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, orchestrated by Baba Yaga V10

v0.0.1-3-g46a489c – 2026-07-28

Contents

AI Literacy: Course Handbook (guidance edition)

This edition carries the marking guidance, the worked answers and the checkpoint data. Open it after attempting the work: it is the same guidance the website shows you once you have answered. The edition to work from is `handbook.pdf`*.*

Course Brief:

The program often called the first ever written computed Bernoulli numbers — a description Unit 1 shows is contested, which is a good place to start. 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.

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

Which path through this module

Six routes. They differ in how much of the module you take and in how much mathematics and code you want; none of them requires programming. Pick one, and change your mind later if it is not the right one – nothing is lost by switching.

Route	Units	Time
The short route	0, 2, 3, 6, 7, 8	201 min (3.4 h)
The core route *(the default)*	0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 17	420 min (7.0 h)
The core route with the code work	0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 17	420 min (7.0 h)
The route without the mathematics	0, 1, 2, 3, 4, 5, 9, 10, 12, 13, 17	349 min (5.8 h)
The responsible-use route	0, 3, 12, 13, 14, 15, 17	207 min (3.5 h)
The complete route	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18	618 min (10.3 h)

The short route. You want the module's actual skill and cannot find seven hours. You keep the part where you build a reference and check something against it, and you give up the parts about writing the work up and defending it.

What it leaves out: Specification and re-prompting, the disclosure and defence you would submit, everything about choosing and working with these systems, and the closing argument about where the observer stands.

The core route. Everybody, unless you have a specific reason to be on another path. It assumes no computer science and no programming, it covers what these systems are for and how to work with them as well as how to check them, and it is the path the time budget is set for.

What it leaves out: The second worked case on sources, agents and automated actions, rights and attribution, representation and accessibility, the historical bug, and the closing second-order argument. The optional extensions and the Python lab work are also outside it.

The core route with the code work. You write code, or you are on a computing programme and want the version that engages with testing rather than only with judgement. Same units as the core path, taken through the Python route with the lab extensions. Recommended if you program, required of nobody.

What it leaves out: The same units the core path misses. The code work teaches no new verification idea; it gives a second way to practise the same one, with the machine doing the comparison.

The route without the mathematics. You do not want to derive Bernoulli numbers by hand and would rather not meet summation or binomial coefficients at all. You still read about Ada's procedure, because it is the demonstration the whole module refers back to, and you do the full method on a case with no arithmetic in it – checking a referenced summary against what its sources actually say.

What it leaves out: The Bernoulli derivation and the verification lab, which is where the method is proved on a case with a decidable answer rather than described. You get the method, on a case where the judgements are yours – and you should know that is the trade you have made.

The responsible-use route. You want the duties rather than the mechanics – consequence and escalation, what leaves your device, who owns the output, and what a generated summary does to the people it describes. Short, and meant to be taken beside another path rather than instead of one.

What it leaves out: Everything about how these systems work and how to check their output. This is why it is a companion path rather than a whole one.

The complete route. You want everything the module offers – all nineteen units, both worked cases, the optional extensions and the lab work. Just over ten hours, and it is meant to be taken over several sittings rather than at once.

What it leaves out: Sustainability beyond the resource section in the tool-choice unit, and industry practice, both of which are signposted rather than taught. Institutional assessment policy, employer policy and jurisdiction-specific legal advice are out of scope by decision – the module states the boundary and tells you to check your own rules.

Chapters are numbered by unit, so a route is a list of chapter numbers to read. The units you skip are not prerequisites for the ones you keep.

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

0.1 Reading

Reading time: about 12 minutes. Activity: about 10 minutes.

0.1.1 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.

0.1.2 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 17. Material that demanded a disclosure it was unwilling to make itself would be asking for trust it does not extend.

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.

One thing is worth saying plainly at the start. **Disclosure is the easy half**, and using AI is not something to apologise for. **The harder half is being able to stand behind the work**: to say you understand it, and that you have evidence it is right.

Keeping “how this was made” apart from “why this is right” is a skill this module returns to repeatedly, and it spends most of its effort on the second one.

0.1.3 The learning contract

One sentence, and it does not have exceptions:

> Using an AI system does not remove your responsibility for the output you put your name to.

Hand something on — submit it, publish it, send it, act on it — and if it is wrong you are answerable for having used 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 is the module's whole subject, and it is worth seeing why it is put this way round. Whether a tool was used is often impossible to establish from the work — Unit 17 goes into that — so it makes a poor foundation for anything. Whether the work is right, and who is answerable for it, can both be established. So the question that runs through every unit is:

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

There is a separate question about **permission**: some material you may not send anywhere, and some tasks have rules about what help is allowed. Where such a rule exists you find out by asking, not by inspecting the finished work, and Units 13 and 17 deal with each in turn. It is a real constraint and a different one. It never answers the question above.

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 *at the level your claim needs*, identify a suitable independent check, or interpret a failed check.

That third qualification matters. You do not have to explain a compiler, or a statistical library, to use one — plenty of reliable tools are opaque inside and checkable outside. What you cannot do is claim more than your check supports while understanding neither. 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 3 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.

0.1.3.1 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.

0.1.4 How the module is organised

Nineteen short units. You do not take all of them – the route you choose picks out a path through them, and the routes page says how long each one is. The middle units run about 25 to 45 minutes each; the opening units are shorter. 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.

(Note: The video and the reading cover equivalent content and follow the same learning path. You can choose to watch the video, read the text, or both—you won't miss any core concepts by picking one format over the other.) - `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 17 assembles it. Start collecting from this unit.

0.1.5 What you need before you start

Stated plainly, because the module has assumed some of this without saying so.

What is assumed throughout. Exact fractions — adding, multiplying and comparing them, and being willing to keep `-1/30` rather than turn it into `-0.0333`. Simple algebra: rearranging an equation to get one term on its own.

What is assumed in one place only. The Bernoulli unit asks you to work a value out by hand from a recurrence, which means reading a summation sign and a binomial coefficient. That is the only unit that does, and if it is unfamiliar, `course/prerequisites.md` works both through on this module's own numbers before you get there.

What is not assumed anywhere. No calculus. No statistics. No prior programming — the table-based route runs the whole module without it. No prior use of an AI system, and no account with any particular provider.

What you need to have. A browser. Python 3 as well, only if you take the code route, and nothing to install beyond it.

If the mathematics is the part you would rather avoid entirely, that is a supported choice rather than a lesser one: the route without the mathematics reaches the same method through a case that has no arithmetic in it. The routes page lists it.

0.1.6 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 7	Ask for Bernoulli values or use the supplied outputs.	Ask for a Python function or use a supplied implementation.
In Unit 8	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 `cp08-bernoulli` checkpoint and the shared `cp08-diagnose`; the table route uses `cp08-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.

0.1.7 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). Nothing here is marked and nothing is collected; if you are taking the module inside a course that says otherwise, that course's rules are the ones that apply. 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 8 code checkpoint runs only in Python.

```
python3 selfcheck.py run <id>  # or in the browser
```

The Python route lives in `course/lab/` and needs nothing beyond a Python 3 installation — no packages to install. The browser route needs nothing at all: each unit's page ends with that unit's

checkpoints, and your progress page collects every checkpoint in the module together with what you have recorded so far. Both work from local files, make no network requests, and keep your answers in this browser. 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, and nobody is stopping you. Reading one before attempting it removes the diagnostic value of that checkpoint: the feedback is built to tell you which misunderstanding you have, and it cannot do that for an answer you did not form. Nothing is lost beyond that, and the guidance is there for afterwards.

0.1.8 If something does not work

This is a self-study module, so there is nobody watching for you to get stuck. Most difficulties have the same few causes:

- **A checkpoint will not run in the browser.** Some browsers block local storage on files opened directly from disk. Use the published course site rather than a downloaded copy, and take a portable backup either way.
- **The Python route will not start.** Nothing in the core module needs it. The browser route covers the same checkpoints, and the table-based work in Units 6, 8 and 16 needs no code at all.
- **You have no access to an AI system.** Every activity that asks you to generate something supplies a specimen instead. You will not be stuck, and the method is the same either way.
- **Something looks wrong in the material.** It may well be. The whole source is published — the link is at the foot of every page — so you can see how any claim was made, and the authors page carries an address for telling us. A module that asks you to check claims has to expect its own to be checked.

If you are taking this module inside a course, whoever is running it comes first: their rules on assessment and AI use are the ones that apply.

0.1.9 Before you go on

It takes about ten minutes and produces the first line of your evidence pack. Then answer the checkpoint at the foot of this page, and start Unit 2.

0.2 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.

0.2.1 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.

0.2.2 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.

0.2.3 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 3 gives you a way to make that decision systematically.

0.2.4 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. Open any unit page and scroll to the end: its checkpoints are there, and answering them records your progress. `progress.html` holds all of them at once. If you were given a link to the module website, use that; if you downloaded a release, open `dist/site/index.html` — it works straight from a file, with no internet connection.

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, sort it out now rather than at Unit 8, where the checkpoints start doing real work. The browser route needs no installation and is the quickest thing to try. If you are taking this module through an institution, ask whoever is running it; otherwise the contact address on the authors page will reach someone.

0.2.5 Now take the checkpoints

`cp00-contract` asks one question about the learning contract. **`cp00-what-this-needs`** asks what the module assumes of you — it is here rather than in Unit 6 so that you find out before you arrive at the derivation, and so that if the answer is unwelcome you can still choose a different route cheaply.

```
python3 selfcheck.py run cp00-contract
python3 selfcheck.py run cp00-what-this-needs  # or in the browser
```

Run them 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 17 asks you to look back at it and say honestly whether you kept it.

0.2.6 Activity Answers

0.3 Unit 0 answers and guidance

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

0.3.1 Marking your 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 17 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.

0.3.2 Five ways the sentence usually goes wrong

Nothing here is graded. What this sentence is for is to be the first entry in your evidence pack, and the thing you look back at in Unit 17. A strong one names an artefact at the level of a single claim, number, function or citation; states the method of the check, not just its intention; and implies a failure condition.

If yours matches a row below, the last column is the fix:

Pattern	Example	What to do instead
Category, not artefact	“I will check the facts.”	Say which fact, in which paragraph. Specificity is the whole exercise.
Check with no possible failure	“I will read it through carefully.”	Say what a bad outcome would look like. Reading is attention, not verification.
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 3 explains why this is not independent evidence.

Pattern	Example	What to do instead
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.

If your sentence fell into either of the last two, the distinction between provenance and correctness is worth settling before you start Unit 3, because the triage activity there assumes it.

0.3.3 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 the two routes ask the same question and give the same feedback.

What the checkpoint is testing is the distinction between disclosing how work was produced and being answerable for whether it is right. If you get it wrong, read the feedback before looking anything up: what it says about the option you chose is more useful than the right answer, because it names the confusion rather than replacing it.

Learners run it with either route:

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

0.3.4 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp00-contract` — The learning contract

Kind: choice.

You use an AI assistant to produce part of work that you submit, and it contains an error. What does this module’s learning contract require?

Options, numbered from zero:

0. You remain answerable for using the output and must be able to support it, even though other people or organisations may also have duties.
1. Only the AI vendor is answerable, since the tool produced the error.
2. No one is answerable, provided you disclosed that you used AI.
3. Only your instructor is answerable if they permitted AI use on the task.

Answer: `0`

Hint: Two different questions can be asked about a piece of work: how it was made, and whether it is right. Ask which of them each option is answering, and whether answering one settles the other.

Diagnostics, by the answer given:

- `1` — A vendor may have duties for design, safety, or claims about its system, but that does not remove your responsibility for material you chose to submit. Accountability can be shared; it is not transferred away from you.
- `2` — Disclosure and responsibility are different things. Disclosure tells the reader how the work was made; it does not transfer accountability for whether the work is right. You need both.
- `3` — Permission to use a tool does not make the instructor solely answerable for your output. Institutions and staff may have policy or review duties, while you remain responsible for material you submit.

Explanation: AI assistance changes how you produce work but does not remove your responsibility for material you submit. Accountability may be shared in a wider system. The oracle, checker, and defence help you support your own use of the output.

Checkpoint `cp00-what-this-needs` — What this module assumes of you

Kind: choice.

Unit 6 asks you to work out a Bernoulli number by hand from a recurrence, using exact fractions and a summation with binomial coefficients. Which statement is true of this module?

Options, numbered from zero:

0. You need to add and multiply exact fractions and to read a summation. Nothing beyond that, and no programming is required on any route.
1. You need to be able to program in Python. The table-based route is a reduced version for people who cannot.
2. You need calculus and some prior study of number theory.
3. You need no mathematics at all, because every calculation is done for you.

Answer: `0`

Hint: Look at what the module actually asks you to do rather than at what the topic sounds like it needs. One unit asks you to compute a value by hand; nothing anywhere asks you to write a program.

Diagnostics, by the answer given:

- `1` — No route requires programming. The table-based route runs the whole module without it and is a complete path rather than a reduced one – it verifies the same claims by comparing against a reference you build yourself, which is what verification means. If you would rather run code, the route with the code work exists and teaches no additional verification idea.
- `2` — Neither appears anywhere. The heaviest mathematics in the module is one derivation from a recurrence in Unit 6, using fractions and a binomial coefficient. If that is unfamiliar rather than unwelcome, course/prerequisites.md works it through on this module's own numbers before you get there.
- `3` — Unit 6 does ask you to compute a value by hand, and that is deliberate – the reference you check against later is worth more when you built it. If you would rather avoid the arithmetic entirely, the route without the mathematics reaches the same method through a case that has none, and it is a full path rather than a fallback.

Explanation: Exact fractions and simple algebra throughout, and one derivation in Unit 6 that uses a summation and a binomial coefficient. No calculus, no statistics, no prior programming, and no prior use of an AI system. Knowing this now rather than in Unit 6 is the point of asking: if the mathematics is unfamiliar, course/prerequisites.md covers it in advance, and if it is unwelcome, there is a route that does the whole method without it.

1 Unit 1 — What generative AI is for

1.1 Reading

Media for this unit:

- **Animation**: *How far away the reference sits* — this unit has an interactive version on the course website (`reference-distance.html`); the still below is one moment from it.

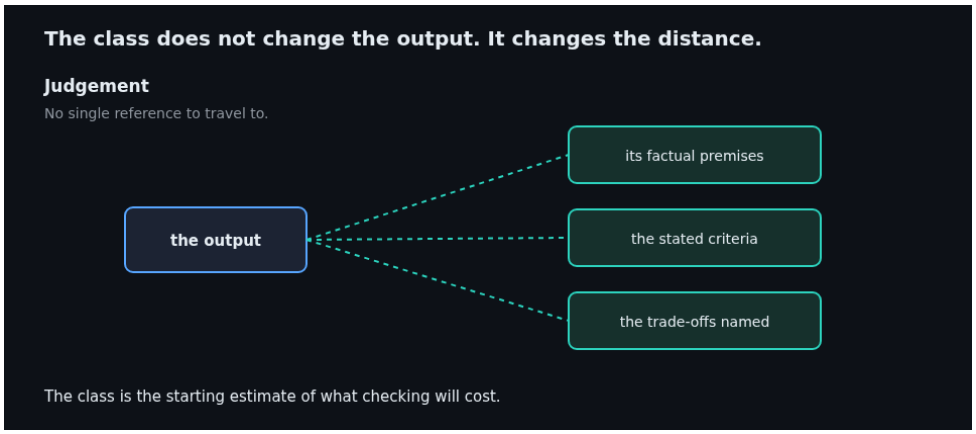

Figure 1: How far away the reference sits. The reference an output is checked against moves further away as the task class changes.

Reading time: about 10 minutes. Activity: about 12 minutes.

1.1.1 The question worth asking

“What is generative AI good at?” is a question that goes stale. Answer it with a list of products and the list is wrong within a year. Answer it with a list of tasks and you have described what the systems could do on the day you wrote it.

This unit answers a different question, one whose answer does not move:

> For this task, where would the evidence come from that the output is right?

That question has only a few possible answers, and they are the useful way to sort the work. Two tasks that look nothing alike — translating a paragraph and reformatting a table — sit in the same class, because in both cases you already hold the thing you would check against. Two tasks that look almost identical — “summarise this document” and “summarise the research on this topic” — sit in different classes, because in one case the document is in front of you and in the other you would have to go and find it.

The class decides what checking costs. That is the whole of this unit.

1.1.2 Five classes, by where the reference lives

A **reference** is whatever you would compare the output against to find out whether it is right. In Unit 6 you will build one by hand; in Unit 8 you will use it. Here we are only asking where it comes from.

Class	What you supply	Where the reference lives	What checking costs
Transformation	The content itself	In your hands already — it is your input	Low: compare output against input
Grounded answering	A source, plus a question about it	In the source you supplied	Low to moderate: read the source at the cited point
Generation to a specification	A statement of what would count as correct	In the specification you wrote	Moderate: test the output against the specification
Recall	A question only	Somewhere external, and you must go and find it	High: the check is a separate research task

Class	What you supply	Where the reference lives	What checking costs
Judgement	A question with no single determinate answer	No single right answer to compare against, but the premises and the criteria are checkable	Check the factual premises; assess the proposal against stated criteria

Read the last column downwards. It is not a ranking of usefulness. It is a ranking of *how much work you have left to do afterwards*, and it is the thing most people do not price before they start.

1.1.2.1 Transformation You supply the content and ask for it in a different form: translated, summarised, reformatted, restructured, rephrased for a different reader, turned from prose into a table.

The reference is your input. Every claim in the output should be traceable to something you handed over, so the check is a comparison you can make at your desk with no other material. The characteristic failure is not invention but **omission**: something in the input that did not survive the transformation. That is easy to miss precisely because the output reads as complete.

1.1.2.2 Grounded answering You supply a source — a document, a page, a specification — and ask a question about it. Some systems fetch the source themselves rather than being handed it [L3].

The reference is that source. The check is to read it at the point the answer relies on. Note what has and has not changed: attaching a source changes *where* you aim the check, not whether one is needed. Two questions remain live. Does the source say what the answer says it says? And is the source the right one to settle this question?

1.1.2.3 Generation to a specification You state what would count as correct, and ask for something that meets it. A routine that satisfies a stated contract; a document in a required structure; a test that fails under a named condition.

The reference is the specification. This is the class the middle of this module lives in: Unit 7 asks for a routine, Unit 8 checks it against a reference you built, and Unit 10 is about what happens to the output when you make the specification more precise. The characteristic failure is that the output meets the specification you wrote rather than the one you meant.

1.1.2.4 Recall You ask a question with no source attached and no specification: a date, an attribution, a requirement, a figure, who first proved something.

The reference is external and you do not have it. The generation step does not consult a record of what is true — Unit 3 explains the mechanism — so a recalled claim arrives with no evidence attached and no marker distinguishing the ones that happen to be right. Checking means finding an authoritative source, which is a research task of its own. Unit 18 is about doing that properly.

This is the class where the gap between how much a claim costs to produce and how much it costs to check is widest. Eight referenced claims can be generated in seconds and take an afternoon to verify.

1.1.2.5 Judgement You ask which of two designs is better, whether an argument is convincing, whether a piece of writing is any good.

There is no reference in the sense the other classes have one, because there is no single answer to be right about. That does not mean there is nothing to check. A judgement rests on factual premises, and those can be wrong: an argument that one design is cheaper depends on a claim about cost. It can also be assessed against things you can state — the objectives, the constraints, the evidence offered, the trade-offs, professional standards, whose interests are served. What you cannot do is settle it by comparison, because there is nothing to compare it against. So the output is a **proposal**, and treating a proposal as a finding is the error. A confident register here is a property of the text, not evidence about the question.

1.1.2.6 How this relates to the triage you meet next Unit 3 asks a second question about the same task: not where the reference lives but **what kind of evidence settles it** — something you can execute and compare, something you must look up in a record, or neither because an error would be cheap and visible.

The two cuts are not rivals, and mostly the first decides the second. A task with its reference in a specification is one you can check by executing and comparing. A task whose reference is a source, whether you supplied it or must go and find it, is one you check by reading the record. Judgement is the class where neither applies, which is why its output is a proposal.

So this unit gives you the cheaper question — where would the evidence come from — and Unit 3 gives you the sharper one, which is what to do about it.

1.1.3 The move worth learning

The classes are not fixed properties of a task. Which one you are in is often your choice, and moving between them is the most useful thing in this unit.

A recall question becomes a grounded question the moment you supply the source. “What does the regulation require?” is recall, and expensive to check. “Here is the regulation — what does clause 4 require?” is grounded answering, and the check is reading clause 4.

The same move works forwards. A vague request becomes generation to a specification the moment you write down what would count as correct, which is exactly what Unit 10 asks you to do, and why it is the unit that changes how much the rest of your work costs.

So the practical question is not “can it do this?” but “which class am I in, and can I move myself into a cheaper one before I start?”

1.1.4 One task, four classes

Take a single question: **does our coursework policy allow me to use an AI system to draft an essay plan?**

Asked with nothing attached, it is **recall**. The answer will name a policy, may quote it, and will read as settled. Nothing in it is evidence: to check it you must find the policy, which is the whole task you were trying to avoid.

Paste the policy in and ask the same question, and it is **grounded answering**. The check is now bounded: find the clause the answer relies on and read it. You may still find that the answer stretched a clause about brainstorming into one about drafting, but you will find it in a minute rather than an afternoon.

Ask instead for a table of every clause that mentions AI, one row per clause with its number and its exact wording, and you are in **transformation**. Every row is checkable against the document you supplied, and the failure to look for is a clause that quietly did not make it into the table.

Ask whether the policy is a *good* policy and you are in **judgement**. There is no clause that settles it. What comes back may be worth reading and is not a finding.

Four classes, one question, and the difference between an afternoon of checking and a minute of it is which one you chose. The same applies to the case this module is built on: “what is the eighth Bernoulli number?” is recall, and “here is the recurrence — compute it” is generation to a specification. Unit 6 has you build the reference, which is what makes the difference payable.

1.1.5 What these systems are not for

Not a list of forbidden tasks. A question:

> If this output is wrong, how would I find out, and what would that cost?

Three situations where the honest answer is bad.

No reference is obtainable at a price you can pay. Questions about the future, about private matters of fact, or at the edge of what is known. There is nothing to check against, so any output is a proposal however it is phrased.

You would not recognise a wrong answer. This is the sharpest one, and it is uncomfortable because it is about you rather than the system. Outside your own competence, every surface cue you normally rely on — fluency, specificity, confident structure — is available to a wrong answer as readily as a right one. The move here is to change class: get a source you can check, or ask someone who can.

The doing is the point. Some work exists to develop your judgement, and an output that arrives without the work has skipped the thing the work was for. That is a statement about what the task is, not a rule about tools. Your institution's rules on assessed work are separate, and they are theirs to state, not this module's.

1.1.6 Summary

- Sort a task by where its reference lives, not by topic or by product. That sorting does not date.
- Transformation and grounded answering put the reference in your hands. Generation to a specification puts it in something you write. Recall puts it somewhere you must go and find. Judgement has none.
- The class sets what checking costs, and most people do not price it before starting. Recall and judgement are the expensive classes, which makes them the ones you owe the most checking — not the ones to avoid.
- Cheap to check is not the same as checked. A transformation can drop the one qualification that mattered, and it will do so fluently.
- You can often move a task into a cheaper class by supplying a source or writing a specification. That is the single most useful habit in this module.
- The limits are not a list of banned tasks but one question: if this is wrong, how would I find out, and what would that cost?

Then answer `cp01-where-the-reference-lives` and `cp01-changing-class` at the foot of this page.

1.2 Unit 1 activity — sort your own week, then move one task

Time: about 12 minutes. Read `page.md` first, at least as far as "The move worth learning".

You are going to apply the five classes to work you actually have, and then do the one thing this unit is for: take a task that would be expensive to check and move it into a class where it is cheap.

1.2.1 Part 1 — five tasks of your own (5 minutes)

List five things you have used, or would plausibly use, an AI system for. Real ones, from your own study or work. If you have never used one, list five you can imagine reaching for this term.

For each, fill in three columns.

The task	Class	Where the reference would come from

For the middle column use one of: **transformation**, **grounded answering**, **generation to a specification**, **recall**, **judgement**.

For the third column be specific. Not "a source" but *which* source, and where you would get it. If the honest answer is "I do not know where I would check this", write that. It is the most useful row you will produce.

Two rules that stop this becoming a paper exercise:

1. **Sort by where the reference lives, not by how the task feels.** A task can matter enormously and still be transformation.
2. **If a task spans two classes, write both.** Most real tasks do, and the parts fail independently. "Draft this email and check the deadline I mentioned" is judgement plus recall, and only one of those halves can be wrong in a way that matters.

1.2.2 Part 2 — move one of them (5 minutes)

Look at your five and find the one with the most expensive reference — usually a recall row, and very often the one where you wrote “I do not know”.

Write two or three sentences answering this:

> What would I have to supply, or write down, to move this task into a cheaper class — and is that thing available to me?

Then say which class it would move to.

Sometimes the answer is that it cannot move. A question about what will happen, or about something nobody has written down, stays where it is. Recording that is a result, not a failure: it tells you the output will be a proposal whatever register it arrives in.

Keep both parts. Unit 17 asks you to assemble an evidence pack, and this table goes into it alongside the verification sentence you wrote in Unit 0 — the two things you contribute before the pack’s numbered sections begin. `course/learner-evidence-template.md` lists everything the pack holds and which unit each part comes from.

