B3`, `B4`, `B5`, `B6`, `B10`, `B12` — agrees exactly.

2. What that pattern means. A single-index disagreement, with everything around it untouched, is not the signature of a bug. Bugs in a recurrence propagate: an error at `B1` would corrupt `B2`, which would corrupt `B4`, and so on down the sequence. Here nothing propagates, which tells you the two programs are computing different things on purpose.

They are. The file uses the **second Bernoulli convention**, `B1 = +1/2`; this module uses the **first**, `B1 = -1/2`. Both are standard, both are published, and they differ in exactly this one value. The colleague's file is not broken. The specification that both of you were working to was incomplete.

This is the failure mode the checker names `b1-sign`. It is worth noticing that the name describes what the output looks like, not a diagnosis of intent — the same output would result from a genuine sign error, and the checker cannot tell those apart. You can, by looking at whether anything else moved.

3. The specification sentence. Any wording is acceptable provided it is testable and unambiguous. A reasonable version:

> Use the first Bernoulli convention, in which `B1 = -1/2`.

The properties that make it a good requirement:

- It names the convention, so a reader can look it up.
- It states the value, so a reader who has never heard of the convention can still comply.
- It is checkable against a single output, which means a test can be written for it directly.

A version such as "use the correct Bernoulli numbers" fails all three: there is no such thing as the correct one here, and nothing can be tested against it. Compare your sentence with the corresponding clause in Unit 6's precise prompt. If yours is close, you have independently reinvented a piece of requirements engineering, which is what specification work mostly is.

13.3 Task 3 — Reading an index from Note G

The completed table, which is the locked mapping used throughout the module:

Ada's label	modern index	value
B1	B2	1/6
B3	B4	-1/30
B5	B6	1/42
B7	B8	-1/30

1. The general rule. Lovelace labels only the non-zero Bernoulli numbers. The modern scheme spends an index on every term, including the odd ones that are zero, so her label numbers are one smaller: her `Bk` is the modern `B(k+1)`, for odd `k`. Her `B7` is the modern `B8`.

Two ways of getting this wrong are worth naming. Treating her labels as modern labels gives `B7 = 0`, since the modern `B7` is an odd index above 1. Doubling — guessing that her `B7` is the modern `B14` — comes from noticing that only even indices are involved, but the rule is a shift by one, not a scaling. Writing out all four pairs makes the pattern visible immediately, which is why the task asks for the whole table rather than a single row.

2. Note G's B7. Note G's worked example computes Ada's `B7`, which is the modern `B8`. Its value is `-1/30`.

A modern routine that returned its own `B7` would return `0`, since every odd modern index above 1 is exactly zero — a plainly visible mismatch. The more dangerous version of this error is the one where nothing looks wrong at all: a routine that renumbers the non-zero terms consecutively returns `1/6` for `B1`, `-1/30` for `B2`, `1/42` for `B3` — its `bernoulli(k)` is the modern `B(2k)`. It skips the zeros as Note G does, though these are not Lovelace's labels either (hers run B1, B3, B5, B7 = modern B2, B4, B6, B8). Every value it returns is a genuine Bernoulli number, correctly computed, exactly represented. Every one is filed under the wrong index.

That combination — real values, wrong labels — is why reading the code carefully does not catch it. There is nothing in the code to catch. The mismatch only exists relative to a specification, and it only becomes visible when you compare against a reference index by index. The module names this failure mode `ada-indexing`, and you will meet a working example of it in Unit 5.

A note on the history: Note G's published table contains an error — established from a facsimile as an inverted division in operation 4 — while its attribution (who introduced it) remains contested. This unit does not depend on the error at all, and Unit 7 handles it properly. See `course/historical-notes.md` before repeating any account of it.

13.4 Task 4 — Optional CS extension

No single correct implementation. A reasonable one:

```
from fractions import Fraction

def first_divergence(claimed: dict[int, Fraction]) -> str:
    for n in sorted(TABLE):
        if n not in claimed:
            return f"B{n}: no value supplied"
        if claimed[n] != TABLE[n]:
            return f"B{n}: claimed {claimed[n]}, expected {TABLE[n]}"
    return f"no divergence up to B{max(TABLE)}"
```

Points worth drawing out in review:

- **Iterating in index order** is what makes "first" meaningful. Iterating over the claimed dictionary in insertion order would report whichever divergence happened to be inserted first, which is not the same thing.
- **Exact comparison.** `!=` between two `Fraction` objects is exact. If a learner has written `abs(a - b) < 1e-9`, that tolerance will pass a float-based implementation for small indices and fail it at `B30`, which is the worst possible behaviour: it hides the bug until the bug is expensive. Comparing a `Fraction` against a `float` in Python does work, and that is part of the hazard — it will silently succeed for `Fraction(1, 4)` and fail for `Fraction(1, 6)`.
- **A missing value is a finding, not a crash.** A routine that has no answer at an index has failed the specification just as surely as one with a wrong answer. The module's own checker treats this as its `raises` failure mode.

If a learner wants to go further, ask them to add the property check that every odd index above 1 is exactly zero. That check needs no table at all — it tests a property rather than a value — and it catches the `ada-indexing` failure mode immediately, because a routine using Lovelace's numbering has no zeros in it anywhere.

13.5 Checkpoints

Attempt these from your own working:

- `cp03-b4-by-hand`
- `cp03-convention`
- `cp03-ada-index`

```
python3 selfcheck.py run cp03-b4-by-hand    # or use the browser self-ch
```

Both routes ask the same questions and give the same feedback. Answers and diagnostics live in `course/lab/checkpoints.py`.

13.6 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.

```
{
  "id": "cp03-b4-by-hand",
  "unit": 3,
  "title": "B4 by hand",
  "kind": "fraction",
  "question": "Using the recurrence sum_{j=0}^{n} C(n+1, j) * B_j = 0 wi
  "answer": "-1/30",
  "diagnostics": {
    "1/30": "Right magnitude, wrong sign. Check the sign of the term you
    "1/6": "That is B2, not B4. Easy to land on if the recurrence was so
    "-1/6": "You are one step from the answer. -1/6 is the negated sum o
    "0": "B4 = 0 usually means the wrong binomial row: row 4 (1, 4, 6, 4
    "-1/42": "You have mixed up denominators: 42 belongs to B6. Recomput
  },
  "hint": "Take n = 4. The identity gives C(5,0)B0 + C(5,1)B1 + C(5,2)B2
  "explanation": "You now know what the answer *should* be before any AI
}
```

```
{
  "id": "cp03-convention",
  "unit": 3,
  "title": "Which convention is this?",
  "kind": "choice",
  "question": "A program prints B0 = 1, B1 = 1/2, B2 = 1/6, and matches
  "answer": 0,
  "options": [
    "It uses the second Bernoulli convention (B1 = +1/2); every other va
    "It has a sign bug that happens to affect only B1.",
    "It is using Lovelace's Note G indexing.",
    "It is correct and this module's oracle is wrong."
  ],
  "diagnostics": {
    "1": "It looks exactly like a sign bug, and that is what makes it da
    "2": "Note G's indexing shifts *which* numbers get which labels, so
    "3": "'The oracle is wrong' is a checkable claim, not a dead end --
  },
  "explanation": "Both conventions are standard and published. Neither p
}
{
  "id": "cp03-ada-index",
  "unit": 3,
  "title": "Reading Ada's index",
  "kind": "integer",
  "question": "Note G computes the number Lovelace calls B7. In modern n
  "answer": 8,
  "diagnostics": {
    "7": "That is the label, not the value's modern index. Lovelace numb
    "14": "Doubling is the right instinct but the wrong rule. Her B1 is
    "6": "You have moved the wrong way. Her label numbers are one smalle
  },
  "explanation": "Ada B1/B3/B5/B7 are the modern B2/B4/B6/B8, so Note G'
}
```

14 Unit 4 — Ask AI to generate the routine

Reading time: about 9 minutes. Activity: about 12 minutes, plus optional extensions. Course items: see `course/course-map.md` for items 2 and 8.

Prerequisite: Unit 3. This unit assumes you have derived B4 yourself and can state which convention this module uses. If you have not, go back — the whole point of the ordering is that your reference exists before the generated output does.

14.1 What you should be able to do afterwards

1. Recognise what an underspecified prompt leaves for the generation to decide.
2. Name the failure modes this module expects in generated Bernoulli code, using the module's own codes.
3. Separate the parts of a generated response that a test could settle from the parts only a source could settle.
4. Produce a saved artefact — prompt, output, date, and your own triage — that Unit 5 will check and Unit 8 will use as evidence.

14.2 The prompt

Here is the prompt for this unit, in full:

> Write a Python function that returns Bernoulli numbers.

That is the entire thing. One sentence, no context, no requirements.

It is important that you do not read this as a trick. It is not a deliberately bad prompt constructed to make a point; it is the sentence people actually type. When you know roughly what you want and the tool is open in front of you, this is what gets written, and a good deal of the time it produces something that works well enough. That is precisely why it is worth examining.

14.2.1 The precise prompt is not in this unit

There is a precise version of this prompt. It states the interface, the convention, the return type, the range and the behaviour at odd indices. You will build it in Unit 6.

It is deliberately withheld until then, and this page is telling you so rather than quietly omitting it. The reason is straightforward: if you see a good specification before you have felt the need for one, you learn that good specifications exist, which you already knew. If you write the vague prompt first, look at what comes back, and only then see the precise version, you learn what each clause of it is for — because you watched the failure it prevents.

14.3 What the sentence leaves open

Read the prompt again and count the decisions it does not make.

- **Which convention?** `B1 = -1/2` or `B1 = +1/2`. Both standard, as Unit 3 established.
- **What return type?** Exact rationals, or floating-point decimals.
- **What range of indices?** Starting where, running how far.
- **What happens at the odd indices?** Exact zeros, or absent, or something else.
- **Does exactness matter?** Nothing in the prompt says it does.

Somebody has to settle every one of those before the code can exist. You settled none of them.

This is the mechanism worth holding on to. A language model produces a statistically plausible continuation of the text it is given. Where your text is silent, the continuation still has to contain *something* — a function

body cannot be partially absent. What it contains reflects the patterns in similar code the model was trained on, not your requirements, because your requirements were never stated.

So an unstated requirement does not appear in the output as a gap or a question. It appears as a confident default. That is much harder to notice than a gap would be, and it is the reason this unit exists.

14.4 The failure modes to expect

The module names five failure modes in generated Bernoulli code. These are the codes the checker emits in Unit 5, so the names are worth learning as they stand.

code	meaning
`b1-sign`	Correct under the *second* convention, `B1 = +1/2`.
`odd-terms`	Odd terms above B1 are not exactly zero.
`ada-indexing`	Only the non-zero terms are numbered, as in Note G.
`off-by-one`	Every value shifted one index.
`not-fraction`	Returns `float` or another inexact type.

A few notes on each.

`b1-sign` follows directly from the missing convention clause. The resulting routine is not broken — it is correct under a different, equally published convention. It disagrees with this module at exactly one index and nowhere else.

`odd-terms` matters more than it looks. `B3` is not approximately zero; it is zero. A routine that omits the odd indices, or returns a tiny non-zero decimal for them, has failed a property that holds exactly.

`ada-indexing` becomes noticeably more likely if your prompt mentions Ada Lovelace or Note G, because text about Note G uses her numbering. A routine produced this way is a faithful reading of the historical source and a broken modern component at the same time.

`off-by-one` is the one that most resists being read. Every value returned is a genuine Bernoulli number, computed correctly and represented exactly. Each is filed under the wrong index. There is nothing in the code for a careful reader to catch, because the error only exists relative to a specification.

`not-fraction` is the default outcome of asking for numerical Python, because most numerical Python is float-based. For small indices it is convincing: `-0.0333...` really is close to `-1/30`. Then the denominators grow. `B30` is `8615841276005/14322`, and accumulated rounding error is no longer invisible. A test that checked `B0` to `B12` with a tolerance would pass such a routine.

What these five have in common is the important part: **none of them makes the code look suspicious.** They are invisible to reading and visible to testing. That asymmetry is the entire argument for the verification lab in Unit 5.

The checker recognises three further conditions — `raises`, `import-error` and `no-function` — which describe code that does not run at all. Those announce themselves, so this unit does not dwell on them.

14.5 The sixth failure mode, which no test catches

Generated code frequently arrives with explanation attached. Comments, a docstring, a paragraph of context. Something along these lines:

```
# Ada Lovelace used this same recurrence in Note G (1843).
def bernoulli(n):
    ...
```

Suppose you check the arithmetic and it is correct. What does that tell you about the comment?

Nothing at all.

The code and the comment were produced by the same process, and they fail independently. The code got checked because code is checkable — you can run it against known values. The historical claim sat directly beside it, phrased with the same confidence, dated with the same specificity, and was checked by nobody.

This is what makes generated output awkward to evaluate: it is a mixture of things with completely different failure modes. Arithmetic you can test. Assertions about the world you can only look up. Both parts can be equally polished, so surface quality alone does not tell you which claim is reliable.

The module is not claiming that the example comment above is false. The point is sharper than that: **you do not know**, and the correctness of the code adjacent to it is not evidence either way. Unit 7 deals with what is actually established about Note G, and `course/historical-notes.md` is careful about which parts are settled and which are contested. Bring that same care to any historical claim that arrives inside generated code.

This is checkpoint `cp04-trust-triage`.

14.6 A worked example: a claim that does not survive checking

The same triage applies to claims that arrive with no code attached at all. Here is one phrased with total confidence:

> Gödel's theorem proves that an AI cannot verify its own output.

You do not need formal logic to triage this sentence. It names a theorem about one kind of system, then applies it to the broad category "an AI" without showing that the theorem's conditions or conclusion transfer. Mark the theorem statement for a source check and the transfer for a separate argument. Unit 9 defines the relevant formal terms, states Gödel's and Tarski's results within their actual scope, and explains why neither directly settles what a language model can verify.

14.7 What to do with the output

Save it. Exactly as generated, without tidying it up. Save the prompt too, word for word, and the date you ran it. The untidied version is the evidence; a cleaned-up version is a summary of the evidence, which is not the same thing.

You will need it twice more:

- **Unit 5** runs it through the checker and names what is wrong with it.
- **Unit 8** uses it in your evidence pack, alongside your verification result and your correctness defence.

Do not run it yet. Two reasons. The first is pedagogical: your predictions are far more informative if they are written down before the checker gives you the answer. The second is practical, and it holds outside this module too.

> **Safety.** Generated code is code from a source you cannot vouch for. Read it before you run it, and run it in a disposable local environment. The checker in Unit 5 imports and executes the file you give it, which is a real consideration and not a formality.

14.8 If you do not have access to an AI system

Use the sample implementations in `course/lab/examples/`:

- `wrong_b1_sign.py`
- `wrong_odd_terms.py`
- `float_output.py`

- `off_by_one_range.py`
- `correct_example.py`

These are real implementations exhibiting the failure modes above, and the activity works identically with them. This is an equal route, not a substitute: in one respect the supplied examples are the better teaching set, because every failure mode is guaranteed to be present and one of the five files is correct, so the exercise of telling them apart is genuine.

The same applies if you have access but would rather not use it, or if what you generate turns out to be flawless. A correct generated routine is a perfectly good result; take one of the examples as a second specimen and triage that too.

14.9 Privacy note

Do not paste confidential, personal, unpublished or assessment-restricted material into an AI system. Nothing in this unit requires it — the prompt is one sentence about a published mathematical sequence. The wider course covers this properly; see `course/course-map.md` for item 5.

14.10 Check yourself

This unit has one checkpoint:

- `cp04-trust-triage` — what you can trust before testing

```
python3 selfcheck.py run cp04-trust-triage    # or use the browser self-
```

Both routes ask the same question and give the same feedback. It is about the comment, not the code, which tends to be the surprising part.

15 Unit 4 activity — generate, then triage

Required work: about 12 minutes. Read `page.md` first. Optional extensions at the end add more if you want them.

You will produce one file and one table. Keep both. Unit 5 checks the file, and Unit 8 uses both as evidence.

Do not run the generated code during this activity. Predictions written before the checker sees them are worth far more than predictions written after.

Safety. Generated code comes from a source you cannot vouch for. Read it before you ever run it, and run it in a disposable local environment.

15.1 Task 1 — Generate (about 4 minutes)

Use whichever AI system you have access to; the exercise does not depend on which you pick. Give it exactly this, and nothing else:

```
Write a Python function that returns Bernoulli numbers.
```

Do not add context, do not mention Ada Lovelace or conventions or fractions, and do not follow up. One prompt, one response.

Save the result to a file — `generated_bernoulli.py` is a reasonable name — exactly as it came back, without tidying the formatting, deleting comments, or fixing anything you notice on the way past. What you noticed is the interesting part, and you will record it in Task 2.

Alongside it record your provenance: the prompt word for word, the date, and the kind of system in general terms (you need not name a product). That is what makes your work checkable by somebody else.

15.1.1 If you do not have AI access

Take one of the files in `course/lab/examples/` as your specimen instead: `wrong_b1_sign.py`, `wrong_odd_terms.py`, `float_output.py`, `off_by_one_range.py`, `correct_example.py`.

Pick one **without reading its docstring first** — the docstrings name the failure mode, which is the thing you are practising spotting. Copy the code into your own file, triage it in Task 2, and read the docstring afterwards.

This route is equivalent. In one respect it is stronger: every failure mode is guaranteed to be present somewhere in that folder, and one of the five files is correct, so telling them apart is a real judgement rather than a formality.

15.2 Task 2 — Triage (about 5 minutes)

Go through your specimen — code, comments, docstrings, and any prose that came with it — and put every substantive part into one of three columns.

Trust	Doubt	Must test
I will rely on this now, and here is my reason.	Plausible, but I cannot confirm it.	And here is the specific check that would settle it.

Three rules make this worth doing:

1. **Every "trust" entry needs a reason**, and "it looks right" is not one. "I derived B4 by hand in Unit 3 and this agrees with it" is one.
2. **Every "must test" entry needs a named check** that somebody else could run and get your answer from. "Test the code" is not a check; "compare `bernoulli(0)` through `bernoulli(30)` against the oracle as exact fractions, index by index" is.
3. **Historical and factual claims go in their own rows**, separate from the code beneath them, because they need a different kind of checking.

Six rows is enough. Use these two prompts to make sure you have covered the ground, and leave the rest for the optional extension:

- Which convention does it use, and what type does it return? Is `1/6` exact, or a decimal?
- What does it do at `n = 3`, at `n = 0`, and at `n = 1`?

15.3 Task 3 — Predict the checker's verdict (about 1 minute)

Before Unit 5 runs anything, commit to a prediction in writing. Using the module's failure-mode codes — `b1-sign`, `odd-terms`, `ada-indexing`, `off-by-one`, `not-fraction` — write down two things:

1. Which codes you expect the checker to report for your specimen, if any.
2. **The first index at which you expect it to diverge** from the reference.

Being wrong here costs nothing and teaches more than being right. The prediction exists so that Unit 5 tells you something about your own judgement, not only about the code, so write it down even if you have no idea.

15.4 Task 4 — Handle the prose separately (about 2 minutes)

If your specimen contained any claim about history, mathematics or attribution — in a comment, a docstring, or the surrounding explanation — copy it out on its own.

Then answer:

1. What kind of evidence would settle whether it is true?
2. Is that the same kind of evidence that would settle whether the code is correct?
3. If the code passes every test you can devise, what have you learnt about the claim?

If your specimen had no such claim, use this one, which is the checkpoint's example:

```
# Ada Lovelace used this same recurrence in Note G (1843).
```

Answer the three questions about it. Do not attempt to establish whether it is true — Unit 7 handles Note G properly, and `course/historical-notes.md` is careful about which parts of that story are established and which are contested. What this task is asking is what you would *need* in order to know.

15.5 Before you move on

Keep these four things together:

1. `generated_bernoulli.py` — or your copied example — exactly as it came.
2. Your prompt, the date, and the kind of system used.
3. Your three-column triage table.
4. Your written prediction of the checker's verdict.

Unit 5 needs items 1 and 4. Unit 8 needs all four for your evidence pack. See `course/learner-evidence-template.md` for the format it will eventually take.

15.6 Now check yourself

This unit has one checkpoint:

- `cp04-trust-triage` — what you can trust before testing

Either route:

```
python3 selfcheck.py run cp04-trust-triage    # or use the browser self-
```

Run this from the `course/lab` folder (setup: `course/lab/README.md`).

The browser route at `build/site/selfcheck.html` has the same question, the same answer and the same feedback, and needs nothing installed.

The checkpoint asks about a comment rather than about code, which catches people out. Attempt it after Task 4, while the distinction between the two kinds of claim is fresh. Tasks 1 to 4 are all you need for it.

15.7 Optional extensions

Optional. These sit outside the core study time, are not required for the checkpoint, and nothing later in the module depends on them.

Extend the triage table (about 4 minutes). Task 2 asked for six rows and two prompts. Carry on with the two that were left out, adding a row for each answer:

- What is the largest index you would trust it at, and why that one?

- Does any comment make a claim that no test could settle?

The mirror-image prediction (about 2 minutes). Task 3 predicted what is wrong. Now predict the opposite: if your specimen turns out to be entirely correct, what single check convinced you of that, and what would that check have missed? Add one sentence on how confident you are and what would change your mind. A check that could not have failed has told you nothing.

Triage a second specimen (about 8 minutes). Take a file from `course/lab/examples/` — without reading its docstring — and run Tasks 2 and 3 on it as well. If your own generated routine looked correct, this is where you get a contrasting case.

For learners who write code (about 10 minutes). Without running your specimen, read it and answer:

- What algorithm is it using — the binomial recurrence, the Akiyama–Tanigawa transform, a hard-coded table, something else?
- Would that algorithm produce exact values if the arithmetic type were changed to `Fraction`? Distinguish an inexact *representation* from an incorrect *method*: these fail differently and are fixed differently.
- Is there an index at which it would be slow rather than wrong?
- Does it define behaviour for negative `n`?

Then write the smallest test you can that would distinguish the correct example in `course/lab/examples/` from all four wrong ones. Note how much of your Unit 3 grounding that test relies on.

15.7.1 Activity Answers

16 Unit 4 answers and guidance

For learners after attempting the activity, and for instructors reviewing it.

Most of this activity has no single correct answer, because what comes back from a generation is not fixed. What *is* fixed is the standard a good triage meets, and that is what this page sets out. Where the activity asks for judgement, the guidance describes what a strong answer does rather than supplying the answer.

Checkpoint answers are not restated here. `cp04-trust-triage` is defined in `course/lab/checkpoints.p` and both self-check routes grade against that file.

Expected time: 12 minutes for the required work, Tasks 1 to 4. The optional extensions at the end of the activity add roughly 4, 2, 8 and 10 minutes, sit outside the core budget, and are not needed for the checkpoint. The guidance below marks which material belongs to them, and does not assume any of it has been done.

16.1 Task 1 — Generate

Nothing to mark. Two things to check about your own work.

Did you save it untouched? The commonest failure at this step is quiet tidying — reformatting, deleting a comment that looked wrong, renaming a variable. Each of those edits is a judgement you made, and once folded into the file it becomes indistinguishable from what was generated. Your evidence pack in Unit 8 depends on being able to tell the two apart.

Did you record provenance? Prompt, date, and the kind of system. This module avoids naming commercial products, and you can too: “a general-purpose conversational AI assistant, accessed via its web interface, on 4 March” is adequate provenance. What matters is that a reader could understand how the artefact came to exist.

If your generated routine turned out to be entirely correct, that is a real and unremarkable outcome — this is a well-known sequence and there is a great deal of correct Bernoulli code in the world. It does not mean

you have missed the point. It means your triage in Task 2 has to justify the trust rather than merely record it, which is the harder version of the exercise. The optional "triage a second specimen" extension gives you a contrasting case from `course/lab/examples/` if you want one.

16.2 Task 2 — Triage

16.2.1 What a strong table looks like

A strong triage is recognisable by three features, none of which is about being right.

Reasons in the "trust" column reference something outside the specimen. The strongest reason available to you at this point in the module is your own Unit 3 working: "this returns `1/6` for `B2`, which matches the value I derived by hand". That is evidence. "The code is clean and well-commented" is not evidence about correctness; it is an observation about style, and generated code is reliably well-styled regardless of whether it works.

Checks in the "must test" column are specific enough that someone else could run them. Compare these two entries:

- "Must test: whether the values are right." — not a check. It names an anxiety.
- "Must test: `bernoulli(n)` for `n = 0` to `30`, compared against `reference_bernoulli.bernoulli` as exact `Fraction` equality, reporting the first index that differs." — a check. It has a range, a comparison method, an authority, and a definition of failure.

The difference is the difference between doubting something and being able to resolve the doubt. A "must test" entry with no named check belongs in the "doubt" column, and moving it there honestly is better work than leaving it where it flatters you.

Claims about the world are on separate rows from claims about behaviour. If one row covers "the recurrence and the historical comment above it", the triage has already blurred the distinction the unit is about.

16.2.2 The prompts, worked through

Task 2 requires the first three of these — convention, return type, and the behaviour at `n = 3`, `n = 0` and `n = 1`. The last two belong to the optional "extend the triage table" extension, so a required-work-only table will not cover them; they are worked through here for anyone who goes on to it.

Which convention does it use? Determinable from the code alone, without running it, in most implementations. Look for a special case at `n == 1`, or a sign flip applied to a single term. `course/lab/examples/correct_example.py` shows what an explicit answer looks like: the Akiyama–Tanigawa transform naturally yields `B1 = +1/2`, so the file negates it and says so in a comment. `wrong_b1_sign.py` runs the same kind of recurrence and simply omits that step. The visible difference between them is one branch.

If you cannot tell from reading, that is a finding in itself, and it belongs in "must test" with the check "evaluate `bernoulli(1)` and compare against `-1/2`".

What type does it return? Look at the imports first. `from fractions import Fraction` at the top is a strong signal; its absence in numerical code is a stronger one. Then look at the literals: `1.0` and `/` on floats produce floats. This is the `not-fraction` failure mode, and it is the default outcome rather than an unusual one, because most numerical Python is float-based.

Worth internalising: `-0.03333333333333333` is not `-1/30`. It is close, it prints convincingly, and it will fail an exact equality test. For small indices nothing appears wrong. `float_output.py` in the examples folder exists to make this concrete — it passes a casual eyeball check across the early indices and diverges once the denominators grow.

What does it do at `n = 3`, `n = 0`, `n = 1`? These three indices are where the specification is least likely to have been guessed correctly. `n = 3` should be exactly zero; a routine that omits odd indices, raises, or returns a small non-zero value has hit `odd-terms`. `n = 0` should be `1`. `n = 1` is where the convention shows. Three single-index checks, chosen because each one isolates a different failure mode — which is what makes them worth more than thirty arbitrary ones.

What is the largest index you would trust it at? The question is designed to be uncomfortable. Most people's honest answer, before testing, is "the ones I have personally checked", which is a much shorter list than the routine's apparent range. That gap between apparent range and verified range is what the differential test in Unit 5 closes. Note also that a float-based routine has no clean answer to this question at all: the failure is gradual rather than sudden, so there is no index at which it starts being wrong.

Does any comment make a claim a test could not settle? See Task 4.

16.2.3 The failure modes, and the missing requirement each one traces back to

This mapping is the substance of the unit, and it is what Unit 6 builds on directly.

failure mode	the requirement that was never stated
`b1-sign`	Which convention. The prompt never said `B1 = -1/2`.
`odd-terms`	That every odd index above 1 must be present and exactly zero.
`ada-indexing`	That modern indexing is required, covering the zero terms.
`off-by-one`	Which index the sequence starts at, and what `bernoulli(0)` means.
`not-fraction`	That returns must be exact `fractions.Fraction`, never `float`.

Read the right-hand column on its own. It is very nearly the precise prompt from Unit 6, arrived at from the other direction. That is the general lesson: a specification is not written in advance by a sufficiently careful person. It accumulates, one clause per failure that has already happened. Every clause in a good prompt is a bug somebody met.

Note too that none of these five is a coding error. Each is a decision that the prompt left open and the generation filled in. The code does what it does correctly; it was answering a question you did not quite ask.

16.3 Task 3 — Predicting the checker's verdict

No correct answer, and no penalty for a wrong prediction. What the task is for is calibration: after Unit 5 runs the checker, you will be able to compare what you expected against what was there, and that comparison tells you something about your own reading that no amount of further reading would.

Two patterns are worth naming in advance.

Predicting no failures because the code looked fine. Very common, and the five named failure modes are precisely the ones that leave code looking fine. `off_by_one_range.py` in the examples folder is the extreme case: clean code, exact arithmetic, every returned value a genuine Bernoulli number, and every one under the wrong index. There is nothing in it for a careful reader to find, because the error exists only relative to a specification.

Predicting a first divergence too late. People tend to nominate a large index, on the reasoning that small cases are easy. In practice the first divergence is very often at `B1`, because that is where the convention

shows. If you predicted `B12` and the answer is `B1`, that is the most informative outcome the task can produce.

The mirror-image prediction — what convinced you a correct routine was correct, and what that check would have missed — is now an **optional extension** rather than required work, so do not expect to see it. It is worth raising in feedback even when it has not been attempted: a check that passes tells you only about what it tested. Checking `B0` to `B12` with a tolerance would pass `float_output.py`, which is wrong, and it would also pass a correct routine. A check that cannot fail for the right reason has not told you anything.

16.4 Task 4 — Handling the prose separately

Taking the checkpoint's example:

```
# Ada Lovelace used this same recurrence in Note G (1843).
```

1. What evidence would settle it? A primary source, or a cited secondary source: the published Notes themselves, or scholarship that quotes them. It is a claim about a document that exists, so it is settled by consulting the document. Note that the claim also has parts that could be true and false separately — a date, an attribution, and an assertion that a *particular* recurrence was used. An answer that treats it as one atomic claim has already lost some resolution.

2. Is that the same evidence that settles the code? No, and the difference is categorical rather than one of degree. The code is settled by execution: run it against known values and observe. The claim cannot be executed at all. Different failure modes, different checks, different kinds of authority — and no amount of one substitutes for the other.

3. If the code passes every test, what have you learnt about the claim? Nothing. This is the whole point of the checkpoint and it is worth stating without softening: correct arithmetic is not weak evidence for an adjacent historical claim, it is *no* evidence. The two were produced by the same process and fail independently. If anything, the checkable part being checked makes the uncheckable part more dangerous, because the specimen as a whole now feels verified.

The module is not asserting that this particular comment is false. The point is that you do not know, and that nothing about the code puts you in a position to find out. Unit 7 addresses what is genuinely established about Note G; `course/historical-notes.md` states which parts of that account are settled and which are contested. Both are careful in a way that a generated comment has no mechanism to be.

A note on why this happens. It is not that the historical part is generated less carefully. Generated text is a single fluent continuation throughout; the arithmetic and the history are produced the same way and are equally plausible on the surface. What differs is not the output but your access to the truth: one part you can settle in seconds by running it, and the other requires a library. Both parts can be equally fluent while differing in reliability. That asymmetry may not announce itself, which is why triage has to be a deliberate step rather than an impression.

The student page's short Gödel example applies the same triage outside code: check what the cited theorem actually establishes, then check the separate argument that transfers it to "an AI". Unit 9 supplies the formal definitions and full scope; they are not required for identifying the unsupported transfer here.

16.5 Optional extensions

None of this is required work, and none of it is needed for the checkpoint. The notes below are for learners who choose an extension, and for instructors fielding questions about one.

Algorithm identification. The examples folder contains two distinct methods, which is deliberate: `reference_bernoulli.py` uses the binomial recurrence and `correct_example.py`

uses the Akiyama–Tanigawa transform. Two independent routes agreeing on exact values is much stronger evidence than one routine agreeing with a copy of itself, and that is a general principle of differential testing rather than a detail of this module.

Representation versus method. The distinction the task is after. A routine using a sound algorithm with `float` arithmetic is fixed by changing the type; the method was never wrong. A routine using Lovelace's indexing is not fixed by any change of type, because the arithmetic was exact all along and the labels were wrong. Diagnosing which of the two you are looking at determines whether the fix is one line or a rewrite, and generated code gives no outward sign of the difference.

Slow rather than wrong. A naive recurrence that recomputes every earlier term from scratch is correct and impractical. Correctness and efficiency are separate properties with separate checks, and the module's checker deliberately tests only the first.

Negative n. Undefined in the contract, and most generated routines say nothing about it — they return something meaningless, or raise an unhelpful error. The checker names this `raises` when it occurs inside the required range. The lesson is that "the specification does not say" and "the behaviour does not matter" are different statements, and code cannot tell them apart.

The smallest distinguishing test. A compact answer checks four indices:

- `bernoulli(1)` — separates `b1-sign` from everything else.
- `bernoulli(3)` — must be exactly zero; catches `odd-terms` and `ada-indexing`.
- `bernoulli(0)` — must be `1`; catches `off-by-one`.
- `isinstance(bernoulli(2), Fraction)` and `bernoulli(2) == Fraction(1, 6)` — catches `not-fraction`.

Four checks, one per failure mode, each isolating a single decision the vague prompt left open. Notice how much Unit 3 that required: you cannot design this test without knowing the convention, the zero property, and the indexing scheme. That is what "build understanding before using the tool" means in practice — not a slogan about diligence, but the reason you were able to write a four-line test instead of thirty arbitrary ones.

16.6 Checkpoint

- `cp04-trust-triage` — what you can trust before testing

```
python3 selfcheck.py run cp04-trust-triage    # or use the browser self-
```

Both routes ask the same question and give the same feedback. The answer and its diagnostics live in `course/lab/checkpoints.py`.

16.7 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.

```
{
  "id": "cp04-trust-triage",
  "unit": 4,
  "title": "What you can trust before testing",
  "kind": "choice",
  "question": "An AI returns a Bernoulli routine with a comment: '# Ada
  "answer": 0,
  "options": [
    "Treat the comment as an unverified historical claim and check it sep
```

```
    "Accept it -- the code was right, so the comment is probably right t
    "Treat the comment as settled and move on; comments do not affect beh
    "Accept it if the model sounds confident about the date."
  ],
  "diagnostics": {
    "1": "Correct code is no evidence at all about an adjacent historica
    "2": "It is true that a comment does not affect execution -- but it t
  },
  "explanation": "Generated output is a mixture of things with completel
}
```

17 Unit 5 — Verification lab

Media for this unit:

- **Interactive Animation**: This unit features an interactive animation (`first-divergence.html`). Please view the web version of the course to interact with it.
- **Interactive Animation**: This unit features an interactive animation (`self-certification.html`). Please view the web version of the course to interact with it.

Reading time: about 16 minutes. Activity: about 24 minutes.

Course items: 2 (how to evaluate AI outputs) and 8 (effective AI use in CS). See `../course-map.md`.

17.1 What this unit is for

In Unit 4 you asked for a routine that computes Bernoulli numbers, and you got one. It looked like Python. It read like Python. It probably ran without complaint. None of that is a result.

This unit turns "it looks right" into "here is what I checked, and here is what the check said". That move — from impression to evidence — is the single most transferable thing in the module, and it applies far beyond Bernoulli numbers.

17.2 Four ideas, named

These four terms are standard in software engineering. They are worth learning by name, because once you have the names you can ask for the thing.