1.2.3 Now take the checkpoints

`cp01-where-the-reference-lives` gives you five tasks and asks you to match each to its class — the same judgement you made in Part 1. **`cp01-changing-class`** is about the move you made in Part 2.

```
python3 selfcheck.py run cp01-where-the-reference-lives
python3 selfcheck.py run cp01-changing-class  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check. Both give the same diagnostics.

1.2.4 Optional extensions

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

Price one check honestly (about 5 minutes). Take the recall row you moved in Part 2 and, before moving it, estimate how long checking it properly would actually take — finding the source, reading the relevant part, confirming the claim as stated. Then do it, and record the real time. The gap between the estimate and the reality is the thing this unit is trying to make visible.

Find a task that resists the move (about 4 minutes). Deliberately look for something you would want to ask that cannot be moved into a cheaper class, and write one sentence on why. Questions about the future, about private facts, and about the edge of what is known are the usual sources. Naming one is worth more than sorting ten easy cases.

1.2.5 Activity Answers

1.3 Unit 1 answers and guidance

Read this after attempting the activity.

There is no single correct table, because the tasks are yours. What can be checked is whether each row’s third column names something you could actually go and get.

1.3.1 Part 1 — what a good row looks like

A strong row has a **specific** reference: not “the documentation” but “the `pandas` merge documentation for the version I have installed”; not “a paper” but “the paper the summary cites, at the page it cites”.

Three patterns that mean the sorting has not quite worked yet:

Pattern	What has happened	What to do
Every row is transformation	You have sorted by what you *supply* rather than by what would settle it. Attaching a document does not make a question about the wider world grounded.	Ask for each row: could I answer this using only what I handed over? If not, the reference is elsewhere.
No row is judgement	Judgement is easy to miss because the answer arrives in the same confident register as everything else.	Look for any row where you asked "which is better", "is this any good", "what should I do".
The third column says "I would just check it"	That is an intention, not a reference. Unit 0's sentence had the same failure.	Name the artefact you would open. If nothing comes to mind, that is the finding.

The rows where you wrote "I do not know where I would check this" are the valuable ones. They are not gaps in your answer; they are the tasks whose checking cost you had not priced, which is the thing this unit exists to make visible.

1.3.2 Part 2 — what moving looks like

The move is almost always one of two things:

- **Supply the source.** Recall becomes grounded answering. The check goes from "find out what the truth is" to "read this document at this point".
- **Write the specification.** A vague request becomes generation to a specification. The check goes from "does this look right" to "does this meet the thing I wrote down".

A good answer names the artefact and says whether you have it. "I would need the 2024 version of the module handbook, which is on the intranet" is an answer. "I would give it more context" is not — context is not a reference, and the whole point of the class is *what you would compare against*.

If you concluded the task cannot move, check that the reason is about the question rather than about effort. "There is no written record of this" is a reason. "It would take a while to find" is a cost, and cost is exactly what this unit asks you to pay deliberately rather than skip silently.

1.3.3 The checkpoints

`cp01-where-the-reference-lives` uses five tasks distinct from your own and tests the same sorting. The two confusions it targets are worth naming, because both survive into later units.

The first is treating **anything with a document attached** as grounded answering. Attaching your own notes and asking for a summary is transformation: the reference is the notes, and the failure to look for is omission. Attaching a policy and asking what it permits is grounded answering: the reference is the policy, and the failure to look for is a clause stretched past what it says.

The second is treating **judgement** as generation to a specification. If you can write down what would count as correct, you are specifying. If you cannot, because there is no fact of the matter, you are asking for a proposal — and no amount of precision in the prompt changes that.

`cp01-changing-class` tests the move itself, and the tempting wrong answer is the one that adds material without changing where the reference lives. More context is not a reference.

1.3.4 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp01-where-the-reference-lives` — Where the reference lives

Kind: match.

Match each task to the class it belongs to, by asking where the evidence would come from that the output is right. Answer with five option numbers in the order of the tasks.

Observations, in order:

1. You paste in your own 600 words of lecture notes and ask for them as a bulleted summary.
2. You attach your department's coursework policy and ask what it says about drafting assistance.
3. You state that a routine must accept every integer from 0 upwards and return an exact fraction, and ask for the routine.
4. You ask which year a named international standard was last revised, attaching nothing.
5. You give two possible titles for your dissertation and ask which is stronger.

Options, numbered from zero:

0. Transformation
1. Grounded answering
2. Generation to a specification
3. Recall
4. Judgement

Answer: `0, 1, 2, 3, 4`

Hint: For each task ask one question: if this output were wrong, what would I open to find out? Your own input, a source you supplied, a specification you wrote, something external you would have to go and find, or nothing at all.

Diagnostics, by the answer given:

- `1,0,2,3,4` — You have swapped the first two. Both attach a document, so both feel grounded, but they differ in what would settle them. Your own notes are the reference for their own summary, so that is transformation and the failure to look for is omission. The policy is a reference for a question about what is permitted, so that is grounded answering and the failure to look for is a clause stretched past what it says.
- `0,3,2,1,4` — You have swapped the policy question with the standard question. The test is not whether an authoritative document exists somewhere – one exists in both cases – but whether you supplied it. The policy is attached, so the check is bounded to reading it. The standard is not, so checking means finding it first, which is a research task of its own.
- `0,1,4,3,2` — You have swapped the routine with the dissertation titles. The distinction is whether you can write down what would count as correct. For the routine you can, and did: accepts every integer from 0 upwards, returns an exact fraction. For the titles you cannot, because there is no fact of the matter about which is stronger. Precision in the prompt does not turn a proposal into a result.
- `0,1,2,4,3` — You have swapped the last two. Asking which year a standard was revised has a determinate answer that someone has written down, so it is recall however hard it is to find. Asking which title is stronger has no such answer, which is what makes it judgement.

Explanation: The five classes differ in one thing only: where the reference lives. That is what sets the cost of checking, and it is why sorting by class is more useful than sorting by topic or by how important the task feels. Two tasks that look alike can sit in different classes, and the difference between them can be an afternoon of work.

Checkpoint `cp01-changing-class` — Moving a task into a cheaper class

Kind: choice.

You need to know whether a particular software licence permits redistribution. Asked with nothing attached, this is recall, and checking it means finding the licence yourself. Which change moves it into a class where the check is bounded?

Options, numbered from zero:

0. Paste in the licence text and ask what its redistribution clause permits.
1. Add more context about your project, your users and why the answer matters.
2. Ask the same question again in a new conversation and see whether the answer is the same.
3. Ask the system how confident it is that its answer about the licence is correct.

Answer: `0`

Hint: The task is expensive because the thing you would check against is somewhere you have not got. Ask which option puts it in front of you, and which options only change what the question says.

Diagnostics, by the answer given:

- 1 — More context changes what the answer is about, not where its reference lives. The question is still one about the licence, and the licence is still not in front of you. Context is not evidence.
- 2 — Repetition tests stability, not correctness. The second answer is produced the same way as the first and can preserve the same unsupported assumption. Unit 3 has the mechanism; nothing about asking twice puts the licence in your hands.
- 3 — A statement of confidence is generated the same way as the claim it is about, so it is another output needing evidence rather than evidence about the first. It also leaves the licence exactly where it was.

Explanation: Supplying the source is the move. It converts recall, where the check is an open research task, into grounded answering, where the check is bounded to reading a named document at a named point. Two questions remain live even then – does the source say what the answer says it says, and is it the right source to settle this – but both are now answerable in minutes. This is the habit Unit 10 formalises for specifications and Unit 18 formalises for sources.

2 Unit 2 — Ada's ordered procedure

2.1 Reading

Reading time: about 8 minutes. Activity: about 12 minutes.

2.1.1 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 6 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).

2.1.2 What Note G actually is

Note G is a prose note, and inside it is a table.

The table is the part this module works from, and what matters is that it is a table rather than a description of a method or an argument that a method would work in principle. It is 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.

2.1.3 Three properties worth naming

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

2.1.3.1 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 3 takes you.

2.1.3.2 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.

2.1.3.3 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 worked trace of one case** and **a general procedure with a stopping rule** — and it is worth saying that a recipe is usually the second of those, not the first. A recipe that says "simmer until it thickens" has a stopping rule and works for any quantity. The contrast is not recipes against algorithms; it is one answer against a family of them.

2.1.4 "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 at least two printed errors. One of them — operation 4, an inverted division — is established from a facsimile; a second, in operation 24, rests on a published reconstruction. Who introduced either stays contested, and "what the faulty table computes" turns out to depend on which errors you repair and what you feed it. This module does not work through the detail here; Unit 16 treats the whole episode as its case study in historical honesty, including the mistake this module itself made with it. 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 17.

2.1.5 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 17 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 3 removes.

2.1.6 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.

2.1.7 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 16 deals with it.

Then answer `cp02-ordered-steps` at the foot of this page.

2.2 Unit 2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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 3, when you decide which parts of a generated output have to be checked and which do not.

2.2.5 Now take the checkpoint

This unit’s checkpoint is **`cp02-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 cp02-ordered-steps  # or in the browser
```

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 10 is about writing specifications precise enough to be tested, and it will ask you to look at this again.

2.2.6 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.

If you 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?

2.2.7 Activity Answers

2.3 Unit 2 answers and guidance

The activity has no single correct output — everyone picks a different task. What you can check is whether you found the gaps, and whether you classified them for the right reason.

2.3.1 A worked example

Here is a first draft of the kind most people 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.

Step	Code	What a literal reader cannot do
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 dwelling on.

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. If you spotted repetition and stopping rules, you have understood control flow, which is the hardest of the three properties to see in everyday language.

2.3.2 Marking your 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? If you repaired the easiest gap you have done the exercise; if you repaired the most dangerous one you have done the thinking Unit 3 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.

2.3.3 Checking your own work

The required work takes about 12 minutes; the three optional extensions add roughly 5, 5 and 15 minutes and are outside the core budget. None of it is graded.

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.

If your result does not look like that, find yourself in the first column:

What happened	Why it usually happens	What to do
You found very few gaps	You are reading co-operatively, supplying the missing meaning as you go.	Swap procedures with someone else. Gaps in a stranger's writing are obvious; gaps in your own are invisible by construction.
Your gaps are all types A, B, F	Surface reading.	Ask yourself directly: which step happens more than once, and how does it stop?
You wrote the procedure as prose	The task you chose was too abstract to break into steps.	Start again with a physical task, with objects you could point at.
You marked background knowledge as a gap	The rule applied too widely.	"The procedure never introduces the tape" is a defect; "the procedure never explains what tape is" is not.

The point, whatever you produced. 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 you seven codes instead of telling you to check your work carefully.

This also sets up the module's central problem. In Unit 7 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.

2.3.4 On the historical material

If you are wondering which parts of the "first program" story you are supposed to believe, the honest answer is the one on the reading 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. `course/historical-notes.md` and Unit 16 are where that is worked through; this activity deliberately does not settle it.

Do not let the contested points be settled by an AI system's confident summary. If you catch yourself doing it, that is a live and slightly awkward demonstration of the module's own thesis.

2.3.5 The checkpoint

This unit's checkpoint is `cp02-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 the two routes ask the same question and give the same feedback.

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 you get it wrong, ask yourself which nouns in the given procedure were never introduced before use — that is the question the feedback pushes you towards.

```
python3 selfcheck.py run cp02-ordered-steps  # or in the browser
```

2.3.6 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp02-ordered-steps` — What the procedure left unsaid

Kind: multi.

A learner writes this procedure for making tea: (1) Boil water. (2) Add a teabag. (3) Pour the water. (4) Wait. (5) Remove the teabag. An Analytical Engine executing this literally would need more. Which of these are genuinely missing from the procedure? Select all that apply.

Options, numbered from zero:

0. How long to wait in step 4.
1. What to add the teabag to – no container is ever named.
2. How much water to boil.
3. The fact that tea is a hot drink.

Answer: `0, 1, 2`

Hint: Read it as a machine would: refuse to supply anything not written down. Which nouns appear without ever being introduced, and which numbers are never given?

Diagnostics, by the answer given:

- `0,1,2,3` — Three of your four are right, but 'tea is a hot drink' is background knowledge, not a step. The distinction matters: an ordered procedure must state every action and quantity, but it is not required to explain what the result is for.
- `0,1` — You have the timing and the missing container. One more is genuinely missing: look at step 1. 'Boil water' never says how much, and a machine cannot supply an amount nobody wrote down. A missing quantity is as much a gap as a missing object.
- `0` — The timing is missing, yes – but look again for the step that assumes an object nobody ever introduced. Ada's diagrams name every variable before using it, and that is not a stylistic preference.

Explanation: Unstated assumptions are invisible to the person who holds them and fatal to the machine that does not. Ada's Note G is so long precisely because every quantity, every operation, and every intermediate variable had to be written down explicitly.

3 Unit 3 — What a language model does differently

3.1 Reading

Media for this unit:

- **Animation**: *One token at a time* — this unit has an interactive version on the course website (`next-token.html`); the still below is one moment from it.

Figure 2: One token at a time. A distribution forms over five candidate next tokens, one is selected and appended, and the distribution re-forms for the next position.

- **Animation**: *Transformer System Layers* — this unit has an interactive version on the course website (`transformer-system-layers.html`); the still below is one moment from it.

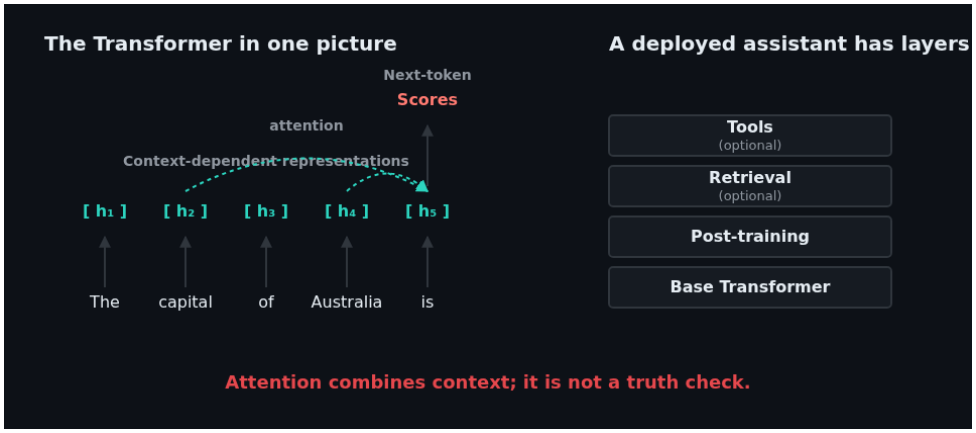

Figure 3: Transformer System Layers. The attention mechanism combining context, and the layered architecture of a deployed assistant around it.

- **Animation**: *Agentic Effort vs Basic Procedure* — this unit has an interactive version on the course website (`agentic-effort.html`); the still below is one moment from it.

Reading time: about 10 minutes. Activity: about 16 minutes.

3.1.1 Start from what changes

Unit 2 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.

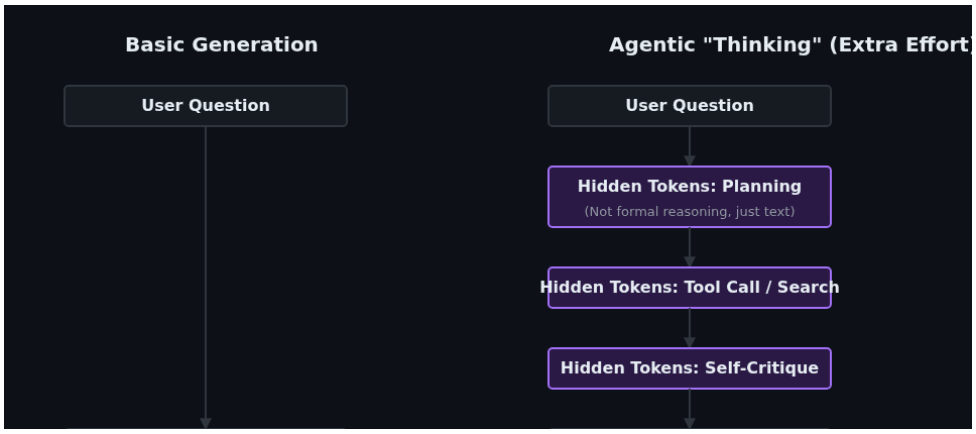

Figure 4: Agentic Effort vs Basic Procedure. Basic generation set against an agentic loop, which spends many hidden tokens before it answers.

3.1.2 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

Some deployed systems generate intermediate text before returning an answer, and some run loops that plan, call tools and revise. Where they call that intermediate text "thinking", what it is mechanically is the same next-token generation used as a scratchpad: the system writes out a breakdown of the problem, or steps, or a critique of its own draft, and may call a search index or a calculator along the way. Those intermediate tokens are often withheld from the reader, which makes the process look silent and internal when it is neither.

Keep the layers apart, because they are audited differently: what the model generates; what it computes with a deterministic tool; what it retrieves from a source; what it shows you as an explanation; and what an orchestrating loop does with all of that. A visible derivation is generated text about the work, not a transcript of it, and it can be fluent and wrong in exactly the way any other generated text can.

Whether that extra effort improves a result depends on the model, the task, the tools available and how the result is judged; it is not a property you can assume, and any figure attached to it dates quickly. The hazard for verification is confusing *computational effort* with *formal reasoning*. A loop that plans and revises is still generating text and choosing actions probabilistically; it is not building a proof. So the principle does not change: a claim you are going to rely on needs a check proportionate to what it would cost to be wrong, however much intermediate work produced it.

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.

3.1.3 Three consequences

3.1.3.1 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 2. 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.

3.1.3.2 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.

3.1.3.3 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.

3.1.4 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 7 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.

3.1.5 The two systems, side by side

	Ordered procedure (Unit 2)	Text generation (this unit)
Same input, same output?	Yes, when implementation and environment are fixed.	Depends on decoding settings and the surrounding service.
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.

	Ordered procedure (Unit 2)	Text generation (this unit)
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.

3.1.6 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. It is a sharper cut of the question Unit 1 asks about where a reference lives, and the two agree: a reference you can execute against gives exact checking, a reference in a record gives source checking, and the class with no reference at all is where neither applies.

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 6, 7 and 8 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 7 puts in front of you.

3.1.7 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 18 teaches from it, and nothing on this page, in the activity, or in either Unit 3 checkpoint depends on either.

3.1.8 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.

Then answer `cp03-risk-triage` and `cp03-system-layers` at the foot of this page.

3.2 Unit 3 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.

3.2.1 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.		
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.		

#	The output	E / S / L	The check I would run
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.

3.2.2 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.

3.2.3 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. Keep this row: Unit 17, Task 4 returns to it and asks you to run the whole method on it.

3.2.4 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.

3.2.5 Now take the checkpoints

`cp03-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. **`cp03-system-layers`** checks the five distinctions from Part 4.

```
python3 selfcheck.py run cp03-risk-triage
python3 selfcheck.py run cp03-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 7 and 8 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.

3.2.6 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.

3.2.7 Activity Answers

3.3 Unit 3 answers and guidance

Check your reason, not your letter. If you wrote **S** for item 1 and justified it by “the postcode rules are published, so I would check the specification before testing anything”, you have understood more than someone who wrote **E** because code is usually **E**.

3.3.1 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.

#	Intended	Why	The check
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.
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.

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

The required work asks for **one** item marked **L**; covering the rest is an optional extension. A good answer names a change in **your 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.

3.3.3 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.
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 8 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.

3.3.4 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 that matter in practice.

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.

3.3.5 Checking your own work

The required work takes about 16 minutes; the three optional extensions add roughly 3, 4 and 5 minutes and are outside the core budget. None of it is graded.

A strong response gets at least eight of the ten broadly right, names a check that could fail for each, and flags at least one item as legitimately spanning two categories.

Five ways this goes wrong. Find yours in the first column:

What your triage looks like	What has happened	What to do
Everything marked E or S; nothing L	You have heard “check everything” and written a policy you will not keep.	The **L** category is not permission to be careless; it is what makes the other two affordable. Ask which items you would actually check next Tuesday.
Everything sorted by stakes	The most common error, and the one `cp03-risk-triage` targets.	Compare item 7 with item 5: both matter, and they need completely different checks.
“I would check it carefully” is your 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.

What your triage looks like	What has happened	What to do
Your justifications say things like “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 — your own item — is the one that matters. It shows what you actually do, not what you can classify. If you honestly wrote “I would not have checked that”, that is the most useful sentence in the exercise; keep it in your evidence pack rather than tidying it away.

3.3.6 The checkpoint

This unit’s checkpoints are `cp03-risk-triage` and `cp03-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 the two routes ask the same question and give the same feedback.

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 feedback names that directly; read it rather than going straight to the answer.

```
python3 selfcheck.py run cp03-risk-triage  # or in the browser
python3 selfcheck.py run cp03-system-layers
```

3.3.7 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp03-risk-triage` — Triage by how it can be wrong

Kind: match.

A language model produces each of these. Match each output to the kind of checking it needs before you rely on it.

Observations, in order:

1. A function that claims to compute compound interest.
2. A statement that a named researcher published a particular paper in 1987.
3. A first draft of a covering email you will rewrite anyway.

Options, numbered from zero:

0. Needs exact checking – run it against known values.
1. Needs source checking – verify against the record.
2. Low-risk drafting – errors are cheap and visible.

Answer: `0, 1, 2`

Hint: For each output ask what you would do to find out it was wrong: run it and compare, look it up in a record, or nothing much because an error would be cheap and visible.

Diagnostics, by the answer given:

- `1,0,2` — You have the first two swapped. Code is checked by *executing* it against values you already know; a citation cannot be executed at all, only looked up. The failure modes are different, so the checks are too.
- `0,2,1` — You have treated the citation as low-risk drafting and the draft email as a claim to verify – the wrong pair is sorted. A citation is a claim about the record and can be flatly wrong in a way nobody notices; the covering email is the one you will rewrite anyway, so its errors are cheap and visible.

- `2,1,0` — This looks like sorting by how serious each item sounds, not by how it would be checked. The interest function feels routine but has an exactly checkable right answer; the draft email feels consequential but is low-risk precisely because you will rework it. Sort by how an error would reach you, not by apparent stakes.

Explanation: Plausibility is uniform across all three – the model is equally fluent whether or not it is right. What differs is how the error would reach you, which is what decides the check. Matching the check to the failure mode is the practical core of evaluating AI output.

Checkpoint `cp03-system-layers` — Separate the assistant's system layers

Kind: match.

Match each function to the layer that performs it. Use each layer once.

Observations, in order:

1. Combines information from relevant token positions in the available context.
2. Shapes instruction-following behaviour after base-model training.
3. Supplies external documents that still need relevance and support checks.
4. Ranks possible continuations; the scores are not a truth predicate.
5. Learns statistical patterns for next-token prediction from a large corpus.

Options, numbered from zero:

0. Attention
1. Post-training
2. Retrieval
3. Next-token scoring
4. Pre-training

Answer: `0, 1, 2, 3, 4`

Hint: Ask whether the function combines existing context, shapes behaviour, fetches material, ranks continuations, or trains the base model.

Diagnostics, by the answer given:

- `2,1,0,3,4` — Attention and retrieval are swapped. Attention combines positions already in the context; retrieval is the separate step that fetches external material. Neither operation establishes that a resulting claim is true.
- `0,1,3,2,4` — Retrieval and next-token scoring are swapped. Retrieval can supply documents, but those documents still need to be checked for relevance and support. Next-token scores rank continuations; they are not truth scores.
- `0,4,2,3,1` — Pre-training and post-training are swapped. Pre-training builds a base next-token model from a large corpus. Post-training is one later layer that shapes instruction following; neither term names the whole deployed product.

Explanation: A deployed assistant is a system of distinguishable layers. Attention combines token context, pre-training learns next-token patterns, post-training shapes assistant behaviour, retrieval can supply external documents, and next-token scoring ranks continuations. None of these is itself a truth predicate, so evidence must still be checked.

4 Unit 4 — Choosing the tool

4.1 Reading

Media for this unit:

- **Animation**: *The check moves; it does not go away* — this unit has an interactive version on the course website (`check-moves.html`); the still below is one moment from it.

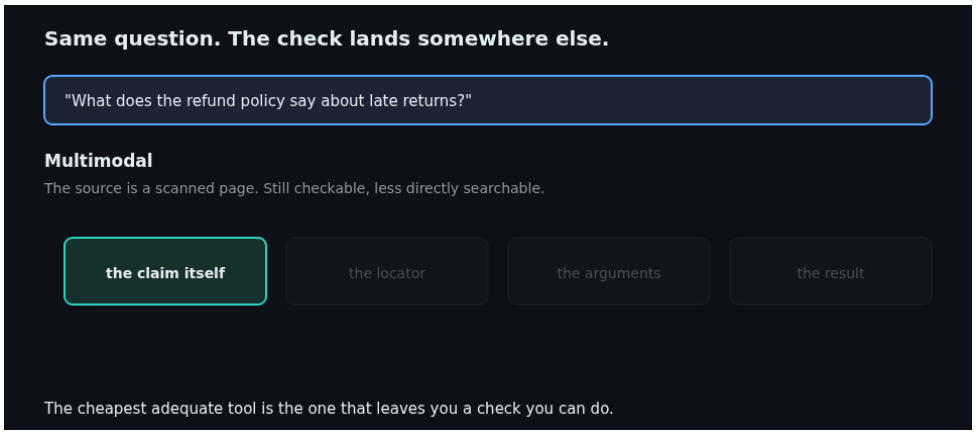

Figure 5: The check moves; it does not go away. One question asked of four system classes; the thing you must check moves each time.

Reading time: about 10 minutes. Activity: about 14 minutes.

4.1.1 What this unit adds

Unit 3 described what a language model does: attention over a context, a distribution over next tokens, and optional layers built around that base — post- training, retrieval, tools. Unit 1 sorted tasks by where the reference lives.

This unit joins them. Deployed systems differ in which of those layers they have, and each layer you add changes **what you have to check** — not by removing checks, but by moving them somewhere else and adding one of its own.

The practical claim is narrow: choosing a system is choosing a checking job. If you know which one you have taken on, you can pick the system whose checking job you can actually do.

4.1.2 Four kinds of system

Not products. Kinds, named by what they add to the base model. A single product may be several of these at once, and may change which it is between one release and the next, so identify what a system is *doing on this occasion* rather than what its name is.

Kind	What it adds	What it changes about checking	The new failure to look for
Plain generation	Nothing	Nothing — you own the whole check	A claim that arrives with no evidence and no marker
Retrieval-backed	Fetches sources	Aims the check at a named source	A real source that does not say what the answer says
Tool-calling	Runs deterministic code	Splits the check in two	A correct computation of the wrong question
Multimodal	Accepts or produces images, audio, video	Puts the reference in something you cannot search	A confident description of what is not there

4.1.2.1 Plain generation A base model with post-training and nothing else. Everything in the answer is generated, so a task in Unit 1's *recall* class stays there: the reference is external and you must go and find it.

There is nothing wrong with this kind. It is the right choice when the class is transformation — you supplied the content, so you already hold the reference — and it is usually the fastest and cheapest option. What it is not is a way of finding things out.

4.1.2.2 Retrieval-backed The system searches, or is given documents, and generates an answer conditioned on what came back [L3]. This is a genuine difference and it is the most useful single upgrade for the recall class, because it moves the task into Unit 1's *grounded answering*.

It does not remove the check. It splits it into two questions that are now answerable:

1. **Does the source say what the answer says it says?** The answer is still generated text about the source, not an extract from it. A summary can stretch a source's qualification, or attach a real citation to a claim the source does not make.
2. **Is this the right source?** Retrieval returns what matches the query, which is not the same as what settles the question. An outdated page, a draft, a forum post and the authoritative specification can all match well.

The second question is the one people skip, because a citation looks like the end of an argument rather than the start of one.

4.1.2.3 Tool-calling The system calls something deterministic — a calculator, a database query, code it wrote and ran, and uses the result.

The check genuinely splits, and the halves are unequal:

- **The computation** is deterministic and, within its own terms, either right or wrong in the ordinary way. This half is a real gain: arithmetic done by a calculator is arithmetic done by a calculator.
- **The decision to call it, with those inputs, for this question** is generated. Nothing deterministic chose which numbers went in or what the result would be taken to mean.

So the failure to look for is a correct computation of the wrong question: the right formula on the wrong column, a currency conversion at an unstated date, a query that answers a slightly different question than the one you asked. The result carries the authority of arithmetic and the fallibility of the choice that produced it.

Unit 8's lab is this idea taken seriously in one direction: run the code, compare against a reference you built yourself.

4.1.2.4 Multimodal The system accepts images, audio or video, or produces them.

What changes is that **the reference stops being searchable**. When the source is a document you can find the sentence the answer relies on. When it is a photograph, a chart or a recording, checking means going and looking, and there is no equivalent of searching for a phrase.

Two specific hazards, both worth naming:

- A description of an image is generated text about the image. It can be detailed, confident and about something that is not in the frame, and unlike a fabricated citation, there is no catalogue in which its absence shows up.
- A chart read by a system is a chart read from pixels. Values between gridlines, a truncated axis, a log scale: all are places where a confident reading can be wrong in a way that reads as precise.

Systems that act rather than answer — tool calls with side effects, multi-step loops that carry out tasks — change the question again, because a wrong answer you can discard and a wrong action you cannot. That is Unit 11's subject and this unit does not anticipate it.

4.1.3 A decision aid you can apply in a minute

For a task you actually have, in order:

1. **Name the class** (Unit 1). Where would the reference come from?
2. **If the class is recall, try to move it.** Can you supply the source? If yes, do that and use whichever system lets you attach it. This single step removes more checking work than any choice of product.
3. **Ask what the answer will contain.** Fetched sources? A computed number? A reading of an image? Each of those is a specific check you are taking on.
4. **Ask whether you can do that check.** Not "is the system good" — can *you*, with the time and access you have, carry out the check its output will need? If not, you have chosen the wrong kind, or the wrong task.
5. **Take the cheapest kind that leaves you a check you can do.** Extra layers are not free, and the next section is about that.

Step 4 is the one that does work. A system that produces claims you cannot check is not more useful than one that produces fewer claims you can.

4.1.4 What it costs to ask

Three costs, and they move together because they have the same driver.

Money. Priced by the amount of text going in and coming out, and by the size of the model. Longer conversations cost more than short ones for the same question, because the conversation's history generally goes with it.

Time. Roughly the same drivers, plus the extra steps: retrieval adds a search, tool-calling adds an execution, and a system that generates intermediate text before answering generates more text.

Energy. Real, and driven by the same quantities: more computation for longer inputs, larger models and more steps.

4.1.4.1 A figure this module will not print

You will have seen numbers for what a single query costs in energy or carbon. This module does not give one, and the reason is the module's own subject.

A defensible figure would have to name the model, the hardware, the length of the query, whether the cost of training the model is amortised into it, and the electricity supply, and it would be out of date within months. Printing a number without those conditions would be exactly the confident-and-unsupported claim this module trains you to catch.

So instead: the shape is reliable even when the numbers are not. **A longer prompt, a longer answer, a bigger model and more steps all cost more of all three.** If you need an actual figure, that is a Unit 1 recall task, and it needs a source that states its conditions and its date.

4.1.4.2 When the cheap option is the right one

Often. The instinct to reach for the most capable available system treats capability as free, and it is not — in money, in time, and in checking work, because a more elaborate system produces more kinds of claim to check.

The cheap option is right when the class is transformation or grounded answering and the check is a comparison you will do anyway; when the task is a draft you will rewrite; and whenever the expensive option's extra output is something you would not check.

The expensive option earns its cost when the extra layer removes a check you could not otherwise do — retrieval on a question you cannot research yourself, a tool call on a computation you cannot perform, and when you will actually use the check it makes possible.

4.1.5 What this unit is not saying

It is not saying retrieval-backed and tool-calling systems are more trustworthy. They are differently checkable. Each removes one kind of uncertainty and adds a different one.

It is not a recommendation of any product. Named products date, change what they are between releases, and are not the level at which this reasoning works.

4.1.6 Summary

- Systems differ by which layers they add to a base model, and each layer moves the check rather than removing it.
- Plain generation leaves the whole check with you; retrieval aims it at a source; tool-calling splits it into a computation and the choice to make it; multimodal puts the reference somewhere you cannot search.
- Choose by asking what check the output will need and whether you can do it, not by which system is most capable.
- Money, time and energy all rise with input length, output length, model size and number of steps. The shape is stable; the numbers are not, and this module declines to print one it cannot defend.
- The cheapest kind that leaves you a check you can carry out is usually the right one.

Then answer `cp04-what-changes-the-check` and `cp04-cheapest-adequate` at the foot of this page.

4.2 Unit 4 activity — run the decision aid on a task you brought

Time: about 14 minutes. Read `page.md` first, at least as far as the decision aid.

Unit 1 asked you to sort tasks. This one asks you to choose, and then to notice what the choice cost you.

4.2.1 Part 1 — the decision aid, on one real task (6 minutes)

Take one task from the table you built in Unit 1 — ideally the one you moved into a cheaper class. Work the five steps and write down each answer.

Step	Your answer
1. Which class is this?	
2. Can I move it? To what?	
3. What will the answer contain that needs checking?	
4. Can I actually do that check, with the time and access I have?	
5. Which is the cheapest kind that leaves me a check I can do?	

Step 3 is where most of the value is, so be concrete. “Sources” is not an answer; “three citations I would have to look up in the library catalogue” is. “A number” is not an answer; “a currency conversion whose date I would have to confirm” is.

Step 4 has only two honest answers, and one of them is uncomfortable. If it is “no”, say so and go back to step 2: either supply something that moves the class, or accept that what you will get is a proposal rather than a result.

4.2.2 Part 2 — the same question, two kinds (5 minutes)

Now ask the same question twice: once of a plain generation system, and once of a system that can fetch or be given a source. If you only have access to one kind, simulate the second by pasting in the source yourself.

Keep both answers. Then write three or four sentences on this:

> What is in the second answer that I can check, that I could not check in the first?

Note what the question is not. It is not which answer is better written, and it is not which answer is right — you may not know that yet. It is about **what each answer gives you to work with**.

Two things to look for specifically:

- Does the grounded answer point at a *place* in the source, or only name the source? “As the policy states” is not a locator.
- Does the grounded answer say anything the source does not? This is the characteristic failure of the retrieval-backed kind and you may well catch one here on your first try.

4.2.3 Part 3 — price it (3 minutes)

For the task in Part 1, write one line each:

- **Money and time:** was the more elaborate option meaningfully slower or dearer for this task? If you do not know, say so — it is the honest answer for most consumer tools, and it is a finding about how visible these costs are.
- **Checking:** did the extra layer add checking work or remove it?

That second line is the one this unit is for. An extra layer earns its cost when it removes a check you could not otherwise do. It does not earn it when it produces more claims you were never going to verify.

4.2.4 Now take the checkpoints

`cp04-what-changes-the-check` gives you four observations about an answer and asks which check each one makes necessary. **`cp04-cheapest-adequate`** is about Part 3's judgement.