Test oracle. A source of correct answers that you trust *independently* of the thing you are testing. A second opinion from the same source is not an oracle. In this lab the oracle is `../lab/reference_bernoulli.py`.

Differential testing. Running two implementations of the same specification on the same inputs and comparing the outputs. Here the two implementations are your generated routine and the oracle.

Property-based checking. Instead of asserting individual values, you assert a statement that must hold across a whole class of inputs — for example, *every* odd Bernoulli number above `B1` is exactly zero.

First divergence. The lowest input at which two implementations disagree. It gives a precise starting point for debugging; it does not establish the cause of later disagreements.

17.3 Why this oracle deserves to be trusted

An oracle that you simply declare trustworthy is not an oracle; it is a second guess with better branding. Here is the actual case for this one, and you can verify every part of it in about a minute.

It uses exact rational arithmetic. Every value is a `fractions.Fraction`. Bernoulli numbers have large numerators and denominators — `B12` is `-691/2730`, `B30` is `8615841276005/14322` — and binary floating point cannot represent them. The oracle never rounds, so any disagreement it reports is a genuine disagreement, never a rounding artefact. This is also why the checker compares values *exactly* rather than "to within a small tolerance".

It is checked against values it did not compute. The oracle carries a table of values transcribed from published Bernoulli-number tables, and it compares its own recurrence against every one of them before it prints anything. Run it:

```
cd course/lab
python3 reference_bernoulli.py
```

The last line of a healthy run reads:

```
OK: oracle agrees with all 22 published values, odd terms are zero.
```

Twenty-two values transcribed from published tables, including odd entries that are zero. The exact table used still has to be recorded and checked by the human release reviewer; `course/references.md` [M2] tracks that blocker. If any entry disagreed, the program would print `FAIL` and exit with an error status, and nothing else in the lab would mean anything until that was resolved.

It is checked at a large index. `B30 = 8615841276005/14322` is in that table deliberately. Small indices hide bugs. A floating-point implementation, or one with a subtle ordering mistake in the recurrence, can agree perfectly with the published values from `B0` to `B12` and still be wrong. By `B30` (whose value is roughly 601,580,873.9) the error is no longer invisible. If you only ever check small cases, you only ever catch large mistakes.

It states its convention out loud. The oracle prints `first Bernoulli numbers (B_n^-), B1 = -1/2` at the top of every report. A reference that does not say which convention it uses cannot settle a disagreement about conventions.

A second implementation using a different algorithm agrees with it. The file `../lab/examples/correct_exa` computes the same values by the Akiyama–Tanigawa transform, which is a different algorithm from the oracle's binomial recurrence. Agreement between separately implemented algorithms is stronger evidence than one routine agreeing with a copy of itself, although both implementations still belong to this module and require human review.

17.4 The property check, and what it catches that a table cannot

Every odd Bernoulli number above `B1` is exactly zero. That is a mathematical fact about the whole sequence, and the lab checks it as a *property*, not as a list of entries.

The difference is not cosmetic.

- A table check asks: *does index 4 return `-1/30`?* It is a list of facts, and it covers exactly the rows you wrote down.
- A property check asks: *is it true, for every odd index above 1 in this range, that the result is exactly zero?* It covers indices you never thought about.

An implementation can match every value in your table and be wrong everywhere else — for instance, one that returns correct values for the indices it was shown and something arbitrary elsewhere.

There is a sharper case. Here is real output from the supplied float-based example:

```
[odd-terms] Odd-index Bernoulli numbers above B1 must be exactly zero.
    Non-zero odd terms found: B3=-1.1102230246251565e-16,
    B5=-6.938893903907228e-17, B7=-6.938893903907228e-17.
```

Those numbers are, to any human eye, zero. Any value comparison with a tolerance would accept them. The property check demands *exact* zero and fails. Tolerances are precisely where inexact arithmetic hides, and a property stated exactly gives it nowhere to hide.

17.5 Running the checker

The tool is `../lab/check.py`. It imports the file you give it, calls its `bernoulli` function for every index from 0 to 30, compares each result against the oracle, reports the first divergence, and names the failure mode.

```
cd course/lab
python3 check.py examples/wrong_b1_sign.py
```

It exits with status `0` on a pass and `1` on a failure, so it can be used in a script. Useful options:

```
python3 check.py --help
python3 check.py examples/float_output.py --max-n 12 --verbose
```

Do not invent other flags; those are the ones that exist.

17.6 Reading the diagnoses

The checker does not stop at "wrong". It names the failure mode, using this fixed set of codes:

code	meaning
`b1-sign`	Correct under the *second* convention, `B1 = +1/2`.
`odd-terms`	Odd terms above B1 are not exactly zero.
`ada-indexing`	Only the non-zero terms are numbered, as in Note G.
`off-by-one`	Every value shifted one index.
`not-fraction`	Returns `float` or another inexact type.
`raises`	Undefined somewhere in the required range.
`import-error`	The file fails on import.
`no-function`	No `bernoulli` defined.

A report also carries a `[divergence]` line, which states the first index where your routine and the oracle disagree. That line locates the fault; the named code identifies what kind of fault it is.

Four wrong implementations are supplied so you can see each diagnosis before you meet it in your own work.

`examples/wrong_b1_sign.py` → `b1-sign`. First divergence at `n=1`: expected `-1/2`, got `1/2`. Nothing in this file is mathematically wrong; it is correct under the second Bernoulli convention. It is wrong *for this specification*. As the checker puts it, this is a specification failure rather than an arithmetic failure — which is the sentence Unit 6 is built on.

`examples/wrong_odd_terms.py` → `odd-terms` and `ada-indexing`. First divergence at `n=1`: expected `-1/2`, got `1/6`. This routine renumbers the non-zero terms consecutively, so its `bernoulli(k)` is the modern `B(2k)`: its `B1, B2, B3` are the modern `B2, B4, B6`. It skips the zeros as Note G does, though these are not Lovelace's labels either (hers run B1, B3, B5, B7 = modern B2, B4, B6, B8). Every value it returns is a genuine Bernoulli number; every one of them is filed under the wrong label. Two diagnoses fire because the same underlying mistake violates two separate requirements.

`examples/float_output.py` → `not-fraction`, plus `odd-terms`. The type fault is reported at `n=0`, and the first value divergence appears at `n=2`, where `1/6` arrives as `0.1666666666666674`.

`examples/off_by_one_range.py` → `off-by-one` and `odd-terms`. First divergence at `n=0`: expected `1`, got `-1/2`. The code is clean, the arithmetic is exact, and every value is a real Bernoulli number shifted one place. This is the failure mode that most resists being read: staring at the source will not find it, because there is nothing ugly to see. Only comparison against a reference finds it.

`examples/correct_example.py` → PASS.

17.7 Safety: the checker executes what you give it

Stop before you run anything.

`check.py` imports and executes the file you hand it. That is the only way it can test the code — and it means the file gets to do whatever it says, with your permissions, on your machine, at the moment of import.

AI-generated code is code from an untrusted source. "Untrusted" is a statement about what you know, not an accusation about anyone's intent. You did not write it, you have not read it, and polished formatting does not establish what the code does.

So, before you run generated code — here or anywhere:

- **Read it.** All of it, including anything above and below the function you asked for.
- **Look for** file writes and deletions, network access, `subprocess`, `os.system`, `eval`, `exec`, decoded or obfuscated strings, and anything that installs packages.
- **Check the imports.** A Bernoulli routine needs `fractions` and possibly `math`. It does not need `os`, `socket`, `requests` or `shutil`.
- **Run it in a disposable environment** — a container, a virtual machine, a throwaway account, or a directory containing nothing you care about.
- **Do not run it as an administrator.**

If you would not paste an unknown script from an internet forum straight into your terminal, apply the same rule here. The fact that you asked for it does not change where it came from.

Treat this as part of the lesson rather than a warning notice. A verification tool that has to execute untrusted input is a real engineering trade-off, and you have just met one.

17.8 Table-based verification

There is a second complete route through this lab that runs no code at all. It is the same method, performed by hand, and it is not a lesser route.

1. Get the reference values. Either run `python3 reference_bernoulli.py`, or use the table below, which is the same data.
2. Ask the AI system for the **values** rather than the program: "list `B0` to `B12` as exact fractions, and give `B30`".
3. Write the two columns side by side, in index order.
4. Go down the rows in order and **stop at the first row that differs**. You have just found the first divergence by hand, using the tool's own method.
5. Check `B30` even if the small values matched.

n	B_n	n	B_n
0	1	10	5/66
1	-1/2	12	-691/2730
2	1/6	14	7/6
3	0	16	-3617/510
4	-1/30	18	43867/798
6	1/42	20	-174611/330
8	-1/30	30	8615841276005/14322

All odd `B_n` for `n > 1` are exactly zero.

What the table route catches: wrong signs (`B1 = +1/2` instead of `−1/2`); missing or non-zero odd terms; values shifted by an index; decimals such as `−0.0333...` offered where an exact fraction was required; and any large-index error, provided you include `B30`.

What it misses: whether the program actually produces the table it printed — a table in a chat window may be generated text rather than program output, and those are different claims; what happens at indices you did not compare; and it does not re-run itself when you change something.

What the automated route misses: it checks agreement with the oracle's stated convention, not whether that convention is the one your task requires; it says nothing about comments, historical claims, licensing or readability; and it cannot tell you whether you understand the code.

Neither route is complete. Both are real verification. If you have Python available, doing the table comparison for `B0` to `B12` *and* running the checker takes about five minutes together and covers more than either alone.

17.9 Where this leaves you

When the checker passes, you have established one thing precisely: your routine agrees with a verified oracle for every index from 0 to 30, returns exact fractions, and satisfies the zero-odd-terms property.

That is evidence. It is not an explanation. Unit 8 will ask you to write two sentences saying why your answer is correct, citing what you checked rather than what produced it.

17.10 The external-reference principle

Behind everything this lab does is a single rule worth naming.

> **The external-reference principle.** A check has to come from a reference the thing being checked does not control. Here that is the oracle: exact, sitting outside the submitted routine, and confirmed against values it did not compute. Asking the same source again is not a check — it is the same source again. (Whether *no* system can ever fully certify itself from the inside is the larger question Unit 9 takes up; the working rule you need here is the concrete one above.)

This is not a new lesson; it is the name for what you have just done. The oracle earns its place precisely because it sits outside the submitted routine, states its convention out loud, and is checked against values it did not compute. Asking the same source again can help you revise an answer, but it does not give you the independence that a separate method or reference does.

This is how real work stays honest, not a classroom simplification. In research and engineering the same rule is enforced by machinery. In BabaYaga — a reasoning-systems project we are building, not yet released, which means you cannot check this claim; notice that, and treat it accordingly — the numbers in its paper are never typed into the prose: a build step reads them out of the files the code produced, and refuses to build, printing *undefined* rather than a stale figure, if a quoted value and its source no longer agree. To trust a computation at all, its results are recomputed into a *separate* tree and compared field by field against a reference kept read-only, so a reproduction can never quietly become its own authority. And the strongest form of the differential testing you just did — which other work of ours uses in earnest — is to implement the same method twice, in two different languages, and compare the two answers row by row; the first row that disagrees is the first divergence, exactly as here. None of this is special to Bernoulli numbers. It is the ordinary discipline of never letting the thing being checked certify itself.

That discipline is also the honest reason to trust *this module*. The routine you are about to check was generated with AI assistance and was not trusted until an external checker — the one you are running — agreed with it. The figures quoted in these pages are compared against the lab's own output by the build that assembles them. If you would rather see the habit in a full, public codebase than take our word for it, the Lucifuge scheduler (published at `https://dev.astarte.org`) checks the example configurations printed in its own documentation as part of its test suite, so a documented example cannot silently stop working — and unlike the BabaYaga claim above, this one you can go and check. You are not rehearsing a toy version of a professional habit; you are using the habit the material itself was made with.

The general version of this idea — that where the observer stands is a problem long before language models, and one this lab does not settle — is an interpretive lens rather than anything the lab proves. The closing unit develops it through the published work on second-order reasoning set out in `course/second-order.md`, which the closing Unit 9 teaches from; nothing in this unit, its activity, or its checkpoints depends on either.

17.11 Checkpoints

Complete this unit at `cp05-diagnose` plus the checkpoint for your route. `cp05-diagnose` runs by either route. The browser/table route uses `cp05-table-evidence` to assess conclusions from manual comparison. `cp05-bernoulli` needs Python and `check.py` and establishes executable behaviour over the checker's stated range. Neither route claims evidence it did not collect.

```
python3 selfcheck.py run cp05-diagnose           # or use the browser se
python3 selfcheck.py run cp05-bernoulli          # Python route only
python3 selfcheck.py run cp05-table-evidence     # table route; browser
```

Full instructions are in `activity.md`.

18 Unit 5 activity — Check what you generated

Time: about 24 minutes. Work through **Route A** or **Route B**. They are equal routes and either one completes the unit; do both if you have time, because they catch different things.

Keep everything you write down. It goes into your evidence pack in Unit 8.

18.1 Before you start: the safety step

This step belongs to both routes and comes first.

`check.py` imports and executes the file you give it. Your Unit 4 code came from a source you did not write and have not read. Read it now, before you do anything else, and answer these three questions in writing:

1. Which modules does it import? A Bernoulli routine needs `fractions` and possibly `math`. Anything else needs a reason.
2. Does anything happen at import time — outside the function body — such as printing, writing a file, opening a network connection, or running a subprocess?
3. Is there anything you cannot explain? Encoded strings, `eval`, `exec`, `os.system`, or an install command.

If any answer worries you, do not run it. Delete the suspicious part, or switch to Route B, which never executes anything.

Then run it somewhere disposable: a container, a virtual machine, a throwaway account, or at least a directory holding nothing you would mind losing. Not as an administrator.

18.2 Route A — Automated verification

18.2.1 A1. Set up (2 minutes)

```
cd course/lab
python3 reference_bernoulli.py
```

Confirm the final line reads:

```
OK: oracle agrees with all 22 published values, odd terms are zero.
```

If it does not, stop. The oracle is not verified, so nothing you check against it means anything.

18.2.2 A2. Meet the diagnoses (4 minutes)

Run the checker against each supplied wrong example, and against the correct one:

```
python3 check.py examples/wrong_b1_sign.py
python3 check.py examples/wrong_odd_terms.py
python3 check.py examples/float_output.py
python3 check.py examples/off_by_one_range.py
python3 check.py examples/correct_example.py
```

Fill in this table from the output before you check the marking guidance below:

file	first divergence at n	expected	got	diagnosis code(s)
`wrong_b1_sign.py`				
`wrong_odd_terms.py`				
`float_output.py`				
`off_by_one_range.py`				
`correct_example.py`	—	—	—	

Then answer in one sentence each:

- Why does `wrong_odd_terms.py` produce **two** diagnosis codes rather than one?
- `off_by_one_range.py` returns only genuine Bernoulli numbers and contains no arithmetic error. What, exactly, is wrong with it, and why would reading the source not reveal it?

18.2.3 A3. Check your own code (5 minutes)

Put the routine you saved in Unit 4 into the exercise stub, or check it where it is. Either works:

```
python3 check.py exercises/ex04_bernoulli.py
```

or, for a file anywhere on your machine:

```
python3 check.py /full/path/to/your_file.py
```

Record:

- **First divergence:** at `n =` _____, expected _____, got _____
- **Diagnosis code(s):** _____
- **In your own words, what the fault is:** _____

If your file passed on the first run, do not skip ahead. Run it again with the comparison extended and confirm it still passes:

```
python3 check.py exercises/ex04_bernoulli.py --max-n 40
```

Then write one sentence saying which clause of the specification your Unit 4 prompt happened to get right by luck rather than by stating it.

18.2.4 A4. Repair (7 minutes)

You have two ways to fix this, and the choice is the interesting part.

- **Repair the code.** Edit the routine directly until the checker passes.
- **Repair the prompt.** Go back to the AI system, add the requirement that was missing, regenerate, and check the new output.

Do at least one. Note which requirement was missing — that observation is exactly what Unit 6 builds on. Then re-run the checker with `--verbose` until you see:

```
PASS - agrees with the oracle for every n from 0 to 30,
       returns exact Fractions, and odd terms above B1 are zero.

Passing is not the end of the job. You still have to be able to say
*why* it is correct. That is the Unit 8 defence.

(Indices compared: 0 to 30)
```

18.2.5 A5. One last question

The checker passed. Write down, in one sentence, something that is still not established about your routine by that pass.

18.3 Route B — Table-based verification

This route uses no Python. It performs the same comparison by hand, and it completes the unit on its own terms.

18.3.1 B1. Get the reference values (2 minutes)

Use the reference table in `page.md`. If you have Python available and want the machine-printed version, `python3 reference_bernoulli.py` prints the same data, but this is optional here.

18.3.2 B2. Get the generated values (4 minutes)

Go back to the AI system you used in Unit 4 and ask for the **values** rather than a program:

> List the Bernoulli numbers B0 to B12 as exact fractions, and also give B30.

Copy what comes back exactly as it is given, including any decimals. Do not tidy it up. Tidying is where evidence gets lost.

If you have no AI access, use a supplied specimen instead, just as Unit 4 does:

1. Open `course/lab/examples/wrong_b1_sign.py` in any text editor. You are reading it, not running it — this route stays code-free.
2. Its docstring states exactly what the routine returns: it is "correct under the *second* Bernoulli convention (`B1 = +1/2`)". So its output table is the reference table with precisely one entry changed: `1/2` at `n = 1`.
3. Write that out as your "generated value" column — every reference value unchanged except `1/2` at `n = 1` — and carry it into B3.

18.3.3 B3. Compare in index order (6 minutes)

Write the two columns side by side, in index order, and go down the rows.

n	reference value	generated value	same?
0	1		
1	-1/2		
2	1/6		
3	0		
4	-1/30		
5	0		
6	1/42		
7	0		
8	-1/30		
9	0		
10	5/66		
11	0		
12	-691/2730		
30	8615841276005/14322		

Stop at the first row that differs. That is the first divergence, found by hand, using the same method the tool uses. Record it:

- **First divergence:** at `n = _____`
- **Which named failure mode does it match?** Use the table in `page.md`.

Then check three things that are easy to miss when scanning:

1. Are the odd rows present and exactly `0`, or are they missing, or "about zero"?
2. Are the values exact fractions, or decimals such as `-0.0333`?
3. Did `B30` come back at all, and is it exactly `8615841276005/14322`?

18.3.4 B4. Repair (6 minutes)

Go back to the AI system, state the requirement that was missing, and ask again. Compare the new table against the reference the same way. Record which single sentence you added and what changed.

If you have no AI access, treat a second supplied specimen as the "regenerated" attempt:

1. Open `course/lab/examples/wrong_odd_terms.py` and read its docstring — again without running anything.
2. Build its output column from what the docstring states: the routine renumbers the non-zero terms consecutively, so its `bernoulli(k)` is the modern `B(2k)`. Its column therefore reads `1` at `n = 0`, then `1/6`, `-1/30`, `1/42`, `-1/30`, `5/66` at `n = 1` to `5`.
3. Compare that column against the reference exactly as in B3, in index order, and record the first divergence.
4. Instead of the sentence you added, record the single sentence a repair prompt would need for *this* specimen — the requirement its docstring says it violates.

18.3.5 B5. One last question

Your two tables now agree. Write down, in one sentence, why that does not establish that a program produced those values.

18.3.6 B6. Check what the table evidence supports (4 minutes)

Open `cp05-table-evidence` in the browser self-check. It gives four observations from manual candidate/reference table comparisons. Match each observation to the strongest conclusion it supports.

This is the browser-assessable counterpart to the executable checkpoint, not a simulation of running code. It checks modern indexing, zero odd terms, exact displayed values, first-convention handling, and the boundary of manual evidence. A matching table supports the cells compared; it does not establish the return type of unseen code, unlisted indices, or that a program produced the table.

18.4 Finish the unit at the checkpoints

The checkpoints are formative and unmarked. `cp05-diagnose` is shared. `cp05-table-evidence` records the manual evidence objective on Route B. `cp05-bernoulli` runs `check.py` against a file and belongs to Route A. The two route-specific checkpoints test the same contract with honestly different coverage.

Python route — from `course/lab`:

```
python3 selfcheck.py run cp05-bernoulli
python3 selfcheck.py run cp05-diagnose
```

To practise both routes, including the table-evidence checkpoint, run the whole unit:

```
python3 selfcheck.py run --unit 5
```

Browser route — open `build/site/selfcheck.html` in any browser, with no installation, and answer `cp05-table-evidence` and `cp05-diagnose`. The browser self-check cannot run code, so it cannot grade `cp05-bernoulli`; it lists that checkpoint and directs you to complete it in the Python lab.

`cp05-bernoulli` requires a completed `exercises/ex04_bernoulli.py`; until you edit the stub it will report that the checkpoint has not been attempted rather than failing you.

To see where you have got to:

```
python3 selfcheck.py list
python3 selfcheck.py progress
```

18.5 Explore, if you want to

The oracle is there to be poked at, not only to judge you:

```
python3 selfcheck.py explore
```

Inside, `b 30` prints `B30` exactly, `float 30` shows the exact value and the float side by side, `table 20` prints a range, `map` shows the Note G index mapping, `check <file>` runs the checker, and `quit` leaves.

18.6 Optional CS extensions

Optional. These are outside the core study time and are not required to complete this unit or the module. Non-programmers lose no core content by skipping them.

Write the test yourself — `exercises/ex05_write_a_test.py`

```
python3 exercises/ex05_write_a_test.py
```

So far the checker has done the judging. Here you write it: a single function that must separate four supplied faulty implementations from one correct one, without wrongly rejecting the correct one. A test that only checks `B4 == -1/30` lets two of the four straight through. The exercise grades itself against the real files in `examples/` and tells you which you caught and which you missed.

Translate between Ada's indexing and ours — `exercises/ex06_ada_translation.py`

```
python3 exercises/ex06_ada_translation.py
```

Implement the mapping from Lovelace's Note G labels to modern indices, returning exact values and rejecting indices that have no Note G label at all. This one looks ahead to Unit 7; you can do it now or after that unit.

18.6.1 Activity Answers

19 Unit 5 answers and guidance

Answer key for the Unit 5 activity. Self-check checkpoints are not answered here; they mark themselves and give their own diagnostic feedback, by either route.

Read this after attempting the activity. Where a question asks for your own routine, the key describes what a good answer looks like rather than giving one.

19.1 Before you start — the safety step

There is no single right answer, but there is a right shape of answer.

1. **Imports.** `fractions` is expected; `math` is reasonable, usually for `comb`. Anything else needs a reason you can state. `os`, `sys` with path manipulation, `socket`, `subprocess`, `urllib`, `requests`, `shutil` and `pathlib` writes have no place in a routine that computes a number from an integer.
2. **Import-time behaviour.** Anything outside a function body runs the moment the checker imports the file. A `print` is harmless and untidy; a file write, a network call or a `subprocess` is neither. Note that a `if __name__ == "__main__":` block does *not* run on import, so a demonstration loop there is fine.
3. **Unexplained code.** The honest answer is often "I could not tell what this line does". That is a finding, not a failure. Code you cannot read is code you cannot vouch for, and the correct response is to remove it or not run it.

A learner who reports "nothing suspicious, here is why" with specific reasons has done this correctly. A learner who reports "it looked fine" has not done it at all.

19.2 A2 — The supplied examples

Exact figures from the checker, at the default `--max-n 30`:

file	first divergence at n	expected	got	diagnosis code(s)
`wrong_b1_sign.py`	1	`-1/2`	`1/2`	`b1-sign`
`wrong_odd_terms.py`	1	`-1/2`	`1/6`	`odd-terms`, `ada-indexing`
`float_output.py`	2	`1/6`	`0.16666666666666674`	`not-fraction`, `odd-terms`
`off_by_one_range.py`	0	`1`	`-1/2`	`off-by-one`, `odd-terms`
`correct_example.py`	—	—	—	none; PASS

Notes on the details that catch people out:

- The `[divergence]` line is not itself a named failure mode. It locates the fault; the codes classify it. A report normally contains both.
- `float_output.py` is reported as `not-fraction` "first seen at n=0", because even `B0 = 1.0` is already the wrong type, while the first *value* divergence is at `n=2`. Type and value are separate failures and are found by separate checks.
- `off_by_one_range.py` picks up `odd-terms` as well, because shifting the sequence moves the non-zero even values onto odd indices. One underlying fault, two violated requirements.

Why does `wrong_odd_terms.py` produce two codes? Because one mistake breaks two separate requirements. Renumbering the non-zero terms consecutively means the odd indices no longer hold zero (`odd-terms`), and it also means every label names a different number from the modern one (`ada-indexing` — though these are not Lovelace's labels either; hers run B1, B3, B5, B7 = modern B2, B4, B6, B8). A diagnostic system that reported only the first symptom would leave you fixing half the fault.

What is wrong with `off_by_one_range.py`? Every value it returns is a genuine Bernoulli number, and every one is filed under the wrong index: its `bernoulli(0)` hands back `B1`, its `bernoulli(4)` hands back `B5`. Reading the source does not reveal it because there is nothing badly written to see — the

arithmetic is exact, the style is clean, and the values are all real. The fault is a relationship between the code and a reference outside the code, and only a comparison against that reference exposes it. This is the strongest argument in the unit for differential testing over code review.

19.3 A3 — Your own code

A good record is specific:

> First divergence at n = 1, expected -1/2, got 1/2. Diagnosis: `b1-sign`. The routine implements the second Bernoulli convention; my prompt never said which convention I wanted, so the specification was incomplete rather than the code being wrong.

Common outcomes, and what each tells you about the prompt that produced them:

diagnosis	what the prompt failed to state
`b1-sign`	which Bernoulli convention to use
`odd-terms`	that odd indices above 1 must return exactly zero
`ada-indexing`	that modern indexing is required, not Note G's
`off-by-one`	the value at named indices, anchoring where the sequence starts
`not-fraction`	that returns must be exact `fractions.Fraction`, never `float`
`raises`	that the function is defined for every `n >= 0`
`no-function`	the required function name and signature
`import-error`	that the module must import cleanly using only the standard library

If it passed first time, the honest answer names the clause you got for free. Typically it is exact arithmetic — a request phrased around Ada Lovelace or exact values sometimes yields `Fraction` without being asked, and sometimes does not. Getting it without asking is luck, and luck is not a property you can rely on next time. That observation is the whole of Unit 6 in one line.

A5 — what the pass does not establish. Any of these is a good answer:

- That the routine is correct for `n > 30`, or for `n > 40` if you extended the comparison. The check is finite.
- That the convention it agrees with is the one your actual task requires. It agrees with *this module's* oracle.
- That you understand it, or could reproduce or repair it.
- That any comment in the file is true. Comments are not executed and are not checked.
- That the code is safe to run on data other than integers, or free of performance problems at large `n`.
- That the algorithm is the one you claimed to use.

19.4 B3 — Table route

The first divergence found by hand is the same index the tool would report, provided you compared in index order and stopped at the first difference. If you scanned for "the obviously wrong one" instead, you may have found a later divergence that is only a consequence of an earlier one.

The three things that are easy to miss when scanning:

1. **Odd rows.** Many generated tables omit the odd indices entirely, listing only `B0, B1, B2, B4, B6, ...`. An omitted row is not the same claim as a row containing zero, and the difference matters: the specification requires `bernoulli(3)` to *return* zero, not to be undefined. If the odd rows are present but written as `0.0`, `~0` or `1e-17`, that is the float failure mode.

2. **Exact fractions.** `-0.0333...` is not `-1/30`. It is close, and closeness is the property the specification rules out. Any decimal at all is a `not-fraction` finding.
3. **B30.** A table that agrees perfectly from `B0` to `B12` and then gives `601580873.9` or `6.0158e8` for `B30` has told you exactly what you needed to know. If `B30` was refused or omitted, treat that as unverified rather than as correct.

B5 — why agreement does not establish that a program produced the values. A table in a chat window is generated text. It may have been produced by a routine, copied from a published table, or generated the same way as any other text. Those are different claims with different reliability, and the table looks identical in all three cases. The values being right tells you the values are right; it tells you nothing about the existence or behaviour of a program. If your task requires a working routine, that is a separate claim needing separate evidence — which is what Route A supplies.

19.5 Marking the two routes

Both routes evidence the same skill: ordered comparison against a trusted reference, stopping at the first divergence, and naming the fault. A learner who completed Route B has verified. Do not treat Route A as the real version.

The distinguishing feature of a good submission by either route is **specificity**: an index, a pair of values, a named failure mode, and a sentence about which requirement was missing. "It was wrong and then I fixed it" is not a verification record.

19.6 Optional CS extensions

These are optional and outside the core study time. Do not treat them as required.

`ex05_write_a_test.py`. The exercise grades itself, so no key is given here. The instructive failure is a test that catches all four faulty implementations by being aggressive and then wrongly rejects `correct_example.py` — the file reports that separately, and says why it is worse than missing a fault: a test that fails good code is a test people learn to ignore. The second instructive failure is a test built from == against a handful of known values, which lets the type-related and range-related faults through. A test that works generally checks a *property* and a *type*, not only a list of values.

The point to draw out afterwards: every clause the learner needed in their test corresponds to a clause the specification should have contained. That list is the one they build in Unit 6.

`ex06_ada_translation.py`. Also self-grading. Note that the exercise requires *rejecting* Ada `B2` and Ada `B0` with a `ValueError`, not only computing Ada `B7` correctly. Learners often implement the mapping and forget the rejection, and the exercise says why that matters: a translation layer that silently accepts nonsense from the other side is how convention bugs travel between systems.

19.7 Checkpoints

`cp05-bernoulli`, `cp05-table-evidence`, and `cp05-diagnose` are self-marking, with their own diagnostic feedback, and their answers are deliberately not reproduced here. Learners reach `cp05-diagnose` by either route; `cp05-bernoulli` belongs to the Python route and `cp05-table-evidence` to the browser/table route:

```
python3 selfcheck.py run cp05-diagnose     # or use the browser self-che
python3 selfcheck.py run cp05-bernoulli    # Python route only
python3 selfcheck.py run cp05-table-evidence  # table route; browser or
```

19.8 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.


```
{
  "id": "cp05-bernoulli",
  "unit": 5,
  "title": "Make the checker pass",
  "kind": "code",
  "question": "Complete course/lab/exercises/ex04_bernoulli.py so that b
  "answer": null,
  "explanation": "You have now done the full loop: a specification precis
  "exercise": "exercises/ex04_bernoulli.py"
}
```

```
{
  "id": "cp05-table-evidence",
  "unit": 5,
  "title": "State what a table comparison establishes",
  "kind": "match",
  "question": "For each observation from a manual candidate/reference ta
  "answer": [
    0,
    1,
    2,
    3
  ],
  "options": [
    "The candidate uses the other B1 convention.",
    "The evidence does not establish the required odd zero rows and mode
    "Only the displayed cells are supported; no program, return type, or
    "Approximate decimals do not establish the required exact fraction v
  ],
  "prompts": [
    "B1 is +1/2; every other displayed value agrees with the reference."
    "B3 and B5 are absent; the remaining displayed exact fractions agree
    "Every displayed cell from B0 to B12 and B30 agrees exactly.",
    "A displayed cell is 0.1666667 where the reference gives 1/6."
  ],
  "diagnostics": {
    "0,1,3,2": "The final two conclusions are swapped. A decimal-versus-f
    "1,0,2,3": "The first two conclusions are swapped. A lone B1 sign diff
  },
  "hint": "Choose the narrowest conclusion licensed by what was actually
  "explanation": "Manual table comparison can establish exact agreement
}
```

```
{
  "id": "cp05-diagnose",
  "unit": 5,
  "title": "Name the failure mode",
  "kind": "choice",
  "question": "For arguments 1, 2, and 3, a submitted routine returns th
  "answer": 0,
```

```
  "options": [
    "It numbers only the non-zero terms, as Note G does.",
    "It uses floating point and the values have drifted.",
    "It has the B1 sign convention backwards.",
    "Nothing -- those are the Bernoulli numbers."
  ],
  "diagnostics": {
    "1": "Float drift produces near-misses -- values like 1e-16 where an
    "2": "The B1 convention only ever changes B1 itself, and here every
    "3": "Each individual value is genuine -- that is what makes this on
  },
  "explanation": "Its B1, B2, B3 are the modern B2, B4, B6: the routine
}
```

20 Unit 6 — Re-prompting, specification, and overreliance

Media for this unit:

- **Interactive Animation**: This unit features an interactive animation (`spec-filter.html`). Please view the web version of the course to interact with it.

Reading time: about 10 minutes. Activity: about 20 minutes.

Course items: 2 (how to evaluate AI outputs) and 4 (AI and critical thinking). See `../course-map.md`.

20.1 The claim

There is a widespread idea that better results come from better wording, and that somewhere there exists a phrase that unlocks a correct answer. This unit argues something less exciting and considerably more useful.

Better prompting is precise requirements work. It is the same skill as writing a specification. You have already been doing it: every failure mode the checker named in Unit 5 was telling you which requirement you had failed to write down.

20.2 Where you started

The prompt Unit 4 began with was of this kind:

> Write a Python function that computes Bernoulli numbers, like the ones in Ada Lovelace's program.

Every word of that is true, and none of it is decidable. It leaves at least five decisions open:

- **Which convention?** `B1 = -1/2` or `B1 = +1/2`. Both are standard and published.
- **Which indexing?** Modern indexing, where every index is numbered including the zero ones, or Lovelace's, where only the non-zero terms carry labels.
- **Which return type?** An exact `Fraction`, a `float`, or a formatted string.
- **Which domain?** Is `bernoulli(0)` defined? `bernoulli(1)`?
- **The zero terms:** returned as `Fraction(0)`, or skipped, or an error.

Something has to settle those five decisions. If the prompt does not, the output does, according to whatever is most frequent in the text the system was trained on. The result is fluent and plausible; whether it is correct is a question the prompt never asked.

20.3 The failure modes were the missing requirements

Put the Unit 5 diagnoses beside that prompt and they stop looking like coding mistakes:

the checker said	you had not stated
`b1-sign`	which Bernoulli convention to use
`ada-indexing`	that modern indexing is required, not Note G's
`not-fraction`	that returns must be exact `fractions.Fraction`
`raises`	that the function is defined for every `n >= 0`
`odd-terms`	that odd indices above 1 return exactly zero
`off-by-one`	the value at named indices, anchoring the sequence
`no-function`	the required function name and signature
`import-error`	that it must import cleanly, standard library only

Not one of those is a wording problem. Every one is a missing requirement. That table is the substance of this unit, and matching its two columns is checkpoint `cp06-spec-clauses`.

20.4 The precise prompt

Here it is in full. It is not a cleverer sentence; it is the same request with the decisions written down. The clause labels are for the activity and are not part of what you would send.