```
python3 selfcheck.py run cp04-what-changes-the-check
python3 selfcheck.py run cp04-cheapest-adequate  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check.

4.2.5 Optional extensions

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

Break a chart reading (about 6 minutes). Find a chart with a truncated axis, a log scale, or values that fall between gridlines, and ask a multimodal system to read a specific value off it. Then read the value yourself. Record what it said, what the value is, and — the useful part — how confident the wording was. This is the fastest way to see why an unsearchable reference is a different problem from an unreliable one.

Find a citation that is real and does not support the claim (about 8 minutes). Ask a retrieval-backed system a question in an area you know, and check not whether the sources exist but whether they say what the answer says they say. The failure you are looking for has a name in Unit 9 — this is a preview of it, and finding one yourself is worth more than being shown one.

4.2.6 Activity Answers

4.3 Unit 4 answers and guidance

Read this after attempting the activity.

4.3.1 Part 1 — what the steps are testing

Step 3 and step 4 are the whole exercise; steps 1, 2 and 5 are bookkeeping around them.

A strong step 3 names **artefacts**, not categories: which citations, which number, which part of which image. A weak one names a kind of thing, which is the same failure Unit 0's verification sentence had — "I will check the sources" cannot be done, because it does not say what would be opened.

Step 4 has a failure mode worth naming: answering it about the *system* rather than about yourself. "It is usually reliable for this" is not an answer to "can I do this check". The question is about your time, your access and your competence, and it is the only step that can tell you to change the task rather than the tool.

If step 4 came out "no" and you did not go back to step 2, that is the thing to correct. A check you cannot do is not a check.

4.3.2 Part 2 — what to look for in the comparison

The intended discovery is that the grounded answer is not *better*, it is *differently checkable*. Three patterns come up:

What you found	What it means
The grounded answer names a source but no place in it	You have provenance, not a locator. Provenance tells you where the answer came from; a locator is what makes the check cheap. Unit 9 treats this as its own failure mode.
The grounded answer says more than the source does	The characteristic failure of the retrieval-backed kind, and the reason attaching a source moves the check rather than removing it.
Both answers agree	Worth nothing on its own. The second answer is not independent evidence for the first — they may share the same source of error. Unit 3 has the mechanism.

That last row catches people out, because agreement feels like confirmation. It is the same error as re-asking a question and taking the repeated answer as confirmation.

4.3.3 Part 3 — pricing

"I do not know what it cost" is the honest answer for most consumer tools, where money and energy are invisible at the point of use and time is the only cost you feel. Recording that is a finding, not a gap: a cost you cannot see is a cost you cannot weigh.

The line that matters is the second. The instinct to reach for the most capable available option treats capability as free. It is not free even when it is unpriced, because every extra layer produces another kind of claim, and a claim you do not check is not a benefit.

4.3.4 The checkpoints

`cp04-what-changes-the-check` matches four observations to the check each makes necessary. The confusion it targets is treating the *presence* of a source or a computed number as the end of the checking rather than the start of it.

`cp04-cheapest-adequate` tests the judgement in Part 3. The tempting wrong answer is the one that picks the most capable system because it is available, which is the reasoning this unit is written against.

4.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp04-what-changes-the-check` — What each kind of system changes about the check

Kind: match.

Match each observation about an answer to the check it makes necessary. Answer with four option numbers in the order of the observations.

Observations, in order:

1. The answer is plain generated prose. Nothing was attached and nothing was fetched.
2. The answer quotes a source it searched for and found itself.
3. The answer contains a figure the system produced by writing and running a short calculation.
4. The answer describes what can be seen in a photograph you uploaded.

Options, numbered from zero:

0. Find an independent record. Nothing in the answer is evidence for the answer.
1. Open the source at the point relied on, and ask both whether it says this and whether it is the right source to settle it.
2. Confirm which inputs were used and which question was actually computed, since the arithmetic can be right and the question wrong.

3. Look at the image yourself at the point the description depends on, because there is no index in which a missing detail would show up.

Answer: `0, 1, 2, 3`

Hint: For each observation ask what you would open to find out whether it is right, and whether the exchange has already put that thing in front of you.

Diagnostics, by the answer given:

- `1,0,2,3` — You have swapped the first two. The distinction is whether a source exists in the exchange at all. With nothing attached and nothing fetched there is no source to open, so the check is a separate research task. With a fetched source there is somewhere specific to look, which is exactly what makes it cheaper.
- `0,1,3,2` — You have swapped the computed figure with the photograph. A calculation is deterministic and inspectable, so the check is about which inputs and which question. An image cannot be searched, so the check is going and looking. Both produce confident-sounding output, but for different reasons and with different remedies.
- `0,2,1,3` — You have swapped the fetched source with the computed figure. A quoted source is checked by reading it; a computed figure is checked by asking what went in. The shared trap is treating either as finished evidence because it looks like evidence.
- `3,1,2,0` — You have swapped the first and last. The photograph and the unattached claim both leave you with work to do, but not the same work. For the image the reference exists and you can look at it; for the unattached claim there is no reference in the exchange at all and you must go and find one.

Explanation: Every layer a system adds moves the check somewhere else and adds one of its own; none of them removes it. Retrieval aims the check at a named source, tool-calling splits it into a computation and the choice to make it, and multimodal puts the reference somewhere that cannot be searched. Knowing which check you have taken on is what lets you choose a system whose checking job you can actually do.

Checkpoint `cp04-cheapest-adequate` — When the cheap option is the right one

Kind: choice.

You have a 900-word draft of your own and you want it restructured under three headings you have already chosen. You have access to a plain generation system and to a slower, dearer one that searches the web and cites sources. Which choice is better reasoned?

Options, numbered from zero:

0. The plain system, because you supplied the content, so the reference is the draft in front of you and searching adds claims you would then have to check.
1. The searching system, because it is the more capable of the two and capability does not hurt.
2. The searching system, because citations would let you show where the restructured text came from.
3. Either, because for a task this simple the choice of system does not matter.

Answer: `0`

Hint: Ask where the reference for this task lives. If you already hold it, consider what a system that fetches more material would add – and whether you would check what it added.

Diagnostics, by the answer given:

- `1` — Capability is not free. Every layer produces another kind of claim, and a fetched source in a restructuring task is a claim you did not need and would now have to check. Treating capability as costless is the reasoning this unit is written against.
- `2` — The restructured text came from your draft, and you have it. A citation would point outside the only reference that matters here. Provenance for your own material is not something a search adds.
- `3` — It does matter, and in a direction worth noticing: the searching system can introduce material that was not in your draft. For a transformation task the failure to look for is omission, and adding a source makes the output harder to compare against the input rather than easier.

Explanation: This is a transformation task in Unit 1's terms, so the reference is the draft you supplied and the check is a comparison you can do at your desk. The cheapest kind that leaves you a check you

can carry out is the right one, and here the extra layer adds work rather than removing it. The expensive option earns its cost when it removes a check you could not otherwise do – not when it is simply available.

5 Unit 5 — Prompts, context and attachments

5.1 Reading

Media for this unit:

- **Animation**: *What is in the window, and what falls out* — this unit has an interactive version on the course website (`context-window.html`); the still below is one moment from it.

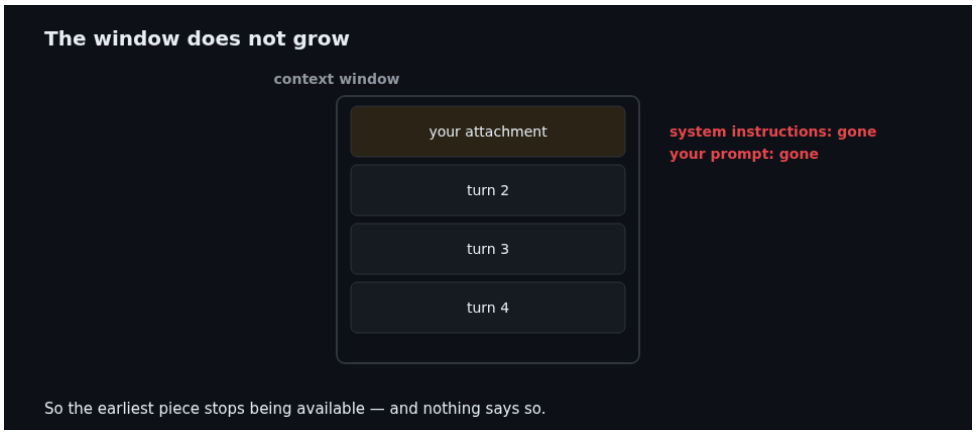

Figure 6: What is in the window, and what falls out. Pieces of context stack inside a fixed frame; when the next will not fit, the earliest leaves.

Reading time: about 9 minutes. Activity: about 14 minutes.

5.1.1 Why the same question gives different answers

You ask something, get a useful answer, ask a colleague to try the same thing, and they get something different. Or you return to a conversation a week later, ask a follow-up, and the answer no longer fits what you remember agreeing.

Neither is mysterious, and neither is evidence of anything about the underlying model. The explanation is almost always the same: **the two questions were not the same question**, because a prompt is not what reaches the system.

This unit is about what actually reaches it, and what follows for checking.

5.1.2 What actually goes in

Unit 3 described generation as conditioned on a context. That context is a single sequence of text assembled just before generation, and your typed prompt is only part of it. The rest typically includes:

- **Instructions you did not write.** A deployed product typically prepends its own standing instructions about tone, refusals, format and role. You do not normally see them, and they can change between releases.
- **Earlier turns of this conversation.** Generally all of them, though a long conversation may be truncated or summarised rather than sent whole. This is why a long conversation costs more per question than a short one, and why something you said twenty messages ago is still shaping the answer.
- **The contents of attachments**, converted to text.
- **Retrieved passages**, if the system searched (Unit 4).
- **Tool results** from earlier in the exchange.

Two consequences follow immediately, and they are the practical content of this unit.

Your prompt is a minority of the context. The thing you are tempted to perfect is one contribution among several. This is why “prompt engineering” framed as finding magic words misses: much of what determines an answer is what else is in the window.

The same words in a different conversation are a different input. So “it gave me a different answer” is not necessarily sampling, or a changed model, or anything about the system at all. It may simply be that the input differed, because the input was never just your sentence.

5.1.3 The window has an edge

The context is finite. Every system has a limit on how much text it can attend to at once, and when a conversation or a document exceeds it, something must go.

What happens then varies and is not usually shown to you. Earlier turns may be dropped, or summarised into something shorter, or a long document may be cut into pieces of which only some are retrieved. All are reasonable engineering; all mean that **material you supplied may not be in the context when the answer is generated**.

This produces a failure mode with a distinctive shape, and it is the one worth carrying out of this unit:

> The system answers as though your document said less than it does — and the answer is confident, because from where it was generated the document *did* say less.

You supplied a forty-page report and asked a question about a clause on page thirty-one. The answer is fluent, cites page four, and is wrong. Nothing about its register distinguishes this from the case where the whole document was present. This is why a **locator** matters — the part of a citation that says where to look, such as a page, a section or a clause. An answer that names a place lets you find out whether it read that place at all.

The practical response is not to trust a limit you cannot see. For anything that matters, ask about a bounded piece of a document rather than the whole of it, and ask for the passage the answer relies on.

5.1.4 Attachments are not references

Attaching a document does two things worth keeping apart.

It **moves the task into grounded answering** (Unit 1): the reference is now something you hold, so the check is bounded.

It does **not** make the answer an extract. What comes back is still generated text conditioned on the document. It can paraphrase past what the document supports, attach a real page number to a claim from elsewhere, or answer from the part of the document that survived the window while sounding as though it read all of it.

So the discipline is short and it is the same one every time:

1. Ask for the passage, not just the answer.
2. Read the passage in the document, not in the answer.
3. Ask whether that passage settles the question you asked.

Step 3 is the one people skip. A passage can be genuinely present, genuinely quoted, and about something adjacent to your question.

5.1.5 Three things that change the answer, and are not magic words

Stating what would count as correct. This is the move from Unit 1: it turns a vague request into generation to a specification, and it is what Unit 10 is about. It changes the answer more reliably than any phrasing trick, because it changes what you can check the answer against.

Giving an example of the form you want. Useful and cheap, with one hazard: an example constrains the shape of the answer, not its truth. A response in exactly the format you demonstrated can be entirely wrong, and the matching format makes it read as compliant.

Starting a new conversation. Underrated. If a conversation has gone somewhere you did not intend, the accumulated context is now working against you, and clearing it is more effective than arguing with it. This is also the honest way to ask a second time: a fresh conversation is a different input, and if you want to know whether an answer depends on how you got there, that is how to find out.

Note what none of these do. None makes the output self-verifying. They change what you get and how easily you can check it; they do not remove the check.

5.1.6 One before and after, on a non-code task

You have attached a twenty-page departmental handbook and you want to know what it says about late submission.

Before.

> What does this say about late submission?

What comes back is likely to be fluent, plausible, and about late submission. It may also be right. The problem is not that it is wrong; it is that **you cannot tell**, and finding out means reading the handbook, which is the work you were trying to bound.

After.

> Using only the attached handbook, list every rule it states about late submission. For each one give the section number and quote the sentence it comes from. If the handbook does not state a rule about something, say so rather than filling the gap.

Three things changed, and none of them is a magic word.

- **It states what would count as correct** — every rule, from this document only. That is Unit 1's move into generation to a specification, and it gives you something to compare the answer against.
- **It asks for locators.** Each row now points at a place you can open. A fabricated section number is visible immediately; an unbounded paraphrase is not.
- **It says what to do about absence.** Silence about a rule and a statement that no rule exists are different findings, and a request that does not distinguish them will get the more fluent one.

What has *not* changed is worth being equally clear about. The second answer is not verified. It is *checkable*, which is different and is the whole point: you can now spend two minutes confirming three section numbers rather than an afternoon reading a handbook to find out whether you were told the truth.

And if a quoted sentence turns out not to be at the section given, you have learned something specific, not "the answer was wrong" but *how* it was wrong, which is what Unit 9 turns into a named vocabulary.

5.1.7 What follows for checking

Everything above cashes out as three habits.

Record the whole input, not just your prompt. When you keep a prompt for your evidence pack — as Unit 7 asks you to — a prompt without its conversation is not a record of what produced the output. Note whether the conversation was fresh, what was attached, and whether anything was retrieved.

Treat a fresh conversation as the reproducible case. If you want to know whether a result holds, ask in a clean context. Repetition inside the same conversation is the least informative kind, because the earlier answer is now part of the input.

Ask for locators, always. "The policy permits this" is unbounded. "Clause 4.2, second sentence" is bounded. The second costs the system nothing extra and turns an open-ended search into a check you can finish, and when the locator turns out not to exist, you have learned something the fluent version would have hidden.

Two things follow that are easy to hear as opposites. Prompting matters, and Unit 10 is about writing a specification precise enough to be worth prompting with. What does not follow is that a better phrase reaches material the system never received: wording cannot recover what is not in the context. Those are different claims and this unit means the second.

Long documents are usable. The limit is one you cannot see from the outside, so the move is to supply the part that matters rather than to hope the whole arrived.

5.1.8 Summary

- What reaches the system is an assembled context: standing instructions you did not write, the whole conversation so far, attachments, retrieved passages and tool results. Your prompt is a minority of it.
- Different answers to "the same question" usually mean the inputs differed, because the input was never just your sentence.
- The context has an edge, and what falls off it is not shown to you. The characteristic failure is a confident answer generated from a document that was only partly present.
- An attachment moves the task into grounded answering; it does not turn the answer into an extract.
- Ask for the passage, read it in the document, and ask whether it settles your question. Record the whole input, not just the prompt.

Then answer `cp05-what-reaches-the-system` and `cp05-locators` at the foot of this page.

5.2 Unit 5 activity — find the edge of the window

Time: about 14 minutes. Read `page.md` first.

Three short experiments on the thing you cannot see. You need one document of your own that is longer than a page or two — a handbook, a long article, a report, a set of lecture notes.

5.2.1 Part 1 — the same question, two contexts (4 minutes)

Ask a question you know the answer to, twice: once at the start of a fresh conversation, and once at the end of a conversation you have already been having about something else.

Record both answers and write one line on how they differ. Length, format, confidence, assumptions carried over from the earlier topic.

Then answer this in a sentence:

> If I had only seen the second answer, what would I have concluded about the system?

The point is not that one answer is better. It is that "the same question" described two different inputs, and nothing on screen said so.

5.2.2 Part 2 — the buried clause (6 minutes)

This is the experiment worth doing carefully.

1. Take your document and find something stated **late** in it — the last quarter — that is specific and checkable. A number, a deadline, a condition, a named exception.
2. Start a fresh conversation, attach the whole document, and ask a question that can only be answered from that late passage.
3. **Ask for the locator**: the section or page, and the sentence it comes from.
4. Go and look.

Record three things: what it said, whether the locator was real, and whether the quoted sentence is at that location and says what the answer said it says.

There are four outcomes and all of them are informative:

What happened	What you have found
Correct answer, correct locator, sentence matches	The passage was in the context, and you now know how to confirm that cheaply
Correct answer, no locator offered	You have an answer you cannot check without reading the document. Ask again and insist
Confident answer, locator points somewhere else	The characteristic failure of this unit. Note whether anything in the wording warned you
The system says the document does not address it	The most useful outcome, and the rarest, because it is the one that required saying no

If your document is short enough that everything fits, say so — that is a result too, and it tells you the bound is above the size you work with.

5.2.3 Part 3 — before and after (4 minutes)

Take the vaguest question you asked in Parts 1 or 2 and rewrite it the way the reading's example does: state what would count as correct, ask for locators, and say what to do about absence.

Ask it. Then write two or three sentences on what changed — not about which answer is better written, but about **what you can now check that you could not check before**.

Keep all three parts. Unit 7 asks you to record a prompt, and after this unit that means recording the whole input: fresh conversation or not, what was attached, what was retrieved.

5.2.4 Now take the checkpoints

`cp05-what-reaches-the-system` is about what the context actually contains. **`cp05-locators`** is about Part 2's outcome.

```
python3 selfcheck.py run cp05-what-reaches-the-system
python3 selfcheck.py run cp05-locators  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check.

5.2.5 Optional extensions

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

Find the edge deliberately (about 8 minutes). Repeat Part 2 with progressively longer documents until the late-passage question starts failing. You will not get a clean threshold — systems handle overflow differently and the behaviour is not documented — and that is the finding. Record the sizes at which it worked and stopped working, and note that you now know something about your tool that its documentation does not tell you.

Test the format trap (about 4 minutes). Ask for an answer in a very specific format — a table with named columns — about something you can check, and see whether a perfectly formatted answer can be wrong. Compliance with a format is the cheapest kind of compliance to obtain and the easiest to mistake for care.

5.2.6 Activity Answers

5.3 Unit 5 answers and guidance

Read this after attempting the activity.

5.3.1 Part 1 — what the difference means

The common finding is that the second answer carries something over: an assumption about your situation, a format from the earlier topic, a level of detail calibrated to a different question. That is the accumulated context doing what it does.

The question at the end of Part 1 is the one that transfers. If you had seen only the second answer, you would probably have concluded something about the *system* — that it is verbose, or that it assumes too much, or that it is good at this. All of those would have been conclusions about a system drawn from evidence about an input. This is a small instance of a large error, and it is worth having made once in a controlled way.

If the two answers were near-identical, that is a real result and not a failed experiment. A short earlier conversation on an unrelated topic may change little. Try again after a long conversation with strong assumptions in it.

5.3.2 Part 2 — the four outcomes

Nothing here is a pass or a fail. What is being tested is whether you can tell which outcome you got, which requires having gone and looked.

The third row is the one to sit with. When a confident answer cites a location that does not support it, look back at the wording and ask honestly whether anything in it warned you. Usually nothing did — and that is the unit's point rather than a complaint about the tool. The register of an answer generated from a partial document is the register of an answer generated from a whole one.

The fourth row deserves comment because it is rare. An answer that says "the document does not address this" has declined to fill a gap, and gap-filling is the more fluent option. If you got one, notice what you asked for: the reading's example prompt explicitly says what to do about absence, and requests that do not distinguish silence from absence tend to receive the more confident of the two.

If everything fitted, your document was inside the bound. Say so and move on. Reporting "I could not reproduce the failure at this size" is a finding with a condition attached, which is exactly the form this module wants.

5.3.3 Part 3 — what changed

The answer being looked for is *checkability*, not quality.

A good response says something like: before, confirming the answer meant reading the document; after, it means opening three named sections. That is the same claim the reading makes about the handbook example, arrived at on your own material.

A weaker response says the second answer was more detailed, or better organised, or more professional. Those are properties of the text. The second answer may also be wrong — the exercise does not establish otherwise — and if it is, you can now find out in two minutes.

5.3.4 The checkpoints

`cp05-what-reaches-the-system` tests the model of the context window. The tempting wrong answers all treat the typed prompt as the input, which is the assumption behind both "prompt engineering as magic words" and "it gave me a different answer, so the model changed".

`cp05-locators` tests Part 2's judgement. The trap is treating a plausible locator as a check. A section number is not evidence until you have opened the section — it is a claim like any other in the answer, and it is the one claim in the answer that is cheap to falsify.

5.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp05-what-reaches-the-system` — What actually reaches the system

Kind: multi.

You type one sentence into a chat product that has a document attached and a conversation already in progress. Which of the following are typically part of the context the answer is generated from? Select all that apply.

Options, numbered from zero:

0. Standing instructions written by the product, which you do not see
1. The earlier turns of this conversation
2. The text of the attached document, or as much of it as fits
3. A record of which of the model's previous answers turned out to be correct
4. Passages the system retrieved, if it searched

Answer: `0, 1, 2, 4`

Hint: Think about what the service must be sent in order to answer at all, and then about what it has no way of knowing. One item on the list would require something in the exchange to have evaluated an earlier answer.

Diagnostics, by the answer given:

- `1, 2` — You have the parts you can see and not the ones you cannot. A deployed product almost always prepends its own standing instructions, and retrieved passages join the context when the system searches. The consequence matters: your prompt is a minority of the input, so perfecting it is not where most of the variation comes from.
- `0, 1, 2, 3, 4` — Four of the five, but not the record of past correctness. Nothing in the exchange tracks which earlier answers turned out to be true; there is no feedback of that kind in the context. This is the assumption behind expecting a system to learn from having been corrected earlier in a conversation – what persists is the text of the correction, not a verdict about it.
- `2` — The attachment is in the context, but so is everything else listed except the correctness record. Treating the document as the input is how the second most common surprise in this unit happens: an answer shaped by twenty earlier turns that you were no longer thinking about.

Explanation: The context is an assembled sequence: standing instructions you did not write, the whole conversation so far, attachments converted to text, retrieved passages, and tool results. What it does not contain is any record of which earlier answers were right, because nothing in the exchange evaluates that. Two things follow. Your prompt is a minority of the input, and the same words in a different conversation are a different input – which is why 'it gave me a different answer' is usually a statement about the input rather than about the model.

Checkpoint `cp05-locators` — What a locator is worth

Kind: choice.

You attached a forty-page report and asked about a condition stated on page thirty-one. The answer is specific and confident, and cites page four. What have you established?

Options, numbered from zero:

0. Nothing yet, and you have a cheap way to find out: open page four and see whether it says this.
1. That the answer is wrong, because the condition is on page thirty-one and not on page four.
2. That the system did not receive the whole document, since it cited the wrong page.
3. That the answer is probably right, since a specific page reference is unlikely to be invented.

Answer: `0`

Hint: A page number is a claim like any other claim in the answer. Ask what it would take to find out whether that particular claim is true, and how long that would take.

Diagnostics, by the answer given:

- `1` — Close, and it skips a step. The answer may be describing something genuinely on page four that also bears on your question, or the report may state the condition in more than one place. A citation you have not opened is a claim, not an error and not evidence. Open it first.
- `2` — A plausible explanation, and not one this observation establishes. A wrong page could come from a document that was only partly in the context, or from a passage that was present and misread, or from a real page four that says something adjacent. Naming a cause before checking is the move this module trains you out of.
- `3` — Specificity is a property of the text, not evidence about the world – this is Unit 3's point, arriving in a new costume. A precise locator is valuable for the opposite reason: it is cheap to falsify. That value is only realised by opening it.

Explanation: A locator is not evidence; it is what makes evidence cheap to obtain. Before you open page four you have established nothing, and after you open it you will have established a great deal in about thirty seconds – either the passage is there and says this, or it is there and says something adjacent, or it is not there at all. Those are three different findings with three different responses, and none of them is available from the answer's register. This is why asking for locators is worth more than asking for confidence.

6 Unit 6 — Meet the Bernoulli numbers

6.1 Reading

Media for this unit:

- **Animation**: *First divergence* — this unit has an interactive version on the course website (`first-divergence.html`); the still below is one moment from it.

Figure 7: First divergence. Two Bernoulli sequences advance in step and part at n=1, where the oracle gives -1/2 and the submitted routine gives +1/2.

- **Animation**: *Two indexings, one term* — this unit has an interactive version on the course website (`ada-indexing.html`); the still below is one moment from it.

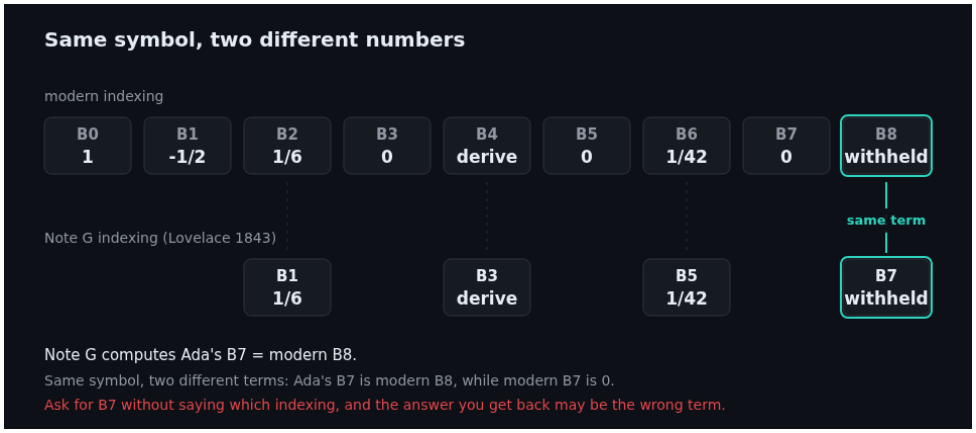

Figure 8: Two indexings, one term. Modern Bernoulli indexing and Note G's indexing start aligned and slide out of alignment, until Ada's B7 sits under the modern B8.

Reading time: about 11 minutes. Activity: about 20 minutes.

6.1.1 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.

6.1.2 Why this unit comes before the AI unit

In Unit 7 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.

6.1.3 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.

6.1.4 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`.

6.1.5 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 2.

6.1.5.1 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 `cp06-b4-by-hand`, and it is worth doing before you read further into this page. Stop here 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.

6.1.6 Post-attempt Reading

6.1.7 Trap 1 — there are two conventions

Here is a scenario you will meet in Unit 8. 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 10.

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 8 this failure mode has a name: `b1-sign`.

6.1.8 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 8 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`.

6.1.9 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 `0.16666666666666666` 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.

6.1.10 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.py`.

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

6.1.11 Check yourself

This unit has three checkpoints:

- `cp06-b4-by-hand` — derive B4 from the recurrence
- `cp06-convention` — identify which convention a program is using
- `cp06-ada-index` — translate a Note G label into a modern index

Run them by either route:

```
python3 selfcheck.py run cp06-b4-by-hand  # or in the browser
```

For `cp06-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.

6.2 Unit 6 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 8, and it belongs in your evidence pack for Unit 17.

6.2.1 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.

6.2.1.1 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.

```
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) * b3
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.

6.2.1.2 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.

6.2.2 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 10's precise prompt, and with the failure-mode name it corresponds to in Unit 8.

6.2.3 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 8 compares index by index against a reference instead of asking whether output "looks like" Bernoulli numbers.

6.2.4 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:
    """Describe the first index where claimed differs from TABLE."""
    ...
```

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 8 hands you.

6.2.5 Now check yourself

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

- `cp06-b4-by-hand` — the value you derived in Task 1
- `cp06-convention` — the disagreement you diagnosed in Task 2
- `cp06-ada-index` — the mapping you worked out in Task 3

Either route:

```
python3 selfcheck.py run cp06-b4-by-hand  # or in the browser
```

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

To attempt all three at once:

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

The browser route, at the foot of this unit's page, has the same three checkpoints, the same answers and the same feedback, and needs nothing installed.

A wrong answer on `cp06-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.

6.2.6 Activity Answers

6.3 Unit 6 answers and guidance

Read this after attempting the activity. Working through it before 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.

6.3.1 Task 1 — B4 by hand

6.3.1.1 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`.

6.3.1.2 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.** You reach `1/6 + 5·B4 = 0` and read 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.

6.3.1.3 Why this task exists The value matters less than its provenance. In Unit 7 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: you can still open the source or the disclosure. The instruction is to attempt the derivation before you do.

6.3.2 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 10's precise prompt. If yours is close, you have independently reinvented a piece of requirements engineering, which is what specification work mostly is.

6.3.3 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 uses Lovelace's own labels returns `1/6` for `B1`, `-1/30` for `B3`, `1/42` for `B5` — her numbering runs `B1, B3, B5, B7` for the modern `B2, B4, B6, B8`, and she has no `B2` at all. Every value it returns is a genuine Bernoulli number, correctly computed, exactly represented. Every one is filed under another century's 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 8.

A note on the history: Note G's published table contains printed errors — the inverted division at operation 4 is established from a facsimile, and a second in operation 24 rests on a published reconstruction — while their attribution (who introduced them) remains contested. This unit does not depend on the error at all, and Unit 16 handles it properly. See `course/historical-notes.md` before repeating any account of it.

6.3.4 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 you wrote `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.

To go further, 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.

6.3.5 Checkpoints

Attempt these from your own working:

- `cp06-b4-by-hand`
- `cp06-convention`
- `cp06-ada-index`

```
python3 selfcheck.py run cp06-b4-by-hand  # or in the browser
```

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

6.3.6 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp06-b4-by-hand` — B4 by hand

Kind: fraction.

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 a fully reduced exact fraction.

Answer: `-1/30`

Hint: Take n = 4. The identity gives C(5,0)B0 + C(5,1)B1 + C(5,2)B2 + C(5,3)B3 + C(5,4)B4 = 0. The coefficients are row 5 of Pascal's triangle: 1, 5, 10, 10, 5. You know the values too: B0 = 1, B1 = -1/2, B2 = 1/6, and B3 = 0 because it is odd and above 1. Substitute all of that, then solve the single remaining unknown. Keep everything as fractions – do not reach for a calculator.

Diagnostics, by the answer given:

- `1/30` — 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.
- `1/6` — That is B2, not B4. Easy to land on if the recurrence was solved one step short. Run it once more, taking n = 4.
- `-1/6` — You are one step from the answer. -1/6 is the negated sum of the known terms: you have correctly reached 5*B4 = -1/6 but stopped before dividing by 5. Divide through by the coefficient on B4.
- `0` — B4 = 0 usually means the wrong binomial row: row 4 (1, 4, 6, 4, 1) instead of row 5. The identity for n = 4 uses n + 1 = 5 on top, so the coefficients are 1, 5, 10, 10, 5. Check which row you used.
- `-1/42` — You have mixed up denominators: 42 belongs to B6. Recompute the coefficient on the highest term for n = 4.

Explanation: 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 8.

Checkpoint `cp06-convention` — Which convention is this?

Kind: choice.

A program prints B0 = 1, B1 = 1/2, B2 = 1/6, and matches this module at every other tested index. Which standard Bernoulli convention is consistent with what it printed?

Options, numbered from zero:

0. The second convention, which takes B1 = +1/2 and agrees everywhere else.
1. The first convention, which is the one this module uses.
2. Lovelace's Note G indexing.

3. None: no published convention gives these values.

Answer: 0

Hint: Two of the printed values disagree with this module and the rest agree. Look at which index the disagreement is at, and ask which published convention differs there and nowhere else.

Diagnostics, by the answer given:

- 1 — The first convention takes B1 = -1/2, which is the value this program did not print. That is the one index where the two conventions differ, and it is the index in question.
- 2 — Note G's indexing shifts *which* numbers get which labels, so B2 and B4 would move too. Here only B1 differs, which points at a convention rather than an indexing scheme.
- 3 — Both values of B1 are published and standard. A disagreement at exactly one index, with every other value matching, is the signature of the convention choice rather than of an error.

Explanation: Both conventions are standard and published, and the second one is consistent with this output. Note what the question asks and what it does not: consistency is not a diagnosis. A sign bug affecting only B1 would print exactly the same values, and no amount of staring at the output separates the two – you would have to read the specification, the documentation or the code. What the comparison establishes is that the disagreement sits where a known convention choice sits, which tells you what to go and look at. This is why the module's contract states B1 = -1/2 explicitly, and why Unit 10 puts that sentence in the prompt: a stated convention makes the question unnecessary.

Checkpoint `cp06-ada-index` — Reading Ada's index

Kind: integer.

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: 8

Hint: Lovelace numbers only the terms that are not zero. Write out the modern sequence, strike out every zero term, and count along to the position she calls B7.

Diagnostics, by the answer given:

- 7 — That is the label, not the value's modern index. Lovelace numbers only the non-zero Bernoulli numbers – the odd ones that vanish never get a label at all, so her label numbers are one smaller: her Bk is the modern B(k+1).
- 14 — Doubling is the right instinct but the wrong rule. Her B1 is the modern B2, her B3 is the modern B4. Write out the first four pairs and read off the pattern.
- 6 — You have moved the wrong way. Her label numbers are one smaller – her Bk is the modern B(k+1) – because the modern scheme spends indices on the zero terms and hers does not.

Explanation: 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.

7 Unit 7 — Ask AI to generate the routine

7.1 Reading

Reading time: about 10 minutes. Activity: about 12 minutes, plus optional extensions.

Prerequisite: Unit 6. 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.

7.1.1 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 8 will check and Unit 17 will use as evidence.

7.1.2 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.

7.1.2.1 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 10.

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.

7.1.3 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 6 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.

7.1.4 The failure modes to expect

The module names five failure modes in generated Bernoulli code. These are the codes the checker emits in Unit 8, 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 a type other than `fractions.Fraction`.

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 a common outcome of asking for numerical Python, which tends to be float-based. Note what is wrong with it and when. The type is wrong from `B0` onwards: the contract asks for a `fractions.Fraction`, and a float is not one, whatever the digits look like. `B0 = 1.0` is exactly representable, so nothing is inaccurate there — the fault is the type, not the value. The inaccuracy arrives separately, because binary floating point cannot represent most of the later Bernoulli fractions at all. For small indices it is convincing anyway: `-0.0333...` really is close to `-1/30`. A test that checked `B0` to `B12` with a tolerance would pass such a routine, and choosing that tolerance is a decision about the task, not a default.

What these five have in common is the important part: **none of them makes the code look suspicious.** A careful reader with the specification in hand can spot some of them — a missing `Fraction`, a special case at `B1`, a loop that steps over the odd indices, but casual reading misses them, and reading alone cannot establish that a routine meets a specification. Testing locates them; reading tells you what to test for. That asymmetry is the argument for the verification lab in Unit 8.

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.

7.1.5 The sixth failure mode, which no test catches

Generated code often 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 16 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 `cp07-trust-triage`.

7.1.6 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 18 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.

7.1.7 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 8** runs it through the checker and names what is wrong with it.
- **Unit 17** 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 8 executes the file you give it — in a separate process with limits, which contains accidents rather than making unread code safe. That is a real consideration and not a formality.

7.1.8 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.

7.1.9 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. Unit 13 covers what leaves your device and why it matters; sustainability is signposted rather than taught, and the resource section in Unit 4 is as far as this module takes it.

7.1.10 Check yourself

This unit has one checkpoint:

- `cp07-trust-triage` — what you can trust before testing

```
python3 selfcheck.py run cp07-trust-triage  # or in the browser
```

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.

7.2 Unit 7 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 8 checks the file, and Unit 17 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.

7.2.1 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.

7.2.1.1 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.

7.2.2 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 6 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`?

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

Before Unit 8 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 8 tells you something about your own judgement, not only about the code, so write it down even if you have no idea.

7.2.4 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 16 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.

7.2.5 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 8 needs items 1 and 4. Unit 17 needs all four for your evidence pack. See `course/learner-evidence-template.md` for the format it will eventually take.

7.2.6 Now check yourself

This unit has one checkpoint:

- `cp07-trust-triage` — what you can trust before testing

Either route:

```
python3 selfcheck.py run cp07-trust-triage  # or in the browser
```

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

The browser route, at the foot of this unit's page, 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.

7.2.7 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.

If you 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 6 grounding that test relies on.

7.2.8 Activity Answers

7.3 Unit 7 answers and guidance

Read this after attempting the activity.

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. `cp07-trust-triage` is defined in `course/lab/checkpoints.py`, 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.

7.3.1 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 17 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.

7.3.2 Task 2 — Triage

7.3.2.1 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 6 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(n)` 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.

7.3.2.2 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 8 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.

7.3.2.3 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 10 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.

failure mode	the requirement that was never stated
`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 10, 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.

7.3.3 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 8 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.

7.3.4 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 16 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 18 supplies the formal definitions and full scope; they are not required for identifying the unsupported transfer here.

7.3.5 Optional extensions

None of this is required work, and none of it is needed for the checkpoint. The notes below are for anyone who takes an extension.

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 6 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.

7.3.6 Checkpoint

- `cp07-trust-triage` — what you can trust before testing

```
python3 selfcheck.py run cp07-trust-triage  # or in the browser
```

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

7.3.7 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp07-trust-triage` — What you can trust before testing

Kind: choice.

An AI returns a Bernoulli routine with a comment: '# Ada Lovelace used this same recurrence in Note G (1843)'. The code passes your spot check of B2 and B4. What is the right response to the comment?

Options, numbered from zero:

0. Treat the comment as an unverified historical claim and check it separately.
1. Accept it – the code was right, so the comment is probably right too.
2. Treat the comment as settled and move on; comments do not affect behaviour.
3. Accept it if the model sounds confident about the date.

Answer: `0`

Hint: The code and the comment are two claims of different kinds. Ask what your spot check established, and whether any amount of testing the code bears on a statement about 1843.

Diagnostics, by the answer given:

- `1` — Correct code is no evidence at all about an adjacent historical claim. They are generated the same way and fail independently – the arithmetic being checkable is precisely why it got checked, and the history did not.
- `2` — It is true that a comment does not affect execution – but it travels with the file, and the next reader will take it as vetted along with the code. 'Does not affect behaviour' and 'settled' are different claims: the historical claim is still unchecked, and moving on leaves it unchecked in something you now vouch for.

Explanation: Generated output is a mixture of things with completely different failure modes: arithmetic you can test, and assertions about the world you can only look up. Both can be equally polished, so surface quality alone does not establish reliability.

8 Unit 8 — Verification lab

8.1 Reading

Media for this unit:

- **Animation**: *First divergence* — this unit has an interactive version on the course website (`first-divergence.html`); the still below is one moment from it.

Figure 9: First divergence. Two Bernoulli sequences advance in step and part at n=1, where the oracle gives -1/2 and the submitted routine gives +1/2.

- **Animation**: *Revision is not independent verification* — this unit has an interactive version on the course website (`self-certification.html`); the still below is one moment from it.

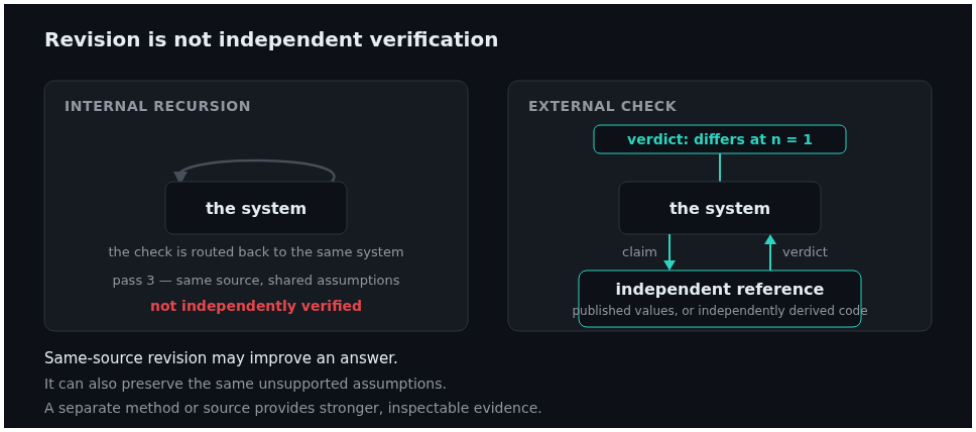

Figure 10: Revision is not independent verification. A claim routed back into the system that made it is revised without independent evidence; the claim sent to a separate reference receives an independently inspectable comparison.

Reading time: about 18 minutes. Activity: about 27 minutes.

8.1.1 What this unit is for

In Unit 7 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.

8.1.2 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.

8.1.3 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 table they came from is recorded at `course/references.md` [M2]: the NIST Digital Library of Mathematical Functions, §24.2, Tables 24.2.1 and 24.2.3, checked value by value on 2026-07-24, all twenty-two agreeing and no correction needed. 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_example.py` 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.

8.1.4 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.

8.1.5 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.

8.1.6 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`	Lovelace’s Note G labels: only the non-zero terms are numbered, and with the odd integers, so her `B1, B3, B5, B7` are the modern `B2, B4, B6, B8` and there is no `B2`.
`off-by-one`	Every value shifted one index.
`one-observation`	`ada-indexing` and `off-by-one` both matched. They agree everywhere except `B0`, so this is one observation with two readings.
`not-fraction`	Returns a type other than `fractions.Fraction`. A `float` is additionally inexact; an `int` is exact and still wrong.
`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 10 is built on.

`examples/wrong_odd_terms.py` → `odd-terms`, `ada-indexing` and `one-observation`. First divergence at `n=1`: expected `-1/2`, got `1/6`. This routine uses Lovelace's Note G labels — only the non-zero terms are numbered, and they take the odd integers, so her `B1, B3, B5, B7` are the modern `B2, B4, B6, B8` and there is no `B2`. Against the modern convention that is inverted: the odd indices carry the numbers and the even ones are empty. Every value it returns is a genuine Bernoulli number; every one is filed under another century's label.

The third diagnosis is the interesting one. A plain off-by-one shift produces *identical* output at every index except `B0`, because her `B_n` at odd `n` is the modern `B_(n+1)` and both schemes give zero at the even indices. So the checker reports both readings and names `B0` as the whole of the evidence separating them — 1 here, where a shift would have put `-1/2`. One value is thin evidence, and settling it means reading the source. A comparison tells you where to look; it does not always tell you what you are looking at.

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

8.1.7 Safety: the checker executes what you give it

Stop before you run anything.

`check.py` executes the file you hand it. That is the only way it can test the code. It does not execute it in its own process: the file runs as a separate process with a timeout, resource limits and an environment stripped of your variables, and only data comes back.

That is a blast-radius reduction, not a security boundary — the child still runs as you, and can read what you can read. It buys you two things: an accident becomes a finding rather than a hung terminal, and your shell's API keys are not handed to code you did not write.

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`.
- **For anything you have not read, or that came from a source you have no reason to trust** — a container, a virtual machine, or a throwaway account. The table route below completes this unit without executing anything.
- **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.

8.1.8 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.

8.1.9 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 17 will ask you to write two sentences saying why your answer is correct, citing what you checked rather than what produced it.

8.1.10 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 18 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.

Take BabaYaga, a reasoning-systems project we are building. It is not released, so you cannot check what follows — notice that, and treat the claim accordingly. In it, no number is typed into the prose of a paper. A build step reads each one out of the files the code produced, and if a quoted value and its source no longer agree it refuses to build, printing *undefined* rather than a stale figure.

Trusting a computation goes further. 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.

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 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, look at the Lucifuge scheduler, published at https://dev.astarte.org. Its test suite checks the example configurations printed in its own documentation, 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 18 teaches from; nothing in this unit, its activity, or its checkpoints depends on either.

8.1.11 Checkpoints

Complete this unit at `cp08-diagnose` plus the checkpoint for your route. `cp08-diagnose` runs by either route. The browser/table route uses `cp08-table-evidence` to assess conclusions from manual comparison. `cp08-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 cp08-diagnose  # or in the browser
python3 selfcheck.py run cp08-bernoulli          # Python route only
python3 selfcheck.py run cp08-table-evidence  # table route
```

8.2 Unit 8 activity — Check what you generated

Time: about 27 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 17.

8.2.1 Before you start: the safety step

This step belongs to both routes and comes first.

`check.py` executes the file you give it, in a separate process with a timeout and resource limits — which limits the damage an accident can do, and does not make unread code safe. Your Unit 7 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.

If you did not read it, or you have no reason to trust where it came from, run the lab in a container or a virtual machine rather than on your own account — or take Route B, which executes nothing. Not as an administrator, either way.

8.2.2 Route A — Automated verification

8.2.2.1 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.

8.2.2.2 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?

8.2.2.3 A3. Check your own code (5 minutes) Put the routine you saved in Unit 7 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 7 prompt happened to get right by luck rather than by stating it.

8.2.2.4 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 10 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 17 defence.

(Indices compared: 0 to 30)
```

8.2.2.5 A5. One last question The checker passed. Write down, in one sentence, something that is still not established about your routine by that pass.

8.2.3 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.

8.2.3.1 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.

8.2.3.2 B2. Get the generated values (4 minutes) Go back to the AI system you used in Unit 7 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 7 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.

8.2.3.3 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`?

8.2.3.4 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 uses Lovelace's Note G labels, so her `B1, B3, B5, B7` are the modern `B2, B4, B6, B8`, and her even indices carry nothing. Its column therefore reads `1` at `n = 0`, then `1/6, 0, -1/30, 0, 1/42` 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.

8.2.3.5 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.

8.2.3.6 B6. Check what the table evidence supports (4 minutes) Open `cp08-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.

8.2.4 Finish the unit at the checkpoints

The checkpoints are formative and unmarked. `cp08-diagnose` and `cp08-what-it-shows` are shared: the first names the failure, the second asks what your check did and did not establish, which is the same question on either route. `cp08-table-evidence` records the manual evidence objective on Route B. `cp08-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 cp08-bernoulli
python3 selfcheck.py run cp08-diagnose
python3 selfcheck.py run cp08-what-it-shows
```

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

```
python3 selfcheck.py run --unit 8
```

Browser route — answer them at the foot of this unit's page, with no installation: `cp08-table-evidence`, `cp08-diagnose` and `cp08-what-it-shows`. The browser self-check cannot run code, so it cannot grade `cp08-bernoulli`; it lists that checkpoint and directs you to complete it in the Python lab.