> **(C1) Interface.** Write a single Python module that defines exactly this function at the top level of the module:
>
> `bernoulli(n: int) -> fractions.Fraction`
>
> **(C2) Convention.** Use the *first* Bernoulli numbers, in which `B1 = -1/2`. Do not use the second convention, in which `B1 = +1/2`.
>
> **(C3) Indexing.** Use modern indexing, in which every index is numbered, including the odd indices whose value is zero. Do not use the nineteenth-century numbering of Ada Lovelace's Note G, in which only the non-zero terms are labelled.
>
> **(C4) Domain.** `bernoulli(n)` must be defined for every integer `n >= 0`. For `n < 0`, raise `ValueError`.
>
> **(C5) Zero terms.** For odd `n > 1`, return `Fraction(0)`. Do not skip those indices, omit them, or raise.
>
> **(C6) Arithmetic.** Return exact `fractions.Fraction` values only. Never return `float`, `Decimal`, or a string, and do not use floating-point arithmetic anywhere in the computation.
>
> **(C7) Anchor values.** These values must be exactly reproduced: `B0 = 1`, `B1 = -1/2`, `B2 = 1/6`, `B3 = 0`, `B4 = -1/30`, `B12 = -691/2730`, `B30 = 8615841276005/14322`.
>
> **(C8) Importability.** The module must import with no side effects: no printing, no file or network access, and no third-party packages — Python standard library only.
>
> **(C9) Comments.** Do not include historical claims about Ada Lovelace or Note G in comments unless I ask for them. I cannot test a comment.

The interface line in C1 is quoted exactly as the checker's contract states it. That is the point of the whole unit in one line: **a specification precise enough to test against is a specification precise enough to prompt with.** If you find you cannot write the prompt, it is usually because you did not yet know what you wanted.

Two clauses deserve a note.

C7 is a trap as well as a requirement. Naming the acceptance values makes the requirement concrete, and it also makes it possible for generated code to satisfy them with a lookup table that fails at every other index. Examples are not a specification. That is why the checker compares thirty-one indices rather than the seven you listed, and why C2 to C6 state properties rather than only values.

C9 is not enforced by any check. No code in the lab reads comments. It is in the prompt because a historical claim in a code comment is generated the same way as the code, fails independently of it, and passes every test in the module. Unit 7 is about exactly that.

20.5 Every clause is a bug that already happened

Read the precise prompt once more and notice what it actually is: a list of things that went wrong.

- C2 exists because a routine came back correct under the *second* convention.
- C3 exists because a routine came back numbering only the non-zero terms, faithfully reproducing Note G.
- C5 exists because a routine skipped the odd indices instead of returning zero.
- C6 exists because a routine came back in floating point, where `B3` is `-1.11e-16` rather than `0`.
- C4 exists because a routine raised an exception at `n = 0`.

Every clause in a good specification is a bug that already happened. That is what separates specification from prompt engineering as it is usually described: you are not searching for magic words, you are writing down a requirement each time reality teaches you one. A specification accumulates in exactly the way a test suite does — one clause per fault that reached you — and it can be reviewed, argued with, and handed to someone else.

20.6 Overreliance

Overreliance is not using AI. It is **relying on generated output past the point where you can check it**. The distance between the two is your own understanding, which makes overreliance situational: the same use is sound for a person who can evaluate the result and unsound for a person who cannot.

Think back to working out `B4` by hand in Unit 3. Five binomial coefficients, a few fractions, no calculator, and out comes `-1/30`. It was slow, and it looked like a detour on the way to the interesting part.

It was the load-bearing step. It is the reason that when a routine returned `1/30`, you knew something was wrong rather than assuming you had misremembered. Without that half-hour, `-1/30` and `1/30` are two equally plausible strings, and you have no basis to prefer either. The oracle, the checker and the property test only convert into a judgement if there is someone who can read the result and act on it.

This is why the module put the mathematics before the generation, and why `cp03-b4-by-hand` never reveals its answer on a wrong attempt. A value you derived independently is strong evidence because the generated answer did not determine it. Authoritative sources, measurements, and separately implemented methods can play the same role in other tasks.

20.6.1 When to stop

There is a practical rule in this. **Pause and re-ground when you can no longer understand and defend what is being produced.** The signal is concrete, not a mood. Pause if you cannot:

1. predict relevant behaviour;
2. explain the transformation from input to output at the level your claim needs;
3. identify a suitable independent check; or
4. interpret a failed check well enough to decide what to revise or escalate.

Subjective confidence is not evidence that any of these capacities remains. Narrow the request until you can follow it, rebuild the missing foundation (as you rebuilt `B4` by hand), or seek qualified review. Do not keep expanding a result that you can no longer evaluate.

20.7 The honest position

This is not an argument against using these systems.

What you gain is real. The precise prompt plus the oracle is genuinely faster than writing the routine from scratch, and the specification you wrote is useful work in its own right — you would need it to brief a colleague, and you would need it to test anything at all.

Two things do not transfer. Responsibility for the result stays with you, whatever produced it. So does judgement: someone has to know what correct looks like, and in your work that someone is you.

And note the luxury of this example. Bernoulli numbers have an oracle. Most of the questions you will put to an AI system do not — there is no reference implementation for the summary of a document, the argument in an essay, or the interpretation of a policy. This case was chosen because it is clean, not because it is typical.

When there is no oracle, you still have the same moves in weaker form:

- Write the specification anyway. It is the part that makes anything checkable.

- Check claims against independent sources rather than against the fluency of the output.
- Produce a second, independent version and compare — by another method, another tool, or another person.
- State plainly what you were unable to verify, rather than letting confident prose stand in for a check.

The activity applies those moves to a citation, a supplied policy passage, and a recommendation containing an unstated value judgement. In each case you must name the claim, an independent check, the judgement that remains, and the point at which you would stop or seek qualified human review.

20.8 A note on the general case

There is a general version of what C1 to C9 are doing. "Correct" is not a property a value carries on its own: `B1 = -1/2` and `B1 = +1/2` are both correct, and they appear to disagree only because the convention each assumes has been left out. A specification clause supplies exactly that missing index — the convention, the range, the type — and it is what makes two answers comparable at all, before any question of which one wins. That argument in general form is in `course/second-order.md`, which the closing Unit 9 teaches from; neither this page, the activity, nor `cp06-spec-clauses` depends on it.

20.9 Checkpoint

Finish this unit at `cp06-spec-clauses`, by either route:

```
python3 selfcheck.py run cp06-spec-clauses    # or use the browser self-
```

Full instructions are in `activity.md`.

21 Unit 6 activity — Annotate the specification, then re-run it

Time: about 20 minutes. Parts 1 to 4 take about 10 minutes; the no-oracle transfer in Part 5 takes about another 10. Only Part 3 differs between the Python and table routes, and either version completes the unit.

Keep what you write. The two prompts and the two results are the core of your Unit 8 evidence pack.

21.1 Part 1 — Annotate the clauses (4 minutes)

The precise prompt in `page.md` has nine labelled clauses. Eight of them remove a failure mode that the checker can name. Fill in the middle column with a clause label, using each of C1 to C8 exactly once.

failure mode	clause that removes it	why that clause makes it impossible
`b1-sign`		
`odd-terms`		
`ada-indexing`		
`off-by-one`		
`not-fraction`		
`raises`		
`import-error`		
`no-function`		

The third column is the part that matters. A clause label on its own is a guess; one sentence saying *what the clause makes testable* is an argument. For example, a clause that names the return type turns "does this look about right?" into a question with a yes-or-no answer.

Then answer two short questions:

1. **C9** — the clause forbidding unrequested historical claims in comments — maps to none of the eight codes. Why not, and what would you have to do instead to check it?
2. Which single clause, if you deleted it from the prompt, would be least likely to be noticed by the checker? Say why.

21.2 Part 2 — Look at your own prompt (2 minutes)

Put your Unit 4 prompt beside the precise prompt and answer in writing:

- Which of C1 to C8 did your prompt already state, in any form?
- Which did it leave open?
- Which of the ones it left open did the generated code happen to get right anyway?

That last group is the interesting one. Getting a requirement satisfied without stating it is luck, and luck is not a property you can rely on the next time.

21.3 Part 3 — Re-run the generation

Take the precise prompt from `page.md` — clauses C1 to C9, without the labels — and send it to the same AI system you used in Unit 4. Then check the result. **A better prompt is not evidence of a better answer**; the new output gets verified exactly like the old one.

21.3.1 Route A — Python (4 minutes)

Read the new code first, as in Unit 5: check the imports, check for anything that runs at import time, and run it somewhere disposable.

```
cd course/lab
python3 check.py /full/path/to/your_new_file.py
```

or paste it into the exercise stub and use:

```
python3 check.py exercises/ex04_bernoulli.py
```

Record for both versions:

	vague prompt (Unit 4)	precise prompt (Unit 6)
first divergence at n		
diagnosis code(s)		
result		

21.3.2 Route B — Table (4 minutes)

Ask the same system for the values produced under the precise prompt:

> Using the specification above, list B0 to B12 as exact fractions, and give B30.

Compare against the reference table in the Unit 5 student page, in index order, stopping at the first row that differs. Record the same three rows as the table above: first divergence, which named failure mode it matches, and the result.

Check the same three easily-missed things: are the odd rows present and exactly `0`; are the values exact fractions rather than decimals; and is `B30` exactly `8615841276005/14322`.

21.3.3 Both routes

Write two sentences comparing the two runs:

1. What changed between the vague and precise results?
2. What did **not** change — that is, which faults survived a fully specified prompt, or which new ones appeared?

If the precise prompt still produced a fault, that is a useful result and not a failed exercise. Add the clause that would have prevented it, and note that you have just done for yourself what clauses C2 to C6 record: a bug happened, so a requirement got written down.

21.4 Part 4 — Overreliance, in one paragraph

Write three or four sentences answering this:

> Which specific piece of understanding, built before you used AI, made you able to judge the generated output? What would you have concluded without it?

Be concrete. Name the value you derived, the convention you knew existed, or the property you knew must hold. "I understood the topic better" is not an answer; "I had derived `B4 = -1/30` myself, so `1/30` was a red flag rather than a plausible alternative" is.

Keep this paragraph. It is most of your Unit 8 correctness defence already written.

21.5 Part 5 — Transfer when there is no oracle (10 minutes)

The Bernoulli task has one exact reference. These three cases do not. For each case, fill one row of the table below.

Case	Claim being relied on	Feasible independent check	Judgement that remains	Stop or escalation condition
Citation				
Policy summary				
Recommendation				

21.5.1 Case A — a citation claim

An AI assistant writes:

> Nguyen (2021) found that first-year students who used weekly retrieval practice improved their examination scores by 18 per cent.

Do not assume that the paper, author, date, statistic, or causal wording is real. State the smallest useful claim you would check first and name where you would check it. If the paper exists, checking the catalogue record establishes only that existence and metadata; supporting the sentence requires reading the paper.

21.5.2 Case B — a supplied policy passage

This is fictional practice text, not your institution's policy:

> Students may use generative AI to brainstorm questions and organise notes. They must not submit generated prose or code as their own work. Any permitted use must be disclosed using the course declaration. Where an assessment brief sets stricter conditions, the assessment brief takes precedence.

The assistant summarises it as:

AI may be used to draft assessed work provided that its use is disclosed.

Compare the summary directly with the supplied passage. Identify the changed permission and the precedence rule that disappeared. State when you would stop interpreting and ask the named course contact for a ruling.

21.5.3 Case C — a recommendation with an unstated value judgement

The assistant recommends an unpaid placement at a famous company over a paid placement at a smaller local employer because it says the famous name will be "better for your career".

Separate checkable claims from the hidden judgement. Salary, travel, duties, training, working conditions, and published destination evidence can be checked from records and, where appropriate, triangulated across independent sources. How much prestige, income, time, accessibility, or particular experience should matter is a value judgement that remains yours.

Use these four checking routes precisely:

- **Authoritative-source comparison:** compare a citation or rule with the publisher, official catalogue, policy owner, or assessment brief.
- **Direct comparison:** compare a summary line by line with the supplied source.
- **Triangulation:** compare genuinely independent records or observations when no single source settles a factual claim.
- **Qualified human review:** escalate interpretation or high-consequence judgement to someone with the relevant role or expertise.

Multiple generated answers agreeing is not triangulation. Consensus can reveal stability and can help locate a question, but agreement alone is not proof and does not make dependent sources independent.

21.6 Finish the unit at the checkpoint

`cp06-spec-clauses` is available by both routes. It is formative and unmarked.

Python route — from `course/lab`:

```
python3 selfcheck.py run cp06-spec-clauses
```

or, equivalently:

```
python3 selfcheck.py run --unit 6
```

Browser route — open `build/site/selfcheck.html` in any browser, with no installation, and answer the same checkpoint there.

To see where you have got to, and to export a record for your evidence pack:

```
python3 selfcheck.py progress
python3 selfcheck.py progress --evidence
```

21.7 Before you finish

You should be leaving this unit with:

- the annotated clause table from Part 1, with a reason in the third column for each of C1 to C8
- the three-group comparison of your own Unit 4 prompt: what it stated, what it left open, and what came right anyway
- the two-run comparison rows from Part 3 — vague prompt against precise prompt
- the overreliance paragraph from Part 4
- the three no-oracle transfer rows from Part 5, each with a stop or escalation condition
- `cp06-spec-clauses` passed

All of these feed directly into the evidence pack you assemble in Unit 8.

21.7.1 Activity Answers

22 Unit 6 answers and guidance

Answer key for the Unit 6 activity. The self-check checkpoint marks itself and gives its own diagnostic feedback by either route, so its answer is not reproduced here.

22.1 Part 1 — The clause mapping

Each clause is tied to the concrete check that becomes possible once the clause exists. That third column is the point: a requirement you cannot check is a preference, not a requirement.

failure mode	clause	why it makes the failure impossible
`b1-sign`	**C2** convention	Fixes the one value the two published conventions disagree about. Without it, both `-1/2` and `+1/2` satisfy the request, and no test can adjudicate. With it, `bernoulli(1)` has exactly one acceptable result.
`odd-terms`	**C5** zero terms	Requires odd indices above 1 to *return* `Fraction(0)` rather than being skipped, omitted or raised. This is what makes the property check — every odd term above `B1` is exactly zero — a test rather than an assumption.
`ada-indexing`	**C3** indexing	Rules out the nineteenth-century scheme in which only the non-zero terms are labelled. Without it, a routine whose `B1, B2, B3` are the modern `B2, B4, B6` is a defensible reading of "like the ones in Ada Lovelace's program".
`off-by-one`	**C7** anchor values	Pins the sequence to named indices — `B0 = 1, B4 = -1/30` — so a shift of one place contradicts a stated value at a stated index instead of merely looking unfamiliar.
`not-fraction`	**C6** arithmetic	Makes the return type part of the contract, so comparison can be exact. A tolerance would accept `-1.11e-16` as zero; exact comparison of `Fraction` values cannot.

failure mode	clause	why it makes the failure impossible
`raises`	**C4** domain	States that every integer `n >= 0` is in the domain, so a routine that works from `n = 2` upwards is a specification violation rather than a design choice, and testing at `n = 0` and `n = 1` becomes meaningful.
`import-error`	**C8** importability	Requires the module to import cleanly with no side effects and no third-party packages, which is precisely the precondition any automated check needs before it can call the function at all.
`no-function`	**C1** interface	Names the function, its parameter type and its return type, so "does this file meet the contract?" is a decidable question. It is also the line the checker itself states.

Question 1 — why C9 maps to none of the codes. Nothing in the lab reads comments. No code is executed to produce them, no test compares them, and a file full of false history passes every check in the module with a clean `PASS`. A historical claim in a comment is generated by the same process as the code beside it and fails independently of it. Checking it means leaving the toolchain entirely: find the primary or secondary source, read it, and cite it — or label the claim as unverified. That is the work of Unit 7, and the reason C9 asks for the claims not to be generated in the first place is that an unrequested claim is one nobody has budgeted time to check.

Question 2 — which clause would be least missed by the checker. The strongest answer is **C8**, importability. Its failure mode is the loudest one there is — the file will not import, and the checker reports `import-error` immediately — but the clause also covers requirements that fail *silently*: a module that prints on import, or writes a file, or reaches the network, still passes every value comparison. The checker never notices, and only reading the code does.

C7 is a defensible alternative answer, for a different reason: its work is mostly done by C1 to C6 already, so deleting it often changes nothing. The good version of that answer notes that C7 earns its place by catching the *shift* faults that the property clauses do not constrain.

Answers naming **C9** are outside the exercise, since C9 is not one of the eight and the previous question already covers it.

22.2 Part 2 — Your own prompt

There is no fixed key. A complete answer names specific clauses in all three groups, including the third:

> My prompt stated C1 in a rough form ("a function called bernoulli") and nothing else. It left C2 to C8 open. Of those, the code happened to satisfy C6 — it returned Fractions without my asking — and C4, because it handled n = 0.

The observation to draw out is that the third group exists at all. A requirement met without being stated is met by coincidence, and a coincidence is not reproducible: the next generation, the next model version, or the same prompt on another day may settle it differently. This is also why "it worked when I tried it" is weak evidence about a system whose output varies between runs.

22.3 Part 3 — Comparing the two runs

Expected patterns, and what each means.

Most faults disappear. The convention, indexing, type and domain faults are the ones a precise clause reliably removes, because each is a single decision that the clause now makes. This is the ordinary result and it is worth stating plainly: the improvement came from specification, not from tone, politeness, or phrasing.

A fault survives. Perfectly possible, and a useful result rather than a failed exercise. Common survivors are performance problems at larger `n`, an unstated error type for `n < 0`, and mixed exact and inexact arithmetic inside the computation even though the return type is a `Fraction`. The correct response is the one the unit teaches: write the clause that would have prevented it, and note that you have just extended the specification the same way C2 to C6 were extended.

A new fault appears. Also possible. Longer specifications produce longer code, and more code has more places to be wrong. Watch particularly for a hard-coded lookup table built from the C7 anchor values, which passes at the seven indices you named and fails elsewhere — the checker compares thirty-one indices, so it catches this, and a learner comparing only the small values by hand may not. If a learner reports this, they have found the sharpest point in the unit: examples are not a specification.

It passed both times. Then the comparison is about confidence rather than outcome. Before, a pass meant the defaults happened to align with the specification; now, a pass means the stated requirements were met. Those are different claims about the same green result, and only the second is repeatable.

22.4 Part 4 — Overreliance

A good paragraph is concrete and names the specific piece of prior understanding. Strong answers look like these:

> I had derived `B4 = -1/30` by hand, so when the routine returned `1/30` I knew the sign was wrong rather than assuming I had misremembered. Without that I would have accepted it, because `1/30` is exactly as plausible a string as `-1/30`.
>
> I knew two Bernoulli conventions existed. Without that, `B1 = +1/2` would have looked like a correct answer, and the disagreement with the oracle would have looked like a bug in the oracle.
>
> I knew every odd term above `B1` must be zero. That turned `B3 = -1.11e-16` from a rounding detail into a failed requirement.

Weak answers are general — “I understood the topic better”, “I was more careful” — and are worth pushing back on, because generality is what fails at the moment of judgement. The test of the paragraph is whether it names something that would have changed a specific decision.

The connection to make explicit when discussing this: the derivation in Unit 3 was not preparation for the lab, it was the thing that made the lab mean anything. Overreliance is not a character flaw or a matter of using AI too much; it is a gap between what you rely on and what you can check. The remedy is on the second side of that gap.

22.5 Part 5 — No-oracle transfer

Strong responses keep four different questions apart: what the output claims, what independent evidence is feasible, what remains a judgement, and when the learner must stop or escalate.

Case	Claim	Suitable check	Remaining judgement	Stop or escalate when
Citation	The paper exists and supports the stated population, method, result, figure, and strength of conclusion.	Search an authoritative catalogue or publisher record for existence and metadata, then read the paper for support. A second generated citation is not independent.	Whether the evidence is relevant and strong enough for the learner's own argument.	The source cannot be located, the statistic or causal wording is absent, or the learner cannot assess the method. Ask a librarian, tutor, or subject specialist as appropriate.
Policy summary	The supplied passage permits drafting assessed work if disclosed.	Direct line-by-line comparison shows the summary is wrong: the passage permits brainstorming and note organisation, prohibits submitting generated prose or code, and gives the assessment brief precedence.	How a real rule applies to a borderline activity not settled by its wording.	The applicable brief differs, the terms conflict, or the learner would be guessing about permission. Ask the named course contact before acting.
Recommendation	The famous unpaid placement produces better career outcomes for this learner.	Check concrete conditions directly and triangulate outcome claims using independent employer information, careers data, and accounts whose dependence is understood.	The weight given to income, prestige, travel, accessibility, duties, learning, and personal constraints.	Material conditions remain unknown, sources all repeat one claim, or the consequence exceeds what the learner can reasonably assess. Seek qualified careers or accessibility advice.

Consensus alone does not move any row to "proved". Several independent sources can strengthen a claim when their evidence and dependence are understood; several outputs or pages repeating the same unsupported statement cannot.

22.6 The honest position, for instructors

Keep both halves. Learners who leave this unit thinking the module is against using these systems have taken the wrong lesson, and so have learners who leave thinking a good prompt is a substitute for a check. The two sentences to preserve are that the specification work is a genuine gain, and that responsibility and judgement do not transfer.

The other half worth protecting is the limitation. Bernoulli numbers were chosen because they have an exact, published, checkable answer. Almost nothing a student will generate in the rest of their degree does. The transferable move is not "run the checker" but the ordered habit underneath it: state the requirement, find an independent reference, compare in a defined order, name what failed, and say plainly what you could not verify.

22.7 Checkpoint

`cp06-spec-clauses` is self-marking, with its own diagnostic feedback, and its answer is deliberately not reproduced here. Learners reach it by either route:

```
python3 selfcheck.py run cp06-spec-clauses    # or use the browser self-
```

22.8 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.


```
{
  "id": "cp06-spec-clauses",
  "unit": 6,
  "title": "Which clause kills which bug",
  "kind": "match",
  "question": "Each clause below was added to the prompt after a specific
  "answer": [
    0,
    1,
    2
  ],
  "options": [
    "A routine correct under the other convention.",
    "Rounding error instead of exact values -- slight at first, unmistaka
    "A routine that numbers only the non-zero terms."
  ],
  "prompts": [
    "'Use the first Bernoulli convention, B1 = -1/2.'",
    "'Return fractions.Fraction, never float.'",
    "'bernoulli(n) must be defined for every n >= 0, returning 0 for odd
  ],
  "diagnostics": {
    "0,2,1": "The last two are swapped. Ask what each clause makes *test
  },
  "explanation": "Every clause in the precise prompt is a bug that alrea
}
```

23 Unit 7 — The historical bug and historical honesty

Media for this unit:

- **Interactive Animation**: This unit features an interactive animation (`ada-indexing.html`). Please view the web version of the course to interact with it.
- **Interactive Animation**: This unit features an interactive animation (`three-buckets.html`). Please view the web version of the course to interact with it.

This unit is optional and it is not part of the module's core path. It sits outside the 2.5–4 hour core time budget, and its checkpoint `cp07-note-g` is an optional checkpoint. A learner who skips this unit has still completed the module: nothing in the core path depends on this page for a checkpoint, an activity, or a definition. The lesson it works through at length — that a claim's *status* is part of the claim — is also recorded in `course/misconceptions.md` entry 12, "Published means true", which is where the core path carries it.

Reading time: about 11 minutes. **Activity:** about 12 minutes. **Course items:** 2, 4, and light 7. **Checkpoint:** `cp07-note-g` (optional). **Before this unit:** Unit 3 (the Bernoulli numbers and Ada's indexing) and Unit 5 (the verification lab).

23.1 How to read the citations on this page

This unit is about the difference between a claim you can stand behind and a claim you have merely heard often. It would be absurd to make that argument sloppily, so every historical statement below carries one of two marks.

- *(Menabrea 1843)* — covered by the primary source: L. F. Menabrea, "Sketch of the Analytical Engine invented by Charles Babbage", translated with Notes A–G by Ada Augusta, Countess of Lovelace, in R. Taylor (ed.), *Scientific Memoirs*, vol. 3, London, 1843. The full bibliographic entry is in `course/references.md`.
- *unsourced* — this module has not verified it. The label is deliberate: the claim appears here because leaving it out would be misleading, not because it has been checked, and each such claim is recorded as an open question in `course/historical-notes.md`. It is the same "unsourced" bucket that section 6 defines.

Fuller detail on every historical claim is in `course/historical-notes.md`; this page does not duplicate it.

23.2 1. What the document is

In 1843, an English translation of Menabrea's account of Babbage's Analytical Engine was published in the third volume of Taylor's *Scientific Memoirs*. The translator was Ada Lovelace, and she appended to the translation a series of her own notes, lettered A to G *(Menabrea 1843)*.

Note G is the last and by some distance the longest. It contains a table that sets out, operation by operation, how the engine would compute a Bernoulli number *(Menabrea 1843)*.

The number worked through is the one Lovelace labels **B7**. Her scheme numbers only the Bernoulli numbers that are not zero, so her labels run ahead of the modern ones:

Lovelace's label	modern index
B1	B2
B3	B4

Lovelace's label	modern index
B5	B6
B7	B8

You met this mapping in Unit 3. The values are tabulated in `course/historical-notes.md` — deliberately not repeated here, because working the value out for yourself is the checkpoint.

23.3 2. The claim this module makes

Note G's published table contains at least one error.

That is stated as established. It is recorded, with its supporting material, in `course/historical-notes.md`.

Read the sentence again and notice how little it says. It does not say where the error is. It does not say what kind of error it is. It does not say whose it was. Everything that sentence leaves out, it leaves out on purpose — and section 3 shows what it took to earn each of the pieces it leaves out.

23.4 3. One claim checked, one still refused

You will encounter a specific account of the error, repeated widely and confidently: that one operation in the table has a **divisor and a dividend the wrong way round**. You will also encounter attributions — to Lovelace, to Babbage, to the compositor who set the type, or to something introduced in translation.

Those are two different claims, and they belong in two different buckets. Which bucket each goes in can change when a check is actually done — and that movement, from "widely repeated" to "checked against the source", is the whole point of the unit.

The description has been checked, and it holds. A facsimile of the printed diagram was examined (recorded as [P2a] in `course/references.md`). The error is in **operation 4**, which acts on the variables `V` and `V` in that order, while producing its stated result `(2n−1)/(2n+1)` requires dividing `V` by `V`. So the operands are inverted, exactly as the common account says. This claim sits in *established*, not *contested* — not because it is repeated often, but because the check was actually done, against the source rather than against other accounts. See `course/historical-notes.md` §4.

The attribution is still refused. Who introduced the slip — the compositor, the translator, Lovelace, Babbage — is not something the table can settle, and the common "it was a typesetting error" is an inference, not a finding. That stays **contested**. Two related things this module has also not established:

- how long the error stood in print before anyone remarked on it (*unsourced*)
- who first published a correction (*unsourced*; both are recorded as open questions in `course/historical-notes.md`)

Notice the shape of that. The check that was available got done, and one claim moved. The check that is not available — into who set the type in 1843 — has not, and that claim did not move. The honest position tracks what has actually been verified, not what would read well.

> **A note on the buckets.** *Contested* does not mean "probably false", and it does not mean "nobody knows anything". It means this module has not satisfied itself that the claim has been checked against the source. Claims move out of this bucket when the check is done — the operand inversion just did. Section 6 says what would move the attribution.

23.5 4. Why an error like this survives

This is the part that transfers, so it is worth being precise about the mechanism.

An error in a private working notebook gets caught or it does not, and either way it affects one person. The Note G table was in a different position entirely. It was in print. It was in a respected series. It was attached to two names readers had reason to respect. And it concerned a computation that very few readers were ever going to sit down and redo by hand.

Here is the uncomfortable part: **every one of those properties reduced the chance that anyone would check it.**

That is not a story about careless readers. It is a story about reasonable readers. Attention is finite. Spending it on a source that has already been vouched for, in a venue that has already applied its own scrutiny, is normally the right call. It was the right call in almost every case — and wrong in this one.

An error that arrives inside something already trusted does not have to defeat scrutiny. It only has to arrive somewhere scrutiny has already been withdrawn.

This is not a nineteenth-century problem that modern tooling has solved. Take this module's own verified example. Across the web, the as-printed Note G table is confidently said to compute −25621/630 — a figure that appears in no scholarly source this module has found. Nor does it survive recomputation: tracing the as-printed operations, inversion and all, gives 139/630 (`course/historical-notes.md` §4). The figure has survived exactly as Note G's error did — repeated inside finished, internally consistent, already-vouched-for accounts — because being published and repeated is not the same as being checked. The fix was not more care; it was a check that recomputes the number from the source and fails if the two disagree. The mechanism is two centuries old; so is the remedy.

23.6 5. Why this is a unit about AI and not about history

Change one variable and hold the rest fixed.

Fluent generated text has the same protective properties. It arrives finished rather than in draft. It is internally consistent, so nothing snags. And it carries no visible seam between the parts that are correct and the parts that are not — the tone does not change, the hedging does not increase, the sentence structure does not wobble.

That last property is the one that matters. A statistically plausible continuation can make an unsupported citation as polished as a correct one. Ordinary style and confidence are therefore insufficient evidence. Some systems can expose sources or calibrated uncertainty signals, but those still need to be checked for the task at hand.

The historical analogy is limited but useful: both readers face a polished claim whose appearance does not establish its source. Modern generation differs in scale, speed, variability, and the possibility of attached retrieval or tools.

23.7 6. The skill: three buckets

The response is not scepticism. Blanket scepticism is as useless as blanket trust and far more exhausting; a person who doubts everything checks nothing, because they have no way to decide where to spend the effort.

The response is **sorting**. Put every claim in front of you into one of three buckets.

Established. You can name where it comes from, and the source is one that you, or someone accountable to you, has actually looked at. Not "I have seen this asserted" — "here is the thing I looked at".

Contested. Sources disagree with each other, or a single account has been copied forward through many retellings without an independent check behind it. Repetition is not corroboration. Ten sources that all trace back to one unchecked account are one source.

Unsourced. You cannot currently say where the claim came from. This is not the same as false, and treating it as false is its own error. It is a claim with no weight yet.

Most claims that go wrong go wrong at the filing stage: something from bucket two or three gets quietly recorded as bucket one, and thereafter is repeated with the confidence appropriate to bucket one. That misfiling is recorded as a named expected error — `course/misconceptions.md` entry 12, "Published means true" — so the module carries it whether or not you work through this unit.

The second half of the sorting habit is knowing **what would move a claim**. A claim you cannot say that about is one you have not really thought about. For the Note G error, the movers are concrete: a facsimile or a physical copy of the 1843 printing; a comparison across separate print runs or later editions, if they differ; surviving manuscript or correspondence material from the period; and a published study that states which copy it worked from (*unsourced*; what manuscript material exists, and where it is held, is recorded as an open question in `course/historical-notes.md`).

23.8 7. What this looks like when you write

Three sentences, on the same subject, at three different strengths.

- "Note G's published table contains at least one error." — a claim this module supports.
- "The error is an inverted divisor and dividend in operation 4." — a claim this module now supports, having checked it against a facsimile [P2a]. It was contested until that check was done; section 3 tells that story.
- "Lovelace made an arithmetic slip in operation 4." — a claim this module does not support and does not make.

The third sentence is the most readable of the three. That is exactly why it is the dangerous one. Confident prose is easier to write and easier to read than careful prose, and generated text defaults to it, because that is the register most text is written in.

When you write up your own work, prefer the shorter, more careful sentence over the longer, more confident one.

23.9 A note on the general case

The three buckets are not a probability scale. "Contested" does not mean "roughly half likely to be true"; it is a statement about the state of the record and about the authority available to settle the claim, which is a different kind of uncertainty from a probability and does not behave like one. Some claims exceed what any currently available source can decide, and reporting that is the correct output rather than a failure to produce one. That distinction has a general form, set out in `course/second-order.md`, which the module's closing Unit 9 teaches from; nothing on this page, in the activity, or in `cp07-note-g` depends on it, and Unit 9 in turn does not assume you have read this one.

23.10 Checkpoint

`cp07-note-g` is an optional checkpoint: it belongs to this unit, it gates nothing, and no other unit requires it. Work through `activity.md` first, then:

```
python3 selfcheck.py run cp07-note-g     # or use the browser self-check
```

Both routes ask the same question and accept the same answers. Neither is a fallback for the other.

23.11 Where to look next

- `course/historical-notes.md` — the full detail behind every historical claim in this module, including the open questions listed above.
- `course/misconceptions.md` entry 12, "Published means true" — the same established-versus-contested lesson in the module's register of expected errors.
- `course/references.md` — full bibliographic entries and the module's source policy.
- Unit 8 — turning "I checked this" into a written defence someone else can read. It does not require this unit.

24 Unit 7 activity — check the table, then sort the claim

Time: about 12 minutes. **Checkpoint:** `cp07-note-g` (optional). **You will need:** the index mapping from Unit 3, and either the table you built in Unit 3 or the lab oracle from Unit 5. Either is sufficient.

Optional unit, outside the core time budget. Nothing here gates anything, and the module is complete without it.

Two tasks. The first takes about four minutes and is the checkpoint. The second takes about eight and is the one worth protecting if you are short of time.

24.1 Task 1 — establish the value independently, then compare

Note G's worked example computes the number **Lovelace labels B7** *(Menabrea 1843)*. Your job is to state what that number should be *without looking at the historical table*, and only then to look.

The order matters. A reference value you obtained after seeing the thing you are checking is not a reference value.

24.1.1 Route A — table-based verification

Equal status with Route B. Use it if you are not running Python, or if you would rather do the reasoning on paper.

1. Take Lovelace's label **B7** and convert it to a modern index using the mapping from Unit 3. Write the modern index down before going on.
2. Look that modern index up in the Bernoulli table you built in Unit 3, or in the table in `course/historical-notes.md`.
3. Write the exact value as a fraction. Not a decimal — the whole argument of this module rests on exactness.

24.1.2 Route B — automated verification

Equal status with Route A.

```
python3 selfcheck.py explore
```

Then, at the `explore>` prompt:

```
map
ada 7
```

`map` prints the index mapping; `ada 7` converts Lovelace's label and returns the oracle's exact value at the corresponding modern index.

24.1.3 Then answer these, in writing, in your evidence pack

Two or three sentences each is plenty.

1. **What did the comparison actually catch?** Be specific about the *kind* of discrepancy a value-by-value comparison can detect.
2. **What would it not have caught?** Name at least one class of error in a printed procedure that comparing final values leaves completely invisible.
3. **What did you have to know before the comparison meant anything?** There is more than one thing here, and one of them is not about Bernoulli numbers at all.

24.1.4 Now run the checkpoint

```
python3 selfcheck.py run cp07-note-g     # or use the browser self-check
```

Both routes ask the same question, accept the same forms of the answer, and give the same diagnostics. Pick whichever you have to hand.

24.2 Task 2 — sort one claim, and say what would move it

This is the harder task and the one that transfers furthest.

Below are seven statements about Note G, all of which circulate. **Pick one** — or work through several if you have time — and do three things with it.

1. **Sort it** into `established`, `contested`, or `unsourced`, as those buckets are defined on the student page.
2. **Say why** you put it there, in one sentence. "It sounds right" is not a reason; "I have not seen a source, only assertions" is.
3. **Say what would move it.** Name a specific piece of evidence that would let you reclassify it — something a person could actually go and consult, not "more research".

Do not look up the answers first. The sorting is the exercise; getting the "right" bucket matters less than being able to defend the placement.