`cp08-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
```

8.2.5 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.

8.2.6 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 16; you can do it now or after that unit.

8.2.7 Activity Answers

8.3 Unit 8 answers and guidance

Answer key for the Unit 8 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.

8.3.1 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.

"Nothing suspicious, here is why", with specific reasons, is doing this correctly. "It looked fine" is not doing it at all.

8.3.2 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`	a float	`not-fraction`, `odd-terms`
`off_by_one_range.py`	0	`1`	`-1/2`	`off-by-one`, `odd-terms`
`correct_example.py`	—	—	—	none; PASS

The float in the third row is exactly `0.1666666666666674`, which is written out here rather than in the cell because a nineteen-character literal does not fit a table column on a printed page.

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.

8.3.3 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 10 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.

8.3.4 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.

8.3.5 The two routes

Both routes evidence the same skill: ordered comparison against a trusted reference, stopping at the first divergence, and naming the fault. If you completed Route B you have verified. Route A is not the real version and Route B the substitute.

What makes the work good, 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.

8.3.6 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 take from this: every clause you needed in your test corresponds to a clause the specification should have contained. That list is the one you build in Unit 10.

`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.

8.3.7 Checkpoints

`cp08-bernoulli`, `cp08-table-evidence`, and `cp08-diagnose` are self-marking, with their own diagnostic feedback, and their answers are deliberately not reproduced here. Learners reach `cp08-diagnose` by either route; `cp08-bernoulli` belongs to the Python route and `cp08-table-evidence` to the browser/table route:

```
python3 selfcheck.py run cp08-diagnose  # or in the browser
python3 selfcheck.py run cp08-bernoulli    # Python route only
python3 selfcheck.py run cp08-table-evidence  # table route
```

8.3.8 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp08-bernoulli` — Make the checker pass

Kind: code.

Complete course/lab/exercises/ex04_bernoulli.py so that bernoulli(n) meets the contract, then run: python3 selfcheck.py run cp08-bernoulli

Explanation: You have now done the full loop: a specification precise enough to test, an implementation, and an oracle that judges it. Note which of the three took the longest to get right – for most people it is the specification.

Checkpoint `cp08-what-it-shows` — What the check established, and what it did not

Kind: multi.

A generated routine passes the checker, and its docstring makes five claims. Which of them does a passing run NOT establish? Select all that apply.

Options, numbered from zero:

0. Returns exact Fraction values for every index from 0 to 30.
1. Uses the same algorithm as the table in Ada Lovelace's Note G.
2. Is correct for every n greater than or equal to 0.
3. Odd-index values above B1 are exactly zero from 0 to 30.
4. Runs in time linear in n.

Answer: `1, 2, 4`

Hint: Ask of each claim: which of them could the checker have found false? It compares returned values against the oracle over a fixed range of inputs, and does nothing else – it does not read the source, and it does not time the run.

Diagnostics, by the answer given:

- `2` — The range claim is the obvious one, and it is not the only one. The checker reads what comes back for the inputs it tried: it never sees the algorithm, and it never times anything. Two more of these claim things no output comparison can show.
- `1, 2` — Close. Those two are unsupported, and so is one more: the checker measures no timing, so a complexity claim rests on reading the code, not on running this check.
- `0, 1, 2, 3, 4` — Two of these the run does establish, over the inputs it covered: the return type and the odd-term property from 0 to 30. Refusing to accept what evidence does support is the mirror image of overclaiming, and it leaves you unable to say anything at all.
- `2, 4` — Both of those are right, and one is missing: nothing in a comparison of returned values can tell you which algorithm produced them. A lookup table and a recurrence pass identically.

Explanation: A passing check supports claims about *observed behaviour over the inputs tested*: the values, their type, and the properties checked across that range. The algorithm used, the behaviour beyond the range, and how the routine scales are all outside what it saw. This is the same discipline whichever route you took: a table comparison establishes what the compared cells show, and says nothing about the rows you did not compare or about how the table was produced.

Checkpoint `cp08-table-evidence` — State what a table comparison establishes

Kind: match.

For each observation from a manual candidate/reference table comparison, choose the strongest conclusion it supports.

Observations, in order:

1. B1 is +1/2; every other displayed value agrees with the reference.
2. B3 and B5 are absent; the remaining displayed exact fractions agree.
3. Every displayed cell from B0 to B12 and B30 agrees exactly.
4. A displayed cell is 0.1666667 where the reference gives 1/6.

Options, numbered from zero:

0. The candidate uses the other B1 convention.
1. The evidence does not establish the required odd zero rows and modern indexing.
2. Only the displayed cells are supported; no program, return type, or unlisted index has been checked.
3. Approximate decimals do not establish the required exact fraction values.

Answer: `0, 1, 2, 3`

Hint: Choose the narrowest conclusion licensed by what was actually compared. Do not infer executable behaviour from displayed cells.

Diagnostics, by the answer given:

- `0,1,3,2` — The final two conclusions are swapped. A decimal-versus-fraction mismatch is visible evidence about exact displayed values. A fully matching displayed table has the opposite limitation: it supports those cells, but says nothing about unseen code, return types, or unlisted indices.
- `1,0,2,3` — The first two conclusions are swapped. A lone B1 sign difference identifies the standard convention mismatch. Missing odd rows leave indexing and the required exact zeros unestablished, even when every row that remains is correct.

Explanation: Manual table comparison can establish exact agreement or divergence for the cells inspected, including convention, indexing, and zero-term evidence. It cannot establish that a program produced the cells, that unseen code returns Fraction, or that unlisted indices work. The Python checker supplies stronger automated coverage, but it too is limited to its stated tests.

Checkpoint `cp08-diagnose` — Name the failure mode

Kind: choice.

A submitted routine returns 1 at argument 0. At arguments 1, 3 and 5 it returns the modern B2, B4 and B6, and at every even argument above 0 it returns zero. Every value it returns is a real Bernoulli number. What is wrong?

Options, numbered from zero:

0. It uses Lovelace's Note G labels, which number only the non-zero terms and number them with the odd integers.
1. Every index has been shifted by one.
2. It uses floating point and the values have drifted.
3. Nothing – those are the Bernoulli numbers.

Answer: `0`

Hint: Every value returned is a genuine Bernoulli number, so nothing is numerically wrong. Two of the options describe patterns that agree everywhere except at one index. Find that index and read what it says.

Diagnostics, by the answer given:

- `1` — A good answer, and it is nearly indistinguishable from the right one – which is the point of this question. A one-place shift produces the same values at every index shown here, because Lovelace's B_n at odd n is the modern B_(n+1) and both give zero at the even ones. What separates them is B0: a shift would move the modern B1 = -1/2 into that slot, and this routine returns 1. Note G's table starts at B1 and leaves B0 alone.
- `2` — Float drift produces near-misses – values like 1e-16 where an exact zero should be – not exact fractions filed under other indices. These values are exact, and each one is a genuine Bernoulli number; the fault is in the labels, not the arithmetic.

- 3 — Each individual value is genuine – that is what makes this one hard. The question is whether each value is filed under the right index.

Explanation: Lovelace numbers only the non-zero terms and numbers them with the odd integers, so her B1, B3, B5, B7 are the modern B2, B4, B6, B8 and there is no Note G B2. Every value here is correct and every label belongs to another century. The harder half of the question is that a plain off-by-one shift produces identical output at every index except B0 – so the checker reports both readings and names B0 as what separates them. One value is thin evidence, and the way to settle it is to read the source rather than stare at the output. That is the same move as the convention trap in Unit 6: a comparison tells you where to look, not always what you are looking at.

9 Unit 9 — Grounding and citation checking

9.1 Reading

Media for this unit:

- **Animation**: *Claim against source, row by row* — this unit has an interactive version on the course website (`claim-divergence.html`); the still below is one moment from it.

Figure 11: Claim against source, row by row. Four claims checked in order against what their cited sources say; the fourth is the first that fails.

Reading time: about 11 minutes. Activity: about 30 minutes.

9.1.1 The second worked case

Units 6, 7 and 8 took one thing seriously: you built a reference by hand, asked for a routine, compared value by value, and stopped at the first divergence. That is the method, and it worked because the case had a decidable answer.

Most of what you will actually check is not like that. There is no oracle for "does this source support this claim". So this unit runs **the same method on a case with no oracle**, and the point is that the method survives intact:

	The Bernoulli case	The source case
The artefact	A routine you asked for	A referenced summary you asked for
The reference	A table you derived from the recurrence	What the cited sources actually say
Building it	Compute B4 by hand	Retrieve and read each source
The comparison	Value against value, by index	Claim against source, one at a time
Where you stop	The first index that disagrees	The first claim the source does not support
Naming the fault	`b1-sign`, `off-by-one`, and the rest	Six codes, in this unit
What you can then say	Correct for these inputs, under this convention	Supported at this strength, by these sources

This is not a lighter version of the lab. Building the reference is still the work, the comparison is still item by item, and the scoped statement at the end is still the deliverable. What changes is that the last column's judgements are yours, because no program can make them.

9.1.2 What a citation from a generative system establishes

By itself: nothing.

A citation produced alongside a claim is another piece of generated text. It sits in the same output, produced the same way, with no more evidence attached than the sentence it follows. Unit 3 has the mechanism; nothing in generation consults a record of what is true, and attaching a bibliography does not change that.

But a citation is the most useful sentence in the output, for a reason that has nothing to do with its reliability: **it is the one part that is cheap to falsify.** An unbounded claim leaves you a research task. A claim with a locator leaves you a bounded one — you know which source to open and where to look. How long that then takes depends on the source: a page you can reach settles it in a minute, a paywalled or dense one does not. Asking for citations is worth doing not because they are evidence but because they are testable.

9.1.3 Three questions, in order

A citation has to survive three questions, and they get very unequal attention.

1. **Does the work exist?** Author, title, year, venue. This is the question everyone asks, and it is the one that fails least often.
2. **Does the locator exist?** The page, section, clause or figure. A real work with an invented page number is common and is invisible unless you look.
3. **Does the source support this claim, as stated, at this strength?** This is where most real failures are, and it is the one that costs actual reading.

Question 3 is the whole exercise. A summary can cite a real paper at a real page that genuinely discusses the topic, and still misstate what it found, and every surface signal, including the correctness of the first two answers, points the wrong way.

9.1.4 Six citation-failure modes used in this exercise

A closed vocabulary for this case, used with no synonyms, exactly as the code failure modes work in the lab. Closed for this exercise, not exhaustive of reality: a nonexistent work, a selective quotation, a compound claim only half supported, a retraction, and several sources that turn out to share one origin are all real failures outside these six.

Code	What it means	A one-line example
`fabricated-locator`	The work exists; the page, section or clause does not	A 240-page book cited at page 361
`stale-version`	The cited passage is reproduced accurately and has been superseded, and the claim is about the current state	The old edition's target, presented as the one in force
`misattributed`	The claimed finding is absent from the cited work, and is supported by another identified source	A real finding, credited to a work that does not contain it
`over-specified`	Same proposition, population and scope, at a precision the source does not give	The source says "approximately 20 percentage points"; the summary says "19.6"
`unsupported-inference`	The source supports a weaker proposition; the summary adds causation, generality, certainty or scope	The source reports an association and says it cannot separate the causes; the summary says one caused the other
`right-source-wrong-claim`	The cited work and passage are the right place to check, and directly report a different value, population, direction or result	A paper squarely on the topic, reporting 840 where the summary says 1,200

9.1.4.1 These are primary diagnoses, not exclusive descriptions The six sit at different layers. `fabricated-locator` is about the pointer, `stale-version` about which edition, `misattributed` about which work, and the last three about what the passage actually supports. A real claim can fail at several layers at once — a fabricated page in a superseded edition making an

unsupported inference is one claim with three faults, and no wording of the definitions will make the categories exclusive in every case.

So you are naming the **primary** fault, and there is an order for deciding it. Work down; the first one that applies is the diagnosis.

1. **Does the locator exist?** No → `fabricated-locator`.
2. **Is an explicitly superseded statement being presented as current?** Yes → `stale-version`.
3. **Is the claimed finding absent from the cited work, but supported by another identified source?** Yes → `misattributed`.
4. **Does the source support the same proposition, population, direction and scope, only less precisely?** Yes → `over-specified`.
5. **Does the source support a weaker proposition, with the summary adding causation, generality, certainty or scope?** Yes → `unsupported-inference`.
6. **Is the cited passage the right place to check, and does it report a different value, population, direction or result?** Yes → `right-source-wrong-claim`.

The order is not arbitrary. It runs from the cheapest check to the most expensive: whether a page exists, then which edition, then which work, and only then what the passage means. That is also the order in which a fault makes the later questions moot — there is no point asking what a passage supports when the passage does not exist.

Two distinctions are worth pinning down, because they are the ones people run together even with the order in front of them.

`over-specified` against `unsupported-inference`. Both are "the source says less than the summary", and the difference is what kind of less. Over- specification keeps the proposition, the population and the scope and sharpens a **quantity** — an approximate magnitude becomes an exact figure, "in the 1990s" becomes 1994. An unsupported inference changes the **logical force** — from association to cause, from some to all, from one setting to any setting.

`misattributed` against `right-source-wrong-claim`. Both are "wrong source", and the difference is **whether the cited work contains the finding at all**. In misattribution it does not: the finding is real and belongs to a different work, so correcting the citation repairs the claim. In right-source-wrong-claim the cited passage is exactly where you would look, and it reports something else, so no citation fixes it, because the claim itself is wrong.

Note what that distinction is *not*. It is not about whether the source is "relevant". Relevance is a matter of degree and a book can be squarely on a subject without containing a particular result. The operational question is narrower and answerable: open the cited work, and see whether the finding is in it.

9.1.4.2 These are not detected by a program

The lab's `check.py` names `b1-sign` by re-running a candidate against an oracle. Nothing equivalent exists here, and nothing will: whether a source supports a claim is a judgement about meaning, and a checker that claimed to settle it would be committing exactly the overreach this module trains you to catch.

So the discipline is not automation. It is the closed list. Having to choose between `unsupported-inference` and `over-specified` forces you to say what kind of wrong you found, and "the citation was bad" does not.

9.1.5 Building the reference

This is the step people skip, and it is the same step you did not skip in Unit 6.

The reference is **what the sources actually say**, in your words, written down before you compare. One row per locator the summary uses, plus a row for any passage that changes how another source may be used:

Source and locator	What the passage supports	Limitation, scope or version note

The third column is the one people leave out, and it is where two of the six failure modes are caught. A passage saying “this supersedes the targets in the 2019 review” supports no claim of its own and decides whether another source may still be cited. So does “we could not separate these two effects”, and so does “across the six authorities studied”.

The middle column is the reference. It is the counterpart of the value you computed by hand, and it has the same property: you produced it, so comparing against it means something.

Record what you could not retrieve. A source behind a paywall, a broken link, an edition you cannot get — those are findings, not gaps in your work. “Could not retrieve” is a legitimate row and it changes what you can claim at the end.

9.1.6 Comparing, and stopping at the first divergence

Take the claims in order. For each, ask question 3: does the passage support this, as stated, at this strength?

Stop at the first claim that fails, and name its code.

The reason is the reason from Unit 8, and it is worth repeating because it feels wrong: later claims may look fine, and their looking fine tells you very little once you know that the summary is capable of producing an unsupported claim in this register. The first divergence is a precise place to start, not a proof about everything after it.

What it is **not** is a licence to stop working. You stop the *diagnosis* there; you still check the rest before you rely on any of it.

9.1.7 What you can say at the end

The deliverable is a scoped statement, exactly as it is in Unit 17.

Not “the summary is accurate”, which nothing you did establishes. Something like:

> Of eight claims, I retrieved sources for six. Five are supported as stated. One (claim 4) cites a section that does not exist in that document (`fabricated-locator`); I found the same claim supported at section 2.1 of the same work, so the claim stands and the locator does not. Two claims I could not check because I could not retrieve the source.

Every part of that is defensible: it says what you checked, what you found, what you could not do, and what follows. Compare it with “I verified the references”, which says none of those things and cannot be interrogated.

9.1.8 What this unit is not saying

It is not saying generated citations are usually wrong. It makes no claim about how often, because this module has no measurement of that and neither should you. The claim is about what a citation establishes before you check it, which is nothing, and how cheaply you can change that.

It is not saying retrieval-backed systems remove the problem. They move it, as Unit 4 said: a fetched source is a real source, and the question of whether it says what the answer says it says is untouched.

9.1.9 Summary

- A citation from a generative system is generated text, and establishes nothing by itself, but it is the cheapest part of an output to falsify, which is why asking for one is worth doing.
- Three questions in order: does the work exist, does the locator exist, does the source support this claim as stated at this strength. Most attention goes to the first; most failures are at the third.
- Six named failure modes, a closed list, assigned by you rather than by a program, because there is no oracle for what a source supports.
- The method is Unit 8’s method: build the reference first, compare item by item, stop at the first divergence, name the fault, and end with a scoped statement rather than a verdict.

Then answer `cp09-diagnose-source` and `cp09-scope-the-claim-source` at the foot of this page.

9.2 Unit 9 activity — check eight claims, then two of your own

Time: about 30 minutes. Read `page.md` first, at least as far as the six codes.

This is the module's second worked case, and it has the same shape as the first: build the reference, compare item by item, stop at the first divergence, name the fault, and finish with a statement you could defend.

Part 1 uses a specimen supplied below, so you can do the whole comparison without leaving this page. Part 2 puts it on a topic of your own, at a scale you can finish honestly.

9.2.1 Part 1 — the specimen (15 minutes)

9.2.1.1 About these sources

> **[FICTIONAL TEACHING SOURCES]** The four sources below are invented for this exercise. Brackenford does not exist, none of these documents exists, and nothing in them should be quoted, cited or repeated anywhere else. They are here so you can carry out a full claim-by-claim comparison in fifteen minutes rather than an afternoon in a library, with every edition fixed and every passage available. Everything needed to adjudicate every claim is in the extracts.

9.2.1.2 The request

```
Write a short referenced summary of the Brackenford kerbside recycling scheme.
Eight claims, each with a citation to a specific location in a source.
```

9.2.1.3 Constructed specimen summary This response was **written for this exercise** to exhibit the failure modes. It is not an observed model output, and nothing here says how often any system produces any of them.

1. Brackenford's kerbside recycling scheme began in 2011
   (Brackenford Borough Council 2019, section 1.1).

2. Participation increased by 19.6 percentage points between 2012 and 2018
   (Hallam 2021, p. 94).

3. Hallam (2021, p. 96) concluded that fortnightly collection causes higher
   contamination of recyclate.

4. Kerbside glass collection was introduced in 2015
   (Brackenford Borough Council 2019, section 4.2).

5. The council's current target is 60 per cent household recycling by 2028
   (Brackenford Borough Council 2019, section 2.3).

6. Okonjo and Lind (2020, p. 112) report that, where authorities printed sorti
   guidance on bin lids, sorting errors fell in the following year.

7. Okonjo and Lind (2020, p. 112) surveyed 1,200 households across three
   northern towns.

8. The 2023 strategy commits to weekly separate food-waste collection from 202
   (Brackenford Borough Council 2023, section 5.2).

9.2.1.4 The sources

```
[FICTIONAL] SOURCE A -- Brackenford Borough Council (2019).
Kerbside Collection Review 2019. A 26-page document. Contents: section 1
Background, section 2 Targets, section 3 Materials, plus appendices. There is
no section 4.

  1.1  "Brackenford's kerbside recycling scheme began in 2011 with the
        collection of paper, card and mixed plastics."
  2.3  "The Council's target is 60 per cent household recycling by 2028."
  3.4  "Glass was added to the kerbside collection in 2015."

[FICTIONAL] SOURCE B -- Hallam, R. (2021). "Kerbside recycling
schemes." Journal of Municipal Waste Studies 14(2), 88-101.

  p. 94  "In Brackenford, participation increased by approximately 20
          percentage points between 2012 and 2018."
  p. 96  "Contamination was higher in the fortnightly-collection authorities.
          We are not able to separate the effect of collection frequency from
          that of housing type, which differs systematically between the two
          groups."
  p. 97  "Where authorities printed sorting guidance on the bin lid itself,
          sorting errors fell in the following year."

[FICTIONAL] SOURCE C -- Brackenford Borough Council (2023).
Waste Strategy 2023-2028.

  5.1  "This strategy supersedes the targets set in the 2019 Kerbside
        Collection Review."
  5.2  "The Council will introduce weekly separate food-waste collection from
        2026."
  5.4  "The Council commits to 65 per cent household recycling by 2028."

[FICTIONAL] SOURCE D -- Okonjo, T. and Lind, S. (2020). Household Sorting
Behaviour. Riverbank Press. 240 pages.

  p. 112  "We surveyed 840 households across three northern towns. The towns
           are not named, at the request of the participating authorities."

  (The book does not discuss labelling or bin design anywhere.)
```

9.2.1.5 What to do Build the reference first, then compare. Do not skip to the comparison — the order is the exercise, exactly as it was in Unit 6.

1. **Build the reference.** One row per locator the summary cites, plus a row for any passage that changes how another source may be used. Use the three columns from the reading: source and locator, what the passage supports, and the limitation, scope or version note.
2. **Compare, claim by claim, in order.** For each, ask: does the cited passage support this, as stated, at this strength?
3. **Record the first divergence.** Which claim number, and which code.
4. **Then keep going** and classify the rest. Stopping at the first divergence is where the *diagnosis* stops, not the work.

Use this table:

#	Supported?	Code, if not	Why, in one line

Work the decision order from the reading rather than picking the label that feels right. Several claims could be described in more than one way; the order is what makes one of them the primary diagnosis.

Optional hint about the distribution of answers

Each failing claim was written to carry one primary fault, and each of the six codes is used once. Open this only if you are stuck — knowing it lets you solve part of the exercise by elimination rather than by judging the claim against the source, which is the thing being practised.

9.2.2 Part 2 — two claims of your own (12 minutes)

Now on a real topic, at a scale you can actually finish.

1. Ask a system for a short referenced summary of something in your own subject. Six to eight claims with citations, exactly as above. Keep the prompt and the output untouched — Unit 7's discipline.
2. **Pick two claims only.** Not all of them. Two claims checked properly is worth more than eight skimmed, and it is the honest amount of work for twelve minutes.
3. Retrieve the sources for those two and build the reference: what does the passage at the cited locator actually say?
4. Classify each. If a claim is supported, say so — that is a result too.

Record what you could not retrieve. A paywall, a dead link, an edition you cannot get: those are findings, and they change what you can claim at the end.

9.2.3 Part 3 — the scoped statement (3 minutes)

Write two or three sentences about your own two claims, in the form the reading gives: what you checked, what you found, what you could not do, and what follows.

Then apply one test to what you wrote:

> Could a reader who does not trust me, and cannot see my screen, tell what evidence exists?

Keep all three parts. Unit 17 asks for an evidence pack, and this comparison table is what goes into it if you took the source route rather than the code route. They are the same deliverable.

9.2.4 Now take the checkpoints

`cp09-diagnose-source` gives you six fresh observations and asks which code each one is. **`cp09-scope-the-claim-source`** is about Part 3.

```
python3 selfcheck.py run cp09-diagnose-source
python3 selfcheck.py run cp09-scope-the-claim-source  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check. Nothing in either checkpoint requires you to have run any code.

9.2.5 Optional extensions

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

Ask for the quotation (about 6 minutes). Go back to one of your Part 2 claims and ask the system to quote the exact sentence its citation rests on. Then compare that quotation with the source. There are three outcomes — the sentence is there and says this, it is there and says something adjacent, or it is not there — and all three are more informative than the original claim was.

Check a claim that has no source at all (about 8 minutes). Take an unsourced claim from the same summary and try to find out whether it is true. Time it. The gap between that and the thirty seconds a locator costs you is the whole argument for asking for citations, and it lands harder when you have paid it once.

9.2.6 Activity Answers

9.3 Unit 9 answers and guidance

Read this after attempting the activity.

9.3.1 Part 1 — the completed comparison

The first divergence is claim 2.

#	Supported?	Code	Why
1	Yes	—	Source A, 1.1 states that the scheme began in 2011, with paper, card and mixed plastics. The claim is at the strength the source supports.
2	No	`over-specified`	Source B, p. 94 says "approximately 20 percentage points", for Brackenford, over exactly that period. Same proposition, same population, same direction, same scope — only the precision is invented. Nothing else about the claim differs, which is what makes this the clean case.
3	No	`unsupported-inference`	Source B, p. 96 reports higher contamination in fortnightly areas *and explicitly says it cannot separate frequency from housing type*. "Causes" states the conclusion the authors decline to state. The locator is correct and the passage is the right one; what changed is the logical force.
4	No	`fabricated-locator`	Source A has no section 4; it ends at section 3 plus appendices. The claim itself is true and is at 3.4 — so the claim stands and the locator does not.
5	No	`stale-version`	Source A, 2.3 says exactly what the claim says: 60 per cent by 2028. The passage exists and is reproduced accurately. It has been superseded — Source C, 5.1 says so, and 5.4 sets 65 per cent — and the claim is about the *current* target. Re-citing does not repair this one: the sentence is wrong about the world, and both the figure and the source have to change.

#	Supported?	Code	Why
6	No	`misattributed`	The finding is real and is at Source B, p. 97, in these words. Source D does not discuss labelling or bin design anywhere, so the finding is absent from the cited work. Correcting the citation to Hallam repairs the claim completely.
7	No	`right-source-wrong-claim`	Source D is exactly where you would check this, p. 112 exists, and the population description matches. It says 840 households. Not 1,200. No citation repairs this, because the claim is wrong.
8	Yes	—	Source C, 5.2 states weekly separate food-waste collection from 2026.

9.3.1.1 The three pairs, and why the order settles them **2 and 3 are both "the source says less".** Claim 2 keeps the proposition, the population, the period and the direction, and sharpens a quantity: approximately 20 becomes 19.6. Claim 3 keeps the quantity — there is none — and changes the logical force from association to cause. If you called claim 3 over-specified, ask what number changed. Nothing did.

6 and 7 are both "wrong source". The question is not which source is more relevant — Source D is a book about household sorting and is perfectly relevant to both. It is whether the cited work *contains the finding*. For claim 6 it does not, and the finding lives in Hallam, so fixing the citation fixes everything. For claim 7 it does, and reports a different number, so nothing about the citation can be fixed.

4 and 5 are both "the cited document does not support this as cited". In 4 the *place* does not exist. In 5 the place exists, says exactly what is claimed, and has been replaced. The decision order puts `fabricated-locator` first for a reason: when a locator does not exist there is nothing to read, so no question about meaning arises.

9.3.1.2 What if you labelled a claim differently Several of these can be *described* in more than one way, and describing one differently is not the same as getting it wrong. What the exercise asks for is the primary diagnosis under the stated order, and the test of your answer is whether it names a repair:

- `fabricated-locator` and `misattributed` say **fix the citation**; the claim survives.
- `over-specified`, `unsupported-inference` and `right-source-wrong-claim` say **fix the claim**; no citation helps.
- `stale-version` says **find the current source and re-check**; both may change.

If your label implies the wrong repair, that is the thing to look at again.

9.3.1.3 The scoped statement for Part 1 Something of this form:

> Of eight claims, six are not supported as cited. Claims 1 and 8 are supported. Claims 4 and 6 are true but wrongly located or wrongly credited, and both are repaired by correcting the citation. Claims 2, 3 and 7 misstate what the sources say and cannot be repaired by re-citing. Claim 5 reproduces a superseded target as current, so both the figure and the source have to change. The first divergence is claim 2. Nothing here establishes anything about the scheme beyond what Sources A to D say — and those are invented.

Note the last clause. The exercise established facts about the *summary*, not about Brackenford, which does not exist.

9.3.2 Part 2 — what a good result looks like

Two claims, properly checked, with the retrieval recorded. That is the whole standard.

Three patterns worth naming:

What happened	What it means
Both claims checked out	A real result, and not a boring one. Record it as "supported as stated" rather than "fine", and note the strength the source supports — that is the part that transfers.
You could not retrieve a source	A finding. "Could not retrieve" is a legitimate row, and it limits what you can claim rather than invalidating your work.
The citation was to a whole work with no locator	You have provenance, not a locator. Go back and ask for the sentence. An unbounded citation is the state Unit 5 warned about.

If you found nothing wrong in either claim, that is not a failed exercise. The module makes no claim about how often generated citations fail, and neither should you: your two claims are two claims.

9.3.3 Part 3 — the test

The sceptical-reader test is the same one Unit 17 applies to the code route, and it fails in the same ways. "I checked the references" does not say which ones, against what, or what happened. "I retrieved two of six sources and confirmed both claims at the cited pages; I could not access the other four" does.

9.3.4 The checkpoints

`cp09-diagnose-source` uses six observations distinct from the specimen, so working through Part 1 first does not give the answers away — it gives you the distinctions, which is the point.

Its three keyed diagnostics are the three confusions that survive: the two pairs above, and the pair of pointer failures. If you got one of those, the diagnostic names the distinction rather than the answer.

`cp09-scope-the-claim-source` tests Part 3. The tempting wrong answer is the one that sounds most thorough, and the correct one is the one that states its limits — including the sources it could not reach.

9.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp09-diagnose-source` — Naming the way a citation failed

Kind: match.

Each observation below describes a claim checked against its cited source. Match each to the failure mode it shows. Answer with six option numbers in the order of the observations.

Observations, in order:

1. The claim is correct, and the finding it states is well established. The work cited does not discuss the topic anywhere.
2. The source reports that two things occurred together and states that its design cannot separate them. The summary says one caused the other.
3. The source says the increase was 'roughly a fifth'. The summary says '19.6 per cent'.

4. The cited edition states the old threshold. The summary presents it as the current requirement, and a later edition has replaced it.
5. The book exists, is by the named author, and has 240 pages. The summary cites page 361.
6. The cited paper is squarely on the topic and the cited page exists. Read at that page, it reports a different value from the one attributed to it.

Options, numbered from zero:

0. misattributed
1. unsupported-inference
2. over-specified
3. stale-version
4. fabricated-locator
5. right-source-wrong-claim

Answer: `0, 1, 2, 3, 4, 5`

Hint: Work the decision order rather than picking the label that fits. Does the locator exist; is a superseded passage being presented as current; is the finding absent from the cited work; did a quantity get sharper; did the logical force change; does the right passage report something else.

Diagnostics, by the answer given:

- `0,2,1,3,4,5` — You have swapped the second and third. Both are 'the source says less than the summary', and the difference is what kind of less. Turning an association the authors refuse to interpret into a cause strengthens the claim, which is unsupported-inference. Turning 'roughly a fifth' into '19.6 per cent' sharpens a quantity, which is over-specification. Nothing about the second observation is a question of precision.
- `5,1,2,3,4,0` — You have swapped the first and last. Both are wrong-source failures, and the question is not which source is more relevant – relevance is a matter of degree and both works are on the subject. It is whether the cited work contains the finding. In the first it does not, and the finding belongs to another identified source, so correcting the citation repairs the claim. In the last the cited page is exactly where you would look and reports something else, so no citation repairs it – the claim itself is wrong.
- `0,1,2,4,3,5` — You have swapped the fourth and fifth. Both are pointer failures, and the difference is whether the pointer ever existed. Page 361 of a 240-page book never existed, which is a fabricated locator. The old edition's threshold did exist and has been replaced, which is a stale version – and the two need different repairs: correct the locator in one case, change the source in the other.

Explanation: The six codes are a closed list for this exercise and you are naming the primary fault under a stated order, not the only true description – a real claim can fail at several layers at once. The test of a label is the repair it names. A fabricated locator and a misattribution are repaired by fixing the citation, because the claim survives. An over-specification, an unsupported inference and a right-source-wrong-claim are not, because the claim itself is what the source does not support. A stale version needs both a new source and a re-checked figure. No program assigns these, because there is no oracle for what a source supports.

Checkpoint `cp09-scope-the-claim-source` — Scoping what the check established

Kind: choice.

You asked for a summary with six referenced claims. You retrieved three of the six sources; the other three were behind paywalls. Of the three you read, two support their claims as stated and one cites a section that does not exist, though you found the same claim supported elsewhere in that document. Which statement can you defend?

Options, numbered from zero:

0. I checked three of the six cited sources. Two claims are supported as stated. One cites a section that does not exist, though the claim is supported at a different section of the same document. Three sources I could not retrieve, so three claims are unchecked.
1. I verified the references and the summary is accurate apart from one citation error.
2. Half the citations could not be confirmed, so the summary should not be relied on.

3. Two of six claims are confirmed, which is a 33 per cent accuracy rate for this summary.

Answer: 0

Hint: Three sources you read and three you could not. Ask what each option claims about the three you did not open, and whether the evidence you have supports saying anything about them at all.

Diagnostics, by the answer given:

- 1 — Two overclaims in one sentence. 'Verified the references' covers three you never opened, and 'the summary is accurate' is a claim about all six from evidence about three. The citation error is also understated – calling it an error without saying the claim itself survived loses the repair.
- 2 — This understates in the opposite direction and reaches a conclusion the evidence does not license. You could not retrieve three sources; that is not the same as their failing, and it says nothing about the two claims you did confirm. 'Unchecked' and 'unsupported' are different findings, and collapsing them is as much a scope error as the confident version.
- 3 — A rate implies a measurement, and this is not one. The denominator includes three claims you did not check, so the number is not an accuracy rate for anything. Numbers carry an air of measurement that prose does not, which is what makes an invented one worse than a vague sentence.

Explanation: A scoped statement says what you checked, what you found, what you could not do, and what follows – and it stops there. The unretrievable sources are the part most people drop, in either direction: either quietly counted as fine, or treated as failures. They are neither. The locator failure is worth stating precisely too, because a claim that survives with a corrected citation needs a different response from one the source does not support at all.

10 Unit 10 — Re-prompting, specification, and overreliance

10.1 Reading

Media for this unit:

- **Animation**: *Specification Filter* — this unit has an interactive version on the course website (`spec-filter.html`); the still below is one moment from it.

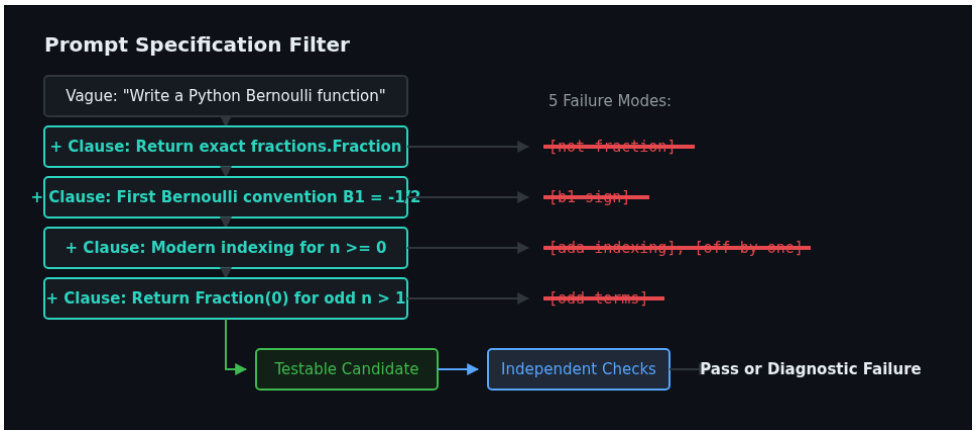

Figure 12: Specification Filter. Precise requirement clauses prohibit specific failure modes, but the resulting candidate still requires independent checks.

Reading time: about 10 minutes. Activity: about 24 minutes.

10.1.1 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 8 was telling you which requirement you had failed to write down.

10.1.2 Where you started

The prompt Unit 7 began with was:

> Write a Python function that returns Bernoulli numbers.

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.

10.1.3 The failure modes were the missing requirements

Put the Unit 8 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 `cp10-spec-clauses`.

10.1.4 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.

C6 asks for more than the checker can see. The checker observes what comes back: the type of every returned value and whether it equals the oracle exactly. "Do not use floating-point arithmetic anywhere

in the computation" is about the *inside* of the routine, and a program can compute in floats and convert before returning. What the checker can establish is the observable half — exact `Fraction` values that agree over the tested range. The rest is a source-review requirement, which is a different kind of check with different limits: reading establishes what the code appears to do, not what it does.

That gap is worth noticing rather than papering over. Several clauses have it. A dynamic check on outputs cannot establish which algorithm was used, how the routine scales, that no lookup table is hidden inside it, or that it behaves outside the range that was tested.

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 16 is about exactly that.

10.1.5 Most of these clauses are bugs that already happened

Read the precise prompt once more and notice what most of it 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.

10.1.6 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 6. 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 `cp06-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.

10.1.6.1 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.

10.1.7 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.

10.1.8 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 18 teaches from; neither this page, the activity, nor `cp10-spec-clauses` depends on it.

10.1.9 Checkpoint

Finish this unit at `cp10-spec-clauses`, by either route:

```
python3 selfcheck.py run cp10-spec-clauses  # or in the browser
```

10.2 Unit 10 activity — Annotate the specification, then re-run it

Time: about 24 minutes. Parts 1 to 4 take about 10 minutes; the no-oracle transfer in Part 5 takes about another 14. 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 17 evidence pack.

10.2.1 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`		

failure mode	clause that removes it	why that clause makes it impossible
`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.

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

Put your Unit 7 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.

10.2.3 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 7. Then check the result. **A better prompt is not evidence of a better answer**; the new output gets verified exactly like the old one.

10.2.3.1 Route A — Python (4 minutes) Read the new code first, as in Unit 8: 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 7)	precise prompt (Unit 10)
first divergence at n		
diagnosis code(s)		
result		

10.2.3.2 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 8 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`.

10.2.3.3 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.

10.2.4 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 17 correctness defence already written.

10.2.5 Part 5 — Transfer when there is no oracle (14 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				

10.2.5.1 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.

Then do two more things, which is where this case meets the rest of the module.

Name the failure mode you would expect. Unit 9 gives six, and this sentence could fail as any of several: the paper may not exist, the figure may be sharper than the source gives, or the causal wording may be stronger than an association the authors reported. Naming which one you are testing for tells you what to open first.

Re-prompt with the source requirements stated. This is the unit's own method applied to a claim rather than to code. Write a specification for the summary the way you wrote one for the routine: every claim carries a locator, no claim is stated more strongly than its source states it, and where the source is silent the summary says so rather than filling the gap. Ask again with that specification, and compare the two outputs the way the Bernoulli routes compare theirs.

10.2.5.2 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.

10.2.5.3 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.

10.2.6 Finish the unit at the checkpoint

`cp10-spec-clauses` is available by both routes. It is formative and unmarked.

Python route — from `course/lab`:

```
python3 selfcheck.py run cp10-spec-clauses
```

or, equivalently:

```
python3 selfcheck.py run --unit 10
```

Browser route — answer it at the foot of this unit's page, 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
```

10.2.7 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 7 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
- `cp10-spec-clauses` passed

All of these feed directly into the evidence pack you assemble in Unit 17.

10.2.8 Activity Answers

10.3 Unit 10 answers and guidance

Answer key for the Unit 10 activity. The self-check checkpoint marks itself and gives its own diagnostic feedback by either route, so its answer is not reproduced here.

10.3.1 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 a request that never said which indexing it meant.
`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.
`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.

failure mode	clause	why it makes the failure impossible
`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 16, 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.

10.3.2 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.

10.3.3 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 comparing only the small values by hand may not. If you found that, you 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.

10.3.4 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 6 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.

10.3.5 Part 5 — No-oracle transfer

A strong response keeps four different questions apart: what the output claims, what independent evidence is feasible, what remains a judgement, and when you 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 your own argument.	The source cannot be located, the statistic or causal wording is absent, or you 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 you would be guessing about permission. Ask the named course contact before acting.

Case	Claim	Suitable check	Remaining judgement	Stop or escalate when
Recommendation	The famous unpaid placement produces better career outcomes for you.	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 you 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.

10.3.6 The honest position

Keep both halves. Leaving this unit thinking the module is against using these systems is the wrong lesson; so is leaving it thinking a good prompt is a substitute for a check. Both of these are true at once: the specification work is a genuine gain, and 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.

10.3.7 Checkpoint

`cp10-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 cp10-spec-clauses  # or in the browser
```

10.3.8 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp10-spec-clauses` — Which clause kills which bug

Kind: match.

Each clause below was added to the prompt after a specific failure. Match the clause to the failure it prevents.

Observations, in order:

1. ‘Use the first Bernoulli convention, B1 = -1/2.’
2. ‘Return fractions.Fraction, never float.’
3. ‘bernoulli(n) must be defined for every n >= 0, returning 0 for odd n > 1.’

Options, numbered from zero:

0. A routine correct under the other convention.
1. Rounding error instead of exact values – slight at first, unmistakable by the large indices.
2. A routine that numbers only the non-zero terms.

Answer: `0, 1, 2`

Hint: For each clause, imagine an output that satisfies every other clause and fails this one. The failure you picture is the one the clause was added to prevent.

Diagnostics, by the answer given:

- `0,2,1` — The last two are swapped. Ask what each clause makes *testable*: naming the return type rules out inexact arithmetic, while pinning down the domain and the zero terms rules out an alternative indexing scheme. You may have expected the indexing clause (C3); note that pinning the domain and the zero terms also rules out Note G numbering.

Explanation: Every clause in the precise prompt is a bug that already happened. That is what separates specification from ‘prompt engineering’: you are not finding magic words, you are writing down a requirement each time reality teaches you one.

11 Unit 11 — Agents and automated actions

11.1 Reading

Media for this unit:

- **Animation**: *Where a check can go in an agent run* — this unit has an interactive version on the course website (`trace-marks.html`); the still below is one moment from it.

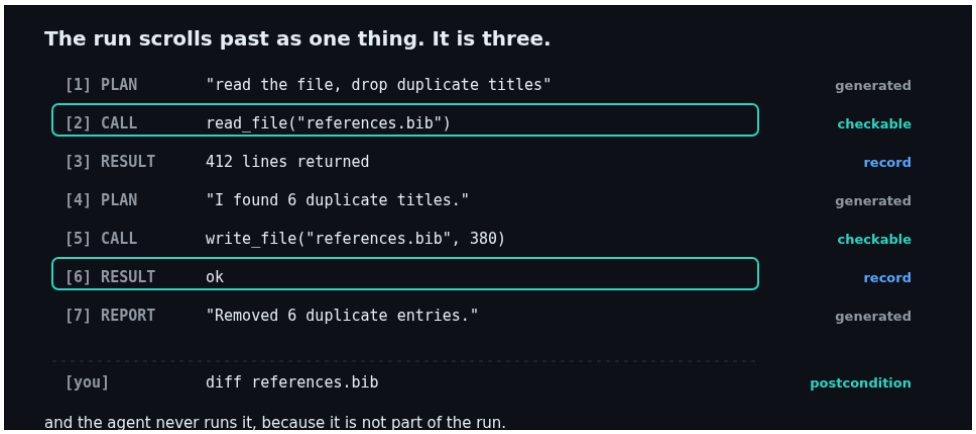

Figure 13: Where a check can go in an agent run. A trace prints line by line, then sorts into generated, record and checkable, marking the three places a check goes.

Reading time: about 12 minutes. Activity: about 14 minutes.

11.1.1 What changes

Everything so far has been about an answer: something you read, check, and then either use or discard. Discarding is cheap, and it is the reason the whole method works. You can take as long as you like over a claim, because until you act on it, nothing has happened.

An **agent** is a system that acts. It calls tools in a loop — plan, act, look at the result, act again, and some of those actions change something outside the conversation. A file is written. A message is sent. A row is deleted. A booking is made.

The method does not change. What changes is one thing, and everything in this unit follows from it:

> An action can change the world before you have looked at it.

Some actions are cheap to take back — a draft, a preview, a file in your own bin. Some are not, and a few cannot be taken back at all. The next section separates those, because the difference decides how much checking has to happen *before* rather than after.

Checking after the fact is still worth doing, because you need to know what happened. But it is no longer *verification* in the sense the rest of this module uses: it cannot prevent the error, only describe it, and what it leaves you is a remedy rather than a correction.

11.1.2 The loop, and where you are in it

A single-shot answer has one place a check can go: after the output, before you use it. An agent run has many, and by default you are in none of them.

A typical run:

1. The system produces a plan. This is generated text.
2. It selects a tool and arguments. This selection is generated.
3. The tool executes, under its own semantics and against the state of the world at that moment.
4. A result is recorded.
5. The system reads the result and decides what to do next. Generated.

6. Repeat, possibly many times.
7. It reports what it did. This report is generated text about the run.

Step 3 is not the safe step. It is tempting to say that the generated parts are the risky ones and the tool merely does as it is told, but that is only true of a tool with no outside world in it — a calculator, say. Most tools have one. A search returns what the index holds today. A file operation depends on what is on disk, on permissions, and on what else is writing at the same time. A network call can time out, half-finish, or succeed after the system has given up waiting. A tool can carry its own bug. And a tool can do exactly what it was told while doing the wrong thing, because what it was told came from step 2.

So there are three distinct places a run goes wrong, and they need different checks:

Where	What goes wrong	What checks it
Selection (step 2)	The right tool with the wrong arguments; the wrong tool	Read the arguments before the call, or approve them
Execution (step 3)	Timeout, partial write, stale data, permission failure, a bug	Look at the result, not just the exit status
Postcondition (step 4)	The call reported success and the world is not as intended	Read back the state: a diff, a re-read, a receipt

`ok` is a claim about a call, not about the world. A write that returns success may have written to the wrong path; a delete that returns success may have matched nothing. The check for step 4 is to look at what is actually there afterwards.

This is Unit 4's tool-calling split, run repeatedly, with the consequences compounding: an error at step 2 produces a result at step 3 that the system then reasons from at step 5.

11.1.3 Reversibility is the axis

Unit 3 sorted outputs by what kind of check they need. For actions, that is the wrong first question. The first question is:

> If this is wrong, can it be undone, by whom, and at what cost?

Four rough bands, and they are not about how likely an error is:

Band	Example	What it changes
Free to undo	A draft file written to your own machine	Check afterwards; the loop is fine unattended
Undoable with effort	A commit, a scheduled job, a change to a shared document	Check afterwards, but budget for the undo
Undoable only by someone else	A message sent, a form submitted, a ticket raised	Confirm before, because the remedy is somebody's time
Not undoable	A payment, a deletion without backup, anything a third party has now seen	Confirm before, every time, without exception

"Undo" is doing a lot of work in that table, and it is worth separating four things it can mean, because they cost different amounts:

- **Undo** — the action is reversed and no trace of it remains that matters. A file restored from your own bin.
- **Compensating action** — the original stands and a second action offsets it. A refund does not unsend a payment; a correction does not unsend a message. This is the usual remedy in the third band, and it is why that band is not the same as the fourth.
- **Recovery** — the state is rebuilt from something kept for the purpose: a backup, a version history, a transaction log. Only available if someone arranged it beforehand.
- **Containment** — the action cannot be undone or offset, and what is left is limiting what follows: revoking a key, notifying the people affected.

A message sent to the wrong person is not undoable and is usually correctable. A disclosure of someone else's personal data is neither, and containment is all there is. Ask which of the four you would actually be reaching for.

Note what is missing from that table: any assessment of how reliable the system is. That is deliberate. Reliability changes how often you will need the remedy; it does not change what the remedy costs. Repeat a bottom-row action often enough with a system that is usually right, and the question stops being whether an unrecoverable error is likely on this run and becomes whether you are willing to carry one at all. How soon depends on the error rate and on whether the failures are independent, neither of which you know, which is the point. You are choosing an exposure, not calculating a date.

11.1.4 What a trace does and does not tell you

Agent products commonly show you a trace: the plan, the tool calls, the results, the summary. Read it — it is far more than a single-shot answer gives you.

Then keep two distinctions.

The tool calls and their results are records — if what you are reading is the system's own log. A call was made with these arguments and returned this. That is an artefact of what happened, and it is the most checkable thing in the run.

That "if" matters, because three different things can look identical on screen:

What you are looking at	What it is worth
The execution log the system wrote as it ran	A record. This is the evidence.
A chat window's rendering of the calls	Usually faithful, but it is a display: it may condense, reorder or omit
The system's account of what it did	Generated text, like any other

The third is the one to watch, because it is written in the same voice as the first. If you cannot tell which you have, that is worth finding out before you rely on it: a product's documentation says whether it shows you the log or a summary of it.

The plan and the summary are generated text about the run. They can be accurate. They are not a transcript, and a summary can describe a run that did not happen that way, not by deception, which would require intent, but because the summary is produced by the same process as everything else and nothing checks it against the log.

So: **check the tool calls, not the narration.** If a report says "I checked the existing entries before adding the new one", the question is whether a call that would have done that appears in the trace.

And note what is often missing. Intermediate steps may be summarised or hidden; a long run may show you a condensed version. The absence of a step from a displayed trace is not evidence that it did not happen, and its presence in a summary is not evidence that it did.

11.1.5 A worked trace, with the checkable points marked

The task: *"Find the duplicate entries in my references file and remove them."*

```
[1] PLAN    "I will read the file, identify duplicates by title,
             and delete the later occurrence of each."   <- generated
[2] CALL    read_file("references.bib")                  <- checkable
[3] RESULT  412 lines returned                            <- record
[4] PLAN    "I found 6 duplicate titles."                 <- generated
[5] CALL    write_file("references.bib", 380 lines)      <- checkable
[6] RESULT  ok                                            <- record
[7] REPORT  "Removed 6 duplicate entries."                <- generated
```

Six things are worth checking, and they are not evenly important.

- **[2] the file.** Right file, right path. Cheap, and the most consequential early check.

- **[4] the six.** “I found 6 duplicates” is a claim with no evidence attached. Which six? A duplicate identified by title alone will merge two genuinely different works that share a title.
- **[5] the write.** This is the step that cannot be undone, and notice that it is a *whole-file overwrite*: 412 lines in, 380 out. Nothing in the trace shows which 32 lines went.
- **[7] the report.** Two claims. “Removed 6 duplicates” and “no other changes”. The second is the one nobody reads and the one most likely to be wrong, because a whole-file rewrite can change formatting, ordering and encoding without anybody intending it.

The right intervention is not more scrutiny at step 7. It is a copy of the file before step 5, which converts the bottom row of the table into the top row, and is the single most useful habit in this unit.

11.1.6 Working with agents, practically

Move actions up the table before you start. Work on a copy. Use a branch. Send to yourself first. Almost every irreversible action has a reversible rehearsal, and arranging one is cheaper than checking carefully.

Bound the run. A loop with a stopping condition you set is a different object from one that runs until it decides it is finished. Say how many steps, which tools, and what it must not touch.

Ask for the plan before the actions. Approving a plan is not verification, and it will not catch an error at step 4. But it catches the wrong-file class of error at the point where it is still free.

Give it the least it needs. An agent can only do what its credentials and its tool list allow. A key that can read but not write, a tool list without the destructive operation, a path or domain it is confined to — each of these turns a class of bottom-row action into one that cannot happen at all. This is worth more than any amount of checking, because it removes the possibility rather than detecting the result.

Ask for a dry run. Many operations have a mode that reports what they would do without doing it. Where one exists, the run becomes free to undo.

Check the calls, not the story. Every time.

None of which is an argument against agents. They do real work, and much of it sits in the top two rows of the table, where letting the loop run unattended is exactly right. Nor is a trace worthless — it is the most inspectable thing any of these systems gives you, which is why it is worth learning to read. The claim is narrower: the record part is evidence, the narration part is not, and the two arrive looking identical.

11.1.7 Summary

- An agent acts rather than answers, and what changes is that the world may change before you have looked at anything. Checking afterwards describes an error; it does not prevent one.
- No step is the safe one. The plan, the choice of arguments, the decision to continue and the final report are generated; the execution depends on state, timing and permissions, and can fail or half-succeed on its own account.
- A result is a claim about a call. Whether the world is as intended is a separate question, answered by reading the state back.
- Sort actions by what recovery costs, not by how reliable the system seems. Reliability changes how often you need the remedy, not what it costs.
- In a trace, calls and results are records *if the trace is the system’s own log*. A rendering in a chat window may summarise, reorder or omit steps, and a narrated account of a run is generated text. Know which one you are reading.
- Move an action up the reversibility table before you start — a copy, a branch, a dry run — rather than checking an irreversible one harder.

Then answer `cp11-reversibility` and `cp11-trace-evidence` at the foot of this page.

11.2 Unit 11 activity — read a trace, then bound a run

Time: about 14 minutes. Read `page.md` first, at least as far as the worked trace.

No agent is required. Part 1 uses a trace supplied below; Part 2 asks you to design the guard rails for a task you might actually give one.

11.2.1 Part 1 — audit a trace (7 minutes)

Below is a complete run. The task given was:

> Tidy my project folder — move anything that is not source code into an `archive` subfolder.

```
[1]  PLAN     "I will list the folder, classify each file, create archive/,
               and move the non-source files into it."
[2]  CALL     list_dir(".")
[3]  RESULT   ["main.py", "utils.py", "notes.txt", "data.csv",
               "old_main.py.bak", ".env", "README.md", "figure1.png"]
[4]  PLAN     "Source files are main.py and utils.py. Everything else is
               not source and should be archived."
[5]  CALL     make_dir("archive")
[6]  RESULT   ok
[7]  CALL     move("notes.txt", "archive/notes.txt")
[8]  RESULT   ok
[9]  CALL     move("data.csv", "archive/data.csv")
[10] RESULT   ok
[11] CALL     move("old_main.py.bak", "archive/old_main.py.bak")
[12] RESULT   ok
[13] CALL     move(".env", "archive/.env")
[14] RESULT   ok
[15] CALL     move("README.md", "archive/README.md")
[16] RESULT   ok
[17] CALL     move("figure1.png", "archive/figure1.png")
[18] RESULT   ok
[19] REPORT   "Tidied the folder. Moved 6 non-source files to archive/.
               Your code is untouched."
```

Answer four questions in writing.

1. **Which lines are records and which are generated text?** Give line numbers for each group.
2. **Which single step should have been confirmed before it ran, and why?** There is a defensible answer here that is not simply "all of them".
3. **The report makes two claims.** Write them out separately and say what evidence in the trace bears on each.
4. **What did the run do that the task did not ask for?** There is at least one thing, and it is the kind of thing that is invisible until somebody looks.

Do not read the answers until you have written something for all four.

11.2.2 Part 2 — bound a run of your own (7 minutes)

Think of a task you would plausibly hand to an agent — tidying files, filling in a spreadsheet, replying to routine messages, updating records.

Write five lines.

The task, in one sentence
The most irreversible action it would need to take
Which band that action is in
The reversible rehearsal: how could that action be made undoable *before* the run?
The stopping condition: how many steps, which tools, what it must not touch

---

The fourth row is the one that matters. Almost every irreversible action has a rehearsal — a copy, a branch, a draft folder, sending to yourself first — and arranging one is cheaper than checking a dangerous run carefully.

If you cannot find a rehearsal, say so. That is a finding, and it means the task is one to do yourself or to supervise step by step.