24.2.1 The statements

A. Note G appeared in 1843, as part of Lovelace's translation of Menabrea's memoir on the Analytical Engine, with her Notes appended.

B. The table published in Note G contains at least one error.

C. The error is an inversion — a divisor and a dividend the wrong way round in one operation of the table.

D. Lovelace introduced the error, and Babbage did not catch it before publication.

E. The error was introduced by the printer when the type was set, not by either author.

F. The error stood uncorrected for more than a century.

G. The as-printed table computes −25621/630.

24.2.2 Write it up like this

```
Statement:       <letter>
Bucket:          established | contested | unsourced
Why:             <one sentence>
What would move it: <a specific, consultable piece of evidence>
```

Keep this. It is worth adding to the evidence pack you assemble in Unit 8 — that pack does not require it, but this is the clearest short demonstration you will produce that you can tell "I checked this" from "I have read this a lot".

24.3 Optional extension — the same test on generated text

Optional. Not required, and it does not gate anything.

Ask an AI system for a short account of the error in Note G. Then apply Task 2's three questions to every factual assertion in what comes back.

You are not looking for it to be wrong. It may well not be. You are looking at something more specific: **whether the register changes** between the parts that are well established and the parts that are contested or unsupported. Does the confidence of the prose track the strength of the evidence?

Record what you find in your evidence pack in one sentence. It is a useful thing to have observed once, first-hand, rather than to have been told.

24.4 Before you move on

You should be leaving this unit with:

- the modern index corresponding to Lovelace's B7, and its exact value, obtained by a route you can describe
- `cp07-note-g` passed
- written answers to the three questions in Task 1
- one sorted statement from Task 2, with its "what would move it"

All four are worth adding to the evidence pack you assemble in Unit 8, as an appendix marked optional. Unit 8 does not require them.

24.4.1 Activity Answers

25 Unit 7 answers and guidance

For learners checking their own work after attempting the activity, and for instructors marking or discussing it.

This key does not restate the value asked for at `cp07-note-g`. Deriving it is the learning objective, and the checkpoint gives targeted diagnostics if you go astray. Checkpoint answers live in `course/lab/checkpoints.py` and nowhere else, by design.

Citation shorthand. *(Menabrea 1843)* = Menabrea, "Sketch of the Analytical Engine invented by Charles Babbage", trans. with Notes A–G by Ada Lovelace, in Taylor (ed.), *Scientific Memoirs*, vol. 3, 1843. Full entry in `course/references.md`.

25.1 Task 1 — the three written questions

25.1.1 1. What did the comparison actually catch?

A value-by-value comparison against a reference you obtained independently catches exactly one class of thing: a **disagreement in a final value at a named index**. That covers a wrong number, a wrong sign, an inexact number where an exact one was required, and — if you are comparing index against index rather than scanning for familiar-looking values — a value that is itself correct but filed under the wrong label.

That last case is worth dwelling on, because it is the one you met at `cp05-diagnose`. Every individual value can be a genuine Bernoulli number while the output as a whole is still wrong.

A good answer names the class of error, not just the outcome. “It caught that the numbers differ” is thin. “It caught a disagreement at a specific index, which is the only thing comparing final values *can* catch” is the answer.

25.1.2 2. What would it not have caught?

Several things, and any one of them earns the mark:

- **A right answer reached by a wrong route.** Two errors that cancel, or an operation that is wrong but happens not to affect this particular example, leave the final value intact.
- **Errors elsewhere in the procedure.** A single worked example exercises a single path. Anything the example does not reach is untested.
- **Errors in the surrounding prose.** A comparison of numbers says nothing about the claims made in the text around the table.
- **The location of the fault.** This is the important one. A final-value comparison tells you *that* something disagrees. It does not tell you *which step* is responsible.

That last point is the hinge of the whole unit. The value comparison supports the claim that the published table contains an error — a claim about final values. It cannot, on its own, underwrite the claim about *where* the error is. That location claim — operation 4, operands inverted — took a different kind of check: reading the facsimile of the 1843 printing [P2a] and recomputing the table’s operations in `noteg.py` (see `course/historical-notes.md` §4). Each kind of claim needs its own kind of check, and the location claim moved to *established* only when that check was done. If a learner connects those two things without prompting, they have understood the unit.

25.1.3 3. What did you have to know before the comparison meant anything?

At least three things, and the third is the one the question is fishing for.

- **The convention.** Under the second Bernoulli convention, `B1 = +1/2`, one value in the sequence differs. Comparing against a reference built on a different convention produces a fake discrepancy.
- **That your reference is independent.** A reference derived from the thing you are checking checks nothing.
- **What “B7” means in the source’s own notation.** This is not a fact about mathematics at all. It is a fact about how a document written in 1843 labels its quantities — Lovelace numbers only the non-zero Bernoulli numbers *(Menabrea 1843)*, so her labels do not mean what the identical modern labels mean.

Reading a historical source requires knowing its conventions, and the conventions are usually not stated in the source, because at the time they did not need to be. Exactly the same is true of a specification handed to a generating system: the requirement most likely to be violated is the one so obvious to you that you did not write it down.

25.2 Task 2 — the seven statements

The bucket assignments below are this module's positions, stated as positions. A learner who places a statement differently but gives a defensible reason and a concrete "what would move it" has done the exercise correctly. The reasoning is the assessable part, not the label.

25.2.1 A. Note G appeared in 1843, with Lovelace's Notes appended

Established. Covered directly by the primary source *(Menabrea 1843)*, and recorded in `course/references.md`.

What would move it: essentially nothing short of a bibliographic error in the volume itself. This is about as settled as a historical claim gets, and it is useful in the exercise precisely as a calibration point — the bucket is not empty.

25.2.2 B. The published table contains at least one error

Established. This module states it, and `course/historical-notes.md` gives the fuller detail.

Note how narrow the statement is. It asserts existence and nothing else — no location, no kind, no author. A learner who marks this "contested" has confused it with statement C, which is the confusion the whole unit is built around. That is a productive mistake and worth discussing rather than simply correcting.

What would move it: a careful published analysis of the original printing concluding that the table is in fact correct as printed.

25.2.3 C. The error is an inverted divisor and dividend in one operation

Established — and it is worth knowing that it only recently was.

For most of this module's development the right answer here was *contested*: the description is the one you meet most often, and wide circulation is not evidence, because many retellings may descend from a single earlier account rather than from the printing itself.

Then the check happened. The "what would move it" for this claim was always specific — *a facsimile of the 1843 printing, read directly, with a trace of the operations against a correct computation* — and it was carried out. The facsimile [P2a] shows the error in **operation 4**, acting on `V` and `V` in that order, where producing the stated `(2n−1)/(2n+1)` needs `V ÷ V`. The operands are inverted. See `course/historical-notes.md` §4 and the unit reading, section 3.

So this claim moved buckets — not because it was repeated more, but because the consultable thing got consulted. **A learner who marks this "contested" is giving the answer that was correct on the older evidence, and should notice from section 3 that the evidence changed.** Reward the reasoning either way; the tell for full marks is whether they cite the facsimile check rather than the claim's popularity. What stays contested is *who* introduced it — items D and E — because no facsimile can show that.

25.2.4 D. Lovelace introduced the error; Babbage did not catch it

Contested. An attribution, and one of several in circulation. Attribution is a strictly harder claim than existence: it requires evidence about the process of production, not just about the printed result.

What would move it: surviving manuscript drafts, proof sheets, or correspondence from the period showing the value at different stages (*unsourced*; what manuscript material exists, and where it is held, is recorded as an open question in `course/historical-notes.md`).

25.2.5 E. The error was introduced by the printer

Contested, and for the same reason as D. It is a competing attribution, and the same evidence would bear on both.

If a learner notices that D and E cannot both be right, and that this alone is enough to place both in "contested" without knowing anything else, that is a genuinely good observation. **Mutual incompatibility among widely repeated accounts is a cheap and reliable contestedness detector.** It costs nothing to apply and it works on generated text.

25.2.6 F. The error stood uncorrected for more than a century

Unsourced. This module has not verified when the error was first remarked on in print, or by whom (*unsourced*; recorded as an open question in `course/historical-notes.md`).

"Unsourced" rather than "contested" is the right call here because the module is not aware of competing accounts — it is simply aware of no checked account at all. Accepting "contested" from a learner is reasonable; what is not reasonable is "established", and the tell is that the statement is quantitative ("more than a century") while nothing supporting the number has been produced.

What would move it: a dated published correction, or a survey of the literature identifying the earliest one.

25.2.7 G. The as-printed table computes −25621/630

Unsourced or contested before checking — and refuted once the check was done.

Before any check, this is a textbook bucket-two-or-three claim: the figure is everywhere online, and cited to no scholarly source. Repetition is not corroboration. But this claim, unlike D and E, is checkable from your desk, and the module checked it: the recomputation in `course/lab/noteg.py` traces the as-printed operations — operation 4 inverted, exactly as the facsimile shows — and the as-printed table gives **139/630**, not −25621/630. See `course/historical-notes.md` §4.

A learner who places this in "established" because the number is so widely repeated has made exactly the misfiling the unit warns about. The recomputation refutes the claim; wide circulation never supported it.

What would move it: nothing now moves it back — a claim refuted by a recomputation you can rerun is settled — but a published derivation showing a different reading of the as-printed operations would reopen the question.

25.3 Common ways this activity goes wrong

Treating "contested" as a polite way of saying "false". It is not. It is a statement about the strength of the evidence available, not about the truth of the claim. Statement C may well turn out to be an accurate description; the module's position is that it has not established this, not that it doubts it.

Sorting by plausibility instead of by provenance. The buckets are about where a claim came from, not about how likely it sounds. A highly plausible claim with no source is unsourced. This is the single most common error in the task.

Producing a vague "what would move it". "More research" and "checking the sources" are not answers. The test is whether you have named something a person could actually go and consult. If you cannot, you have not yet worked out what the claim depends on.

Looking it up before sorting it. Understandable, and it destroys the exercise. The skill being trained is what you do with a claim *before* you have resolved it, because in practice most claims are never resolved — they are used, or not used, in whatever state you found them.

25.4 For instructors

- The unit is deliberately short and deliberately under-coloured. If it feels thin next to the rest of the module, that is the design responding to the risk flagged in the plan: this is the unit where invented historical detail is most likely to creep in, from any author, human or otherwise.
- The most valuable discussion prompt is not "which bucket" but **"what would move it"**. That question generalises to essentially every claim a student will meet in the rest of their degree.
- Statement F is the useful one for cohorts that find the exercise easy. It is quantitative, memorable, entirely uncontroversial in tone, and unsupported — which is a good description of a large fraction of confident writing.
- If the course team later resolves the specific description or the attribution from primary evidence, the change belongs in `course/historical-notes.md` first. This unit deliberately holds no independent historical position that could drift out of step with it.

25.5 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.

```
{
  "id": "cp07-note-g",
  "unit": 7,
  "title": "Comparing against the historical table",
  "kind": "fraction",
  "question": "Note G's worked example computes Lovelace's B7. Using the
  "answer": "-1/30",
  "diagnostics": {
    "-1/42": "Close to the right region of the table but one term off. Ma
    "1/30": "The magnitude is right and the sign is not. Check the moder
    "1/42": "That is the modern B6, which is Lovelace's B5 -- one label
  },
  "hint": "Two steps, in this order. First: which modern index does Ada'
  "explanation": "Being able to state the correct value independently is
}
```

26 Unit 8 — Disclosure, defence, and responsible use

Media for this unit:

- **Interactive Animation**: This unit features an interactive animation (`evidence-pack.html`). Please view the web version of the course to interact with it.

Reading time: about 17 minutes. **Activity:** about 10 minutes. **Course items:** 6 and 8, with light 3 and 7. **Checkpoint:** `cp08-disclosure`. **Before this unit:** Unit 6 (re-prompting and specification). Unit 7 (historical honesty) is optional and is not assumed here.

This is the last unit of practical work. It gives you the two written statements that turn everything you have done into something another person can read and rely on. Unit 9 closes the module afterwards by naming the general question all of this has been circling.

26.1 1. Two sentences that are not the same sentence

> "Parts of this were drafted with AI assistance."
>
> "This is correct because I checked it against values I derived myself, and here is what the check showed."

The first is a **disclosure**. It answers *how was this made*.

The second is a **defence**. It answers *why is this right*.

They are separate statements, they answer different questions, and neither one substitutes for the other.

26.1.1 Why the substitution is tempting

Disclosure feels like it should be enough. You have been honest. You have hidden nothing. Surely the responsible thing is done.

It is half done. A reader who knows exactly how your work was produced still knows nothing about whether it is correct. Provenance is not evidence. An honest disclosure attached to a wrong answer produces an honest wrong answer, and the honesty does not repair the wrongness.

The failure runs the other way too. Work you can defend in complete detail, with the assistance undeclared, is not rescued by being correct. The reader needed to know how it was made in order to know what to check — and to know how much of your judgement is behind it.

Two statements, two questions — but they do not carry equal weight.

26.1.2 Which one carries the work

Both belong in honest work, and disclosing AI assistance is the easy, blameless part: using AI is not a problem, and you should say plainly where you used it. But a disclosure never makes anything correct. The claim that carries the work is the other one — being able to say, on your own authority, **"I understand this well enough to defend it, I have evidence that it is right, and I can show you both."**

That is a claim of *ownership*, and it has two separable parts. **Understanding** is being able to predict and reproduce what the thing does: your model of it maps its behaviour, and you have checked that the map holds — for which the verification lab gathers evidence. **Correctness** is the further claim that the thing you understand is also right — that the specification it meets is the one you wanted. The two come apart: you can understand something perfectly and have it be wrong. The module's authors mapped the as-printed Note G table well enough to predict what it computes — `139/630`, not the number the web repeats (`course/historical-notes.md` §4): a perfect map of a wrong table. Understanding without a correctness check is not yet a defence; disclosure without either is not a defence at all.

So disclose, always, and without shame — then spend your effort on the defence. It is the part that decides whether the work is any good.

Ownership also gives you a stopping boundary. Pause rather than approve work when you can no longer predict relevant behaviour, explain the transformation that produced the result, identify a suitable independent check, or interpret a failed check. A confident feeling and a polished output are not evidence. Narrow the claim, rebuild the missing understanding, or ask a qualified reviewer before putting your name to it.

26.2 2. The disclosure

A disclosure is short, factual, and about process. It carries three things:

1. **What you used AI assistance for** — drafting, translating, reviewing, debugging, explaining, summarising.
2. **At which stage** — first draft, revision, checking, or throughout.
3. **What you did about it afterwards** — how the output was verified, and what you changed.

The required detail depends on the context. Follow the applicable institutional, assessment, publication, or workplace rule. A tool and version can be useful for reproducibility, terms-of-service review, or incident investigation, although the name alone says little about what happened. Record it when required or materially useful, together with the scope and stage of use.

26.2.1 Reusable template

Copy this, delete what does not apply, and keep it to two or three sentences.

```
AI-use disclosure
-----------------
AI assistance was used to <what: draft / translate / review / debug / ex
<which part of the work> at the <stage> stage.

The output was <not used verbatim / revised after verification / used ver
after checking>. Specifically: <what you changed and why>.

All <values / claims / code> were verified by <table-based comparison aga
reference I derived / running the module's checker / consulting the cite
source>, and I am responsible for the final content.

<Tool and version, when required or materially useful.>
```

26.2.2 Professional code-review disclosure

The same structure can be used outside coursework. For example, a code review or design note can record what was generated, how it was checked, and who is taking responsibility for the final change.

```
## AI Assistance Disclosure
- **Scope:** AI assistance used for an initial draft of `bernoulli.py`.
- **System:** <tool, model/version, and date, when required or useful>.
- **Prompt record:** <full prompt or a stable link/hash to the retained
- **Verification:**
  - Code audited for untrusted imports and security risks.
  - Tests run against the reference oracle.
  - Edge cases checked: `B1 = -1/2`, exact `Fraction` returns, odd terms
    `B1`, and the large-index case `B30`.
```

```
- **Evidence:** <link to the retained test output or review record>.
- **Accountability:** <named person or role> reviewed and accepted the c
```

26.2.3 Worked academic example

```
AI-use disclosure
-----------------
AI assistance was used to draft the initial version of the bernoulli() r
in section 3, at the first-draft stage.

The output was revised after verification: the first version used the seco
Bernoulli convention (B1 = +1/2) and returned floats. I re-specified the
requirement and replaced it.

All returned values were verified against the module oracle for n = 0 to
for n = 30, and I am responsible for the final content.
```

Notice what the disclosure does *not* do. It does not argue that the routine is correct. It reports that verification happened. That is a different sentence, and it comes next.

26.3 3. The defence

A defence is about evidence. It carries three things:

1. **What you checked.**
2. **What the check showed.**
3. **Which specification or assumption the check was against.**

That third item is the one people leave out, and leaving it out hollows out the other two. “All tests passed” is worth very little if a reader cannot tell what the tests were testing for. Passing tests against the wrong convention is exactly the failure this module has been circling since Unit 3.

26.3.1 What a defence never cites

- That the system stated the answer was correct.
- That the output sounded confident, or was long, detailed, or well formatted.
- The name or version of the tool. That is provenance; it belongs in the disclosure.

This is not a point of etiquette. Ordinary confident wording can accompany correct or incorrect generated text, so wording alone does not establish correctness. Research has found that specially elicited or trained confidence measures can discriminate answers on some evaluated tasks, but calibration varies by model, task, prompting method, and distribution [L4]. This module gives you no validated confidence measure for the Bernoulli task, so a confident phrase or self-rating is not part of the defence.

A defence can rest on **test results, checked reasoning, primary sources, independent measurements, or qualified review**, chosen to match the claim. The source must provide relevant evidence rather than repeat the assertion being checked.

Independent-check principle: the evidential basis must not be controlled only by the system whose output is under review. Asking the same model to endorse its first answer, or trusting a routine’s own claim that it passed, produces another assertion rather than independent support. Use a separately established reference, direct observation, a different method whose dependence you understand, or a qualified reviewer as appropriate to the claim.

26.3.2 Two-sentence defence template

```
Correctness defence
-------------------
<Final result> is correct under <the stated specification or convention>,
verified by <the specific check> which showed <the specific outcome>.

The <discrepancy / assumption / convention> I found was <what it was>, w
resolved by <what you did>.
```

26.3.3 Both routes produce a real defence

These routes have equal standing in this module, but they do not produce identical evidence. Each defence must state its actual coverage and limitations.

Table-based verification:

```
The values I checked from B0 to B8 agree under the first Bernoulli conven
(B1 = −1/2), verified by comparing B0 to B8 term by term against the valu
derived from the recurrence by hand, which agreed at every index.

The discrepancy I found was at B1, where the generated version gave +1/2
traced this to the second convention and corrected the specification.
```

Automated verification:

```
The routine agreed with the oracle under the interface contract in Unit
n = 0 to 20 and n = 30; the checker reported no divergence over those in

The discrepancy I found was float output at higher indices, reported as
`not-fraction`, which I resolved by requiring exact Fraction returns.
```

Both name a specification. Both name a check. Both name what it showed. Both name a discrepancy and its resolution. That is what makes them defences rather than assertions.

26.4 4. The evidence pack

The pack is yours. It is a private formative self-check: you are not asked to submit it, it is not marked, and nobody collects it. Export it or show it to someone only if you decide to.

You have already produced most of this without being asked to collect it.

#	Item	Where it came from
1	Original prompt	Unit 4
2	Generated output or code	Unit 4
3	Verification result	Unit 5
4	Discrepancy notes	Unit 5
5	Revised prompt	Unit 6
6	Final result	Unit 6
7	AI-use disclosure	this unit
8	Two-sentence correctness defence	this unit

The structure and the completion criteria are in `course/learner-evidence-template.md`. Use that file as the container; this page only explains items 7 and 8.

The pack works identically from either route. A pack built on table-based verification is complete, and a pack built on the checker is complete. What is being evidenced is the same thing: that you established what correct looked like, compared against it, found the disagreement, and can say what you did.

If you took the Python route, you can export your self-check record:

```
python3 selfcheck.py progress --evidence
```

That prints a Markdown table you can paste straight into the pack. Read the note the tool prints underneath it: the record shows which checkpoints you passed, and is **not** a claim that you can defend the answers. An AI system can help draft a defence, but it cannot supply evidence that was never gathered or accept accountability on your behalf. You must check the statement against your record and be able to support it yourself.

26.5 5. Where the boundary is

Do not paste confidential, personal, unpublished, or assessment-restricted material into an AI system.

Four short reasons, one per category. Confidential material is not yours to disclose. Personal data about other people is not yours to send. Unpublished work is not yours to circulate, and that includes other people's drafts. And some assessed material you are simply not permitted to share, whatever your own view of the risk.

This module signposts that boundary rather than teaching it. Privacy — and sustainability alongside it — is covered properly by wider course **Item 5**; see `course/course-map.md` for the item list and this module's coverage level.

26.5.1 Rules about AI use in assessed work

Whether AI assistance is permitted in a particular piece of assessed work, and in what form, is set by your own institution or course. This module does not set those rules and does not state them. Follow whatever your institution says, and ask your tutor if you are unsure.

26.6 6. Ada's claim, read carefully

In Note G, writing about the Analytical Engine, Lovelace stated that it "has no pretensions whatever to originate anything" *(Lovelace 1843, Note G — full entry in `course/references.md`)*.

The sentence gets quoted a great deal in discussions of AI, usually as a settled verdict. It is worth slowing down, because there are two easy readings and both are wrong.

The first easy reading: she settled it, and language models therefore originate nothing. But consider what she was describing. The Analytical Engine would execute a procedure specified completely in advance, by her, operation by operation. Every quantity it would handle was named beforehand. A system trained on a large corpus is not in that position: it produces combinations its authors did not write and could not have predicted in advance, and the specification of its behaviour is nothing like a Note G table. Her sentence was exactly true of the thing in front of her. Extending it to a different kind of system is an argument that has to be made, not a quotation that can be deployed.

The second easy reading: she was writing about clockwork, so the remark is obsolete. But this treats "originate" as though its meaning were obvious. It is not. If origination requires intention, or understanding, or a purpose held by the thing doing it, then producing a novel combination is not sufficient and her claim survives intact. If origination just means producing something that did not exist before and was not specified in advance, then the question turns on facts about what these systems produce. Both positions are held by

serious people. **This module does not resolve it, and you should be suspicious of any short treatment that does** — including a confident one you might be handed by a language model, which will produce a well-turned answer to a question its own output cannot settle.

Her sentence still prompts a useful accountability question: who specified, deployed, used, reviewed, and approved the result? For a student's own submission, the student remains answerable for using generated material. In a modern organisation, responsibility can also be shared among users, reviewers, deployers, employers, and vendors. AI assistance does not remove the learner's responsibility, but the module does not claim that no one else has duties.

26.7 7. What you can now do

- Distinguish an ordered procedure from statistically generated continuation, and say why the difference changes how you check the output.
- Derive a reference value before trusting a produced one, and explain why that order cannot be reversed.
- Verify by table-based comparison or by automated checker, and name the specific failure mode rather than reporting "it is wrong".
- Turn a failure you have observed into a specification clause that prevents it.
- Write a disclosure that reports how work was made, and a defence that rests on evidence rather than on authority.
- State the scope and limits of that evidence, and retain ownership of the final claim.
- And, if you did the optional Unit 7: sort a claim into established, contested, or unsourced, and say what evidence would move it.

The closing standard is not "I generated this and disclosed it." It is "I understand the claim, I gathered independent evidence appropriate to it, I can defend its scoped correctness, and I take ownership of using it." Disclosure belongs alongside that standard for transparency; it does not satisfy it.

26.8 8. Where this connects

This module leads on some course items and only touches others. The canonical list of items, names, and coverage levels is `course/course-map.md` — this page does not restate them.

- The items this module covers strongly are the ones you have just been doing: evaluating outputs, critical thinking, academic integrity, and effective use in computing.
- The items it touches lightly — responsible AI and professional practice — are represented here only by the parts the Ada and Bernoulli example naturally supports: accountability, disclosure, and a review-style workflow.
- Item 5 is signposted only. Nothing on this page is a substitute for it.

If you take one habit forward, ask what independent evidence supports the result and whether you understand it well enough to defend the claim within that evidence's limits. Record how it was made as a separate transparency statement. If you also did optional Unit 7, use its three claim-status buckets when deciding what still needs checking.

26.9 A note on the general case

The split between the two statements is an instance of something wider. Disclosure carries **provenance** — where the thing came from; a defence carries **warrant** — the evidence, and the specification the evidence was gathered against. A candidate answer is not warranted merely by fluent writing or honest provenance. Its support must be appropriate to the claim and separable from the assertion being checked. The module's

closing Unit 9 interprets that distinction through the second-order material set out in `course/second-order.md`; neither this page, the evidence pack, nor `cp08-disclosure` depends on it.

This is practical. The module's checker computes its verdict from outputs compared with a separately established oracle; the routine's own docstring cannot make it pass. In research or professional review, the corresponding support might be a measurement, a primary source, an independently implemented method, or review by someone qualified and accountable. The form changes with the claim; the principle does not.

26.10 Checkpoint

Assemble your evidence pack first — the checkpoint is easier and considerably more meaningful once you have written a real defence rather than imagined one.

```
python3 selfcheck.py run cp08-disclosure     # or use the browser self-c
```

Both routes ask the same question and give the same diagnostics.

27 Unit 8 activity — assemble the pack, write the defence

Time: about 10 minutes. **Checkpoint:** `cp08-disclosure`. **You will need:** your prompts and outputs from Units 4 and 6, and your verification results from Unit 5. If you did the optional Unit 7, bring your claim sorting as well; the pack does not require it.

Three short tasks, then the checkpoint. The optional CS extension at the end sits outside the core time budget.

27.1 Task 1 — assemble the evidence pack (about 4 minutes)

Open `course/learner-evidence-template.md` and fill in items 1 to 6 from work you have already done. You are collecting, not producing. The pack stays with you: it is not submitted and nobody collects it.

#	Item	Where you made it
1	Original prompt	Unit 4
2	Generated output or code	Unit 4
3	Verification result	Unit 5
4	Discrepancy notes	Unit 5
5	Revised prompt	Unit 6
6	Final result	Unit 6

If you took the Python route, add your self-check record:

```
python3 selfcheck.py progress --evidence
```

That prints Markdown. Paste it in.

If you took the table-based route, attach the table you built and the comparison you made instead. **This is not the lesser route.** The pack is evidencing that you established what correct looked like, compared against it, and can account for the difference — and a hand-built table evidences that as directly as a checker run does.

If item 4 is empty because nothing ever disagreed, say so explicitly and say what you checked. "No discrepancy found, having compared X against Y" is a result. A blank space is not.

27.2 Task 2 — write the disclosure (about 2 minutes)

Use the template on the student page. Two or three sentences.

Cover: what you used AI assistance for, at which stage, and what you did about the output afterwards.

Then read it back and apply one test:

> Does any sentence in this disclosure argue that the work is *correct*?

If yes, that sentence is in the wrong statement. Move it to Task 3. A disclosure that has started arguing for correctness has stopped being a disclosure.

27.3 Task 3 — write the defence (about 4 minutes)

Two sentences. This is the shortest task and the hardest one.

Sentence one: what your final result is, which specification or convention it is correct under, what check you ran, and what the check showed.

Sentence two: the discrepancy, assumption, or convention you had to resolve, and how you resolved it.

27.3.1 Then run this checklist against what you wrote

- ☐ Does it name a **specific check**, not "I verified it"?
- ☐ Does it say **what the check showed**, not just that it was run?
- ☐ Does it name the **specification or convention** the check was against?
- ☐ Is the evidential basis independent of the system whose output is under review, rather than another assertion from that system?
- ☐ Does it rely on the model's confident wording, unsupported self-rating, or tool identity as proof? **If so, remove that.** Tool identity belongs in the disclosure. A confidence measure belongs in a defence only if it has been validated and calibrated for the relevant task, which this activity does not provide.
- ☐ Could a reader who does not trust you, and who cannot see your screen, tell what evidence exists?

That last question is the real one. A defence is written for a sceptical reader, not a sympathetic one.

27.4 Task 4 — the checkpoint

```
python3 selfcheck.py run cp08-disclosure     # or use the browser self-c
```

Both routes ask the same question and give the same diagnostics. Do this after Tasks 2 and 3, not before: the checkpoint asks you to sort statements into disclosure and defence, and it is a different exercise once you have written both yourself.

27.5 Optional reflection

Not assessed, not required, and worth five minutes if you have them.

In Note G, Lovelace wrote that the Analytical Engine "has no pretensions whatever to originate anything" *(Lovelace 1843, Note G)*.

Write one paragraph. Not on whether she was right about language models — that question is genuinely open, and the student page explains why a short confident answer to it should be distrusted. Write instead on this:

> Which parts of the work in your evidence pack did you originate, on your own understanding of the word, and how would you show it to somebody else?

The interesting answers are usually not about the code.

27.6 Optional CS extension

Optional. Outside the core time budget. It does not gate anything, and the module is complete without it.

Take a small routine — twenty or thirty lines is plenty. Your Unit 5 solution works, or anything of your own.

Pick **one** of three jobs and ask an AI system to do it:

- **Translate** the routine into another language you can read.
- **Review** it and report defects.
- **Debug** it after you have introduced a fault deliberately.

Then produce a **test-based account**, which is the whole point of the exercise:

1. **Before**, write down what your tests currently cover. Not what you think the code does — what is actually checked.
2. **Run those tests** against whatever comes back. For a translation, port the tests too, and note honestly whether porting them was harder than porting the code.
3. **Record where the assistance helped**, with evidence. A defect it reported that your tests had not caught is evidence. A defect you already knew about is not.
4. **Record where it did not**, with evidence. Reported defects that are not defects. Real defects it did not report. Changes that altered behaviour your tests did not cover — which is the most instructive category, because it tells you where your test suite is thin.
5. **State what you would now add to your tests**, and why.

Note the shape of step 5. The most durable value in this exercise is usually not the code that came back. It is finding out which parts of your own program were never really being checked.

Add the account to your evidence pack as an appendix, marked optional.

27.7 Before you finish

Your pack should contain all eight items, with the disclosure and defence as separate statements that do not do each other's jobs. Check that once more.

Then, if the work is for assessment, check what your own institution or course permits and requires. This module does not set those rules.

That is the practical work finished. Unit 9 closes the module by naming the general question behind it, and your pack is what you take into it.

27.7.1 Activity Answers

28 Unit 8 answers and guidance

For learners checking their own work, and for instructors marking or discussing it.

`cp08-disclosure` is a comprehension check on the distinction taught in this unit, so this key discusses the distinction rather than listing the checkpoint's options. Checkpoint answers live in `course/lab/checkpoints.py` and nowhere else.

Citation shorthand. *(Lovelace 1843, Note G)* = Note G, in Lovelace's translation of Menabrea, "Sketch of the Analytical Engine invented by Charles Babbage", in Taylor (ed.), *Scientific Memoirs*, vol. 3, 1843. Full entry in `course/references.md`.

28.1 The distinction, in one table

	Disclosure	Defence
Answers	How was this made?	Why is this right?
Is about	process and provenance	evidence
Contains	what, at which stage, what you did next	what you checked, what it showed, against which specification
May cite	the tool/version when required or materially useful	test results, checked reasoning, primary sources, measurements, qualified review
Does not establish correctness by itself	—	confident wording, unsupported self-rating, tool identity
Fails by	omission or vagueness	asserting rather than evidencing

The single most useful test: **a disclosure that starts arguing for correctness has stopped being a disclosure**, and a defence that reports only what was used has never started being one.

28.2 Task 2 — marking the disclosure

A sound disclosure names the use, the stage, and what happened afterwards. It is two or three sentences and it is boring. Boring is correct.

Accept:

> "AI assistance was used to draft the first version of the routine. The output was revised after verification: it used the wrong sign convention for B1 and returned floats. I re-specified and replaced it."

Query:

> "AI was used, but everything was carefully checked and the final answer is correct."

The first clause is a disclosure and the rest is an unevidenced defence wedged into it. Neither statement is now doing its job: the disclosure has become vague about what was actually used, and the correctness claim has no check attached. Ask the learner to split it in two and watch what happens — usually the correctness half turns out to have nothing behind it, which is exactly why it had to lean on the disclosure for support.

Query:

> "I used AI."

True, and useless. A reader cannot tell what to check. Ask: used for what, when, and what did you do with the output?

Note on tool names. Learners often assume naming the tool and version is the whole disclosure. It is not. Include those details when required or when they aid reproducibility or review, but also state the scope, stage, and subsequent action. A product name alone does not tell the reader how the work was made.

28.3 Task 3 — marking the defence

The four criteria: a **specific check**, a **stated outcome**, a **named specification or convention**, and **no appeal to authority**.

Accept — automated route:

> "The routine is correct under the Unit 6 interface contract, verified by running the checker over n = 0 to 20 and n = 30, which reported no divergence from the oracle. The discrepancy I found was float output at higher indices, reported as `not-fraction`, resolved by requiring exact Fraction returns."

Accept — table-based route:

> "My table is correct under the first Bernoulli convention, B1 = -1/2, verified by comparing every value against ones I derived from the recurrence by hand, which agreed at every index. The discrepancy I found was at B1, where the generated version gave +1/2; I traced it to the second convention and corrected the specification."

These are of **equal weight**. If a marking scheme, a rubric, or an instructor's instinct treats the second as the weaker submission, that is the scheme failing, not the learner. Both name a specification, a check, an outcome, and a resolution. That is the whole standard.

Reject:

> "The code is correct — the AI is very reliable for this kind of task and the output was detailed and well commented."

Nothing here is evidence. Reliability in general says nothing about this instance; detail and comment quality are properties of the text, not of its correctness. This is the failure the checkpoint is built around.

Reject:

> "All tests passed."

Which tests, showing what, against what specification? A routine that returns Lovelace's indexing passes any test written by someone who also assumed Lovelace's indexing. Passing tests are evidence only in combination with a stated specification — which is why the third criterion exists and why it is the one most often missing.

Query:

> "I checked it thoroughly and I am confident it is right."

The learner's own confidence is also not the evidence. Ask what "thoroughly" consisted of: what they compared, observed, sourced, or reproduced. The answer is often good and simply was not written down.

28.4 Common misconceptions in this unit

"Disclosure is the responsible part; defence is showing off." They are two halves of one obligation. Honest provenance plus an unchecked answer is an honest wrong answer.

"If I disclose, I am covered." Disclosure transfers no accountability. This is `cp00-contract` returning at the end of the module, and it is worth naming the callback out loud.

"If it is correct, disclosure is a formality." It is not. The reader needs to know how the work was made in order to know what to check and how much of your own judgement stands behind it.

"The no-code pack is the lightweight one." Addressed above. Watch for it in peer discussion as well as in marking.

"A passed checkpoint is a defence." The self-check record is a result. A defence is an account of why a result means what you say it means. `selfcheck.py progress --evidence` prints this caveat itself; if a learner has pasted the record in place of item 8, the tool has already told them so.

28.5 The Ada reflection

There is no answer key for this and there should not be.