11.2.3 Now take the checkpoints

`cp11-reversibility` is about the axis this unit sorts on. **`cp11-trace-evidence`** is about Part 1's first and third questions.

```
python3 selfcheck.py run cp11-reversibility
python3 selfcheck.py run cp11-trace-evidence  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check.

11.2.4 Optional extensions

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

Rewrite the task so the run cannot go wrong the same way (about 5 minutes). Take the folder-tidying task from Part 1 and rewrite the instruction so that the specific failure you identified in question 4 could not happen. This is Unit 10's method — specification — applied to an action rather than to an output, and the comparison between the two versions is the point.

Find the rehearsal you already use (about 3 minutes). You almost certainly already have one habit of this kind — working on a copy, drafting before sending, a staging area. Write down which one, and which band it moves an action from and to. The habit generalises better once it has a name.

11.2.5 Activity Answers

11.3 Unit 11 answers and guidance

Read this after attempting the activity.

11.3.1 Part 1 — the four questions

11.3.1.1 1. Records and generated text **Records:** lines 2, 3, 5–18 — every CALL and every RESULT. These are artefacts of what happened.

Generated: lines 1, 4 and 19 — the plan, the mid-run classification, and the report.

Line 4 is the one people put in the wrong group, because it appears between two tool calls and reads like a finding. It is not: nothing was consulted to produce "source files are main.py and utils.py". It is the classification the whole run then acts on, and it is generated.

11.3.1.2 2. The step to confirm **Line 13, the move of `.env`.**

The defensible reasoning is not that moving a file is dangerous — the other moves are all trivially reversible with another `move`. It is that `.env` conventionally holds credentials, so this move changes where a secret sits, and it will break anything that reads it. It is also the file whose relocation is least likely to be noticed, because nothing lists it by default.

Answering "line 5, creating the directory" is not right — that is free to undo. Answering "all of them" is not wrong so much as not a judgement: an agent that must confirm every step is not doing a job you have delegated.

If you named line 11 (`old_main.py.bak`) instead, that is a reasonable second answer for a different reason: it is a backup, and archiving a backup buries the thing someone would reach for in a hurry.

11.3.1.3 3. The two claims in the report

Claim A: "Moved 6 non-source files to archive/." Supported. Lines 7–18 show six moves, all returning ok. This claim is backed by records.

Claim B: "Your code is untouched." Not supported, and the trace shows why. `old_main.py.bak` is a backup *of code*, and `.env` is configuration the code needs to run. Nothing was deleted, so the claim is not exactly false — but "untouched" is doing work it has not earned, and it is the sentence a reader will rely on.

The general shape: the quantitative claim is checkable against the calls, and the reassuring claim is the generated one. Reports tend to end with a reassuring claim, and it tends to be the one nobody checks.

11.3.1.4 4. What it did that was not asked

Two things, and either counts.

It archived `.env`. The task said "anything that is not source code". A literal reading includes `.env`; a reasonable reading does not, because configuration is part of how the project runs. This is the gap-in-the-instruction failure from Unit 2, appearing in a run that carries it out rather than in a procedure you read.

It archived `README.md`. Documentation is not source code by a strict reading, and moving it out of the project root breaks the convention every tool and person relies on.

Both are cases of the system doing exactly what was said. The remedy is not a better system; it is Unit 10's remedy — say what "not source code" excludes — or this unit's, which is to make the whole thing reversible first.

11.3.2 Part 2 — the rehearsal

The rows are all in service of the fourth. A good answer names a *specific* mechanism: "run it on a copy of the folder", "have it write to `drafts/` and move things myself", "let it prepare the messages and send them from my own client".

Two weak patterns:

- **"I would check the output carefully."** Attention is not a rehearsal, and by the time you are checking, the action has happened. This is the same failure as Unit 0's "I will read it through carefully".
- **"I would use a reliable tool."** Reliability changes how often you need the remedy, not what it costs. The table in the reading deliberately has no column for it.

If you found a task with no available rehearsal, that is the most useful answer in Part 2. It tells you the task is one to supervise step by step or to do yourself — which is a decision, made in advance, rather than a discovery made afterwards.

11.3.3 The checkpoints

`cp11-reversibility` tests the sorting axis. The tempting wrong answers all substitute a judgement about the system for a judgement about the action, which is the error the reading's table is shaped to prevent.

`cp11-trace-evidence` tests the record-versus-narration distinction. The trap is the mid-run plan line, which sits among the tool calls and reads like a result.

11.3.4 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp11-reversibility` — Sorting an action by what it costs to be wrong

Kind: choice.

You are deciding whether to let an agent run unattended on a task whose final step sends an email to an external client. The product is well regarded and has been accurate for you so far. What is the reasoning that fits this unit?

Options, numbered from zero:

0. Confirm before the send, because a sent message can only be remedied by someone else's time, and how reliable the system has been changes how often you need that remedy rather than what it costs.
1. Run it unattended, because the system has been accurate on this kind of task and the risk is therefore low.
2. Run it unattended but read the report carefully afterwards, so any mistake is caught quickly.

3. Confirm before every step, because any step in an agent run can be wrong.

Answer: `0`

Hint: Separate two questions: how often the system is wrong, and what it costs when it is. Ask which of those a track record speaks to, and which one decides whether this should run unattended.

Diagnostics, by the answer given:

- `1` — This substitutes a judgement about the system for a judgement about the action. A track record changes the frequency of errors, not the cost of one, and an unrecoverable action taken often enough by a mostly-accurate system is an unrecoverable error eventually. The reading's table has no column for reliability on purpose.
- `2` — Reading the report afterwards tells you what happened; it cannot make the message unsent. Once an action has left your control, checking has stopped being verification and become description. It is still worth doing – but not instead of the confirmation.
- `3` — True and not useful. An agent that must confirm every step is not doing a delegated job, and treating every action as equally serious means the genuinely irreversible one gets the same attention as creating a folder. The point of sorting by reversibility is to spend your attention where the remedy is expensive.

Explanation: Actions sort by what it costs to undo them, not by how likely the system is to be right. A sent message sits in the band where the remedy is somebody else's time, so it earns a confirmation before it happens. The stronger move, where it is available, is to change the band rather than to check harder – have the agent draft the message and send it yourself, which turns an action you cannot undo into an output you can discard.

Checkpoint `cp11-trace-evidence` — What in a trace is evidence

Kind: multi.

An agent run displays a plan, a numbered sequence of tool calls with their results, a mid-run line saying which files it has classified as source code, and a closing report. Which of these are records of what happened, rather than generated text about it? Select all that apply.

Options, numbered from zero:

0. The tool calls, with their arguments
1. The results the tools returned
2. The mid-run line stating which files were classified as source
3. The closing report summarising what was done

Answer: `0, 1`

Hint: Ask of each line whether something happened to produce it, or whether it was written about what happened. One of them sits between two tool calls and reads like a result.

Diagnostics, by the answer given:

- `0,1,2` — The mid-run classification is the one that catches people out, because it sits between two tool calls and reads like a result. Nothing was consulted to produce it – no tool was called to decide which files are source code. It is generated text, and it is the text the rest of the run then acts on, which makes it the most consequential generated line in the trace.
- `0,1,2,3` — Only the calls and their results are records. Both the classification line and the closing report are generated text about the run. The report is the more familiar trap and the classification line the more dangerous one, because an error there propagates into every action that follows.
- `0` — The results are records too. A tool returned what it returned, and that is an artefact of what happened rather than a description of it. Discarding the results leaves you with what was attempted and not with what occurred.

Explanation: A trace mixes two kinds of line that look identical. Calls and results are artefacts: a call was made with these arguments and this came back. Plans, mid-run reasoning and reports are generated text about the run, produced by the same process as any other output, with nothing checking them against the log. So a report claiming 'I checked the existing entries first' is answered by looking for a call that would have done that – not by reading the sentence more carefully.

12 Unit 12 — High-stakes use and escalation

12.1 Reading

Media for this unit:

- **Animation**: *Two axes, not one scale* — this unit has an interactive version on the course website (`two-axes.html`); the still below is one moment from it.

Figure 14: Two axes, not one scale. One task moves up the amount-of-check axis while its kind of check stays fixed.

Reading time: about 9 minutes. Activity: about 14 minutes.

12.1.1 A second axis, not a fourth scheme

You now have three ways of sorting a task, and they are all the same question asked at different depths: where does the reference live (Unit 1), what kind of check does that imply (Unit 3), and what does the system you chose add to the checking (Unit 4).

All three answer **what kind of check**. None answers **how much**.

That was deliberate. Unit 3 told you to sort by how an error would reach you and *not* by how important the task feels, because sorting by importance is the most common way people get the kind of check wrong. But importance has to come back somewhere, or you will spend the same effort confirming a spelling as confirming a dosage.

This unit is where it comes back, as a separate axis:

> **Kind** of check comes from where the reference lives. **Amount** of check comes from what being wrong would cost.

Two axes, asked in that order. Getting them the wrong way round is the error Unit 3 was guarding against, and keeping them separate is what lets you use consequence without it corrupting the diagnosis.

12.1.2 What “how much” is made of

Four things, and they interact rather than add up.

Consequence. If this is wrong and nobody notices, what happens, and to whom? The last three words matter. A great deal of low-stakes-to-you work is high-stakes to somebody else — a summary of someone’s medical history, a reference for a colleague, an accessibility statement.

Reversibility. Unit 11’s axis, generalised beyond agents. Can the error be found and fixed later, by you, cheaply? A draft you will reread is recoverable. A number that goes into a report that goes to a client is not, in practice, even though nothing was deleted.

Your own uncertainty. Not the system's — yours. Could you recognise a wrong answer here? This is the one people leave out, and it is the one that turns a manageable task into an unmanageable one, because outside your competence every surface cue you rely on is available to a wrong answer too.

Whether it is yours to decide. Competence and authority are different, and you can have one without the other. You may understand perfectly well what a contract clause means and still not be the person who may accept it; a clinician may be competent to read a scan they are not the responsible clinician for. When the decision belongs to someone else, escalation is not a confession that you could not work it out. It is the correct route.

They interact rather than add up, because a bad enough answer on any one of them can carry the whole thing. An irreversible action in an area you cannot assess is not made safe by being about something unimportant. There are no scales here and nothing is being multiplied — this is a way of noticing which of the four is about to decide the matter.

12.1.3 The decision table

For a claim you are about to rely on:

Consequence if wrong	Recoverable?	Can you assess it?	What that means
Small	Yes	Yes	Spot-check. The class of the task sets the check; do it once.
Small	Yes	No	Use it as a draft, do not present it as established, and keep the way back open. Say which part you could not check.
Large	Yes	Yes	Full check of the kind the class implies, and record what you did.
Large	No	Yes	Full check *before* the irreversible step, plus a second pair of eyes if one is available.
Large	—	No	**Escalate.** Do not proceed on your own judgement.
Any	—	Not yours to decide	**Escalate**, whatever you personally could work out.

The bottom two rows are the point of the unit. The first is the one people convert into the row above by deciding they can probably assess it after all. The second catches the case where you can assess it perfectly well and the decision still is not yours.

Row two is worth dwelling on, because it is the row that keeps this table consistent with Unit 0. Not everything gets checked; that was true there and it is true here. What changes is that you say so — a result you could not assess, used as a draft and labelled as one, is a different act from the same result presented as though it had been checked.

12.1.4 Escalation is a skill, not an admission

Escalating means handing a decision to someone qualified to make it. It is a normal professional move, and doing it early is cheaper than doing it late.

What to hand over. Not "the AI said this, is it right?" That wastes the other person's time and buries what you actually know. Hand over four things:

1. The claim you need settled, stated precisely.
2. What you have already checked, and what it showed.
3. What you could not check, and why.
4. What decision is waiting on it.

That is the evidence pack from Unit 17, used for its other purpose. A pack is the right thing to bring to somebody else, precisely because it separates what is established from what is not.

When. Four triggers, any one of which is sufficient:

- Acting on the claim would decide something for an identifiable person, or commit an organisation. Regulated areas — health, law, finance, safety, immigration, employment — are where this happens most often, which is why they are worth naming. The trigger is the decision, though, not the topic: looking up what a published clinical term means is not the same act as advising somebody on their treatment, and a rule that escalated both would be ignored within a week.
- The decision is not yours to make, whatever you could work out.
- You cannot construct any independent check.
- The checks you can construct disagree with each other.
- You notice you are looking for a reason to accept the answer. This one is about you rather than the material, and it is the most reliable of the four.

12.1.5 One scenario per kind of subject

Different subjects meet this at different points. Find yours; the shape is the same.

Health and care. A generated summary of a condition or an interaction, used to decide something about a real person. Consequence severe, reversibility often nil, and the assessment requires training. Escalate.

And while you are there, keep two checks apart, because they answer different questions and only one of them is available to you:

- **Does the source say this?** Open the official guidance and read it. This is a real check and it is worth doing. It is not weakened by the guidance possibly being in the system's training data: you are comparing the claim against the document, and the document is the authority on its own contents. Unit 9 is this check.
- **Is this right for this person?** The guidance being quoted correctly does not tell you it applies to the case in front of you, that it is current, or that it outweighs something else in that person's circumstances. No amount of reading settles that, and it is what the qualified person is for.

The first check is yours. The second is the escalation. Confusing them in either direction is a mistake — skipping the reading because "it might be in the training data anyway", or treating a correct quotation as a clinical judgement.

Law, finance and administration. A summary of what a regulation, contract or policy requires. Here the source usually exists and is retrievable, so this is a Unit 9 grounding task and the check is real. Escalate when the question is whether a rule *applies to this case*, which the document cannot settle.

Engineering and the physical world. A calculation, a tolerance, a material property. Often exactly checkable, so do the check. Escalate on anything with a safety case attached, where the standard is not "I verified it" but "the qualified person signed it".

Research and publication. A claim, a citation, a summary of a literature. Unit 9 is this. Escalate when a claim you cannot verify is load-bearing for a conclusion you intend to publish, because the cost lands on readers who cannot see how it was made.

Teaching, assessment and your own study. The consequence is to your own learning, which makes it easy to discount and easy to regret. The rules that apply are your institution's, and they are the institution's to state. What this module says is narrower: an output that arrives without the work has skipped the thing the work was for.

12.1.6 What this unit is not saying

It is not saying high-stakes work is off limits. It is saying the amount of checking scales with the cost of being wrong, and that at some point the right amount stops being available to you alone.

It is not a list of forbidden topics. The triggers are about your situation — what you can check, what you can undo, who bears the cost, not about subject matter. The same question can be routine in one setting and an escalation in another.

12.1.7 Summary

- Kind of check comes from where the reference lives. Amount of check comes from what being wrong would cost. Ask them in that order and keep them separate.
- Amount is consequence, reversibility, your own uncertainty and whose decision it is, taken together — a low score on any one can carry the whole decision.
- "Large consequence, and I cannot assess it" is the row that means stop, and it is the row people talk themselves out of.
- Escalating well means handing over a claim, what you checked, what you could not check, and what decision is waiting, not handing over an output.
- The most reliable trigger is noticing that you are looking for a reason to accept the answer.

Then answer `cp12-two-axes` and `cp12-escalation-trigger` at the foot of this page.

12.2 Unit 12 activity — the two axes, and one escalation you would actually make

Time: about 14 minutes. Read `page.md` first.

12.2.1 Part 1 — same class, different amount (5 minutes)

Here are three tasks that all sit in the *same* class — each is a claim about the world, checkable only against a source.

	The task
A	Confirming the opening hours of a library before you walk there
B	Confirming a submission deadline before you plan your week around it
C	Confirming a dosage interval that will be written into care notes for someone else

For each, write three short answers:

1. **Kind of check** — what would you compare it against? (This should be nearly the same answer for all three, and that is the point.)
2. **Amount** — consequence, recoverability, and whether you could assess it.
3. **What you would actually do**, in one line.

The exercise is to see the first column stay constant while the third changes completely. If your answer to question 1 varies a lot between the three, look again — you may be letting importance decide the kind, which is the error Unit 3 warned about.

12.2.2 Part 2 — the row you talk yourself out of (4 minutes)

Find a real case from your own study or work that belongs in the table's bottom row: **large consequence, and you could not confidently assess a wrong answer.**

Write four lines.

The claim
Why the consequence is large, and who bears it
Why you could not assess it
Who is qualified to settle it, and is that person reachable

If you genuinely cannot find one, use this instead: think of something you *did* accept without being able to check it, and write the same four lines about that. Almost everyone has one, and it is more useful than an invented case.

12.2.3 Part 3 — write the handover (5 minutes)

Take the case from Part 2 and write what you would actually send. Four parts, as the reading gives them:

1. The claim you need settled, stated precisely.
2. What you have already checked, and what it showed.
3. What you could not check, and why.
4. What decision is waiting on it.

Then apply one test:

> Have I handed over a **question**, or have I handed over an **output** and asked someone to deal with it?

The second wastes the other person's time and hides what you already know. The first is a colleague asking a colleague something.

Keep Part 3. It is the same artefact as the Unit 17 evidence pack, used for its other purpose: not to defend a claim you are making, but to hand over one you are not prepared to make.

12.2.4 Now take the checkpoints

`cp12-two-axes` is about Part 1. **`cp12-escalation-trigger`** is about Part 2.

```
python3 selfcheck.py run cp12-two-axes
python3 selfcheck.py run cp12-escalation-trigger  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check.

12.2.5 Optional extensions

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

Find where the axes are already separated for you (about 5 minutes). Many professions have this built in — sign-off, second-checking, four-eyes rules, supervision requirements. Find one in a field you know and write down what it treats as the trigger. You will usually find it is consequence and irreversibility, not the likelihood of error, which is the same conclusion this unit reaches from a different direction.

Rewrite a refusal (about 4 minutes). Find a time you declined to answer something, or should have. Write it as a handover rather than a refusal: the claim, what you checked, what you could not, and what is waiting. The difference between "I do not know" and a handover is roughly four sentences, and it is most of the difference between being unhelpful and being useful.

12.2.6 Activity Answers

12.3 Unit 12 answers and guidance

Read this after attempting the activity.

12.3.1 Part 1 — what the three tasks show

The kind of check is the same for all three. Each is a claim about the world with a source that exists: the library's own posted hours, the official assessment timetable, the prescribing reference. In Unit 1's terms all three are recall unless you supply the source; in Unit 3's terms all three need source checking. Nothing about consequence changes that.

The amount is completely different.

	Consequence	Recoverable	Could you assess it	What follows
A	You walk somewhere for nothing	Yes, immediately	Yes	One look at the official page. Done.

	Consequence	Recoverable	Could you assess it	What follows
B	You mis-plan a week; possibly a late submission	Partly, and expensively near the deadline	Yes	Check the official timetable, not a summary of it, and note the date you checked.
C	Harm to a person who is not you	No	Almost certainly not	Escalate. This is the bottom row.

Three things worth noticing.

Row C is not distinguished by being "medical". It is distinguished by the three factors: the cost lands on someone else, it cannot be taken back, and you are outside your competence. A logistics figure with the same three properties gets the same answer.

Row B is the interesting one. It looks routine and is the one most people under-check, because the consequence is entirely to them and feels manageable. Note that its recoverability *decreases with time*, which is a property worth looking for generally.

"Could you assess it" is about you. Two people can get different correct answers for row C — a prescriber and a student are not in the same position — and the table is doing its job when that happens.

12.3.2 Part 2 — the row people talk themselves out of

The conversion from "large consequence, cannot assess" to "large consequence, can probably assess" happens in one of three ways, and they are worth recognising:

- **Familiarity read as competence.** You have seen a lot of writing on this, so the answer feels checkable. Recognising the vocabulary is not the same as being able to tell a right answer from a plausible one.
- **Partial checks read as whole ones.** You confirmed the parts you could, and the confirmed parts stand in for the rest. Unit 9's scoped statement exists to stop exactly this.
- **The check that is not independent.** You looked it up somewhere that is plausibly where the claim came from in the first place. Agreement between an output and its own likely source is not confirmation.

If you used the "something I did accept" variant, that is the better version of this exercise. A case you actually let through tells you which of the three above is your habit.

12.3.3 Part 3 — question or output

The test has a clear signal. A handover that begins "Can you check this for me?" and pastes an output is handing over your whole problem including the parts you had already solved. A handover that begins "I need to know whether X, and here is what I have established" is a question, and it can be answered in minutes.

The second also does something the first does not: it makes your own work visible, including its limits. That is why it is the same artefact as the evidence pack. The pack defends a claim you are making; the handover explains a claim you are declining to make. The structure is identical because both are answers to "what is established, and how?"

12.3.4 The checkpoints

`cp12-two-axes` tests the order of the two questions. The tempting wrong answers let consequence decide the *kind* of check, which is precisely the error Unit 3 guarded against and which this unit is careful not to reintroduce.

`cp12-escalation-trigger` tests what actually licenses stopping. The trap is subject matter: the triggers are about your situation — what you can check, what you can undo, who bears the cost — not about the topic being a serious one.

12.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp12-two-axes` — Which question does consequence answer

Kind: choice.

Two tasks: confirming a library's opening hours, and confirming a dosage interval that will be written into someone else's care notes. Both are claims about the world with an authoritative source that exists. What differs between them?

Options, numbered from zero:

0. The amount of checking, and who should be doing it. The kind of check is the same for both – compare the claim against the authoritative source.
1. The kind of check. A serious claim needs a different sort of evidence from a trivial one.
2. Nothing that affects the method. Both are source-checking tasks and the same procedure applies to each.
3. The reliability you can expect from the system, which is lower on specialised topics.

Answer: `0`

Hint: Both claims are settled by opening the same kind of thing. Ask what that means for the kind of check, and then ask separately what differs – and whether what differs is about the claim or about you.

Diagnostics, by the answer given:

- `1` — This lets consequence decide the kind of check, which is the error Unit 3 warned about and the reason this unit introduces amount as a separate axis. Both claims are settled by comparing against an authoritative source; seriousness does not change what would count as evidence, only how much of it you need and whether you are the right person to weigh it.
- `2` — Half right, and it stops too early. The kind of check is indeed the same, which is why they sort together. But the second claim is irreversible, falls on someone else, and requires training to assess – so the same procedure emphatically does not apply. Same kind, different amount, and different person.
- `3` — This is a claim about the system, and this module makes no such measurement – nor should you. Even if it were true it would change how often you need the remedy rather than what the remedy costs, which is the distinction Unit 11 drew for actions and this unit generalises.

Explanation: Two axes, asked in order. Where the reference lives decides the kind of check; what being wrong would cost decides how much checking and by whom. Keeping them separate is what lets you use consequence without letting it corrupt the diagnosis – a task can be enormously important and still be checked by a single comparison against one document, and a task can feel routine while sitting in the row where you should not be deciding alone.

Checkpoint `cp12-escalation-trigger` — What licenses stopping

Kind: multi.

Which of the following are sufficient on their own to stop and hand the question to someone qualified? Select all that apply.

Options, numbered from zero:

0. You cannot construct any independent check of the claim
1. The checks you can construct disagree with each other
2. You notice you are looking for a reason to accept the answer
3. The topic is a serious one that you find intimidating

Answer: `0, 1, 2`

Hint: Three of these are about your situation – what you can check, what you can undo, what you have started doing. One is about how the topic makes you feel.

Diagnostics, by the answer given:

- `0, 1, 2, 3` — The first three, but not the fourth. Finding a topic intimidating is a fact about your confidence, and confidence runs in both directions – people are unnervingly comfortable in some areas where they cannot assess anything, and anxious in others where they can check perfectly well. The triggers are about your situation: what you can check, what you can undo, who bears the cost.
- `0, 1` — The third one belongs. Noticing that you are looking for a reason to accept an answer, rather than looking for whether it is right, is the most reliable of the four triggers precisely because it is about the process you are in rather than about the material. It is also the one you can apply when you have no other information.
- `3` — This is the only one of the four that is not a trigger. Seriousness of topic is not the axis – the same question can be routine in one setting and an escalation in another, depending on what you can check and who bears the cost of being wrong.

Explanation: Escalation is licensed by your situation, not by subject matter: no available independent check, checks that disagree, or the recognition that you have started looking for permission rather than for evidence. The fourth option describes a feeling, and feelings about a topic track competence poorly in both directions. Handing over well means handing over a question – the claim, what you checked, what you could not, and what decision is waiting – rather than an output for somebody else to deal with.

13 Unit 13 — Privacy and data handling

13.1 Reading

Media for this unit:

- **Animation**: *What crossed, and what stayed* — this unit has an interactive version on the course website (`what-crossed.html`); the still below is one moment from it.

Figure 15: What crossed, and what stayed. Four pieces cross a device boundary when a learner sends one question with one attachment.

Reading time: about 9 minutes. Activity: about 12 minutes.

13.1.1 The question this unit answers

Unit 7 says, in one paragraph, not to paste confidential or personal material into an AI system. That is a sound instruction and it is not enough to act on, because it does not say what happens to what you paste, and without that you cannot judge the cases the sentence does not cover.

This unit says what leaves your device, what is retained, and what “do not paste this” means when the material is not obviously secret.

It does **not** state your institution’s rules or the law. Those exist, they differ by place and by organisation, and they are not this module’s to invent. What the module can give you is the shape of the thing, so that when you read your own rules you know what they are talking about.

13.1.2 What leaves the device

With a hosted service — anything you reach through a browser or an app, rather than a model running on your own machine — what you supply leaves your device. Exactly what happens to it then is a property of the product, and products differ. The rule that holds regardless:

> Assume material you submit to a hosted service may be transmitted to and processed by the provider and whoever it works with, unless that product’s documentation and settings establish otherwise.

That covers your prompt, the conversation, any file you attach, and anything a tool fetched on your behalf.

Three things follow that people are routinely surprised by.

An attachment goes whole. You asked about one clause; the document went. If the file has tracked changes, comments, hidden columns, or a second sheet, those are text too. Whether the whole of it then reaches the model is a separate question — some products index or chunk a file and retrieve parts of it — but the file left your machine either way, and that is the part that matters for privacy.

Earlier turns keep going. A conversation carries its history, so something you pasted twenty turns ago is generally still in play rather than gone. Whether it is re-sent verbatim, summarised, or retrieved from storage varies by product.

Deleting the conversation is not deletion. It removes it from your view. Whether and when the provider's copies, backups and abuse-monitoring logs go is a question about that service's policy, not about the button.

13.1.3 What is retained, and by whom

This is where a module that cannot date itself has to be careful. Retention depends on the service, the plan you are on, and settings that change. So rather than facts that will be wrong next year, three questions whose answers you can go and find, and which your own rules will assume you have found:

1. **Is this input used to train future models?** Often configurable, often different between a free tier and a paid or institutional one. This is the question consumer defaults are least likely to answer the way you would.
2. **How long is it stored, and who can read it?** Providers commonly retain conversations for a period for abuse monitoring, which means staff access under stated conditions.
3. **Where is it processed, and under what terms?** Server location is part of this and not the whole of it: which law applies can depend on who the data is about, who is responsible for it, and what the contract says. You are not expected to settle that. You are expected to notice that it is a question, and to ask whoever handles it where you work or study.

If you cannot answer all three for the service in front of you, that is itself the finding, and it makes the classification below strict rather than optional.

13.1.4 A classification you can apply in seconds

Not a legal taxonomy. Four bands, sorted by who is harmed if the material becomes readable by someone else.

Band	What it covers	What it settles
Public	Already published; you could put it on a poster	The privacy question, mostly. Not the rights question
Yours alone	Your notes, your drafts, your code, your unsubmitted work	That nobody else's privacy is at stake. Not whether you are permitted
Someone else's	Anything identifying another person, or told to you in confidence	That you must not paste it as it stands
Not yours to decide	Employer or client material, unpublished research, anything under an agreement	That the decision is not yours. Ask before, not after

These bands are one dimension, not a permission slip. That is the most important sentence on this page. A band tells you about the *privacy* risk — who is harmed if this becomes readable by someone else. It does not tell you whether you may use the material, and three other questions sit alongside it:

- **Rights.** Public does not mean unrestricted. A published paper, a photograph, a piece of code on a public repository and a database of public records all carry licences or terms, and Unit 14 is about exactly this. "I found it in public" settles nothing about what you may do with it.
- **Purpose and terms.** The service you are pasting into has its own terms, and your institution or employer may have approved some services and not others.
- **Assessment rules.** Material that is entirely yours may still be assessment-restricted, and that is a rule about the task rather than about the data.

So the first row means "the privacy risk here is low", and the second means "no third party is exposed by this". Both leave the other three questions open.

The third and fourth bands are the ones that matter most, and they are the ones a sentence about "confidential information" does not reach.

Someone else's is broader than it sounds. A message a friend sent you. A draft your colleague shared. An interview transcript. A photograph with a person in it. Health, financial or immigration details about anybody. The test is not "is this marked confidential" but "did somebody else give me this expecting it to stay where I put it".

Not yours to decide is the band people miss entirely, because nothing about the material feels sensitive. A dull internal spreadsheet is often covered by an agreement someone else signed. The point of the band is that the decision is not yours to make on the material's apparent importance.

13.1.5 Working with material you cannot paste

The useful move is almost never "do not use the tool". It is to change what you send.

Redact and generalise. Replace names, dates, identifiers and distinctive details. "A patient" not a name; "a large employer in the sector" not the employer. The task usually survives this untouched, because you were asking about structure, not about the person.