The unit's position is that whether a language model originates anything is a genuinely contested philosophical question, and it turns on what "originate" is taken to require — intention, understanding, and purpose on one reading; novelty and non-specification on another. Both readings are held by serious people *(Lovelace 1843, Note G is the source of the quotation; the interpretive question is not settled by it)*.

What to accept: any position argued from a stated meaning of the word.

What to challenge, in either direction: a position that treats the question as already closed. "Lovelace settled this in 1843" and "that is obviously obsolete" are the same error wearing opposite clothes — both skip the step where you say what origination would require.

The productive discussion prompt is the one in the activity: which parts of *your own* work did you originate, and how would you show it to somebody else? Learners usually arrive at the specification, the choice of what to verify, and the judgement about what counted as a discrepancy — rather than at any text they wrote. That is the right destination, and it is better reached than asserted.

28.6 The CS extension

Optional, outside the core budget, and not a gate.

The assessable content is step 4, **where the assistance did not help**, and within that the third category: changes that altered behaviour the tests did not cover. A learner who reports only step 3 has written a testimonial rather than an account.

Two things worth watching for:

- **Translation exercises expose test coverage faster than review exercises do.** Porting the tests is usually harder than porting the code, and learners notice this themselves. It is a good moment.
- **Reported defects that are not defects** are as informative as missed ones, and learners under-record them because they feel like non-events. Ask for them specifically.

The intended landing point is step 5: the learner adds tests they did not previously have, because the exercise revealed which parts of their own program were never really being checked. If that happens, the extension has done its job regardless of how the AI performed.

28.7 For instructors

- **The module states no assessment policy.** It says only that the rules for AI use in assessed work are set by the learner's own institution or course. That is deliberate: this module is not the source of such a

policy and must not read as though it were. If your institution has one, supply it alongside the module rather than writing it into the unit text.

- **The evidence pack is private and is not collected.** `course/learner-evidence-template.md` carries the completion criteria and the learner checks their own pack against them. It is not submitted, not marked, and not reviewed by default. Treating it as a coding assessment inverts the design: the pack evidences a verification habit, not programming ability.
- **Equal status between the routes.** If a learner chooses to show you a pack, read table-based and automated packs against the same four defence criteria. The routes differ in medium, not in standard.
- **Item 5 is signposted, not taught.** The privacy paragraph is a boundary reminder. If it starts growing into a privacy lesson during review, that is scope creep away from `course/course-map.md`.
- **The Ada section is the one to protect in editing.** It is the place where a reviewer's own view most easily hardens into the module's position. It should read as unresolved, because it is.

28.8 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.

```
{
  "id": "cp08-disclosure",
  "unit": 8,
  "title": "What a defence has to contain",
  "kind": "multi",
  "question": "You submit work containing AI-generated code. Which of th
  "answer": [
    0,
    1
  ],
  "options": [
    "The specific checks you ran and what they showed.",
    "Which convention or assumption you fixed, and why.",
    "The name and version of the AI tool you used.",
    "That the model stated the answer was correct."
  ],
  "diagnostics": {
    "0,1,2": "The tool name belongs in your *disclosure* -- it says how
    "0,1,2,3": "Two of these are evidence and two are not. A tool name d
    "0": "Test results are the backbone, but they are only meaningful ag
  },
  "explanation": "Disclosure and defence answer different questions: 'how
}
```

29 Unit 9 — Alles Lüge! First- and second-order reasoning

Media for this unit:

- **Interactive Animation**: This unit features an interactive animation (`self-certification.html`). Please view the web version of the course to interact with it.
- **Interactive Animation**: This unit features an interactive animation (`formal-systems-leap.html`). Please view the web version of the course to interact with it.

  The module's closing unit, and the last required one. No new practical work: it names the question the earlier units kept running into, says where that question comes from, and closes off the three overclaims easiest to reach from it. Units 0 to 8 make complete sense without it.

Reading time: about 12 minutes. **Activity:** about 12 minutes. **Course items:** 1, 2, and 4. **Checkpoint:** `cp09-sort-the-claims`. **Before this unit:** Unit 8 (disclosure, defence, and the evidence pack).

No prior study of logic or cybernetics is assumed. Every specialist term needed for the activity is defined below; the optional further reading carries the advanced detail.

29.1 What this unit does, and what it does not do

This unit covers **established, published** material: a distinction from cybernetics, where it came from, and two classical theorems about self-reference. Its job is to **raise the issue and say what "second order" means**. It proves nothing about language models and teaches no unpublished research. That restriction is deliberate: the module has spent three hours training you to ask what supports a claim, and a closing unit that quietly imported an unsettled framework would fail its own test in front of you.

Bracket labels — [C1], [C4], and so on — resolve to `course/references.md`, and `course/second-order.md` gives the fuller treatment this page summarises.

29.2 1. What you have already done

Five things, all done with your hands before you met any of the language here.

- **B4** derived from the recurrence on paper, before you saw any generated code.
- The checker reporting `b1-sign` rather than "wrong".
- Two valid Bernoulli conventions disagreeing at **B1** and nowhere else, because neither program had said which one it used.
- Ada's **B7** and the modern **B7**: one label, two different terms.
- A disclosure and a defence, written separately.

Section 3 says how each of those reads in this unit's vocabulary. Underneath all five is one question: **where is the observer standing?** When you check something, are you outside the system you are checking, or part of it? That question did not begin with artificial intelligence, and it has a substantial published literature going back to the 1940s. The rest of this page is what that literature calls it.

In the plain language used here:

- the **observer** is the person or system making a claim or carrying out a check;
- the **context** is the convention, specification, setting, and limits under which the claim is made; and
- the **record** is the retained evidence of what was observed or checked.

These three details make a claim easier to evaluate. They say who did what, under which conditions, and where the evidence can be inspected.

29.3 2. First order and second order

The distinction comes from cybernetics and is usually put like this.

- **First-order cybernetics** — the cybernetics of *observed* systems. The observer stands outside, describing a system without being part of it.
- **Second-order cybernetics** — the cybernetics of *observing* systems. The observer is included in the system being described.

Heinz von Foerster formulated the distinction in those terms [C1]. Margaret Mead's 1968 address "Cybernetics of Cybernetics" is commonly cited as the point where the reflexive turn was named [C2].

The consequence is small to state and large in effect. An observer inside the system is not taking a free look from nowhere: observing takes time, costs something, and can change what is observed. Claims stop being simply true or false and start carrying an index — who observed, in what context, with what record. Maturana's compact version: *anything said is said by an observer* [C3, p. 8].

A note on precision, since this unit is about exactly that. This line is very often quoted as *everything* said is said by an observer. It is not: page 8 of [C3] reads *anything*. The *everything* form is the title of a later Maturana essay (the 1987 piece in Thompson's *Gaia: A Way of Knowing*; see the note under [C3] in `course/references.md`), and it has propagated by repetition into places that cite the 1980 book for the wording it does not contain.

Two notes on the vocabulary, because both catch people out.

- This is **not** the first-order/second-order distinction in logic. The same words name different contrasts in different fields; this page means the cybernetic one.
- Including the observer makes a claim **more** checkable, not less: an indexed claim says what would have to be compared with what. Naming the Bernoulli convention turned an unresolvable disagreement into a resolved one.

29.3.1 Where it comes from

The **Macy Conferences on Cybernetics** were ten meetings held between 1946 and 1953, funded by the Josiah Macy Jr. Foundation and chaired by Warren McCulloch [C4]. Their mixed disciplines and the later development of the reflexive question are described in the optional further-context section.

Second-order cybernetics is a recognised research tradition with a literature, not a fringe or recent position. It is also a tradition rather than a theorem — a category, not a criticism. A programme can be decades old, published and influential while remaining a way of framing problems rather than something anyone proved. Telling those apart is the exercise at the end of this unit.

Keep two kinds of claim separate. Second-order cybernetics supplies an **interpretive lens**: a published way to frame observer-dependent problems. The logic results in section 4 are **proved theorems** within stated formal conditions. The lens does not follow from the theorems, and the theorems do not prove the module's interpretation.

29.4 3. The mapping, and what it is not

Adapted from `course/second-order.md` §3. The left column is what you did; the right is a reading of it.

A reading, not a proof. None of it is a deduction from the theorems in section 4. Reject the whole second-order framing and you still have a working oracle, a checker, and a correct routine.

What the module does concretely	The second-order reading
The AI cannot tell you whether its Bernoulli routine is right; the oracle can.	The check has to come from somewhere the system being checked does not control. An observing system observing itself is the hard case.
`B1 = -1/2` and `B1 = +1/2` are both correct and they disagree.	The claim was incomplete without its context. Supply the index — which convention — and the disagreement dissolves.
Ada's B7 and the modern B7 are different numbers under the same symbol.	Two observers, two contexts, one symbol. Not an arithmetic error by either.
The oracle is trustworthy because it is exact, external, and independently checked against published values.	Agreement between independent records, rather than access to a view from nowhere.
The checker names `b1-sign` or `ada-indexing` instead of reporting "wrong".	*Which kind* of disagreement this is determines what to do about it.
Note G's error is established as an inverted division in operation 4; only its attribution stays contested.	Saying which part is which is itself part of the claim.
Disclosure says how it was made; defence says why it is right.	Provenance and warrant are different things, and neither substitutes for the other.

One working account of understanding, not a settled definition. On the structural account used in this module, your internal model supports understanding when its parts, relationships, and predictions continue to correspond reliably with the target under relevant checks. The lab gathers evidence for that correspondence through predicted values, property checks, and the first place two records diverge.

That capacity is separate from the target's empirical correctness. You can model a wrong table accurately, as the module's authors did with the as-printed Note G table. In that case you understand what the table says without making the table right. A defence therefore needs evidence both that your model corresponds to its target and that the target is appropriate and correct for the claim you are making.

Other accounts emphasise causes and counterfactuals: understanding includes being able to say what would happen after an intervention, or what would change under a different condition. These are alternatives and useful additions, not positions this short module settles. `course/second-order.md` gives the optional formal structural statement (`f : M1 → M2`) and its limits. The practical rule remains: pause when you can no longer understand and defend what is being produced. In operational terms, pause if you cannot predict relevant behaviour, explain the transformation, identify an independent check, or interpret a failed check. Confidence is not a fifth form of evidence.

29.5 4. The classical results on self-reference

Three plain definitions come first:

- A **formal system** uses explicit symbols, starting assumptions, and rules for deriving one statement from others.
- A **theorem** is a statement established by proof under stated rules and assumptions.
- A **truth predicate** is an expression inside a language that would classify sentences of that language as true or not true.

Two established theorems are the rigorous core of "a system cannot fully account for itself from inside", and their *limits* matter as much as their content.

- **Gödel's incompleteness theorems** (1931): any consistent formal system strong enough for arithmetic contains true statements it cannot prove [C6].

- **Tarski's undefinability theorem** (1936): such a system cannot define a truth predicate for its own sentences [C7].

29.5.1 What they do not say

This subsection matters more than the two lines above it, so read it slowly.

Both are results about **formal systems of a specific kind**: a formal language with a defined satisfaction relation and enough expressive power to talk about its own sentences. They do **not** straightforwardly transfer to language models, to institutions, or to people, and this unit does not claim they do. Their role is to establish that limits on self-reference are real and provable *somewhere* — much weaker, and much safer, than asserting a limit on a particular AI system.

Anyone who tells you that Gödel proves something about machine intelligence is making a leap. The conclusion may still hold; it has not been shown, because the theorem is about a different kind of object, and the step across is an argument someone must make and defend. Noticing that leap — in a paragraph that is fluent, confident, and cites a real theorem — is the skill this module teaches.

29.6 5. What this does *not* license

Three overclaims are easy to reach from here. This unit closes each one off.

29.6.1 "Nothing can be known" — overclaim

The move is from a guarantee to a method: state your context, keep records, compare independent ones. The module is its own counterexample. You finished Unit 5 with evidence that your routine agreed with the oracle over the tested indices and satisfied the checked properties under a stated convention. That scoped conclusion is stronger than felt certainty because it states its limits. The module itself was made by that same method, not exempted from it: generated with AI assistance and then held to external checks before it reached you, with its AI-use disclosure recorded once, at module level, in the repository, as the honest record of how — which is the difference between trusting it and taking it on faith.

"Und doch ich hör' nur Lügen" — *and yet I hear only lies* — (Lacrimosa, "Alles Lüge") [Q1]. That is the position this section refuses. Total distrust is not scepticism but a way of stopping checking: someone who doubts everything cannot decide where to spend the effort.

29.6.2 "Therefore AI cannot be trusted at all" — overclaim

The observer question applies with equal force to human reasoners, to the oracle, to peer review, and to the published Note G table that was wrong for a long time. Nothing here establishes how often a language model is right or wrong. The response is architectural, not attitudinal: put the check outside the thing being checked, index the claim, say what kind of support you have.

29.6.3 "Gödel and Tarski prove that AI cannot verify itself" — overclaim

They do not; see section 4. This is the most interesting of the three, because both halves are true and the join is not: real theorems, real systems, and an unstated argument between them. That shape reads extremely well, and it is exactly what Unit 4 trained you to catch.

29.7 6. Further context (optional)

Nothing in this section is required for the activity or checkpoint.

29.7.1 People and disciplines around the Macy Conferences

Participants came from mathematics, engineering, neurophysiology, anthropology, and psychiatry. They included Norbert Wiener, John von Neumann, Claude Shannon, Gregory Bateson, Margaret Mead, and Heinz von Foerster [C4]. The reflexive question was developed afterwards by von Foerster and the Biological Computer Laboratory [C1], Bateson [C5], and Maturana and Varela [C3].

For the complete bibliography, use `course/references.md`: [C1] to [C5] cover second-order cybernetics and its history; [C6] and [C7] cover Gödel and Tarski. The fuller conceptual treatment is in `course/second-order.md`.

29.7.2 The register, and the rule that governs it

This is the only unit that uses the module's gothic register at full strength, including two short song quotations. It is a deliberate choice with a justification, not decoration: Ada Lovelace was Byron's daughter [G1], and the 1816 Villa Diodati gathering is where the ghost-story challenge associated with *Frankenstein* [G3] and Polidori's *The Vampyre* [G4] is conventionally dated [G2]. The connection is thematic; no claim is made that this literature influenced Lovelace's work.

Notice what those citations carry. [G2] is honest that the nearest thing to a primary account is Mary Shelley's own introduction to the 1831 edition, and it still records the date of the ghost-story challenge as contested — a range, not a night. So: connection supported, summer conventionally dated rather than documented to the day, influence claim not made.

"It's not the red of which we bleed" (Einstürzende Neubauten, "Sabrina") [Q2] — a note about warrant. Nothing here descends from above. The red accent in these materials always means divergence and never travels without a word beside it; warrant is assembled the same way, locally, from records you can point at, by someone who can be named. Both quotations are short, attributed and used for commentary; their entries and the rights position governing their survival to release are in `course/references.md` [Q1], [Q2].

The one rule. The register may carry the argument and may never substitute for it. Every claim on an atmospheric slide still carries its citation or its uncertainty label. If a sentence in this unit sounds good and cites nothing, it is a defect, and you are entitled to say so.

29.8 7. What you can now do

- Say what the first-order/second-order distinction is, and where it came from.
- Name three things you did in this module that the second-order reading illuminates, while saying plainly that illumination is not proof.
- State Gödel's and Tarski's results within their actual scope, and say why neither settles anything about a language model on its own.
- Tell a proved theorem from an established research programme from an overclaim, when all three appear on the same well-written page.

29.9 Checkpoint

Work through `activity.md` first: the checkpoint asks you to sort claims made *by this unit*, and it is a different exercise once you have sorted some yourself. It is the last thing the module asks of you. From `course/lab/`:

```
python3 selfcheck.py run cp09-sort-the-claims     # or use the browser s
```

Both routes ask the same question and give the same diagnostics.

29.10 Where to look next

- `course/second-order.md` — the fuller treatment this unit summarises.
- `course/references.md` — full entries for [C1] to [C7], the gothic-context entries, the two song quotations, and the source policy.
- `course/course-map.md` — where this module stops and the wider course continues.

30 Unit 9 activity — sort the claims, then sort your own

Time: about 12 minutes. **Checkpoint:** `cp09-sort-the-claims`. **You will need:** the student page for this unit, and your evidence pack from Unit 8.

This is the module's closing activity. Two tasks and a checkpoint. Task 1 takes about five minutes. Task 2 takes about six and is the one that transfers furthest, so protect it if you are short of time.

No prior logic or cybernetics is assumed. Use the unit reading's definitions of observer, context, record, formal system, theorem, and truth predicate; the task tests distinctions made there rather than background knowledge.

30.1 Task 1 — sort claims by what supports them

Below are six statements. **Choose three to analyse, and include C and D.** The others are available for discussion or extension. Every statement either appears in this unit or is a short step from something that does.

Put each statement into one of three buckets.

bucket	test
proved theorem	A published result with a date and a standard citation. Name it.
established research programme	Published, influential, decades old — and still not a theorem. Name the tradition or the researchers.
overclaim	It does not follow. Name what it contradicts, or name the step that is missing.

For each of your three, write **one sentence** saying why, and **one line** saying what would change your placement: for a theorem, what would unsettle it; for a research programme, what would settle it either way; for an overclaim, which specific piece of your own work or which missing step refutes it.

One further instruction, and it is not a hint. **If a statement does not fit any of the three buckets, say so, and name the instrument that does fit.** A historical claim, for instance, is sorted as *established*, *contested* or *unsourced* — the scheme the optional Unit 7 works through, and one you can apply here without having done it. Being handed the wrong scheme and using it anyway is a real failure mode, and refusing it is a correct answer.

Do not look the statements up before sorting them. Sorting a claim you have not resolved is the actual skill, because in practice most claims are used, or not used, in whatever state you found them.

30.1.1 The statements

A. Any consistent formal system strong enough to express arithmetic contains true statements that it cannot prove.

B. Mind is better described as a property of a system of relationships than as something belonging to an isolated individual.

C. The Macy Conferences on Cybernetics were ten meetings held between 1946 and 1953, funded by the Josiah Macy Jr. Foundation and chaired by Warren McCulloch.

D. Gödel's incompleteness theorem shows that a language model cannot verify its own output.

E. Because an observing system is part of what it observes, a report an AI system produces about its own behaviour carries no information at all.

F. Gödel's and Tarski's results concern formal systems of a specific kind, so carrying them across to people or institutions requires a separate argument.

30.1.2 Write it up like this

```
Statement:          <letter>
Bucket:             proved theorem | established research programme | over
Citation / whose / what it contradicts:   <one line>
Why:                <one sentence>
What would change it:   <one line>
```

Three of these are harder than the rest, and they are hard in three different directions. If nothing gave you trouble, you have probably sorted by how the sentences sound rather than by what supports them, which is the failure mode this whole module is built to prevent.

30.2 Task 2 — now sort a claim of your own

Open the evidence pack you assembled in Unit 8. You are going to turn the same method on your own work, which is the only version of it that costs you anything.

30.2.1 Step 1 — pick the claim

Take **item 8**, your two-sentence correctness defence. If your pack has several claims, take the one you would most mind being wrong about.

Copy it out as it stands. Do not improve it first.

30.2.2 Step 2 — name its index

A claim made by an observer inside the system carries an index. For your own sentence, write down which of these it **states**, and which it merely **assumes**.

index	for your defence, this means
observer	who ran the check, and who is accountable for the result
context	which convention or specification the result is correct *under*

index	for your defence, this means
record	what was written down, and where another person can find it now

Most defences state one or two of these and assume the rest. That is normal, and it is worth knowing precisely which one you left implicit, because that is the one that will be read differently by someone who is not you.

30.2.3 Step 3 — sort the clauses

Do **not** force your own result into Task 1's buckets. A test result is neither a published theorem nor a research programme, and reusing the wrong instrument here is the same error Task 1 asks you to spot. Use the instrument that fits this kind of claim:

- **Record-supported result** — the clause says what check ran, under which specification and range, and where another person can inspect or repeat it.
- **Reasoned judgement** — the clause interprets the record, or makes a choice that the evidence does not settle. Name whose judgement it is.
- **Overclaim** — the clause claims more than the evidence supports. The usual form is a check that passed over one range being reported as correctness for all possible inputs.

A defence may contain all three. Mark the clauses rather than assigning the whole paragraph one flattering label. Then write one line: **what evidence would strengthen the claim, or what qualification would make its scope accurate?**

30.2.4 Step 4 — one honest sentence

Finish with this, in your pack:

> The part of my defence that rests on a record is …, and the part that rests on my judgement is …

If those two turn out to be the same thing, you have found something worth knowing.

30.3 Task 3 — the checkpoint

Do this after Tasks 1 and 2. The checkpoint asks you to sort claims made *by this unit*, and it is a different exercise once you have sorted three statements and one of your own.

From `course/lab/`:

```
python3 selfcheck.py run cp09-sort-the-claims     # or use the browser s
```

Both routes ask the same question, accept the same answers, and give the same diagnostics. Neither is a fallback for the other. Pick whichever you have to hand.

30.4 Optional extension — where the question came from

Optional. Outside the core time budget. It does not gate anything.

Spend five minutes finding out one thing about the Macy Conferences that this unit did not tell you: a participant it does not name, a topic that recurred across the ten meetings, or a disagreement that was never resolved. Write two or three sentences on what you found and where you found it.

Then answer one question in one line: **is the source you used a primary record of the meetings, or someone's later account of them?** Both are useful and they are not the same thing, and saying which one you have is the habit this module has been building since Unit 4.

30.5 Before you finish

You should be leaving this unit with:

- three sorted statements, including C and D, each with a citation, an attribution, or a named missing step
- one defence of your own, split into record-supported result, judgement, and any overclaim, with its three index entries marked stated or assumed
- one line saying what evidence or qualification it needs
- `cp09-sort-the-claims` passed

Add the whole thing to your evidence pack as an appendix. It is the shortest demonstration you will produce that you can tell a theorem from a research programme from a sentence that merely sounds settled.

That is the module. Wider course Items 3, 5, and 7 go further than this material does; `course/course-map.md` shows where.

30.5.1 Activity Answers

31 Unit 9 answers and guidance

For learners checking their own work after attempting the activity, and for instructors marking or discussing it.

This key does not restate the answer to `cp09-sort-the-claims`, and it does not paraphrase it either. Sorting claims by what supports them is the learning objective, and the checkpoint gives targeted diagnostics if you go astray. Checkpoint answers live in `course/lab/checkpoints.py` and nowhere else, by design. The six statements discussed below are deliberately disjoint from the three the checkpoint uses: no statement here is the checkpoint's, and no statement here is a restatement of one of the checkpoint's.

Scope. This unit teaches established, published material only. There is no research framework in it to mark, and answers that reach for one have gone outside the unit.

Citation shorthand. Bracket labels resolve to `course/references.md`: *[C1]* von Foerster, *[C2]* Mead 1968, *[C3]* Maturana and Varela, *[C4]* the Macy Conferences, *[C5]* Bateson, *[C6]* Gödel 1931, *[C7]* Tarski 1936; *[G1]* to *[G4]* support the register; *[Q1]* and *[Q2]* are the songs. Fuller treatment of this unit's argument: `course/second-order.md`.

The assessable part is the reasoning, not the label. A learner who places a statement differently but names a real citation, a real tradition, or a real missing step has done the exercise. A learner who lands on the intended bucket and writes "it sounds like a theorem" has not.

31.1 Task 1 — the six statements

31.1.1 A. Incompleteness

Proved theorem. Gödel 1931 [C6]. Any consistent formal system strong enough to express arithmetic contains true statements it cannot prove.

The tell is that it has a date, a name, and a place in every logic textbook. A learner who marks it "established research programme" because it is unfamiliar has confused *unfamiliar* with *unsettled*, and that confusion is worth naming out loud — it is the mirror image of the error waiting in statement B.

What would unsettle it: an error in the proof, or a change to the hypotheses. Note that the statement is conditional — consistent, sufficiently strong — and almost every misuse of it in the wild starts by dropping one of those conditions.

31.1.2 B. Mind as a property of a system of relationships

Established research programme. Bateson [C5], and the wider tradition [C1], [C3]. Published, developed over decades, widely built on, and still an account rather than a theorem.

The tell is the phrase "is better described as". The statement is not reporting a result; it is proposing that a particular way of describing something is the right one. Descriptions of that shape are argued for, adopted, extended, and sometimes displaced. They are not looked up and settled.

Say plainly in discussion that this category is **not** a polite way of saying "wrong" or "weak". It is a statement about what kind of support the claim has.

What would settle it either way: sustained independent development that makes the account do work nobody could do without it, or a demonstration that it fails on the cases it was built for. Note that "better described as" invites the question *better for what purpose*, and a learner who asks it has found the seam.

31.1.3 C. The Macy Conferences

None of the three buckets fits, and that is the point. This is a **historical claim**, and historical claims are sorted as established, contested, or unsourced — the instrument the optional Unit 7 works through, and named in the activity itself so a learner who skipped Unit 7 can still reach for it.

Sorted that way it is **established** — `course/references.md` [C4] records the series, the dates, the funder and the chair.

The best answers do two things: notice that the bucket scheme does not fit, and reach for the other instrument instead. A learner who does only the first has still done something worth marking, because recognising that you are holding the wrong instrument is most of the skill. Do not treat having done Unit 7 as a requirement here.

Watch for a learner who marks it "established research programme" because cybernetics is a research programme. The statement is not about the programme; it is about ten meetings, their dates, their funder and their chair. Those are facts with a record, not a way of framing problems.

31.1.4 D. "Gödel shows a language model cannot verify its own output"

Overclaim, and the most interesting one on the list, because both halves of it are true and the join is not.

Gödel 1931 [C6] is a proved theorem. It is a result about formal systems of a specific kind. A language model is not that object. Carrying the result across is an **argument**, requiring premises about what corresponds to what, and no such argument appears in the sentence.

The failure here is not falsity. The conclusion might even be defensible on other grounds. The failure is a theorem being used as a licence for a claim about a different kind of system, with the transfer step unstated. That move is common, it reads extremely well, and it is precisely the thing this module has been training you to catch.

Accept a learner who says "the theorem is proved, the transfer is not, and the sentence hides the seam" over one who writes "overclaim" with nothing behind it.

31.1.5 E. "An AI system's report about itself carries no information at all"

Overclaim. This is the second overclaim from the student page wearing practical clothing, pushed one step further.

Two separate faults. First, it does not follow: that an observer is inside the system says nothing about whether its reports correlate with anything. Second, it is operationally empty — if a self-report carries nothing, you have no way to decide which outputs deserve a check, which in practice means you check nothing, which is exactly where blanket trust also lands you.

What refutes it from your own work: Unit 5. Generated output that was correct, generated output that was wrong, and a check that told you which was which. You were not applying a presumption; you were comparing against a record.

Note for discussion: describe behaviour, not intent. "The system reported X" is the right form. "The system believed X" is not, and a learner who slips into the second has drifted from the module's language rules.

31.1.6 F. Scope of Gödel and Tarski

Proved theorem, in the sense that matters here: this is a statement about what [C6] and [C7] actually say, and you can check it by reading the theorems' hypotheses. Accept "established" with that reasoning.

The generous alternative reading is that F is a methodological remark rather than a result. A learner who says so, and then says the remark is checkable against the theorem statements, has understood it better than the bucket alone shows.

What must not happen is a learner marking F an overclaim. F is the caution, not the leap. If someone has read the caution as itself an overreach, they have inverted the section they most needed.

31.2 Task 2 — a claim of your own

31.2.1 Step 2, the index

Typical results, and none of them is a failure:

- **Observer** — almost always implicit. The pack has one author and it feels redundant to name them. It is not redundant to a reader who has three packs on their desk.
- **Context** — usually stated, because Unit 8's template forces it. This is the one the module worked hardest on, and it shows.
- **Record** — often half-stated: "the checker reported no divergence" names the check but not where the output now lives, or whether it can be produced again.

A good answer marks at least one of the three as assumed. A learner who marks all three as stated has almost certainly read their own sentence generously, which is the one reading a sceptical reader will not give it.

31.2.2 Step 3, the clauses

A well-built pack usually contains a **record-supported result**: there is a record, the reader can consult or reproduce it, and the sentence states the range, convention, and properties actually checked. Do not call this a "proved theorem" or an "established research programme". Reusing Task 1's instrument on this kind of claim is the same error statement C is there to provoke.

The instructive answer is when a learner finds an **overclaim** in their own writing. The common form is a scope slip: a check that ran over a specific range, under a specific convention, reported as though it established correctness in general. That is not dishonesty and it does not need punishing. It is the same structure as the Note G table — a narrow supported claim and a broad readable one, and the broad one is easier to write.

Reasoned judgement is usually present beside the record: which discrepancy mattered, which convention was the sensible default, and what counted as resolved. It should be attributed to the learner by name, not dressed up as something more.

31.2.3 Step 4, the honest sentence

The intended discovery is that the record-backed part of a defence is usually smaller than it felt while writing it, and that the remainder is not worthless — it is judgement, which is exactly the thing the module says must be attributable to a person. A learner who concludes "so my defence is weak" has misread the exercise. The point is to be able to say which part is which.

31.3 Common ways this activity goes wrong

Sorting by how the sentence sounds. Statements B, D and F are all written in the register of established results. That is not a trick; it is what most writing about this material looks like, including writing produced by a language model. The sorting has to be done on provenance, not on prose.

Treating "established research programme" as a downgrade. It is not. A programme can be decades old, published, influential and worth taking seriously while still not being a theorem. The unit says so explicitly and the checkpoint tests it.

Reading the unit as a case against AI systems. If a learner leaves saying "so it cannot be trusted", the second overclaim has not been closed properly. The materials establish no base rate for correct output. They teach learners to match the strength and independence of a check to the claim and its risk.

Collapsing the whole thing into "everything is relative". The index is the opposite of that. An indexed claim is *more* checkable than an unindexed one, because it says what would have to be compared with what. The Bernoulli convention is the proof: naming it turns an unresolvable disagreement into a resolved one.

31.4 For instructors

- **This unit is the final core unit, and the earlier ones must still stand without it.** Nothing in Units 0 to 8 may be rewritten to depend on it for a checkpoint, an activity, or a definition; Units 2 and 5 to 8 carry a short pointer only. Being last in the sequence is not a licence to make it load-bearing for anything that comes before it.
- **The unit raises the issue; it does not settle it.** The scope is fixed to established, published material. If the delivery starts extending the argument into new territory — however interesting — it has left the unit.
- **Section 4's second half is the unit.** "What they do not say" is the teaching point, not a caveat attached to one. If a session runs short, cut a mapping-table row, never that subsection.
- **The most valuable discussion prompt is statement D.** It generalises: a real theorem, a real system, and an unstated transfer between them is the shape of a large fraction of confident writing about AI and formal limits.
- **Register discipline.** Every claim on an atmospheric slide keeps its citation or its uncertainty label. If a slide review produces a frame that is only atmosphere, cut the frame rather than the citation.
- **Pending work this unit depends on**, none of which belongs in this directory: full entries and resolution of any open markers for [C1] to [C7]; the open markers already recorded against [G1] to [G4] and [Q1] to [Q2]; and confirmation of the rights position on the two song quotations, which must be recorded in `docs/instructor-review-log.md` before release.
- **Glossary.** The terms this unit now needs are *first-order cybernetics*, *second-order cybernetics*, *observer*, and *Macy Conferences*. Terms belonging to the framework the unit no longer teaches should not be maintained on its account.

- If the status of any claim in `course/second-order.md` changes, that file is updated first and this unit follows. This unit holds no independent position that could drift out of step with it.

31.5 Checkpoint data

The self-check questions for this unit, as structured data. `course/lab/checkpoints.py` and the browser self-check read these blocks directly, so the questions have one source and cannot drift. Editing a block changes both routes.


```
{
  "id": "cp09-sort-the-claims",
  "unit": 9,
  "title": "Sort the claims",
  "kind": "match",
  "question": "Unit 9 makes claims of three different kinds, and its own
  "answer": [
    0,
    1,
    2
  ],
  "options": [
    "A proved theorem, with a date and a standard citation.",
    "An established research programme: published, influential, decades o
    "An overclaim: it does not follow, and your own lab work contradicts
  ],
  "prompts": [
    "No sufficiently expressive formal system can define a truth predicate
    "Descriptions should include the observer, because an observer insid
    "Because no system can fully account for itself, nothing can really
  ],
  "diagnostics": {
    "1,0,2": "You have the first two swapped. Tarski's undefinability theo
    "0,2,1": "The third claim is the one that does not follow. In Unit 5
    "2,1,0": "Reversed. Ask which of these you could look up in a logic
  },
  "hint": "One has a date and a name attached to it in any logic textboo
  "explanation": "This is the module's last exercise and it turns the me
}
```

32 Learning paths

This module is a fixed sequence of units, but you do not have to take all of it, and you do not have to take it the same way as everybody else. This page sets out the routes through it. For each one it says who it suits, which units you take and in what order, how long the material declares it will take, what you will be able to do at the end, and what you will not.

The last of those matters most. The useful question is not how many units you covered. It is what you can do afterwards that you could not do before, and where the edge of that is. Every route below is described in those terms.

All timings come from the `duration_minutes` block in each unit's `metadata.yml`, added up. Reading is measured at 180–220 words per minute with inspection allowances for code, tables, equations, and media; activities use a dry run of the required tasks. Your own pace will differ, so treat these as a planning range rather than a promise.

32.1 The units and what each costs

Unit	Title	Minutes
0	Human introduction and how to use this module	23
1	Ada's ordered procedure	25
2	What a language model does differently	31
3	Meet the Bernoulli numbers	37
4	Ask AI to generate the routine	27
5	Verification lab	46
6	Re-prompting, specification, and overreliance	35
7	The historical bug and historical honesty *(optional)*	29
8	Disclosure, defence, and responsible use	32
9	Alles Lüge! ('it's all lies!') First- and second-order reasoning	30

The core path is Units 0 to 6, then 8 and 9. Its declared central estimate is **286 minutes**, about four hours and forty-six minutes; the measured reading range makes the route approximately 281–300 minutes. Unit 9 is not an appendix: it is the closing unit and it is required. Unit 7 is optional, sits outside that budget, and adds 29 minutes if you take it. Titles and course-item numbers come from `course-map.md`, which is canonical; the unit table with checkpoints and status is in `index.md`.

The 2.5–4 hour planning target is advisory. The measured route exceeds it because the required readings need more time than their early estimates allowed, Unit 6 includes transfer to claims with no automatic oracle, and Units 3 and 5 retain the derivation and verification practice that make the module more than a set of claims about AI. Optional extensions and Unit 7 remain outside this total.

The units have prerequisites, listed in each `metadata.yml`, and they are real rather than decorative. Unit 5 cannot check an artefact you did not generate in Unit 4. Unit 4 has nothing to compare against unless you derived the reference in Unit 3. Unit 3's traps only land if Unit 1 has shown you what an exactly specified procedure looks like. This is why the routes below differ in where they *stop* rather than in which units they skip out of the middle. Unit 7 is the one unit that lifts out cleanly: it depends on Units 3 and 5, and nothing depends on it, which is exactly what makes it safe to leave out.

32.2 Route 1 — The short route

Units 0, 1, 2, 3, 4, 5, in that order. 189 minutes, about three hours and nine minutes.

Who it suits. You want the module's actual skill and you cannot find three and five hours. You are willing to give up the parts about writing up and defending the work in order to keep the part where you check something.

This is the shortest route that still teaches something defensible, and it is worth saying plainly that it is not very short. The saving over the core route is about an hour and a half. There is no thirty-minute or ninety-minute version of this module, and offering one would be dishonest, because the module's point is not a fact you can be told. It is a thing you do: you ask an AI system for a computation, you build or obtain a reference independently, you compare them index by index, and you find where they part. That sequence needs a generated artefact, a trusted reference, and a comparison. Remove any of the three and there is nothing left to verify. Units 0 to 5 are the smallest set that gives you all three.

What you will be able to do afterwards.

- Say what changes when output is produced by next-token continuation rather than by an ordered procedure, and why that changes how you check it.
- Triage an AI output in seconds into what a test could settle and what only a source could settle.
- Derive `B4` by hand from the recurrence and show your working, so that you hold a reference value you did not get from the thing you are checking.
- Explain why two correct-looking programs disagree about `B1`, and say which convention this module uses.
- Take a generated routine or a generated table of values, compare it against the reference index by index, find the first index where it diverges, and name the failure mode using the module's codes rather than saying "it is wrong".

What you will miss.

- Unit 6: turning a failure you observed into a specification clause that prevents it next time. You will be able to find bugs but not to systematically stop producing them.
- Unit 8: the disclosure and the defence. You will have checked your work and you will not have practised saying, in writing, how it was made and why it is right. Whether any course asks you for it is up to your own institution — this module sets no assessment rules — but it is the part most worth being able to produce.
- Unit 9: the closing unit, which offers one published way of reading everything you did — an interpretive lens, not the reason the verification works, which the oracle and the independent references already supply — and asks you to sort three of its own claims by what actually supports them. Without it you have the technique and not the reason it works, and the reason is the part that travels to problems that have nothing to do with Bernoulli numbers.

Unit 7 is absent from that list, but it is not something this route gives up: it is optional for everybody, and the core route does not include it either. See "Optional material" below.

32.3 Route 2 — The core route

Units 0 to 6, then 8 and 9, in that order. 286 minutes, about four hours and forty-six minutes. This is the default.

Who it suits. Everybody, unless you have a specific reason to be on another route. It assumes no computer-science background and no programming. It is the route the module is designed around and the one the time budget is set for.

Spread it over as many sittings as you like. Natural breaks fall after Unit 3 (you now hold the reference), after Unit 5 (you have found the divergence), and after Unit 6. Units 8 and 9 come to 62 minutes together and

make a good final sitting: one writes the work up, the other says why the whole procedure was sound.

Unit 7 is not in this list, and its absence is a decision rather than an oversight. The module ran long, and the lesson Unit 7 works through at length — that published does not mean true — is also recorded as expected error 12 in `misconceptions.md`, so the core path carries it either way. Skipping Unit 7 leaves no hole. Taking it is a good use of 29 minutes once you have the core.

What you will be able to do afterwards. Everything in the short route, and:

- Take a failure you have actually observed and write it as a specification clause, which makes the same failure easier to catch or rule out next time — though a generative system offers no guarantee it will never recur.
- Say why asking the same system to check its own answer is not independent verification.
- Write two different things that are commonly confused: a **disclosure**, which reports how the work was made, and a **defence**, which says why it is right. A defence that cites the tool rather than the check is not a defence, and you will be able to see that in your own writing.
- Hold a complete evidence pack: prompt, output, what you checked it against, by what method, what diverged, what you changed, and why the final answer stands.
- State the general argument behind the whole module in one sentence, and point at three things you actually did that are instances of it.
- Tell a first-order treatment of a problem from a second-order one, and say which indices a claim is carrying — who observed it, in what context, with what record.
- Distinguish a proved theorem from an established research programme that nobody proved, and both from an overclaim, when all three appear on the same well-written page. That is the reading skill that makes the rest of it usable outside this module, and Unit 9 asks you to apply it to Unit 9.

What you will miss. The optional Unit 7, the optional CS extensions, and the per-unit optional extensions at the end of the Unit 1, 2 and 4 activities. None of it gates anything. A learner who finishes Unit 9 has completed the module, and the module says so in its own canonical map.

32.4 Route 3 — The core route plus CS extensions

Units 0 to 6, 8 and 9, plus the optional Python work. 286 minutes of declared core time, plus about 30 minutes of extensions that carry an estimate and three that do not.

Who it suits. You write code, or you are on a computing programme and want the version of this that engages with testing rather than only with judgement. The module's focus is computer science, and this is the route that shows it. It is recommended if you program, and it is not required of anybody.

Take the core route as written, and add the extensions at the points they appear:

Where	What	Timed?
Unit 1, optional extensions	Rewrite your procedure as a variable table in the style of Note G: name each object, and for each step record the operation, the variables read, and the variable written.	About 15 minutes, declared in the activity.
Unit 3, Task 4	Write a function that reports the *first* disagreeing index, not every one.	About 5 minutes, declared in the activity.

Where	What	Timed?
Unit 4, optional extensions	Read a specimen without running it and say which algorithm it uses, whether it would be exact under `Fraction`, and where it would be slow rather than wrong. Then write the smallest test that separates the correct example from the four wrong ones.	About 10 minutes, declared in the activity.
Unit 5, optional CS extensions	`exercises/ex05_write_a_test.py` — write a single test that separates four faulty implementations from one correct one, without rejecting the correct one. It grades itself against the real files.	Not declared.
Unit 5, optional CS extensions	`exercises/ex06_ada_translation.py` — implement the mapping from Lovelace's Note G labels to modern indices, returning exact values and rejecting labels that do not exist. It looks ahead to Unit 7; do it now, or after Unit 7 if you take that unit.	Not declared.
Unit 8, CS extension	Take a routine of your own, ask an AI system to translate, review, or debug it, and produce a test-based account of where the assistance helped and where it did not.	Not declared.

The extensions sit outside the core time budget by design. Three of the six now carry a declared estimate, about 30 minutes between them; the other three do not, so no honest total is available for this route. The declared figure is 286 minutes for the core; plan a further sitting for the extensions rather than trying to fit them into the same one.

What you will be able to do afterwards. Everything in the core route, and:

- Write a test rather than run one. You will find out, concretely, that a test checking `B4 == -1/30` lets two of the four faulty implementations straight through, and you will have to do better than that.
- Distinguish an inexact *representation* from an incorrect *method* when reading someone else's numerical code, because those fail differently and are fixed differently.
- Convert between two indexing schemes correctly, including rejecting the indices that have no counterpart at all — which is where off-by-one errors actually live.
- Say what your existing test suite covers, run it against AI-assisted changes, and produce evidence for where the assistance helped and where it did not. Changes that altered behaviour your tests did not cover are the most informative category, because they tell you where your suite is thin.

What you will miss. Only the optional Unit 7, and the optional extensions in Units 1, 2 and 4 that are not about code. Note also what the extensions do not add: they do not teach a different verification idea from the core route. They give you a second way to practise the same one, with the machine doing the comparison.

32.5 Route 4 — The complete route

Units 0 to 9, including the optional Unit 7, plus the CS extensions if you want them. 315 minutes of declared time, about five hours and fifteen minutes.

Who it suits. You have the time, and you want the historical case worked through properly rather than summarised.

The one thing this route adds to the core is Unit 7, and it is worth being precise about why Unit 7 sits outside the core, because "optional" is easy to misread as "filler". Unit 7 is not lighter than the rest and it is not remedial. It is the unit that works hardest on sources: it cites a primary text, it states plainly which part of the standard account is established and which is contested, and it refuses to settle in passing something the record does not settle. That refusal is the module practising in front of you what it asks of you everywhere else. It was made optional because the first full assembly ran well over the target and the module needed to be shorter, and because its transferable lesson is recorded in `misconceptions.md` entry 12 so the core path does not lose it. Length was the reason. Quality was not.

What you will be able to do afterwards. Everything in the core route, and:

- Establish a value independently *first* and only then compare it against a famous printed table, so that a primary source becomes something you can check rather than something you defer to.
- Sort a claim into established, contested, or unsourced, and say what specific evidence would move it between those buckets. For the Note G error those movers are concrete: a facsimile or physical copy of the 1843 printing, a comparison across print runs, surviving manuscript material.
- Spot the misfiling that most bad claims start with — something contested or unsourced quietly recorded as established, and thereafter repeated with the confidence that only the first bucket earns. Repetition is not corroboration: ten sources that all trace back to one unchecked account are one source.
- Write at three different strengths, so that the confidence of your sentence matches the state of the record rather than the state of your memory.

What you will miss. Nothing this module offers. Note the boundary anyway: this module does not cover fairness, representation, privacy, sustainability, or industry practice properly. It signposts them. The wider course covers them, and `course-map.md` records which items this module leads on and which it only touches.

32.6 Route 5 — The no-code route

Units 0 to 6, 8 and 9, in order, using table-based verification throughout. 286 minutes, the same as the core route.

Who it suits. You do not run Python and do not intend to. This is a complete route to the full set of the module's core outcomes. It is equal in standing to the route that runs code and produces a complete evidence pack. It is not a reduced version of anything.

What changes is the method of comparison, not the presence of it.

- **Unit 5** provides two full routes through the verification lab, Route A and Route B. Route B is the table one. You ask the AI system for the *values* rather than the program, write the two columns side by side in index order, go down the rows, and stop at the first row that differs. That is the first divergence, found by hand, by the same method the checker uses.
- **Checkpoints** use the same browser self-check. Twelve core checkpoints are shared; Unit 5 substitutes `cp05-table-evidence` for the Python-only `cp05-bernoulli` so that each route is assessed against evidence it can actually produce.
- **The evidence pack** has a table-based version worked through in full in `learner-evidence-template.md`.

What you will be able to do afterwards. Everything listed under the core route. The capabilities are the same list, reached by a different comparison method.

What you will miss, stated exactly. The table route and the automated route each miss things, and the module is explicit about both. The table comparison will not tell you whether a program actually produces the table it printed — a table in a chat window may be generated text rather than program output, and those are two different claims. It says nothing about indices you did not compare. It does not re-run itself when something changes. Against that, the automated checker cannot tell you whether the convention it enforces is the one your task requires, says nothing about comments, historical claims, licensing, or readability, and cannot tell you whether you understand the code. Neither route is complete. Both are real verification. If Python is available to you and you are not on this route by preference, doing the table comparison for `B0` to `B12` and running the checker takes about five minutes together and covers more than either alone.

32.7 Optional material, and what optional means here

Optional in this module means: work that nothing else depends on, that no checkpoint requires, and that is worth doing once the core argument has landed. It does not mean easier, lesser, or intended for anybody in particular. The core path is the same for everybody, and the optional material is where the module put what would not fit in it.

There are three kinds, and they are different in scale.

Unit 7 — a whole optional unit, 29 minutes. Described under Route 4 above. Its checkpoint `cp07-note-g` is an optional checkpoint, and it carries an optional extension of its own that runs the same three-bucket test on generated text. It requires Units 3 and 5, and nothing requires it.

Per-unit optional extensions — Units 1, 2 and 4. At the end of each of those activities there is a section headed “Optional extensions”, each task carrying its own estimate. These are not new busywork: they are the tasks that used to be part of the required activity and were moved out when the module was shortened. Nothing was deleted. If an activity felt like it stopped early, this is where the rest of it went.

- **Unit 1** — about 25 minutes across three: repair the remaining gaps in your procedure, run the literal-reader pass on somebody else’s, and (for programmers) rewrite the procedure as a Note G-style variable table.
- **Unit 2** — about 12 minutes across three: name the de-risking condition for every item you marked L, build the full list of things that feel like independent verification and are not, and rewrite your three weakest checks as checks somebody else could run.
- **Unit 4** — about 24 minutes across four: extend the triage table, make the mirror-image prediction, triage a second specimen from `lab/examples/`, and (for programmers) read a specimen without running it.

CS extensions. Route 3, and the table there.

One piece of ordering advice, since all of this competes for the same evening. Take the core first. Optional work is what you reach for once the core argument has landed; reached for before that, it costs you the end of the module and gives you no way to tell what you have gained.

32.8 If you only have 45 minutes

Then you cannot do this module, and it is better to say so than to sell you a route that ends before the point.

What 45 minutes genuinely buys is orientation. Complete **Unit 0** (23 minutes), then use the remaining time to begin Unit 1 and keep your answers separately so you can resume. You will finish the learning contract: using an AI system does not remove your responsibility for how you use and submit its output.

You will not have checked anything. You will not have derived a reference, seen a plausible output be wrong, or found a first divergence. Unit 2 is also written to follow Unit 1, so you will meet references to a contrast you have not been shown.

Do not skip directly to Unit 2 to manufacture a 45-minute mini-route: Unit 2 assumes the ordered-procedure contrast from Unit 1. Starting the short route and resuming later is more coherent than claiming completion before any verification has happened.

32.9 Where you can stop, and where you should not

Some stopping points leave you with something coherent. Others leave you holding the setup without the payoff, which is the worst of both — you spent the time and you did not get the skill.

Good places to stop.

- **After Unit 0.** Honest but thin. You have the contract and know how the module works. Nothing has been verified.
- **After Unit 3.** A defensible small win: you can derive `B4` by hand, you know the two conventions exist and which one is in use, and you can translate between Lovelace's Note G labels and modern indices. You have a reference. You have not used it on anything.
- **After Unit 5.** The best early stopping point in the module, and the end of the short route. You have generated something, checked it against an independent reference, found the first divergence, and named the failure mode. That is the skill the module exists to teach.
- **After Unit 8.** A real stopping point, and the end of the practical arc: you have verified something, specified against the failure, and written both the disclosure and the defence. It is not the end of the module. Unit 9 is 30 minutes and it is where the thing you did gains a general reading you can carry elsewhere — offered as one published position, not a proof.
- **After Unit 9.** You have finished the module.

Bad places to stop.

- **After Unit 4.** The single worst place. You have a generated artefact, a triage of it, and no verification. You have done the part that was already easy and stopped before the part that is the module. Twenty-two more minutes of video and reading gets you into Unit 5.
- **Part-way through Unit 5.** If you have set up the lab, or written out the two columns, and stopped before comparing them, you have paid the setup cost and taken none of the return. Finish the comparison.
- **After Unit 6, without Unit 8.** You can verify and you can specify, and you have not written the disclosure or the defence. Unit 8 is 32 minutes and it is the unit that turns everything before it into something you can hand to somebody else and stand behind.
- **After Unit 7, having taken it in place of Units 8 and 9.** Unit 7 is good, and it is not a substitute for the end of the module. If it is a choice, finish the core and come back to Unit 7 afterwards; nothing in it expires.

32.10 Checkpoints, on every route

Every unit has at least one self-check checkpoint. Twelve core checkpoints and the optional Unit 7 checkpoint are available in both the browser and terminal interfaces. Unit 5 additionally has one Python-only executable checkpoint and one browser-compatible table-evidence checkpoint; a learner completes the one for their route. All definitions still come from one shared source.

```
python3 selfcheck.py run <id>          # or use the browser self-check
```

Whichever route through the module you take, the checkpoints are how you know it worked. They are not decoration at the end of a unit and they are not a quiz. A wrong answer that matches a known misunderstanding tells you which one you have hit and points you at the unit that deals with it, which is why taking them in the unit is worth more than taking them at the end. Some will not show you the answer, because deriving it is the exercise; you get a hint aimed at your next step.

The checkpoint ids for each unit are listed in `index.md` and in `course-map.md`. The terminal route and the lab live in `lab/README.md`.

32.11 Your evidence pack

You accumulate an evidence pack as you go, and Unit 8 assembles it. It is a private formative self-check. It is yours, it is not submitted, and it is not marked. Its only purpose is to let you see, in one place, the distance between “I generated this” and “I understand and can defend this” — and to let you notice when that distance is larger than you assumed.

Because it is not submitted, the incentive to tidy it up runs the wrong way. Paste the prompt as you sent it, including the part you got wrong or left out. The gaps in that prompt are the evidence. The template, with a worked example for each route, is in `learner-evidence-template.md`.

33 Historical Notes

The authoritative reference for every historical claim in this module. Unit materials cite this page rather than restating history in their own words, so that a correction here propagates instead of leaving stale copies behind.

Each claim carries a verification status: **established**, **contested**, or *unsourced*. The labels are defined in `course/references.md`. Reference labels in square brackets — [P2], [S1] — resolve there.

A warning that is also the lesson: this is the part of the module where confident, fluent, wrong text is easiest to produce and hardest to notice. That applies to AI-generated drafts of this page, and it applied to the nineteenth-century printers too.

33.1 1. The people and the machine

Established. Charles Babbage designed the Analytical Engine, a general-purpose mechanical computing machine, from the 1830s onwards. It was never completed in his lifetime. Its architecture separated a "store" (memory) from a "mill" (processing), and it was to be programmed with punched cards adapted from the Jacquard loom. [S1], [S4]

Established. In 1840 Babbage lectured on the Engine in Turin. Luigi Federico Menabrea wrote up an account in French, published in 1842. [P1]

Established. Augusta Ada King, Countess of Lovelace, translated Menabrea's article into English and appended seven Notes, A to G, of her own. The translation with Notes appeared in 1843 in Taylor's *Scientific Memoirs*, volume 3. Her Notes run to roughly three times the length of the article she was translating (a commonly repeated characterisation of the 1843 text, not a figure the text itself states). [P2]

Established. Note G contains a table setting out, operation by operation, how the Engine would compute a Bernoulli number. [P2]

33.2 2. "The first program"

Contested — and the contest is the teaching point.

The claim "Note G contains the first computer program" is useful shorthand and a poor historical statement. What is actually disputed, and by whom:

- **What Note G contains** is not in dispute: an explicit, ordered, operation-by- operation specification of a computation, including working variables and a repeated group of operations, intended for a general-purpose machine. [P2]
- **Whether that constitutes "the first program"** depends on a definition nobody agreed in advance. Babbage had written sequences of operations for the Engine earlier, in unpublished notebooks. [S1], [S4]
- **How much of the Notes was Lovelace's own work** is genuinely disputed. Stein [S2] argues for a substantially smaller contribution than the popular account; Toole [S3] argues for a larger one; Hollings, Martin and Rice [S5] work through the surviving correspondence in detail. Babbage's own account [P3] is a primary source written by an interested party decades later.

How the module handles this. Unit 1 states only the checkable core: Note G was published in 1843 and contains an ordered operation table intended for the Analytical Engine. It reports "first program" as a common, contested retrospective description rather than claiming priority. It does not adjudicate the authorship question or pretend the question is settled in either direction.

A claim being contested does not make it unknowable or make all positions equal. It means the honest statement is “sources disagree, here is the disagreement”, which is a different skill from repeating whichever version you met first.

33.3 3. The indexing convention in Note G

Established, and verified computationally.

Lovelace numbers only the Bernoulli numbers that are not zero. The odd-index Bernoulli numbers above the first are all zero, and in her scheme they simply do not receive labels. Her label numbers are therefore one smaller than the modern indices — her Bk is the modern Bk+1:

Lovelace’s label	modern index	value
B	B	1/6
B	B	-1/30
B	B	1/42
B	B	-1/30

Note G’s worked example computes her B , which is the modern B = -1/30.

This mapping is implemented and tested in `course/lab/reference_bernoulli.py` (`ada_to_modern_index`, `ADA_NOTE_G_INDEX_MAP`) and covered by `tests/test_reference_be`

Why the module makes so much of this. It is a specification hazard of exactly the kind that still causes defects: two systems, the same symbol, different meanings, and no error message when they meet. A learner who asks an AI to “reproduce Ada’s Bernoulli calculation” and compares the result against a modern table will see a mismatch that is nobody’s arithmetic error. The checker reports it as `ada-indexing`.

33.4 4. The error in the published table

Established: the table printed in Note G contains an error, and it is in **operation 4**.

Established (facsimile [P2a], accessed 2026-07-23): operation 4 acts on the variables ^{2}V and ^{2}V, in that order, and its stated result is $(2n-1)/(2n+1)$. Since V holds $2n-1$ and V holds $2n+1$, producing that result requires dividing V by V. The table prints the operands the other way round. See the facsimile recorded at [P2a]; the diagram is headed “Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G (page 722 *et seq.*)”, which is also the source for the page number now given in [P2].

Established (facsimile plus correction of record): the correct ordering is $V \div V$; the published $V \div V$ inverts the division. The location and the inversion are established by the facsimile [P2a] and the archival correction of record [S8], and corroborated by [S7], a detailed independent reconstruction, self-published. The corrected table computes $-1/30$ — Ada’s B , the modern B .

Established (recomputed in this module): what the *as-printed* table actually computes is $139/630$, not the $-25621/630$ that encyclopaedia and blog accounts [S6] repeat. The module reconstructed Note G’s 25 operations and ran them with exact arithmetic (`course/lab/noteg.py`): the corrected run reproduces the documented $-1/30$, which is what earns trust in the reconstruction, and the as-printed run then yields $139/630$. The misprint corrupts only the A term (operation 9 overwrites the affected variable before the multiplicative loop), so it cannot produce a value as large as $-25621/630 \approx -40.7$. The much-repeated $-25621/630$ appears in no scholarly source and does not survive recomputation — which is exactly the lesson of this unit, now demonstrated on the unit’s own example.

Still contested: how it got there. The common description is a *typesetting* error rather than a mistake in the procedure Lovelace devised — Babbage's 1864 memoir records that she caught an error for him, and this one is often read as introduced in printing. But that attribution is an inference, not something the table itself can settle, and whether Babbage prepared or checked the table bears on the unsettled authorship question in section 2. Do not state the printer, or any individual, as the cause.

A caveat that is itself the lesson. The facsimile consulted is a single compressed web image [P2a]. From it, the operand order (`V` before `V`) and the location (operation 4) are legible and match the secondary account; the division *operator* is small enough at that resolution to be easy to misread — on a first pass it can look like a plus sign. The value chain has now been recomputed from the operation sequence (`course/lab/noteg.py`), so the `139/630` result rests on the module's own tested code rather than on repetition; a higher-resolution facsimile or the physical page in *Scientific Memoirs* vol. 3 would still settle the fine typography beyond doubt. What is settled now is the location, the operand inversion, and — by computation — the value the faulty table produces.

Why this restraint is the lesson, not a limitation. Unit 7's teaching point is not "here is an amusing bug in a famous document". It is that a published, respected, widely-cited table carried an error for a long time because readers trusted it instead of checking it, and that the specific story now attached to that error has itself propagated by repetition. A fluent AI system will produce a confident, detailed account of the Note G error on request. So will many websites. Being able to say "that is the common claim, here is what is actually established, here is what would settle it" is the transferable skill — and a module that modelled the opposite behaviour while teaching it would be worthless.

33.5 5. The "originate anything" passage

Established that the passage appears in Note G [P2]. Lovelace writes that the Analytical Engine "has no pretensions whatever to originate anything", and that it can do "whatever we know how to order it to perform".

Contested what it implies for modern systems. Unit 8 asks learners to compare the claim with what a language model does, and is written to leave the question open rather than resolve it. Both of the easy readings are available and neither is obviously right:

- A language model produces text no human wrote, so in some sense it originates.
- A language model produces statistically likely continuations of its training data given a prompt, and "we ordered it to perform" exactly that.

The honest position is that Lovelace was writing about a specific machine she understood in detail, that the passage is being applied well outside its original context, and that the argument turns on what "originate" means — which is a philosophical question the module does not settle. Present it as a live question. Quote briefly; the source is public domain, so the limit is pedagogical rather than legal.

33.6 6. Claim register

The quick-reference summary. Anything a unit asserts about history should appear here.

claim	status	source
Babbage designed the Analytical Engine; never completed in his lifetime	established	[S1], [S4]
Store/mill architecture, punched-card input	established	[S1]

claim	status	source
Menabrea published a French account in 1842	established	[P1]
Lovelace translated it and added Notes A–G, published 1843	established	[P2]
The Notes are about three times the length of the article	secondary characterisation, consistent with the page counts in [P2] but not stated by it	[P2]
Note G contains an operation-by-operation Bernoulli computation	established	[P2]
Note G's example computes Ada's B = modern B = -1/30	established	[P2], verified in lab
Ada's B /B /B /B = modern B /B /B /B	established	verified in lab
"The first program ever written"	contested	see section 2
Extent of Lovelace's own contribution to the Notes	contested	[S2], [S3], [S5]
The published table contains an error, in operation 4	established	facsimile [P2a], section 4
The error is an inverted division, `V ÷ V` for `V ÷ V`	established	facsimile [P2a], [S8]; corroborated by [S7] (self-published reconstruction)
The as-printed table computes `139/630` (not the repeated `−25621/630`)	established (recomputed)	`course/lab/noteg.py`; [S7]
The error was a typesetting slip, not authorial	contested	see section 4
"No pretensions whatever to originate anything" appears in Note G	established	[P2]
What that passage implies about language models	contested	see section 5

34 The verification lab and self-check

This directory holds the tools you use to check your own work: a trusted reference implementation of the Bernoulli numbers, a differential checker you can point at any Python file, and a self-check you can run repeatedly as you work through the units.

Two routes exist, and they are equals. The **browser route** (`build/site/selfcheck.html`) needs no installation and covers the same checkpoints. The **Python route** described here suits you if you already run Python or want to see the checker work on real code. Choose on preference and circumstance, not on confidence. Nothing in the core module depends on installing anything.

Separately from both, the **table-based verification** path in Units 3, 5 and 7 lets you check generated output against a small table you fill in by hand. That is a first-class method — it is how the oracle itself was validated against published values — not a substitute for anything.

34.1 Requirements

Python 3 and nothing else. Every file here uses only the standard library (`fractions`, `math`, `argparse`, `json`, `importlib`, `subprocess`, `pathlib`). There is no `pip install` step, no virtual environment, and no network access at any point.

Verified on Python 3.13.14. Anything from Python 3.9 onwards should work.

On Windows, use `py` or `python` wherever these pages say `python3`.

Run all commands from inside `course/lab`, because the example and exercise paths in these commands are relative to that folder:

```
cd course/lab
```

34.2 Safety: read before you run

`check.py` **imports and executes** the file you hand it. So does `selfcheck.py` when it runs a code checkpoint for you. There is no sandbox.

AI-generated code is code from an untrusted source. It is not malicious by design, but it was not written by anyone who is answerable for it, and you cannot tell what it does by looking at how confident it sounds. Treat it exactly as you would treat a script downloaded from a stranger:

- **Read the whole file first.** Look for anything that touches the filesystem, the network, or the shell. A Bernoulli routine needs none of those.
- **Prefer a disposable environment**: a scratch directory, a container, a throwaway virtual machine, or a machine with nothing on it you would mind losing. Not your only copy of your coursework.
- **Do not paste confidential, personal, unpublished, or assessment-restricted material into an AI system** in order to get the code in the first place.
- Notice that you had to make this decision. That noticing is part of the module, not an obstacle to it.

34.3 What each file is

file	what it is
`reference_bernoulli.py`	The **test oracle**. Computes `B_n` exactly using `fractions.Fraction` and the binomial recurrence, and checks itself against 22 values transcribed from published tables. Everything else in the lab judges against this.

file	what it is
`noteg.py`	A faithful reconstruction of Ada's Note G table, run with exact arithmetic. The corrected table computes $-1/30$; the as-printed table (with the operation-4 misprint) computes $139/630$ — **not** the $-25621/630$ repeated online. Run it: `python3 noteg.py`. Used by Unit 7 to recompute a much-copied number instead of repeating it.
`check.py`	The **differential tester**. Imports a submitted file, compares its `bernoulli` against the oracle index by index, reports the *first* divergence, and names the failure mode.
`selfcheck.py`	The **learner CLI**. Lists checkpoints, runs them, records your progress locally, and gives you an interactive oracle to explore.
`checkpoints.py`	Loads all 15 checkpoint questions, answers, diagnostics and hints from each unit's `activity.md` and grades your answers. The browser self-check is generated from the same data, which is what stops the two routes drifting apart.
`examples/`	Five small implementations: one correct, four broken in specific, named ways. Use them to see what each diagnostic looks like before you meet it in your own work.
`exercises/`	Stubs you complete. Each carries its task in its docstring.
`tests/`	The test suite for the framework itself — the oracle against published values, the checker against known-bad examples, the checkpoint definitions against their own diagnostics. This exists so that the tools you are trusting have themselves been checked.

34.3.1 The example implementations

file	what is wrong with it	code
`examples/correct_example.py`	Nothing. Deliberately uses a *different* algorithm from the oracle (Akiyama–Tanigawa rather than the binomial recurrence), so agreement means something.	—
`examples/wrong_b1_sign.py`	Correct under the second Bernoulli convention, $B_1 = +1/2$.	`b1-sign`
`examples/wrong_odd_terms.py`	Numbers only the non-zero terms, as Note G does.	`odd-terms`, `ada-indexing`
`examples/float_output.py`	Returns `float`. Convincing at small indices, wrong everywhere.	`not-fraction`, `odd-terms`
`examples/off_by_one_ramanujan.py`	Every value is a genuine Bernoulli number, filed one index too early.	`off-by-one`, `odd-terms`

34.3.2 The exercises

file	checkpoint	unit	status
`exercises/ex04_bernoulli.py`	`cp05-bernoulli`	5	core
`exercises/ex05_write_a_test.py`	—	5	optional CS extension
`exercises/ex06_ada_translation.py`	—	7	optional CS extension

The optional CS extensions are labelled optional in the module because they are optional. Skipping them costs you no core content.

34.4 The interface contract

This is the contract `check.py` enforces, and the same words appear in the Unit 6 precise prompt:

```
bernoulli(n: int) -> fractions.Fraction
```

Defined for every `n >= 0`; first-Bernoulli convention `B1 = -1/2`; exact `Fraction` returns only, never `float`; returns `Fraction(0)` for odd `n > 1`.

That is the whole specification. It is short enough to read and precise enough to test against — and a specification precise enough to test against is a specification precise enough to prompt with.

34.5 Failure-mode codes

These are the names `check.py` emits. They are the module's fixed failure-mode codes, and the module uses these names and no synonyms.

code	meaning
`b1-sign`	Correct under the *second* convention, `B1 = +1/2`.
`odd-terms`	Odd terms above B1 are not exactly zero.
`ada-indexing`	Only the non-zero terms are numbered, as in Note G.
`off-by-one`	Every value shifted one index.
`not-fraction`	Returns `float` or another inexact type.
`raises`	Undefined somewhere in the required range.
`import-error`	The file fails on import.
`no-function`	No `bernoulli` defined.

`check.py` also prints three structural markers alongside these: `divergence` gives the first index where your values and the oracle's disagree, `no-file` means the path does not exist, and `not-callable` means `bernoulli` exists but is not a function. They locate the problem; the eight codes above name it.

One report often carries several codes. `examples/off_by_one_range.py` produces `divergence`, `odd-terms` and `off-by-one` together, because a one-index shift also drags non-zero values onto the odd indices. Read the codes as a description of the shape of the fault, not as a count of separate bugs.

34.6 Commands

Every transcript below is real output from running the command in this directory. Where a command is interactive, what you type is shown after the prompt.

34.6.1 Print the oracle's table

```
$ python3 reference_bernoulli.py
Bernoulli oracle - convention: first Bernoulli numbers (B_n^-), B1 = -1/2

  n   B_n
  --  --------------------------
   0  1
   1  -1/2
   2  1/6
   3  0
   4  -1/30
   5  0
   6  1/42
   7  0
   8  -1/30
   9  0
  10  5/66
  11  0
  12  -691/2730

Larger index, where float implementations fail:
  B30 = 8615841276005/14322

Note G indexing (Lovelace's label -> modern label -> value):
  Ada B1  ->  modern B2  =  1/6
  Ada B3  ->  modern B4  =  -1/30
  Ada B5  ->  modern B6  =  1/42
  Ada B7  ->  modern B8  =  -1/30

OK: oracle agrees with all 22 published values, odd terms are zero.
```

Exit status 0. If the oracle ever disagreed with its published table it would print `FAIL:` with the offending indices and exit 1. Start here: this is the table you check everything else against, and it is short enough to copy onto paper for the table-based route.

34.6.2 Check a file against the oracle

```
$ python3 check.py examples/wrong_b1_sign.py
======================================================================
Checking: examples/wrong_b1_sign.py
Oracle convention: first Bernoulli numbers (B_n^-), B1 = -1/2
======================================================================

FAIL - 2 problem(s) found.

[divergence] First divergence at n=1: expected -1/2, got 1/2.
    Everything below this index matched. The first divergence is the
    best place to start debugging. It does not prove that later
    differences have the same cause.
    See: Unit 5 (differential testing against an oracle)

[b1-sign] B1 has the wrong sign for this module's convention.
    Your routine returns B1 = +1/2. That is the *second* Bernoulli
    convention, and it is not wrong mathematically -- but this module
```

```
    fixes the *first* convention, B1 = -1/2. The conventions differ
    only at B1, but leaving that choice unstated is still a
    specification failure, not an arithmetic failure.
    See: Unit 3 (the convention trap), Unit 6 (state the convention in t

Fix the specification or the code, then run this check again.
```

Exit status 1 on failure, 0 on success. A passing run looks like this:

```
$ python3 check.py examples/correct_example.py --verbose
======================================================================
Checking: examples/correct_example.py
Oracle convention: first Bernoulli numbers (B_n^-), B1 = -1/2
======================================================================

PASS - agrees with the oracle for every n from 0 to 30,
       returns exact Fractions, and odd terms above B1 are zero.

Passing is not the end of the job. You still have to be able to say
*why* it is correct. That is the Unit 8 defence.

(Indices compared: 0 to 30)
```

Options:

flag	effect
`--max-n N`	Highest index compared. Default 30.
`--verbose`	Adds a line reporting how many indices were compared, on both passing and failing runs.

Point it at your own file the same way: `python3 check.py my_bernoulli.py`. Read the file first — see the safety section above.

34.6.3 List the checkpoints

```
$ python3 selfcheck.py list
Self-check checkpoints
----------------------------------------------------------------------

Unit 0
  [ ] cp00-contract        (choice)    The learning contract

Unit 1
  [ ] cp01-ordered-steps   (multi)     What the procedure left unsaid

Unit 2
  [ ] cp02-risk-triage     (match)     Triage by how it can be wrong
  [ ] cp02-system-layers   (match)     Separate the assistant's system l

Unit 3
  [ ] cp03-b4-by-hand      (fraction)  B4 by hand
  [ ] cp03-convention      (choice)    Which convention is this?
  [ ] cp03-ada-index       (integer)   Reading Ada's index
```

```
Unit 4
  [ ] cp04-trust-triage    (choice)    What you can trust before testing

Unit 5
  [ ] cp05-bernoulli       (code)      Make the checker pass
  [ ] cp05-table-evidence  (match)     State what a table comparison est
  [ ] cp05-diagnose        (choice)    Name the failure mode

Unit 6
  [ ] cp06-spec-clauses    (match)     Which clause kills which bug

Unit 7
  [ ] cp07-note-g          (fraction)  Comparing against the historical

Unit 8
  [ ] cp08-disclosure      (multi)     What a defence has to contain

Unit 9
  [ ] cp09-sort-the-claims (match)     Sort the claims

Run one with:  python3 selfcheck.py run <id>
```

`[x]` marks a checkpoint you have passed. The kind in brackets tells you what sort of answer is expected: a fraction such as `-1/30`, a whole number, one option number, several option numbers, an ordered list of option numbers, or — for `code` — a file you edit.

34.6.4 Run one checkpoint

```
$ python3 selfcheck.py run cp03-ada-index
------------------------------------------------------------------
cp03-ada-index  (Unit 3)  -  Reading Ada's index
------------------------------------------------------------------
Note G computes the number Lovelace calls B7. In modern notation,
where every index is numbered including the zero ones, which B_n is
that? Enter the number only.

Answer with a whole number.

Your answer (or 'skip'): 8

PASS
Ada B1/B3/B5/B7 are the modern B2/B4/B6/B8, so Note G's worked example
produces the modern B8 = -1/30. A historical notation that means
something different from the identical modern notation is a
specification hazard, and it is exactly the kind of thing a language
model will blur together.
```

A wrong answer that matches a known misconception gets that misconception named rather than a bare “incorrect”:

```
$ python3 selfcheck.py run cp03-b4-by-hand
------------------------------------------------------------------
cp03-b4-by-hand  (Unit 3)  -  B4 by hand
------------------------------------------------------------------
Using the recurrence sum_{j=0}^{n} C(n+1, j) * B_j = 0 with B0 = 1 and
```

```
the first-Bernoulli convention B1 = -1/2, work out B4 by hand. Enter
it as an exact fraction, for example -1/30 or 5/66.

Answer as an exact fraction, e.g. -1/30

Your answer (or 'skip'): 1/30

Not yet.
Right magnitude, wrong sign. Check the sign of the term you moved
across the equals sign when you solved for B4 -- the whole sum equals
zero, so the highest term takes the sign of everything else negated.
```

You get three attempts per session, and typing `skip` (or pressing Enter on an empty line) moves on. On checkpoints where deriving the value *is* the point, such as `cp03-b4-by-hand`, a failure never reveals the answer. It gives you the next step instead.

The `code` checkpoint delegates to `check.py`, and recognises an untouched stub rather than reporting a crash:

```
$ python3 selfcheck.py run cp05-bernoulli
------------------------------------------------------------------------
cp05-bernoulli  (Unit 5)  -  Make the checker pass
------------------------------------------------------------------------
Complete course/lab/exercises/ex04_bernoulli.py so that
bernoulli(n) meets the contract, then run: python3 selfcheck.py run
cp05-bernoulli

Not attempted yet - the stub is still unedited.
Open exercises/ex04_bernoulli.py and complete it, then run this again.
```

`run` also accepts `all` (or no target at all) to work through every checkpoint in order, and `--answer TEXT` to supply a single answer without being prompted.

34.6.5 Run every checkpoint in a unit

```
$ python3 selfcheck.py run --unit 3
------------------------------------------------------------------------
cp03-b4-by-hand  (Unit 3)  -  B4 by hand
------------------------------------------------------------------------
Using the recurrence sum_{j=0}^{n} C(n+1, j) * B_j = 0 with B0 = 1 and
the first-Bernoulli convention B1 = -1/2, work out B4 by hand. Enter
it as an exact fraction, for example -1/30 or 5/66.

Answer as an exact fraction, e.g. -1/30

Your answer (or 'skip'): -1/30

PASS
You now know what the answer *should* be before any AI produced it.
Deriving the value independently gives you one strong reference for
judging generated output; an authoritative table or separately
implemented method can also provide independent evidence. This value
catches the sign-convention bug in Unit 5.

------------------------------------------------------------------------
cp03-convention  (Unit 3)  -  Which convention is this?
```

```
------------------------------------------------------------------------
A program prints B0 = 1, B1 = 1/2, B2 = 1/6, B4 = -1/30. Its author
says it is correct and they are not lying. What is going on?

  (1) It uses the second Bernoulli convention (B1 = +1/2); every other
value agrees.
  (2) It has a sign bug that happens to affect only B1.
  (3) It is using Lovelace's Note G indexing.
  (4) It is correct and this module's oracle is wrong.

Answer with one option number.

Your answer (or 'skip'): 1

PASS
Both conventions are standard and published. Neither program is
broken; the *specification* was incomplete. This is why the module's
contract states B1 = -1/2 explicitly, and why Unit 6 puts that
sentence in the prompt.

------------------------------------------------------------------------
cp03-ada-index  (Unit 3)  -  Reading Ada's index
------------------------------------------------------------------------
Note G computes the number Lovelace calls B7. In modern notation,
where every index is numbered including the zero ones, which B_n is
that? Enter the number only.

Answer with a whole number.

Your answer (or 'skip'): 8

PASS
Ada B1/B3/B5/B7 are the modern B2/B4/B6/B8, so Note G's worked example
produces the modern B8 = -1/30. A historical notation that means
something different from the identical modern notation is a
specification hazard, and it is exactly the kind of thing a language
model will blur together.

------------------------------------------------------------------------
Session: 3 of 3 checkpoints passed.
------------------------------------------------------------------------
```

34.6.6 See how far you have got

```
$ python3 selfcheck.py progress
Passed 7 of 15 checkpoints.

  Unit 0  [#]  1/1
  Unit 1  [#]  1/1
  Unit 2  [##]  2/2
  Unit 3  [###]  3/3
  Unit 4  [.]  0/1
  Unit 5  [...]  0/3
  Unit 6  [.]  0/1
```

```
  Unit 7  [.]  0/1
  Unit 8  [.]  0/1
  Unit 9  [.]  0/1

Next up: cp04-trust-triage (Unit 4)
```

34.6.7 Export a record for your evidence pack

```
$ python3 selfcheck.py progress --evidence
# Self-check record

Generated 2026-07-24 00:23 UTC.

Passed 7 of 15 checkpoints.

| checkpoint | unit | passed | attempts |
|---|---|---|---|
| cp00-contract | 0 | yes | 1 |
| cp01-ordered-steps | 1 | yes | 1 |
| cp02-risk-triage | 2 | yes | 1 |
| cp02-system-layers | 2 | yes | 1 |
| cp03-b4-by-hand | 3 | yes | 1 |
| cp03-convention | 3 | yes | 1 |
| cp03-ada-index | 3 | yes | 1 |
| cp04-trust-triage | 4 | no | 0 |
| cp05-bernoulli | 5 | no | 0 |
| cp05-table-evidence | 5 | no | 0 |
| cp05-diagnose | 5 | no | 0 |
| cp06-spec-clauses | 6 | no | 0 |
| cp07-note-g | 7 | no | 0 |
| cp08-disclosure | 8 | no | 0 |
| cp09-sort-the-claims | 9 | no | 0 |

This record shows which checkpoints were passed. It is not a claim that
the learner can defend the answers; that is what the Unit 8 defence is f
```

That output is Markdown, so you can redirect it straight into your evidence pack:

```
$ python3 selfcheck.py progress --evidence > my-selfcheck-record.md
```

Nothing sends it anywhere. Exporting it is a decision you make.

To move your checkpoint results into the browser course, or to keep a restorable backup, write the portable JSON record:

```
$ python3 selfcheck.py progress --backup my-ai-literacy-record.json
Portable learner record written to my-ai-literacy-record.json.
```

The browser progress page imports the same format. The readable Markdown evidence record is for you or an instructor to read; the JSON file is the format that can restore progress.

34.6.8 Explore the oracle

```
$ python3 selfcheck.py explore
Bernoulli oracle - exploration mode
Convention: first Bernoulli numbers (B_n^-), B1 = -1/2
```

```
Commands:
  b <n>            value of the modern B_n            e.g. b 30
  ada <n>          value of Lovelace's B<n>           e.g. ada 7
  map              the Note G index mapping table
  table [n]        modern B_0 .. B_n (default 12)
  float <n>        B_n as an exact fraction and as a float, side by side
  check <file>     run the differential checker on a file
  help             this list
  quit             leave

explore> b 4
  B4 = -1/30
explore> ada 7
  Ada B7 = modern B8 = -1/30
explore> map
  Ada B1  ->  modern B2  =  1/6
  Ada B3  ->  modern B4  =  -1/30
  Ada B5  ->  modern B6  =  1/42
  Ada B7  ->  modern B8  =  -1/30
explore> float 30
  exact  B30 = 8615841276005/14322
  float  B30 = 601580873.9006424
  -> the float is NOT equal to the exact value.
explore> table 6
  B0   = 1
  B1   = -1/2
  B2   = 1/6
  B3   = 0
  B4   = -1/30
  B5   = 0
  B6   = 1/42
explore> quit
```

This is where the module stops being a reading and starts being a laboratory. Look up any value you want to check by hand, watch `float 30` fail, and run `check <file>` on anything you have generated without leaving the session.

34.6.9 Clear your progress

```
$ python3 selfcheck.py reset
Progress cleared.
```

Running it again when there is nothing to clear:

```
$ python3 selfcheck.py reset
No progress to clear.
```

34.6.10 Write the test yourself (optional CS extension)

The stub reports that it is a stub:

```
$ python3 exercises/ex05_write_a_test.py
Write your test here, then run this file.
```

Once you have filled in `catches_the_bug`, the exercise grades your test against the real example implementations. A test that only checks `B4 == -1/30` looks reasonable and lets two of the four faults

straight through:

```
$ python3 exercises/ex05_write_a_test.py
Running your test against every example implementation.
------------------------------------------------------------
  MISSED wrong_b1_sign        (the other B1 convention)
  MISSED wrong_odd_terms      (Note G non-zero-only indexing)
  caught float_output        (floating point, not exact)
  caught off_by_one_range    (index shifted by one)
  passed correct_example      (a correct implementation)
------------------------------------------------------------
Not yet.
  Slipped through: wrong_b1_sign, wrong_odd_terms
  Ask what property each of those violates that a correct one does not.
```

A test that checks the type, the convention, the zero odd terms and a large index separates all four:

```
$ python3 exercises/ex05_write_a_test.py
Running your test against every example implementation.
------------------------------------------------------------
  caught wrong_b1_sign        (the other B1 convention)
  caught wrong_odd_terms      (Note G non-zero-only indexing)
  caught float_output        (floating point, not exact)
  caught off_by_one_range    (index shifted by one)
  passed correct_example      (a correct implementation)
------------------------------------------------------------
PASS - your test separates all four faults from the correct version.

Now notice what you had to know in order to write it. Every clause in
your test corresponds to a clause the specification should have had.
That is the same list you build in Unit 6.
```

Note that wrongly rejecting `correct_example` is reported as an error too. A test that fails good code is worse than no test, because you learn to ignore it.

34.6.11 Check the framework itself

The lab's own test suite runs with `pytest` if you have it, and with a standard-library fallback runner if you do not:

```
$ python3 tests/run_tests.py
...
============================================================
81 passed, 0 failed, 6 skipped
============================================================
```

The counts change as the suite grows. A skipped test is one this plain runner cannot drive (it needs pytest fixtures, and runs under `pytest` instead); it is not a failure. You are not required to run this — it exists so that the tools judging your work have themselves been checked against published values and known-bad examples.

34.7 What is recorded, and what is not

- The self-check is **formative and unmarked** unless your instructor says otherwise. It exists so you can find out what you have understood while there is still time to do something about it.
- Progress is stored **locally**, in `.selfcheck-progress.json` in this directory. It uses the portable learner-record format documented in `course/learner-record-schema.json`

and records checkpoint answers, pass state, attempts, and timestamps. Keep a separate backup if the work matters.

- **Nothing is uploaded.** There is no network code anywhere in this directory, no analytics, and no account. Delete the file, or run `selfcheck.py reset`, and the record is gone.
- If your instructor asks for evidence, you produce it yourself with `progress --evidence` and hand it over deliberately. See `course/learner-evidence-template.md`.
- The answers are in each unit's `activity.md` (which `checkpoints.py` reads), in plain sight. This is a self-check, not an exam. Reading them costs you the only thing the checkpoint was offering.

34.8 The browser route

`build/site/selfcheck.html` runs the same 15 checkpoints, with the same ids, the same expected answers and the same diagnostic wording, in a supported browser. It is generated from the same activity data by `scripts/build_selfcheck.py`, which is why the two routes cannot drift apart. It is self-contained, makes no network requests, and can work from a `file://` URL. Browser storage behaviour for local files varies, so use the published course site for durable local progress and keep a portable backup either way.

Use it if you would rather not install or run Python, if you are working on a machine where you cannot, or simply because you prefer it. It is not a reduced version of the Python route, and choosing it does not put you on a lesser path through the module. What it does not include is the checker running against real code — for that, either use the Python route or follow the table-based verification worksheet in Unit 5, which reaches the same conclusion by hand.

34.9 Where this fits

you are working on	start with
Unit 3, grounding yourself in the numbers	`python3 reference_bernoulli.py`, then `selfcheck.py run --unit 3`
Unit 4, judging generated code	read the file, then `python3 check.py <file>`
Unit 5, the verification lab	`exercises/ex04_bernoulli.py` and `selfcheck.py run cp05-bernoulli`
Unit 6, writing the precise prompt	the interface contract above, quoted verbatim
Unit 7, the historical table	`selfcheck.py explore`, then `map`
Unit 8, disclosure and defence	`selfcheck.py progress --evidence`

Related material: `course/course-map.md` for how the units map to the wider course, `course/glossary.md` for the terms used here, `course/misconceptions.md` for the errors these tools are built to catch, `course/historical-notes.md` for Note G and its citations, and `course/references.md` for sources.

35 Glossary

Terms used across this module, with the unit that introduces each one. Unit titles and numbers come from `course/course-map.md`; this file does not restate them.

Where a term has a loose everyday meaning and a precise one here, the precise one is what the module means.

35.1 Index

term	introduced in
algorithm	Unit 1
ordered procedure	Unit 1
determinism	Unit 1
state	Unit 1
language model	Unit 2
next-token prediction	Unit 2
attention	Unit 2
pre-training	Unit 2
post-training	Unit 2
retrieval	Unit 2
plausibility	Unit 2
hallucination	Unit 2
independent verification	Unit 2
convention	Unit 3
exact rational arithmetic	Unit 3
floating point	Unit 3
ground truth	Unit 3
off-by-one	Unit 4
verification	Unit 5
test oracle	Unit 5
differential testing	Unit 5
property-based check	Unit 5
first divergence	Unit 5
overreliance	Unit 6
established, contested, unsourced	Unit 7
disclosure	Unit 8
defence	Unit 8
provenance	Unit 8
warrant	Unit 8
formative and summative	Unit 0
first-order and second-order reasoning	Unit 9
observer	Unit 9
Macy Conferences	Unit 9
understanding	Unit 8, Unit 9

35.2 Definitions

35.2.1 algorithm

A finite, unambiguous description of how to compute something: a fixed set of steps that, applied to valid input, produces the intended output and stops. Every step must be executable without judgement or interpretation by whatever carries it out.

Introduced in Unit 1. Note G is the module's example: a computation written out in enough detail that a machine with no knowledge of Bernoulli numbers could carry it out.

35.2.2 ordered procedure

The plain-language name this module uses for an algorithm written as a numbered sequence of operations, each one acting on named quantities in a stated order. The emphasis is on *ordered*: the steps are not a list of things to do, but a sequence in which position carries meaning, because each step depends on the result of the ones before it.

Introduced in Unit 1.

35.2.3 determinism

The property that the same input, run through the same procedure, always produces the same output. A deterministic procedure has no dependence on chance, on timing, or on anything not written down. `reference_bernoulli.bernoulli(4)` returns `-1/30` today, tomorrow, and on any machine.

Introduced in Unit 1. Contrast with a language model, whose output for the same prompt may differ between runs.

35.2.4 state

The set of values a procedure is currently holding, and which it may change as it proceeds. In Note G these are the engine's numbered variables; in the lab oracle they are the partial sums accumulated inside the recurrence. Tracking state is what lets you say precisely where a computation has got to, and therefore where it went wrong.

Introduced in Unit 1.

35.2.5 language model

A statistical model of sequences of text, fitted to a large body of existing text, which assigns probabilities to what token might come next given what has come before. Text is generated by sampling from those probabilities repeatedly.

A language model is a description of regularities in text, not a store of verified facts and not an agent with intentions. Describing what it produces is accurate; describing what it "believes" or "wants" is not.

Introduced in Unit 2.

35.2.6 next-token prediction

The mechanism by which a language model produces text: given the sequence so far, it produces a probability distribution over possible next tokens, one is selected, it is appended, and the process repeats. A token is a short chunk of text — often a word or part of a word.

Two consequences matter for this module. First, the base generation objective is a likely continuation, not a direct correctness test. Second, correct and incorrect output can both be fluent, so fluency alone does not establish correctness.

Introduced in Unit 2.

35.2.7 attention

The mechanism inside a transformer language model that lets the representation at each position combine information from relevant positions elsewhere in the available context. It is a way of combining context; it is not a database lookup, a citation, or a truth test.

Introduced in Unit 2.

35.2.8 pre-training

The first stage of building a language model: adjusting its many numerical parameters to improve next-token prediction over a large training corpus. The result is the base model — a description of regularities in that text, not a store of verified facts.

Introduced in Unit 2.

35.2.9 post-training

Further training applied to a base model — on demonstrations and preference judgements — so a deployed assistant follows instructions and behaves as its makers intend. It changes how the model responds; it does not make a generated claim correct by definition.

Introduced in Unit 2.

35.2.10 retrieval

A system layer some deployed assistants add: fetching documents or search results and placing them in the model's context before it generates. Retrieval can ground an answer in a real source, but the generated text still has to be checked against that source — attaching a document is not the same as citing it correctly.

Introduced in Unit 2.

35.2.11 plausibility

The property of reading as though it could be right: well-formed, idiomatic, consistent in tone, and shaped like a correct answer. Language-model training and generation strongly reward plausible continuation.

Plausibility is not correctness, and — importantly — it is not evidence of correctness either. A plausible wrong answer can look as polished as a plausible right answer, which is why the module builds a separate means of checking.

Introduced in Unit 2.

35.2.12 hallucination

The established term for output from a generative model that is fluent, well-formed, and false: an invented citation, a non-existent function, a confident date that no source supports.

The word is a metaphor and should not be taken literally. The model is not perceiving anything, not mistaken about anything, and not doing something different from what it does when it is right. It is producing a statistically plausible continuation in both cases; "hallucination" simply names the subset of those continuations that do not correspond to the relevant source, specification, or world. Surface reading may expose some errors, but it cannot establish the claim; that requires appropriate evidence.

Introduced in Unit 2. Used consistently in this sense throughout the module.

35.2.13 independent verification

A check whose evidential basis is separable from the assertion being checked: for example, an authoritative source, an exact calculation, a test oracle built by a different method, or accountable review against explicit criteria.

Re-prompting or self-critique can improve an answer [T9], but the same system may preserve the same assumptions. Improvement is useful; it is not the same claim as independent verification.

Introduced in Unit 2; practised in Unit 5.

35.2.14 convention

A choice about notation or definition that is agreed rather than derived, where a different choice would be equally valid and would give different results. The Bernoulli numbers have two published conventions: the **first** Bernoulli numbers, with `B1 = -1/2`, and the **second**, with `B1 = +1/2`. Every other value is identical.

Conventions cause a distinctive kind of fault. Two programs can both be correct and defensible while disagreeing at `B1`, because the specification never said which convention was meant. They agree at every other index, but the unresolved meaning of `B1` remains a specification defect. This module fixes the first convention, `B1 = -1/2`, and states it in the contract.

Introduced in Unit 3. Failure-mode code: `b1-sign`.

35.2.15 exact rational arithmetic

Arithmetic on fractions represented as an exact numerator and denominator, with no rounding at any point. Python's `fractions.Fraction` does this: `Fraction(1,3)` is one third, not an approximation to it.

The lab oracle uses exact rational arithmetic because Bernoulli numbers have large numerators and denominators — `B30 = 8615841276005/14322` — and comparing generated output against a reference is only meaningful if both sides are exact. An approximate comparison has to choose a tolerance, and any tolerance loose enough to pass rounding error is loose enough to pass a real bug.

Introduced in Unit 3.

35.2.16 floating point

The binary approximation to real numbers used by default in most programming languages, including Python's `float`. It is fast, fixed in size, and inexact: most fractions cannot be represented, so results carry small errors that accumulate.

For small Bernoulli indices, floating-point output looks convincing — `-0.03333...` really is close to `-1/30`. By `B30` the error is unmistakable. Producing floats where exact values were specified is a failure of the specification's arithmetic requirement, not a rounding detail to be tolerated.

Introduced in Unit 3. Failure-mode code: `not-fraction`.

35.2.17 ground truth

The independently established answer against which a result is judged. In this module, ground truth for the Bernoulli numbers comes from published tables and from the value you derive yourself by hand in Unit 3 — not from the program being tested, and not from a second generated answer that agrees with the first.

The ordering matters: ground truth has to be established *before* you look at the output you are judging, or you will find yourself deciding that whatever you got must have been right.

Introduced in Unit 3.

35.2.18 off-by-one

An indexing error in which every value is correct but shifted by one position, so that item *n* returns what belongs at *n+1* or *n-1*. It is the archetypal programming mistake because it survives inspection: the arithmetic is right, the values are all genuine, and nothing looks wrong until you compare index by index against a reference.

Introduced in Unit 4. Failure-mode code: `off-by-one`.

35.2.19 verification

Establishing that a result is correct by checking it against something independent of the process that produced it. Reading the output again, asking the model whether it is sure, or noting that it looks reasonable are none of them verification, because none of them is independent.

In this module verification takes three concrete forms, all equally valid: comparing against a table you filled in by hand, comparing against a published source, and comparing against a test oracle by running code.

Named in Unit 0 as part of the learning contract; practised throughout and made central in Unit 5.

35.2.20 test oracle

A trusted source of correct answers, used to judge a piece of code under test. `course/lab/reference_bernoull` is this module's oracle.

An oracle is only worth as much as its own validation, so this one is checked in two directions: against 22 values transcribed from published tables, which it did not compute, and against properties that must hold regardless of how it computes anything.

Introduced in Unit 5.

35.2.21 differential testing

Running two implementations of the same specification on the same inputs and comparing the results, on the principle that two independent routes to the same answer are much stronger evidence than one route agreeing with itself. `course/lab/check.py` does this: your routine against the oracle, index by index.

The independence is what carries the weight. `examples/correct_example.py` deliberately uses a different algorithm from the oracle — the Akiyama–Tanigawa transform rather than the binomial recurrence — so that agreement is informative rather than circular.

Introduced in Unit 5.

35.2.22 property-based check

A check that tests a property which must hold for *every* valid input, rather than comparing specific input-output pairs. The module's example: every odd Bernoulli number above `B1` is exactly zero, so `B_(2k+1) = 0` for all `k >= 1`.

Property checks catch faults that a table of examples misses, because they do not depend on having chosen the right examples. They also state, in code, something a reader would otherwise have to know.

Introduced in Unit 5. Failure-mode code: `odd-terms`.

35.2.23 first divergence

The lowest index at which a routine under test and the oracle disagree. The checker reports this specifically because it gives a reproducible boundary: every lower tested input matched. It is often a useful place to start debugging, but it does not prove that later failures share the same cause.

Introduced in Unit 5.

35.2.24 overreliance

Depending on a tool beyond the point where you can tell whether its output is right. It is not the same as using AI heavily; it is using AI in place of the understanding that would let you check it.

The practical test is whether you could detect an error if there were one. If a generated answer would look identical to you whether it was correct or not, you are relying on it rather than using it. This is why the module has you derive `B4` by hand before any code is generated: the understanding has to come first, or there is nothing to check against.

Introduced in Unit 6.

35.2.25 established, contested, unsourced

The module's three buckets for the status of a claim. **Established**: you can name where it comes from, and the source is one that you, or someone accountable to you, has actually looked at. **Contested**: sources disagree, or a single account has been copied forward through many retellings without an independent check behind it. **Unsourced**: you cannot currently say where the claim came from — which is not the same as false.

A claim's bucket is part of the claim, and a claim moves buckets when a check is actually done, not when it is repeated more.

Introduced in Unit 7, §6 of its reading; the same status labels are used throughout `course/references.md`.

35.2.26 disclosure

A short, factual statement of how AI was used in producing a piece of work: what was generated, at what stage, and what was done with it afterwards. Disclosure answers the question *how was this made*.

It is a statement about provenance. It is not a claim about quality, and it does not remove the user's responsibility for the output. Other people and organisations may also have duties in a wider system.

Named in Unit 0; templated and practised in Unit 8. See `course/learner-evidence-template.md`.

35.2.27 defence

A short statement of why the work is correct, citing the checks that were run and the reasoning that was followed. Defence answers the question *why is this right*.

A defence rests on evidence: a value derived by hand, a table compared, a checker run and its output. It never rests on the authority of the tool that produced the work. "The model was confident" and "the code looked right" are not defences.

Disclosure and defence are separate statements answering separate questions, and the module asks for both. The gap between them is the gap between "I generated this" and "I understand and can defend this".

Introduced in Unit 8.

35.2.28 provenance

Where a thing came from: who or what produced it, by what process, at what stage, and what happened to it afterwards. A disclosure is a statement of provenance. Provenance is worth recording honestly, and it is not evidence of correctness — knowing how a piece of work was made tells you what to check, not whether it is right.

Introduced in Unit 8.

35.2.29 warrant

What entitles you to assert a claim: evidence you can show — a value derived by hand, a comparison recorded, a checker's output — together with the specification or assumption the evidence was gathered against. Warrant is about what you can point at, not about who produced the text. A defence carries warrant; a disclosure carries provenance; neither substitutes for the other.

Introduced in Unit 8; taken up again in Unit 9, where a warranted claim carries its index — who observed, in what context, with what record.

35.2.30 first-order and second-order reasoning

In Unit 9, **first-order reasoning** describes an observed system from the outside; **second-order reasoning** includes the observer in the system being described, so that claims carry a three-part index: who observed, in what context, with what record.

This use of "second-order" comes from the second-order cybernetics tradition [C1], developed from the Macy Conferences onwards. It is not the distinction between first-order and second-order logic, and it is not presented as settled terminology across every field.

Introduced in Unit 9.

35.2.31 observer

In Unit 9's sense, whoever or whatever is doing the describing or the checking. The observer question is: when you check something, are you outside the system you are checking, or part of it? Including the observer makes a claim more checkable, not less, because an indexed claim — who observed, in what context, with what record — says what would have to be compared with what. Maturana's compact version: *anything said is said by an observer* [C3, p. 8].

Introduced in Unit 9.

35.2.32 Macy Conferences

Ten meetings on cybernetics held between 1946 and 1953, funded by the Josiah Macy Jr. Foundation and chaired by Warren McCulloch [C4], with deliberately mixed participants — among them Wiener, von Neumann, Shannon, Bateson, Mead and von Foerster. The reflexive question behind second-order cybernetics surfaced there and was developed afterwards. A recognised research tradition with a literature — and a tradition rather than a theorem.

Introduced in Unit 9.

35.2.33 understanding

This module uses a **working structural account**, not a settled definition: your internal model of a target — another agent, a process, a piece of code, or a theory — supports understanding when its parts, relationships, and predictions continue to correspond reliably with the target under relevant checks. The verification lab

gathers evidence for that correspondence through input–output agreement, property checks, and the first divergence. It does not measure understanding directly or completely.

Understanding is **separate from correctness**. It is a structural property — how well your model maps the target — not a claim that the target is right. You can understand a flawed framework perfectly: the module's authors mapped the as-printed Note G table and predicted what it computes without that value being correct. The capacity to map (understanding) and the empirical accuracy of the map (correctness) are two different things, and a defence needs both.

For the formal statement — understanding as a structure-preserving map `f : M1 → M2` *whose validity is measured by consistency, and its decoupling from truth — see* `course/second-order.md`*. Causal accounts emphasise explaining interventions; counterfactual accounts emphasise predicting what would change under different conditions. They are alternatives and possible additions; this module uses the structural account without claiming to settle the question.*

Introduced in Unit 8; formalised through the second-order material in Unit 9.

35.2.34 formative and summative

Formative work exists to tell you — and possibly your instructor — what you have understood while there is still time to act on it. It is not marked, or is marked only for feedback.

Summative work is assessed and contributes to a result.

The self-check in `course/lab/` and the browser route are formative and unmarked unless your instructor states otherwise. The distinction matters for how you should use them: on formative work, discovering that you were wrong is the useful outcome, and a checkpoint you failed twice before passing has taught you more than one you passed first time.

Introduced in Unit 0.

36 AI Skills for Verification

As part of this course, you have learned the crucial distinction between a deterministic, ordered procedure (like Ada Lovelace's original algorithm) and the statistical, predictive nature of a modern language model. The core lesson is that **human verification** is the bridge between AI generation and reliable truth.

36.1 Facts vs. Opinions, Beliefs, and Values

When verifying claims, it is critical to distinguish between **facts** (which can be independently verified through evidence, computation, or historical record) and **opinions, beliefs, values, or ethics** (which are subjective, normative, or culturally situated).

An AI cannot determine what is ethically "right" or what you "should" believe; it can only predict what words commonly follow others in discussions of ethics. When an AI outputs a statement of value or opinion, it is not a "fact" to be verified, but a statistical reflection of its training data. - **How to handle them:** Always explicitly frame these statements. Use phrases like *"According to perspective X…"* or *"A common ethical argument is…"* rather than presenting them as objective truths. Verify the *existence* of the argument, not its absolute correctness.

To help you apply these principles in your own work, we have designed two "AI Skills" (also known as custom instructions, system prompts, or personas) that you can use with any Large Language Model (LLM). These skills force the AI to produce outputs that are much easier for a human to verify.

36.2 1. The "Show Your Work" Skill (For Mathematics & Algorithms)

Purpose: This skill forces the LLM to act more like an ordered procedure by explicitly writing out its state changes. By breaking down calculations into discrete, observable steps, you can trace errors back to their source without having to guess where the statistical generation went wrong.

How to use: Paste this text into your LLM's system instructions, custom instructions, or at the start of your prompt:

> "You are an assistant designed to produce checkable, deterministic-style outputs for mathematical and algorithmic problems. 1. Do not just provide the final answer or leap to the conclusion. 2. Break your reasoning down into the smallest possible discrete steps. 3. For each step, explicitly state the current inputs, the operation you are performing, and the new resulting state or value. 4. If applying a formula or referencing a convention (like the Bernoulli numbers), explicitly state which convention you are using before calculating. 5. Your goal is not just to be right, but to provide an output that a human can easily verify line-by-line against a reference."

36.3 2. The "Review & Triangulate" Skill (For Prose & Source Checking)

Purpose: This skill addresses the first and second-order reasoning principles taught in the course. It prevents the LLM from hallucinating certainty by forcing it to map its claims to independent evidence, and requires it to give you the tools to triangulate the truth yourself.

How to use: Paste this text into your LLM's system instructions, custom instructions, or at the start of your prompt:

> "You are a critical review assistant. When I ask you to review a text, summarise a topic, or answer a question: 1. State the boundaries of your knowledge: clearly separate established facts from your own generated extrapolations. 2. If you make a historical, scientific, or factual claim, tag it with `[Needs Verification]` if it is not something universally considered a basic fact. 3. Never claim that your output is definitively 'true'. Instead, present it as a 'candidate response'. 4. Suggest 1-2 independent ways I can verify your claims (e.g., 'To verify this, you should check primary source X or run a calculation using method Y')."

36.3.1 How to Practice

Try using the **“Show Your Work”** skill on the Bernoulli sequence exercise from Unit 4, and observe how much easier it is to spot exactly where the language model makes an arithmetic mistake.

Try using the **“Review & Triangulate”** skill to ask about Ada Lovelace’s role in writing Note G (Unit 7), and see how the model frames historical uncertainty and provides avenues for independent verification.

37 References And Source Policy

37.1 Source policy

This module teaches people to check claims. It would be indefensible if its own claims were unchecked. Three rules apply to everything written here.

1. Historical and mathematical claims carry a citation or an uncertainty label. Every factual claim in the module maps to an entry below or is plainly labelled contested or unsourced. A claim with neither is a defect, and `docs/instructor-review-log.md` treats it as a blocker. Author-only drafting flags must be resolved before material reaches this reference list.

2. Claims about current AI carry a date. Model capabilities, tool names, pricing, institutional policy, benchmark results, and industry practice all age badly. Any such claim is written either so it does not date (a statement about how next-token prediction works) or with an explicit date attached (a statement about what a particular system did in a particular month). Undated claims about "what AI can do now" do not belong in the materials.

3. Named commercial systems are avoided unless pedagogically necessary. The module teaches a method that outlives any particular product. Naming a model version in the teaching text creates a maintenance burden and implies an endorsement. Where a learner needs *an* AI system for an activity, the material says "an AI assistant of your choice".

37.1.1 Verification status labels

Used throughout `course/historical-notes.md` and the unit materials:

label	meaning
established	Supported by a primary source, or by consistent secondary sources with no serious dispute.
contested	Sources disagree, or the claim is widely repeated but its evidential basis is unclear. State the disagreement; do not pick a side.
unsourced	The source is not currently known. This does not mean false; state what evidence could settle it.

An author-only drafting flag is not a learner-facing uncertainty category and must not survive to release. *Unsourced* is the module's student-facing bucket, defined in Unit 7, for a claim deliberately left open together with the evidence that could move it.

37.1.2 Pre-release freshness review

Before any student-facing release, a reviewer must re-read every dated claim and either refresh it or remove it. This is a named gate in `docs/instructor-review-log.md`.

37.2 Primary sources

[P1] Menabrea, L. F. (1842). "Notions sur la machine analytique de M. Charles Babbage." *Bibliothèque Universelle de Genève*, new series, vol. 41, no. 82, October 1842. The original French account of Babbage's Analytical Engine, based on Babbage's 1840 lectures in Turin. This is the article Lovelace translated.

[P2] Menabrea, L. F., trans. A. A. L. [Augusta Ada Lovelace] (1843). "Sketch of the Analytical Engine Invented by Charles Babbage… with notes by the translator." In R. Taylor (ed.), *Scientific Memoirs*, vol. 3,

London: Richard and John E. Taylor, pp. 666–731. The translation plus Lovelace's Notes A–G. The Notes are commonly characterised as roughly three times the length of the article — a secondary characterisation, consistent with the page counts here but not something this page-run itself states. **Note G** contains the Bernoulli-number diagram and the "no pretensions whatever to originate anything" passage. Public domain. The diagram itself is headed "for the computation by the Engine of the Numbers of Bernoulli. See Note G (page 722 *et seq.*)", so the Bernoulli material sits at **page 722 and following**; the surrounding Notes span the pp. 666–731 range commonly cited for the whole translation. Confirmed against the facsimile [P2a].

[P3] Babbage, C. (1864). *Passages from the Life of a Philosopher.* London: Longman, Green, Longman, Roberts, & Green. Babbage's own retrospective account, including his description of the Analytical Engine and of Lovelace's work on the Notes. A primary source that is also an interested party — useful, not neutral.

37.2.1 Facsimiles

The Note G table must be checked against a facsimile rather than a transcription, because the error the module discusses is precisely the kind of thing a transcription silently corrects.

[P2a] Facsimile of the Note G diagram. Photograph of the printed "Diagram for the computation by the Engine of the Numbers of Bernoulli" from *Scientific Memoirs* vol. 3, as offered by Sophia Rare Books: `https://www.sophiararebooks.com/pictures/3544a.jpg` (accessed 2026-07-23; 1000×654 JPEG, retained at `media/unit-07-historical-bug/facsimile-noteg.jpg` so the check does not depend on the URL persisting). Legible from it: the diagram heading and page reference (722 *et seq.*); that the error is in operation 4; and that operation 4 acts on $^{2}\mathrm{V}$ and $^{2}\mathrm{V}$ in that order. **Not** resolvable at this resolution: the fine typography of the division operator, which is easy to misread. For anything turning on the exact glyph, a higher-resolution facsimile or the physical volume is needed. See `course/historical-notes.md` §4.

Higher-resolution scanned copies of *Scientific Memoirs* vol. 3 are also available through several digital library collections and should be recorded here if used.

37.3 Secondary sources

[S1] Bromley, A. G. (1982). "Charles Babbage's Analytical Engine, 1838." *Annals of the History of Computing*, 4(3), 196–217. Technical reconstruction of the Engine's architecture. The standard reference for what the machine could actually do.

[S2] Stein, D. (1985). *Ada: A Life and a Legacy.* Cambridge, MA: MIT Press. A sceptical assessment of Lovelace's mathematical contribution. Represents one pole of a genuine and continuing scholarly disagreement.

[S3] Toole, B. A. (1992). *Ada, the Enchantress of Numbers: A Selection from the Letters of Lord Byron's Daughter and Her Description of the First Computer.* Mill Valley, CA: Strawberry Press. A collection of correspondence, taking a considerably more expansive view of Lovelace's contribution than [S2]. Read alongside it, not instead of it.

[S4] Swade, D. (2000). *The Difference Engine: Charles Babbage and the Quest to Build the First Computer.* New York: Viking. On Babbage's machines and the modern reconstruction effort.

[S5] Hollings, C., Martin, U., and Rice, A. (2018). *Ada Lovelace: The Making of a Computer Scientist.* Oxford: Bodleian Library. A recent treatment drawing on the Lovelace archives, notable for engaging directly with the disputed questions rather than taking a side by omission. Useful for Unit 7's distinction between established and contested claims.

[S6] The operation-4 error, and the recomputed value. Operation 4 is printed as $\mathrm{V} \div \mathrm{V}$ where $\mathrm{V} \div \mathrm{V}$ is correct, commonly read as a typesetting rather than an authorial error. The *location and operand inversion* are established by the facsimile [P2a] and the scholarly reconstructions [S7],

[S8]. A specific corrupted value, −25621/630, is repeated in encyclopaedia and blog sources (https://en.wikipedia.org/wiki/Note_G, accessed 2026-07-23) but appears in no scholarly source. The module reconstructed the 25-operation table and ran it with exact arithmetic (course/lab/noteg.py): the corrected table reproduces the documented −1/30, and the as-printed table computes **139/630**, not −25621/630. The repeated online value does not survive recomputation and is retained only as an example of a much-copied but unverified figure.

[S7] Glaschick, R. (2016). *Ada Lovelace's Calculation of Bernoulli's Numbers.* Paderborn, 29 September 2016. https://rclab.de/_media/analyticalengine/aal_noteg_glaschick_v1.2.pd (accessed 2026-07-23). A detailed independent reconstruction, self-published — a step-by-step working of the Note G table that states the operation-4 operand swap directly ("V must be divided by V ") and groups it with typographical-domain errors; independently confirms the target value B = −1/30.

[S8] Campbell-Kelly, M. (ed.) (1989). *The Works of Charles Babbage,* vol. 3. London: Pickering. The archival correction of the Note G table (p. 159, note a); cited by [S7] and by Misa (2015) as the fullest correction of record.

37.3.1 A note on tertiary sources

Popular articles, encyclopaedia entries, and blog posts about Note G's table error are abundant, mutually inconsistent, and frequently trace back to each other rather than to the table. They are useful for finding out *what is commonly claimed* — which Unit 7 asks learners to examine — and are not sufficient support for a factual assertion in the materials. Where the module reports a common claim, it says that it is a common claim.

37.4 Mathematical references

[M1] Bernoulli numbers, conventions. Two standard conventions differ in the value of B1: the *first* Bernoulli numbers (B_n^−) take B1 = −1/2, and the *second* (B_n^+) take B1 = +1/2. All other values agree. This module fixes the first convention; see course/lab/reference_bernoulli.py.

[M2] Published value tables. The oracle's PUBLISHED_TABLE was checked against *NIST Digital Library of Mathematical Functions*, Release 1.2.7 of 2026-06-15, F. W. J. Olver et al. (eds.), Chapter 24, §24.2(iv), Tables 24.2.1 and 24.2.3, National Institute of Standards and Technology, https://dlmf.nist.gov/24.2.T1 and https://dlmf.nist.gov/24.2.T3 (accessed 2026-07-24). The subsection identifies its printed source as M. Abramowitz and I. A. Stegun (eds.) (1964), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards Applied Mathematics Series 55, Washington, DC: U.S. Government Printing Office, pp. 809–810; the electronic source corresponds to the corrected tenth printing of December 1972.

The comparison covers B0–B12, every even value through B30 = 8615841276005/14322, and the odd zeros represented in the oracle. The table and DLMF equation 24.2.2 use the module's convention B1 = −1/2. All 22 stored values agree exactly; no correction was needed.

[M3] Defining recurrence. The oracle uses sum_{j=0}^{n} C(n+1, j) * B_j = 0 for n >= 1, with B0 = 1, which yields B1 = −1/2 directly. course/lab/examples/correct_example.p deliberately uses a different method (the Akiyama–Tanigawa transform) so that agreement between them is evidence rather than duplication.

37.5 Language models and current AI systems

These references support the compact account in Unit 2. They describe distinct layers of a modern assistant: transformer architecture, pre-training and post-training, retrieval, and calibrated self-evaluation. Unit 2 does

not imply that every product uses every technique.

[L1] Vaswani, A. et al. (2017). "Attention Is All You Need." *Advances in Neural Information Processing Systems*, 30. arXiv 1706.03762. https://arxiv.org/abs/1706.03762 (accessed 2026-07-24). Introduced the Transformer architecture. Its attention mechanism combines information from different token positions to produce context-dependent representations. This citation supports the architecture sketch in Unit 2, not a claim that attention provides a truth check or a human-readable explanation.

[L2] Ouyang, L. et al. (2022). "Training Language Models to Follow Instructions with Human Feedback." *Advances in Neural Information Processing Systems*, 35, 27730–27744. arXiv 2203.02155. https://arxiv.org/abs/2203.02155 (accessed 2026-07-24). Shows one influential post-training pipeline: supervised demonstrations followed by preference data and reinforcement learning from human feedback. It supports the distinction between next-token pre-training and training an assistant to follow instructions.

[L3] Lewis, P. et al. (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." *Advances in Neural Information Processing Systems*, 33, 9459–9474. arXiv 2005.11401. https://arxiv.org/abs/2005.11401 (accessed 2026-07-24). Combines a pre-trained generator with an external document index. It supports Unit 2's distinction between generation from model parameters and a system that also retrieves inspectable sources.

[L4] Kadavath, S. et al. (2022). "Language Models (Mostly) Know What They Know." arXiv 2207.05221. https://arxiv.org/abs/2207.05221 (accessed 2026-07-24). Finds that specially elicited or trained self-evaluation scores can discriminate correct from incorrect answers on some evaluated tasks, with calibration and out-of-distribution limits. This is why Unit 8 says ordinary confident wording is not a correctness defence while avoiding the stronger, false claim that every model-derived confidence measure carries no information.

37.6 Second-order reasoning

Background for course/second-order.md and Unit 9. Read that page first: it marks the boundary between the established results below and the module's own synthesis, and the module's claims discipline applies to its own capstone exactly as it applies to everything else.

[T2] Tarski, A. (1936). "Der Wahrheitsbegriff in den formalisierten Sprachen." *Studia Philosophica*, 1, 261–405. The undefinability of truth: under the theorem's formal conditions, arithmetical truth for the object language is not definable within that same language. This is a scoped result about formal languages. Applying it to a language model, a person, or an organisation requires a separate argument and is not a corollary of Tarski's theorem.

Publication history, checked. Volume and page range are **established**: the German text is *Studia Philosophica* vol. 1, pp. 261–405, Leopold Blaustein's translation of the Polish original with a postscript added (Stanford Encyclopedia of Philosophy, "Alfred Tarski", bibliography, https://plato.stanford.edu/entries/tarski/, accessed 2026-07-23; that bibliography lists the German text as 1935, which is the datum the contested-year discussion below turns on). The **year is contested**, and the disagreement is recorded here rather than settled. The volume is dated 1935 and the Stanford Encyclopedia of Philosophy cites it as 1935; Peter Milne, "On the Year of Publication of Tarski's 'Der Wahrheitsbegriff in den formalisierten Sprachen' ", *History and Philosophy of Logic*, 46(2), 273–286 (DOI 10.1080/01445340.2024.2334173, published online 26 April 2024), argues from correspondence that the journal issue did not appear until 1936, and that what circulated in late 1935 were preprints lacking two corrections and a short addendum. The module cites 1936; a reader who finds 1935 elsewhere has not found an error.

The other versions, and why a page number depends on which one. The Polish original is *Pojęcie prawdy w językach nauk dedukcyjnych*, Warsaw: Nakładem Towarzystwa Naukowego Warszawskiego, 1933, VII + 116 pp., in the series *Prace Towarzystwa Naukowego Warszawskiego, Wydział III: Nauk Matematyczno-Fizycznych*, no. 34 (catalogue record, Kujawsko-Pomorska Biblioteka Cyfrowa, permalink https://kpbc.umk.pl/dlibra/publication/265521/edition/262893, accessed 2026-07-23, which confirms the series title, the Warsaw imprint and the VII + 116 pagination; the

"Wydział III" number 34 as cited by the Stanford Encyclopedia of Philosophy). The English translation, "The Concept of Truth in Formalized Languages", is in *Logic, Semantics, Metamathematics*, 2nd edn 1983 [1956], pp. 152–278. Three texts, three paginations: a page reference to "Tarski" means nothing unless it says which one.

[T3] Gödel, K. (1931). "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I." *Monatshefte für Mathematik und Physik*, 38, 173–198. The first incompleteness theorem. Closely related to [T2]; the two are frequently conflated in secondary accounts, and Unit 9 should keep them distinct.

[T7] von Foerster, H. (2003). *Understanding Understanding: Essays on Cybernetics and Cognition.* New York: Springer-Verlag. Collected essays from the tradition that includes the observer in the system being described. Confirmed against the book's own front matter: "© 2003 Springer-Verlag New York, Inc.", ISBN 0-387-95392-2, xii + 362 pp., with the printing line "9 8 7 6 5 4 3 2 1" — the first printing, and there is no later edition. The Library of Congress CIP data on the same page carries the year 2002 (`Q315.5 .V64 2002`), which is why some catalogues give 2002; the imprint year is 2003. Chapter and page references used by the module are to this printing. The 2003 Springer edition is also online with per-chapter DOIs under the stem `10.1007/0-387-21722-3` (e.g. chapter 14, "Ethics and Second-Order Cybernetics", pp. 287–304, DOI `10.1007/0-387-21722-3_14`; the chapter DOI, its title, page range and 2003 year confirmed via the Crossref record, accessed 2026-07-23).

[T8] Maturana, H. R. and Varela, F. J. (1980). *Autopoiesis and Cognition: The Realization of the Living.* Boston Studies in the Philosophy of Science, vol. 42. Dordrecht, Holland: D. Reidel. The companion reference for the same tradition. Confirmed against the book's own copyright page: published by D. Reidel Publishing Company, P.O. Box 17, Dordrecht, Holland; series *Boston Studies in the Philosophy of Science*, vol. 42; copyright 1980; ISBN 90-277-1015-5 (cloth) and 90-277-1016-3 (paperback). The same 1980 volume is reprinted by Springer under the stable DOI `10.1007/978-94-009-8947-4`, whose Crossref record (accessed 2026-07-23) confirms the title, the 1980 date, ISBN 978-90-277-1016-1 and the Boston Studies series — the Crossref record uses the later series name "Boston Studies in the Philosophy and History of Science", the same renaming noted below. No edition statement is printed — this is the first English book publication. Sold and distributed in the USA and Canada by Kluwer Boston Inc., Hingham, MA, which is why several catalogues record the place of publication as Boston; the imprint is Dordrecht. The series was later renamed *Boston Studies in the Philosophy and History of Science*, and records that use the newer name for volume 42 are describing the same book. The second of the two essays, "Autopoiesis: The Organization of the Living" (from p. 63), first appeared in Spanish as *De Máquinas y Seres Vivos*, Editorial Universitaria S.A., 1972.

[T9] Madaan, A. et al. (2023). "Self-Refine: Iterative Refinement with Self-Feedback." *Advances in Neural Information Processing Systems*, 36. Experiments on seven tasks found that iterative feedback and revision using the same language model could improve outputs over one-step generation. This is dated, task-bounded evidence that same-model revision can be useful; it does not make the revision independent verification. Primary paper: https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

A note on how these are used. [T2] and [T3] are established results with settled citations; they are stated as established. [T7] and [T8] are named as a research tradition rather than as results. [T9] is dated empirical evidence, not a general guarantee about self-correction. Unit 9's checkpoint asks the learner to sort these categories themselves, so the categories must be visible in the material rather than smoothed into a single voice.

37.7 Second-order cybernetics

Background for `course/second-order.md` and Unit 9, which cite these as [C1] to [C7]. They are the sources for the cybernetic tradition itself, as distinct from the formal results at [T2] and [T3]. All of it is established, published material, and none of it is a claim about language models.

[C1] von Foerster, H. — second-order cybernetics, the cybernetics of *observing* systems. The distinction between the cybernetics of observed systems, where the observer stands outside, and the cybernetics of observing systems, where the observer is included in what is described, is formulated in his work. The usual English source is the collected papers *Understanding Understanding: Essays on Cybernetics and Cognition*, New York: Springer, 2003 — the same volume listed above as **[T7]**. Cite [C1] for the first-order/second-order distinction and [T7] for the volume; they are one source under two uses.

The formulation, located. It is stated in von Foerster's own words in "Ethics and Second-Order Cybernetics", chapter 14 of [T7] (DOI 10.1007/0-387-21722-3_14, Crossref record accessed 2026-07-23), **pp. 287–304** — an interview with Yveline Rey, first published in French in *Systèmes, Éthique, Perspectives en thérapie familiale*, ed. Y. Rey and B. Prieur, Paris: ESF éditeur, 1991, pp. 41–55, per the chapter's own source note. On **p. 303**: "I would say, first-order cybernetics is the cybernetics of observed systems, while second-order cybernetics is the cybernetics of observing systems." The first half of the pair is given separately on p. 299: "the whole conceptual machinery of 'early' cybernetics, first-order cybernetics; or as I would say, the cybernetics of observed systems." Cite one of those two pages; do not attribute the formulation to the volume at large. (Chapter 13 of the same volume, pp. 283–286, is von Foerster's own "Cybernetics of Cybernetics", which is a different text from Mead's at [C2].)

The Biological Computer Laboratory. Much of the work was done there. It was founded by von Foerster at the University of Illinois at Urbana-Champaign in **1958** and closed in **1976** — "founded by Professor Heinz von Foerster in 1958, made Illinois an international center of cybernetics research until the lab's demise in 1976" (Illinois Distributed Museum, University of Illinois, https://distributedmuseum.illinois.edu/exhibit/biological_computer_labor accessed 2026-07-23). The 1958 founding is confirmed independently by the University of Illinois Archives: "he established the Biological Computer Laboratory in 1958" (https://archives.library.illinois.edu/von-foerster-and-the-bcl/, accessed 2026-07-23). Give 1958–1976 rather than "the late 1950s to the mid 1970s".

[C2] Mead, M. (1968). "Cybernetics of Cybernetics." In H. von Foerster, J. D. White, L. J. Peterson and J. K. Russell (eds), *Purposive Systems: Proceedings of the First Annual Symposium of the American Society for Cybernetics*, New York: Spartan Books, pp. 1–11. Commonly cited as the point at which the reflexive turn — cybernetics applied to itself — was named. The address is associated with the founding of the American Society for Cybernetics. The module states only that it is *commonly cited* as that point, which is what the available sources support. The venue, editors and year are given in this form by the American Society for Cybernetics itself (https://asc-cybernetics.org/CofC/wp-content/uploads/2010/08/mead.html, accessed 2026-07-23, which prints the citation "Mead, M (1968) Cybernetics of Cybernetics, in Foerster, H von, White, J, Peterson, L and Russell, J (eds) (1968) *Purposive Systems*, New York, Spartan Books"); the page range is given by P. Pangaro, "Why do we want to live in cybernetics?", *Cybernetics and Human Knowing*, 23(2), 2016, 9–21, whose reference list reads "Mead, M. (1968). Cybernetics of cybernetics. In von Foerster, H., White, J. D., Peterson, L. J., & Russell, J. K. (Eds.), Purposive Systems (pp. 1–11). New York: Spartan Books", and who quotes Mead at pp. 9 and 10 — in-text page citations that fall inside the stated range. *Contested: the occasion of the address.* The society's first annual symposium is usually dated 26–27 October 1967 at the National Bureau of Standards, Gaithersburg, Maryland, but descriptions of the volume also place the symposium itself in 1968. The module's claim rests only on the printed text, whose date, 1968, is not in dispute. Do not assert a delivery date in the materials.

[C3] Maturana, H. R. and Varela, F. J. (1980). *Autopoiesis and Cognition: The Realization of the Living.* The same volume as **[T8]**: cite [C3] for the observer, [T8] for autopoiesis. Publisher, place, series and edition are established and are recorded at [T8] (Boston Studies in the Philosophy of Science, vol. 42; Dordrecht: D. Reidel, 1980).

The formulation, located — and corrected. It belongs to Maturana's essay "Biology of Cognition", first issued as Biological Computer Laboratory Research Report BCL 9.0, Urbana, IL: University of Illinois, 1970, and reprinted in this volume at pp. 5–58. In section III, "Cognitive Function in General", under the sub-heading "The Observer", numbered statement (1), on **p. 8**, the sentence reads:

> Anything said is said by an observer. In his discourse the observer speaks to another observer, who could be himself; whatever applies to the one applies to the other as well.

Checked against a page facsimile of the 1980 Reidel text, and against the 1970 BCL report, which has the same wording.

So the module's own working phrase was slightly wrong. The commonly quoted form *everything said is said by an observer* is **not** the wording on p. 8. It is the title of a later Maturana essay — "Everything said is said by an observer", in W. I. Thompson (ed.), *Gaia: A Way of Knowing: Political Implications of the New Biology*, Lindisfarne Press, 1987. Both forms are Maturana's; they are not the same text. Quote "Anything said is said by an observer" and cite p. 8 of this volume, or quote "Everything said is said by an observer" and cite the 1987 essay by title. Do not attach the 1987 wording to the 1980 page. Unit 7 asks learners to watch a claim drift as it is repeated; this is the same thing happening inside this file, and it is worth saying so in the teaching.

[C4] The Macy Conferences on Cybernetics (1946–1953). A series of ten meetings funded by the Josiah Macy Jr. Foundation and chaired by Warren McCulloch, deliberately mixing mathematicians, engineers, neurophysiologists, anthropologists and psychiatrists. The standard secondary history in English is **Heims, S. J. (1991),** *The Cybernetics Group*, Cambridge, MA: MIT Press. The surviving transcripts have been edited by Claus Pias, which is the route to the primary material.

The Pias edition — there are two, and they are not the same book. The bilingual original is Claus Pias (ed.), *Cybernetics — Kybernetik: The Macy-Conferences 1946–1953*, Zürich and Berlin: Diaphanes, in **two volumes**: vol. 1, *Transactions / Protokolle*, 2003, 734 pp., ISBN 978-3-935300-35-3; vol. 2, *Essays and Documents / Essays und Dokumente*, 2004, 512 pp., ISBN 978-3-935300-36-0 (records of the Deutsche Nationalbibliothek). The later English-only edition is Claus Pias (ed.), *Cybernetics — The Macy Conferences 1946–1953: The Complete Transactions*, Zurich and Berlin: Diaphanes, 2016, first printing, 734 pp., ISBN 978-3-03734-598-6 (Deutsche Nationalbibliothek). It is distributed in North America by the University of Chicago Press, whose catalogue page is `https://press.uchicago.edu/ucp/books/book/distributed/C/bo23348570.html` (accessed 2026-07-23); note that that page lists its printing as ISBN 978-3-03734-607-5, 752 pp., differing from the Diaphanes printing above, so cite the exact printing consulted. Cite whichever was actually consulted: "the Pias edition" on its own is ambiguous, and the two have different contents and different pagination.

The count of ten. Ten is the **core series** and does not include the supplementary meetings. The ten conferences on "Circular Causal and Feedback Mechanisms in Biological and Social Systems", later "Cybernetics", ran from 8–9 March 1946 to 22–24 April 1953 (American Society for Cybernetics, Macy conference summary, `https://www.asc-cybernetics.org/foundations/history/MacySummary.htm`, accessed 2026-07-23, which lists all ten with dates — first conference 8–9 March 1946, tenth 22–24 April 1953). The separately sponsored related meetings are not among them: a 1946 sociological sub-conference organised by Bateson, and the New York Academy of Sciences symposium on teleological mechanisms, whose papers appeared as *Annals of the New York Academy of Sciences*, 50(4), 1948, 187–278. The Pias volumes use the same count of ten. "A series of ten meetings" as written above is therefore correct, read as the core series.

[C5] Bateson, G. (1972). *Steps to an Ecology of Mind: Collected Essays in Anthropology, Psychiatry, Evolution, and Epistemology.* San Francisco: Chandler Publishing Company. Bateson was a Macy participant [C4], and this collection is the standard source for his part in the tradition. This entry is the reference for the Bateson attribution in `course/second-order.md` and Unit 9, which previously carried no source at all.

The 1972 imprints, and why the pagination does not carry. The first edition is Chandler Publishing Company, San Francisco, 1972, 545 pp., ISBN 0-8102-0447-9, LCCN 75-169581. A Ballantine Books paperback, New York, 1972, 541 pp., appeared the same year by arrangement with Chandler. Records that merge the two — giving "Ballantine Books; Chandler Pub. Co., New York, 1972" in one line — are conflating imprints, and the two differ in pagination. The current reissue is University of Chicago Press, 2000, 565 pp., ISBN 978-0-226-03905-3, with a new foreword by Mary Catherine Bateson (publisher's page

https://press.uchicago.edu/ucp/books/book/chicago/S/bo3620295.html, accessed 2026-07-23, which confirms the 2000 year, the 565 pp., the ISBN and the foreword). The 1972 Chandler first edition carries LCCN 75-169581 and ISBN 0-8102-0447-9; that LCCN and OCLC 258014 are recorded in OpenLibrary edition OL17854018M(https://openlibrary.org/books/OL17854018M, accessed 2026-07-23), though that OpenLibrary record itself merges the Chandler and Ballantine imprints — the exact conflation warned against just above — so it is cited for the identifiers, not the pagination. Cite the edition actually used and give its pagination; page numbers do not transfer between these three.

[C6] Gödel (1931), first incompleteness theorem. Listed in full above as **[T3]** and not repeated here. course/second-order.md cites it as [C6] and Unit 9 cites it as [C6]; [T3] is the same paper under its earlier label.

[C7] Tarski (1936), undefinability of truth. Listed in full above as **[T2]** and not repeated here, including the record of that volume's contested date and of the three separately paginated versions of the text. course/second-order.md cites it as [C7], and Unit 9 now cites [C7] directly; [T2] is the same paper under its earlier label. Anything said about the paper's year or pagination belongs at [T2], where the disagreement is set out, and not restated here.

A note on how these are used. [C1] to [C5] support one claim and no more: that second-order cybernetics is a named research tradition with a literature and a history going back to the 1940s. They are not evidence about what any AI system does, and course/second-order.md says so on its own account.

37.8 Gothic and Romantic context

Supporting the visual-register decision recorded for the module and the short passage in Unit 1. These entries exist so that the register carries a citation rather than an atmosphere.

[G1] Marchand, L. A. (1957). *Byron: A Biography.* 3 vols. New York: Alfred A. Knopf. The standard scholarly biography, and the reference for Byron's departure from England in 1816 and his subsequent life abroad. Ada Lovelace was born in December 1815; Byron left in 1816 and did not see her again.

Confirmed from the book itself (volume III front matter, contents and index): Alfred A. Knopf, New York, 1957, "a Borzoi book", first edition, LC catalog card number 57-7547, published simultaneously in Canada by McClelland & Stewart. The 1957 Knopf edition is catalogued at OpenLibrary as edition OL42986375M (https://openlibrary.org/books/OL42986375M, accessed 2026-07-23), under the work record for Marchand's *Byron: A Biography* (LC card 57-7547 is from the volume itself, above). **Three volumes, continuously paginated** — vol. 1, chapters I–XII, pp. 3–476; vol. 2, chapters XIII–XXIII, pp. 477–942; vol. 3, chapters XXIV–XXX, pp. 943–1264, followed by notes, a list of sources and the index. Because the pagination is continuous, a page reference identifies the volume by itself.

The 1816 departure falls in **volume 2**, at the end of chapter XV, "1816 The Separation" (pp. 563–608), running into chapter XVI, "1816 Switzerland" (from p. 609). The index is explicit: "Dover, 607–8", and under Byron, "preparations for going abroad, 602; engaged Dr. Polidori, 602–3; … signed deed of separation, 605; … to Dover in Napoleonic coach, 607; … at Churchill's grave, 607–8; … through lane of spectators to boat, 608; … end of an epoch, 609; … at Ostend, 610". Cite **pp. 605–610** for the departure as a whole, or pp. 607–8 for Dover.

[G2] The Villa Diodati, summer 1816. The gathering on Lake Geneva at which Byron, Percy Bysshe Shelley, Mary Godwin (later Mary Shelley), Claire Clairmont and John Polidori spent part of the summer, and from which the ghost-story challenge that produced [G3] and [G4] is conventionally dated. The nearest thing to a primary account is Mary Shelley's own introduction to the 1831 edition of *Frankenstein*; Polidori's diary is the other contemporary source.

A citable edition of the diary. Polidori, J. W., *The Diary of Dr. John William Polidori, 1816, Relating to Byron, Shelley, etc.*, edited and elucidated by William Michael Rossetti, London: Elkin Mathews, 1911. This is the standard text, with the standing caveat that Rossetti worked from the copy expurgated by

Polidori's sister Charlotte, the original having been destroyed. Reissued in the Cambridge Library Collection (Literary Studies), Cambridge University Press, 2014, 240 pp., ISBN 978-1-108-07228-1, DOI 10.1017/CBO9781107444713. The 1911 text is in the public domain and is available complete as Project Gutenberg ebook 55017 (https://www.gutenberg.org/ebooks/55017, accessed 2026-07-23; confirmed as Rossetti's edition of Polidori's 1816 diary), which is the copy the dates below were checked against.

Dates that are established, from the diary and from the Shelley-Godwin Archive's *Frankenstein* chronology (which follows Nora Crook's "Chronology of Life and Works"):

- Byron sailed from Dover on the evening of **25 April 1816**, reaching Ostend on 26 April. Polidori heads the embarkation entry "April 26"; Rossetti's note corrects this to 9 p.m. on 25 April on the evidence of a published letter of Byron's, and this small conflict between a diarist and a letter is itself worth showing to learners.
- The Shelley party — Percy Bysshe Shelley, Mary Godwin and Claire Clairmont — left London on 2 May and reached Lake Geneva on **13 May 1816**, first at the Hôtel d'Angleterre in Sécheron, moving "towards the end of May" into the Campagne Chapuis, also called Campagne Mont Alègre, near Cologny, about eight minutes' walk below the Villa Diodati (Rossetti's note).
- Byron and Polidori reached Sécheron on **25 May 1816**. The Diodati lease was signed on **6 June**: "we got the paper making us masters of Diodati for six months to November 1 for 125 louis". They moved in on **10 June 1816**.
- The Shelley party left Geneva on **29 August 1816**.

Contested: the date of the ghost-story challenge. The conventional date is 16 June 1816. The Shelley-Godwin Archive chronology (https://shelleygodwinarchive.org/contents/frankenstein/franke
chronology/, accessed 2026-07-23) declines to fix it, recording three possibilities — before 15 June, on 15 June, or on 16 June — and noting only that 16 June is "the date often given". The diary fixes an outer bound rather than a date: on 17 June, "The ghost-stories are begun by all but me"; on 18 June, "Began my ghost-story after tea", the same evening as Byron's recitation of *Christabel* and Shelley's fit. Rossetti's note on the 17 June entry points out that this contradicts the standard sequence, in which the *Christabel* incident *preceded* the challenge — "The ghost-stories … had already been begun … not later than June 17, whereas the *Christabel* incident happened on June 18." **Give a range, not a night**, and do not write "on the night of 16 June" as settled fact.

[G3] Shelley, M. (1818). *Frankenstein; or, The Modern Prometheus. In Three Volumes.* London: Printed for Lackington, Hughes, Harding, Mavor, & Jones, 1818. Published anonymously. The revised 1831 edition carries the author's introduction describing the Villa Diodati summer, and differs from the 1818 text in ways that matter if the novel is quoted rather than merely cited.

Imprint, confirmed. Three volumes; London; "Printed for Lackington, Hughes, Harding, Mavor, & Jones"; 1818. The imprint is given in this form by the New York Public Library's record for its Berg Collection copies and by the catalogue record for the Internet Archive facsimile of volume I (https://archive.org/details/shelleyfrankenstein01, accessed 2026-07-23, whose record gives the bracketed printer "Printed [by Macdonald and Son] for Lackington…"), and the publisher and year are given the same way by the Frankenstein Variorum, a scholarly digital edition (https://frankensteinvariorum.org/about, accessed 2026-07-23) collated against a photographic facsimile of the 1818 text. The NYPL record adds that the print run was about 500 copies, of which it holds seven. The Internet Archive record supplies the printer in brackets — "Printed [by Macdonald and Son] for Lackington…" — i.e. as a cataloguer's addition rather than from the title page, so do not print it as if it were on the book. The publication date of 1 January 1818 is standardly given but was not confirmed from a primary source here; if the module needs the day, check it before printing it.

[G4] Polidori, J. W. (1819). "The Vampyre; A Tale." *The New Monthly Magazine and Universal Register*, vol. XI, no. 63 (1 April 1819), pp. 195–206. London: Henry Colburn. First published in April 1819 and initially, and wrongly, attributed to Byron — which is itself a usable example of an attribution that was widely repeated before it was checked.

Checked against the scanned issue, page by page. The scanned issue used is the scan-backed Wikisource edition of *The New Monthly Magazine and Universal Register*, Volume XI, No. 63 (https://en.wikisource.org/wiki/The_New_Monthly_Magazine_and_Universal_Re accessed 2026-07-23), transcribed from the Wikimedia Commons scan of Volume XI (file *The New Monthly Magazine and Universal Register - Volume 011.djvu*). The running head on the first page of the tale reads "1819.] *The Vampyre; a Tale by Lord Byron.* 195"; the tale ends on p. 206, where "A Pedestrian Tour round Florence" begins under the running head "[April 1,", which also fixes the issue date. The volume and number are confirmed by the signature line on p. 193: "New Monthly Mag.—No. 63. Vol. XI". The tale is preceded, from p. 193, by the unsigned "Extract of a Letter from Geneva, with Anecdotes of Lord Byron", which supplies the Villa Diodati frame; booksellers' catalogues that cite the piece as "pp. 193–194" are citing that introduction rather than the tale, so **pp. 195–206** is the range for the tale itself. The false attribution is on the title as printed, not merely in circulation: "a Tale by Lord Byron". A disclaimer by Polidori is reported in the following month's issue; that follow-up was not verified here.

The module makes no claim that this literary inheritance influenced Lovelace's own work. That the family connection exists is established; what she made of it is not a claim this module supports.

37.9 Music quoted

Unit 9 carries short song quotations as part of its register. They are recorded here so that each one has a source and a rights position rather than appearing as unattributed text on a slide.

Rights position. The position is to **reference the song and quote a minimal set of lyrics under the educational quotation exception**, rather than to clear rights or to drop the material. The lyrics remain in copyright; the module reproduces no work in whole or in substantial part. The position is recorded in full in docs/licensing.md, which fixes four conditions that every quotation in the module must meet:

- **short** — no more than a line or two from any single work;
- **used for criticism, review, or illustration** — carrying an argument, never decoration;
- **sufficiently acknowledged** — title, artist, and a full entry on this page;
- **fair dealing** in the relevant jurisdiction.

A quotation that fails any of the four does not go in, and the argument never depends on a lyric, so removing one costs the module nothing. By the module's own rule the register may carry an argument and may never substitute for one. The per-quotation sign-off is item **B11** in docs/instructor-review-log.md.

[Q1] Lacrimosa. "Alles Lüge." Title track of the single *Alles Lüge*, Hall of Sermon, Switzerland, 1993. Referenced, and quoted minimally under the educational quotation exception. *Not from the album, despite the usual attribution.* The band's own discography lists *Alles Lüge* (1993) as a single, separately from the studio album *Satura* (1993), and the original six-track *Satura* — "Satura", "Erinnerung", "Crucifixio", "Versuchung", "Das Schweigen", "Flamme im Wind" — does not contain it (band's own discography, https://lacrimosa.com/en/project/alles-luege-1993/, accessed 2026-07-23; MusicBrainz release group for the 1993 single, type Single, https://musicbrainz.org/release-group/1e938aaf-09e1-37d6-b004-2af4e6213d91, and for the album *Satura*, https://musicbrainz.org/release-group/d320ced3-5fa5-3725-94ca-8be674182bd0, both accessed 2026-07-23; the single-versus-album distinction and the six-track original *Satura* are corroborated by both, so the 1993 single date is not single-sourced). The song was added to later *Satura* reissues as a bonus track, which is where the common "from *Satura*" attribution comes from. Cite the 1993 single. MusicBrainz also dates the recording's first appearance to a label sampler, *German Mystic Sound Sampler, Volume III* (1992); that is single-sourced, is not relied on, and should not be printed without a second source.

[Q2] Einstürzende Neubauten. "Sabrina." Opening track of the album *Silence Is Sexy*, Mute (UK) / Potomak (Germany), 2000. Referenced, and quoted minimally under the educational

quotation exception. Release date 3 April 2000 on the UK Mute release (MusicBrainz release https://musicbrainz.org/release/e0ceeda4-fa04-30f0-b511-75117b673c7d, accessed 2026-07-23, giving date 2000-04-03, country GB, label Mute, catalogue CD STUMM 182, opening track “Sabrina”). This exact release date rests on that single database and is recorded here as single-sourced, not independently corroborated.

37.10 Citing in module materials

In-text, cite by bracket label: `(Lovelace 1843, Note G)` for [P2], `[S1]` for a technical point. Every citation must resolve to an entry on this page. If it does not, it is a defect.