Redaction reduces risk; it does not by itself make material anonymous or make its use authorised. A combination of details can identify someone with every name removed — a role, a date and a place is often enough — and an organisation may forbid sending even de-identified material to a service it has not approved. Removing the names is the beginning of that judgement rather than the end of it.

Abstract the question. You rarely need to send the document to get what you want. "Here is a clause I have rewritten with all identifying detail removed — does it say what I think it says?" is a different question from pasting the contract, and it usually gets the same answer.

Bring the tool to the data. Where a locally-run model or an institutionally-approved service is available, the first question — what leaves the device — gets a different answer, and that is often the whole difficulty solved. It does not remove the question. A local application may still check for updates, download models, sync to cloud storage, or send diagnostics, and the device it runs on has its own exposure. Ask what leaves, rather than assuming that local means nothing does.

Watch re-identification. Removing a name is not always removing a person. A combination of details — a role, a place, a date, an unusual circumstance — can identify somebody as surely as a name, especially in a small institution. Ask whether the remaining details would let a colleague work out who it is.

13.1.6 The one about your own work

Your unsubmitted coursework is in the "yours alone" band as far as *privacy* is concerned. Whether you may use an AI system on it is a different question entirely, governed by your institution's rules on assessed work, which this module does not state and cannot.

Keeping those two questions apart matters, because they have different answers and different authorities. "It is my own work so there is no privacy problem" can be true at the same time as "and I am not permitted to do this".

13.1.7 What this unit is not saying

It is not saying hosted services are unsafe. It is saying that what leaves your device is a fact you can establish, and that the classification depends on whose material it is rather than on how secret it feels.

It is not giving you a rule to follow instead of your institution's. Where they conflict, theirs governs. This gives you the vocabulary to read theirs.

13.1.8 Summary

- With a hosted service the whole context leaves your device, including the parts of an attachment you were not asking about, and it leaves repeatedly as a conversation grows.
- Deleting a conversation removes it from your view; what it does to stored copies is a question about the service.
- Three questions to answer about any service you use for anything but public material: is it used for training, how long is it kept and who can read it, and where is it processed.

- Classify by whose material it is: public, yours alone, someone else's, or not yours to decide. The last two are the ones a warning about "confidential information" misses.
- The move is usually to change what you send — redact, generalise, abstract the question — rather than to abandon the task.

Then answer `cp13-what-leaves` and `cp13-whose-material` at the foot of this page.

13.2 Unit 13 activity — classify, then redact

Time: about 12 minutes. Read `page.md` first.

13.2.1 Part 1 — answer the three questions about your own tool (4 minutes)

Pick the AI service you actually use most. Find out, from its own documentation or settings:

Question	What you found	Where you found it
Is my input used to train future models? Can I change that?		
How long are conversations kept, and who can read them?		
Where is it processed?		

Two rules for this part.

Record where you found each answer. This is Unit 9's discipline on a question about your own tools. A locator makes the answer checkable and dateable; without one you have a memory.

"I could not find out" is an answer. Write it, and note how long you looked. If a provider does not make this findable in a few minutes, that is a fact about the service and it should push your classification stricter.

13.2.2 Part 2 — sort eight items (4 minutes)

Put each into one band: **public**, **yours alone**, **someone else's**, **not yours to decide**.

	Item
1	A paragraph from a published journal article
2	Your own half-finished essay
3	A screenshot of a group chat, to ask what someone meant
4	An internal spreadsheet of your employer's monthly figures
5	A CV a friend sent you, to ask how to improve it
6	Your own lecture notes, typed by you in a lecture
7	An anonymised case study from a textbook
8	A draft paper a colleague shared for comment

Then answer one question in a sentence: **which one did you find hardest, and why?** There is a defensible disagreement available on at least two of them, and noticing where it is matters more than the label.

13.2.3 Part 3 — redact one, and test it (4 minutes)

Take item 5, or any real piece of someone else's material you might genuinely have wanted help with.

1. Rewrite it so the task survives and the person does not appear: remove names, dates, identifiers, and distinctive details.
2. Then apply the re-identification test: **could a colleague who knows the setting work out who this is?** Look particularly at combinations — a role plus a place plus a time is often enough in a small institution.
3. If the answer is yes, generalise further or abandon the task.

Write down what you removed and what you had to change beyond names. The second list is usually the interesting one.

13.2.4 Now take the checkpoints

`cp13-what-leaves` is about Part 1. **`cp13-whose-material`** is about Part 2.

```
python3 selfcheck.py run cp13-what-leaves
python3 selfcheck.py run cp13-whose-material  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check.

13.2.5 Optional extensions

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

Find your institution's actual rule (about 6 minutes). Locate the policy that governs AI use with personal or institutional data where you study or work, and record where it is and what it says about the four bands. This module cannot tell you the rule; it can tell you that you should know where it lives. Note whether it addresses the "not yours to decide" band at all — many do not, and knowing that is useful.

Open an attachment as text (about 4 minutes). Take a document you might have attached and look at what a text extraction of it actually contains: tracked changes, comments, metadata, hidden rows, the second worksheet. The gap between what you thought you were sending and what would have gone is the point of the exercise.

13.2.6 Activity Answers

13.3 Unit 13 answers and guidance

Read this after attempting the activity.

13.3.1 Part 1 — what a good answer looks like

The locator column is what separates this from a vague impression. "The privacy policy, section on data retention, read today" is an answer; "I think it does not train on my data" is a memory, and this is the module that distinguishes those.

If you found the answers in a marketing page rather than a policy or a settings screen, note that. Marketing text about privacy is a claim from an interested party, which is a Unit 9 problem: right source, possibly wrong claim.

"I could not find out" pushes the classification stricter, not looser. An unanswerable question about retention means you treat the service as if the answer were the least favourable one, because you have no basis for anything else. That is the same move as Unit 12's bottom row.

13.3.2 Part 2 — the eight items

	Band	Why
1	Public	Already published.
2	Yours alone (for privacy)	See the reading on the separate assessment question — that is not a privacy matter and has a different authority.
3	Someone else's	Other people wrote those messages, in a setting they took to be private. Your being in the chat does not make it yours to publish.
4	Not yours to decide	The dullness of the figures is irrelevant. It is the employer's material and probably under an agreement someone else signed.

	Band	Why
5	Someone else's	A CV is dense with personal data — name, employers, dates, sometimes address. Sending it to help is still sending it.
6	Yours alone, with a caveat	You typed them, so the words are yours. If they quote a named person's unpublished remarks, that part is not.
7	Public	Published and already anonymised by someone with the responsibility to do so.
8	Not yours to decide, and someone else's	Unpublished research shared in confidence. Two bands at once, and both say ask first.

The two where reasonable people differ are 6 and 7.

Item 6 divides on whether lecture notes are your words or a record of someone else's. In practice both: your paraphrase is yours, a verbatim quotation of an unpublished remark is not, and the useful habit is noticing that a single document can span bands.

Item 7 divides on how much you trust the anonymisation. Published and professionally anonymised is a reasonable basis; "it looked anonymised to me" is not, and the re-identification question from Part 3 applies to material you receive as well as material you produce.

Item 4 is the one most people get wrong, and it is worth dwelling on. Nothing about a routine internal spreadsheet feels sensitive, which is exactly why the band exists: the decision is not yours to make on the material's apparent importance, because you are not the person who agreed the terms.

13.3.3 Part 3 — what you had to change beyond names

The second list is where the learning is. Common answers:

- a job title that only one person holds;
- a date that pins the event;
- an unusual circumstance that is the whole reason you were asking;
- the name of a small organisation, which narrows the population to a handful.

That last category is the awkward one, because the distinctive detail is often the thing you needed help with. When removing it destroys the question, you have found a task that should not be sent — and that is a result, not a failure of redaction.

If you concluded that no colleague could identify the person, check whether you tested that against someone who knows the setting or against yourself. You are the worst possible judge of whether your own redaction works, because you already know the answer.

13.3.4 The checkpoints

`cp13-what-leaves` tests the model of what a hosted service receives. The trap is thinking in terms of what you asked about rather than what you sent.

`cp13-whose-material` tests the classification. The trap is sorting by how sensitive material feels rather than by whose it is — which is why the internal spreadsheet is in the question.

13.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp13-what-leaves` — What actually leaves your device

Kind: multi.

You attach a 40-page report to a hosted AI service and ask a question about one paragraph on page 3. Which of the following leave your device? Select all that apply.

Options, numbered from zero:

0. The paragraph you asked about
1. The other 39 pages of the report
2. Tracked changes and comments stored in the file
3. The earlier turns of this conversation, re-sent with your question

Answer: `0, 1, 2, 3`

Hint: Distinguish what you asked about from what you sent. Then consider what a document contains beyond the words you can see when you open it.

Diagnostics, by the answer given:

- `0` — Only the paragraph would leave if you had pasted only the paragraph. You attached the file, so the file went – converted to text, including the parts you were not asking about. This is the most common surprise in this unit and the reason 'attach the document' and 'quote the passage' are different decisions.
- `0,1` — The report went whole, and so did the material stored inside the file that is not visible when you read it: tracked changes, comments, hidden rows, additional worksheets. A text extraction sees them. So does the service.
- `0,1,2` — Nearly. The conversation goes too, and it goes again with every turn – so anything you pasted earlier has left your device many times rather than once. That is also why a long conversation costs more per question than a short one.

Explanation: With a hosted service the whole assembled context leaves, not the part of it you are thinking about. That includes an attachment in full, the material stored inside the file that you do not see when you read it, and the entire conversation so far, re-sent each turn. None of this makes hosted services unsafe – it makes what you attach a decision rather than a convenience, and it is why redacting or quoting a passage is a genuinely different act from attaching the document.

Checkpoint `cp13-whose-material` — Whose material is it

Kind: match.

Match each item to its band. Answer with four option numbers in the order of the items.

Observations, in order:

1. A paragraph from a published journal article
2. Your own half-finished essay
3. A screenshot of a group chat, so you can ask what someone meant
4. A routine internal spreadsheet of your employer's monthly figures

Options, numbered from zero:

0. Public – paste freely
1. Yours alone – your call, though assessment rules are a separate question
2. Someone else's – do not paste, remove or replace first
3. Not yours to decide – ask before, not after

Answer: `0, 1, 2, 3`

Hint: Ask who would be harmed if this became readable by someone else, and who could give permission for it to be sent.

Diagnostics, by the answer given:

- `0,1,3,2` — You have swapped the last two. The test is not how sensitive the material feels – a group chat is chatty and an internal spreadsheet is dull. It is whose material it is. Other people wrote those messages expecting them to stay where they were put, which is someone else's. The spreadsheet belongs to an organisation that agreed terms you did not sign, so the decision is not yours to make.

- `0, 1, 2, 2` — The spreadsheet is not in the same band as the chat, and the difference is who could authorise sending it. For the chat, the people in it could. For employer material, the answer lies with whoever holds the agreement – which is why the band is defined by the decision not being yours rather than by the material being secret.
- `0, 0, 2, 3` — Your own unfinished essay is not public, and the distinction matters in both directions. As a privacy question it is yours alone. Whether you may use an AI system on assessed work is a separate question with a different authority – your institution's – and keeping the two apart is what stops 'it is my own work' from answering a question it does not answer.

Explanation: The bands sort by whose material it is, not by how secret it feels. That is why a dull internal spreadsheet and a private message land in different bands from each other and both away from your own drafts. The two bands a warning about 'confidential information' misses are the last two – material somebody gave you expecting it to stay put, and material whose terms were agreed by someone other than you.

14 Unit 14 — Copyright and attribution

14.1 Reading

Media for this unit:

- **Animation**: *One question, three answers* — this unit has an interactive version on the course website (`three-columns.html`); the still below is one moment from it.

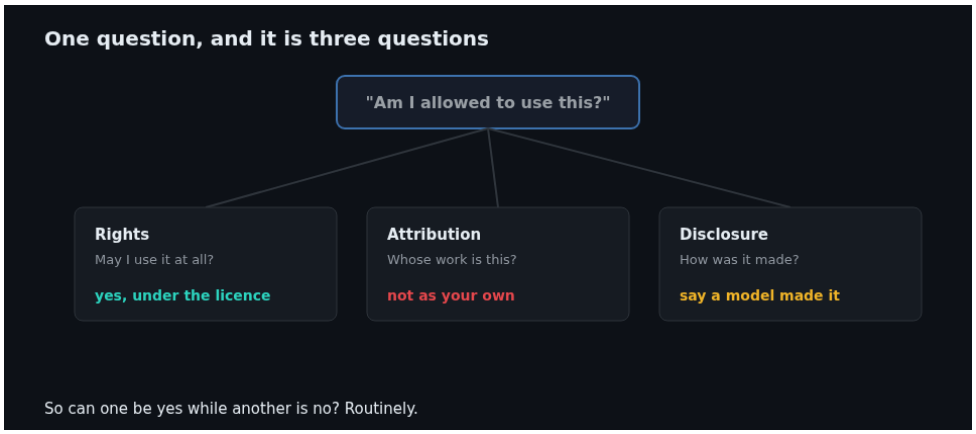

Figure 16: One question, three answers. One permission question splits into rights, attribution and disclosure, which answer differently about the same artefact.

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

14.1.1 Two questions people run together

Unit 17 is about **disclosure**: saying how a piece of work was made. This unit is about two different questions that sit beside it and are constantly confused with each other.

> **Rights.** What may lawfully be done with this — copied, published, sold, built on. **Attribution.** Who should be credited, and what an honest account of the work says.

They come apart in both directions, which is the useful thing to know. Something can be entirely lawful to use and dishonest to present as your own. Something can be scrupulously credited and still an infringement. Neither question answers the other.

Add Unit 17's disclosure and there are three, and this module's position is that you owe all three separately.

14.1.2 What this module will not tell you

Copyright in generated output is genuinely unsettled, and it is unsettled differently in different countries.

That is not a hedge to avoid the question. It is the answer, and stating it plainly is more useful than a confident summary would be. A page that told you "generated output is not copyrightable" or "training on public material is fair" would be doing the thing this whole module trains you to catch: a confident claim in an area where the record does not support one, delivered in a register that does not distinguish the settled parts from the contested ones.

So this unit does two things instead. It separates what is broadly settled from what is not, so you know which of your questions has an answer. And it gives you a practice for attribution, which is not a legal question and which you can get right today.

14.1.3 What is broadly settled, and what is not

Reasonably settled.

- Copyright protects a particular expression, not an idea, a fact or a style.

- Material you feed *into* a system does not stop being covered by whatever covers it. A licence you are bound by still binds you.
- If generated output reproduces a substantial part of an existing protected work, the ordinary rules about that reproduction apply — the fact that a machine produced it does not create an exemption.
- Terms of service are a separate matter from copyright and bind you by agreement. A provider can grant or restrict commercial use of output regardless of what copyright says.

Not settled, and contested differently by jurisdiction.

- Whether generated output attracts copyright at all, and if so whose.
- Whether training on protected material without permission is lawful, and under what conditions.
- What obligations attach to output that resembles a particular existing work without copying it.

Notice which of these you actually need. Most practical questions — may I use this in my dissertation, may I publish this on a client's site — are answered by your institution's or client's rules and by the terms of service, both of which exist and are readable. The unsettled questions bite mainly when you need to assert a right over something, or when output resembles an identifiable work.

14.1.4 Attribution: the part you can get right

Nothing above prevents you from giving an honest account. Attribution is a practice, not a permission, and it has three parts.

Credit what a person made. If your work rests on someone's argument, data, figure or code, cite them. That obligation is entirely untouched by how you found the material. A generated summary that led you to a source does not make the source uncited — it makes it a source you found in an unusual way and have now read.

Never cite a source you have not opened. This is Unit 9, and it is the point where the two units meet: attribution rests on the citation being real, and a citation you have not opened is a claim you have not checked.

Say what you did, separately. How the work was made is Unit 17's disclosure. It answers a different question from "whose ideas are these", and running them together produces the two characteristic failures:

Failure	What it looks like	Why it is wrong
Disclosure standing in for credit	"Written with AI assistance" and no citations	Says how it was made and nothing about whose ideas it rests on
Credit standing in for disclosure	A full bibliography, no mention of assistance	Says whose ideas, and nothing about how the text was produced

You owe both, and they belong in different places: citations where the claim is made, and a disclosure statement about the work as a whole.

Do not credit the system as an author. Not because of a rule about who counts as an author, but for a reason this module can defend: an author is someone answerable for the work. Naming a system as an author states that something can answer for the claims, and nothing in the system does. Record the tool the way you would record any other instrument — in the account of how the work was made.

14.1.5 A worked case

You are writing a literature review. You ask a system for an overview; it produces a summary with six citations. You read four of the six papers, use two of them substantially, and rewrite everything in your own words.

What you owe:

- **Cite the two papers** you used, at the claims they support. You read them; they are sources.
- **Do not cite** the four you did not read, and do not cite the two you read but did not use.
- **Disclose** that you used an AI system to produce an initial overview, that you retrieved and read the sources yourself, and that the text is your own.

- **Say nothing about the system in the bibliography**, because the bibliography answers "whose ideas", and the system did not have any.

The awkward case in that list is the one people ask about: what if the *framing* of your review came from the summary? Then say so in the disclosure. "The structure of section 2 follows an outline produced by an AI system, which I then revised" is a true sentence, it costs you nothing, and it is exactly the kind of thing a reader is entitled to know.

14.1.6 What this unit is not saying

It is not legal advice, and it is not your institution's rule. Both exist and govern; this gives you the categories to read them with.

It is not saying attribution is optional because rights are unsettled. The opposite: attribution is the part you can get right regardless of how the legal questions resolve, which is why it is where this unit spends its effort.

14.1.7 Summary

- Three separate obligations: rights (what may lawfully be done), attribution (whose ideas), and disclosure (how it was made). Each needs answering on its own.
- Copyright in generated output is genuinely unsettled and varies by jurisdiction. This module says so rather than summarising it confidently.
- What is settled: your inputs keep their licences, reproduction of a protected work is still reproduction, and terms of service bind you separately.
- Cite what a person made, never cite a source you have not opened, and keep credit and disclosure in their separate places.
- Do not name a system as an author — an author is someone answerable for the work.

Then answer `cp14-three-obligations` and `cp14-settled-or-not` at the foot of this page.

14.2 Unit 14 activity — separate the three, then write them

Time: about 12 minutes. Read `page.md` first.

14.2.1 Part 1 — sort six questions (4 minutes)

Each question below is really about **rights**, **attribution**, or **disclosure**. Label each, and for one sentence say who could answer it.

	The question
1	May I put this generated diagram in a poster I am selling?
2	Should the paper whose argument I am summarising appear in my bibliography?
3	Does my reader need to know an AI system drafted this section?
4	The output looks a lot like a figure I have seen before. What now?
5	I found this source through a generated summary and then read it. Do I cite it?
6	Am I allowed to use an AI system for this coursework at all?

Two of these have answers this module can give you, two have answers your institution or a contract can give you, and two do not have settled answers at all. Say which is which.

14.2.2 Part 2 — the two failures (3 minutes)

Here are two disclosure-and-credit statements for the same piece of work.

A. "This report was written with AI assistance."

B. A full bibliography of eleven sources, correctly formatted, and no other statement.

For each, write one line saying **what a reader still does not know**.

Then write a third version that does the job of both, in no more than three sentences.

14.2.3 Part 3 — your own worked case (5 minutes)

Take a real or realistic piece of work of your own where you used an AI system. Write down four things:

Cite: whose ideas the work rests on, and where each citation goes
Do not cite: what you will *not* cite, and why
Disclose: how the work was made, in one or two sentences
Unsettled: any rights question you cannot answer, and who could

The second row is the one that does the teaching. A source you did not read, a paper you read but did not use, and a system that produced an outline are all things that do not belong in a bibliography — for three different reasons. Say which reason applies to each of yours.

Keep Part 3. The disclosure line is the same artefact Unit 17 asks for, and this is the version of it that also answers "whose ideas".

14.2.4 Now take the checkpoints

`cp14-three-obligations` is about Part 1. **`cp14-settled-or-not`** is about knowing which questions have answers.

```
python3 selfcheck.py run cp14-three-obligations
python3 selfcheck.py run cp14-settled-or-not  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check.

14.2.5 Optional extensions

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

Read one set of terms of service (about 6 minutes). Find what the service you use says about ownership of output and about commercial use. Record the clause and the date you read it. This is a Unit 9 grounding exercise on a document that actually governs you, and most people have never opened it.

Find where your discipline has already decided (about 5 minutes). Many journals, conferences and professional bodies have published a position on declaring AI assistance. Find one in your field, note what it requires, and compare it with the three obligations in the reading. The interesting question is which of the three it addresses and which it leaves silent.

14.2.6 Activity Answers

14.3 Unit 14 answers and guidance

Read this after attempting the activity.

14.3.1 Part 1 — the six questions

	About	Who can answer it
1	**Rights**	Partly the terms of service, which exist and are readable; partly genuinely unsettled, since whether the output attracts copyright and whose it is varies by jurisdiction.
2	**Attribution**	This module. Yes — you are resting on their argument, and how you found it is irrelevant.
3	**Disclosure**	This module, and your institution or publisher. Yes; Unit 17 is about how.
4	**Rights**	Not settled in general. If it reproduces a substantial part of a protected work, ordinary rules apply; resemblance without copying is the genuinely open case. Escalate — this is Unit 12's bottom row.
5	**Attribution**	This module. Yes, cite it — you read it. Finding a source in an unusual way does not change what you owe its author.
6	**Neither, and not this module's**	Your institution. It is a rule about assessed work, not about rights or credit.

The two the module answers are 2 and 5, and they are the same question twice — which is deliberate, because the second one is where people hesitate. Reading a paper you found through a generated summary makes it a source you read. Nothing about the route to it changes the obligation.

Question 6 is the one that most often gets attached to the wrong authority. It is not a copyright question and not an attribution question, and no amount of careful citation answers it.

14.3.2 Part 2 — what each version leaves out

Version A tells the reader how the text was produced and nothing about whose ideas it rests on. A reader cannot tell whether the claims are the author's, drawn from sources, or invented — and there is no way to follow anything up.

Version B tells the reader whose ideas and nothing about how the text was produced. A reader may reasonably assume the prose is the author's own work, and that assumption has not been either confirmed or corrected.

A third version that does both:

> The argument in sections 2 and 3 follows Okonjo (2020) and Hallam (2021), cited at the relevant claims. An AI system was used to produce an initial outline and to redraft two paragraphs for length; all sources were retrieved and read by me, and the final text is mine.

Three sentences, and every part of it is checkable. Note what it does *not* do: it does not claim the work is correct. That is a defence, and Unit 17 handles it separately.

14.3.3 Part 3 — the "do not cite" row

Three different reasons, and separating them is the point:

- **A source you did not read.** Not cited because you have not checked it, and a citation asserts that you have. Unit 9's `fabricated-locator` and `misattributed` are what happens when this rule is broken.
- **A source you read but did not use.** Not cited because your work does not rest on it. A bibliography is not a reading list, and padding one implies support that is not there.

- **The system that produced an outline.** Not cited because a bibliography answers “whose ideas”, and an outline is not an idea belonging to anyone. It goes in the disclosure, where the question is “how was this made”.

If your answer put the system in the bibliography, that is the most common version of this error and it usually comes from a good instinct — wanting to be transparent. The instinct is right and the location is wrong: naming a system as an author states that something can answer for the claims, and nothing in the system can.

14.3.4 The checkpoints

`cp14-three-obligations` tests the separation. The trap is treating disclosure as covering credit, which is the failure Version A shows.

`cp14-settled-or-not` tests something less common in a checkpoint: knowing which questions have answers. The tempting wrong answers are the confident ones, in both directions — and a module that trains you to catch confident-unsupported claims should not make one here.

14.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp14-three-obligations` — Three obligations, separately owed

Kind: match.

Match each statement to the obligation it discharges. Answer with three option numbers in the order of the statements.

Observations, in order:

1. A bibliography listing the eleven works whose arguments and data the report rests on
2. A sentence saying an AI system produced the initial outline and redrafted two paragraphs
3. A check that the licence on the dataset used in the report permits redistribution

Options, numbered from zero:

0. Attribution – whose ideas this rests on
1. Disclosure – how the work was made
2. Rights – what may lawfully be done with it

Answer: `0, 1, 2`

Hint: Ask what question a reader would be answering with each one: whose ideas are these, how was this made, or what am I permitted to do with it.

Diagnostics, by the answer given:

- `1,0,2` — You have swapped the first two, and this is the substantive confusion in the unit rather than a slip. A bibliography answers whose ideas; a statement about AI assistance answers how the text was produced. Neither answers the other, which is why a report with only one of them leaves a reader unable to tell something they are entitled to know.
- `0,2,1` — You have swapped the last two. A statement about how the work was made is disclosure – it is about process. A question about what a licence permits is about rights, and it is answered by reading the licence rather than by describing your own conduct.
- `2,1,0` — You have swapped the first and last. A bibliography is not a rights document; listing a work does not establish that you may reuse it, and clearing rights to a dataset does not credit anyone. The two obligations come apart in both directions – lawful and uncredited, or credited and infringing.

Explanation: Three obligations, owed separately, discharged in different places. Citations go where the claim is made; a disclosure statement covers the work as a whole; a rights question is settled by reading a licence, terms of service, or a rule – and sometimes cannot be settled at all. Running any two together

produces work that looks accounted for and leaves a reader unable to answer a question they are entitled to ask.

Checkpoint `cp14-settled-or-not` — Which questions have answers

Kind: choice.

A colleague asks whether the images an AI system generated for your shared poster are copyrighted, and if so by whom. What is the defensible reply?

Options, numbered from zero:

0. That this is genuinely unsettled and differs by jurisdiction, so the answerable questions are what the terms of service permit and what our institution requires – and both of those we can go and read.
1. That generated output is not copyrightable, because copyright requires a human author.
2. That the copyright belongs to whoever wrote the prompt, since that is the creative contribution.
3. That it does not matter for a poster, since nobody enforces copyright at that scale.

Answer: `0`

Hint: Two of these options are confident answers that contradict each other, which is itself informative. Ask which option separates the question that has no settled answer from the ones that do.

Diagnostics, by the answer given:

- `1` — This is a real position in some jurisdictions and it is stated here as though it were settled everywhere. That is the exact move this module trains you to catch – a confident claim in an area where the record does not support one, in a register that does not separate the settled parts from the contested ones.
- `2` — Also a real position, also contested, and stated with the same unwarranted confidence as the previous option. Note that it directly contradicts that one, which is itself informative: when two confident answers to the same question contradict each other, the honest description of the field is that it is unresolved.
- `3` — This substitutes a prediction about enforcement for an answer about rights, and it is the kind of reasoning that is fine until it is not. It also ignores the two questions that do have answers – the terms of service and your institution's rules – both of which may prohibit the use regardless of who owns what.

Explanation: Some questions in this area are genuinely open, and saying so is more useful than a confident summary. The move is to separate the unanswerable question from the answerable ones sitting beside it: terms of service bind you by agreement and are readable, and institutional rules exist and are readable, and between them they settle most practical decisions. Reserve the unsettled question for when you actually need it – when you must assert a right, or when output resembles an identifiable work.

15 Unit 15 — Bias, representation and accessibility

15.1 Reading

Media for this unit:

- **Animation**: *One pair, then three* — this unit has an interactive version on the course website (`vary-one-thing.html`); the still below is one moment from it.

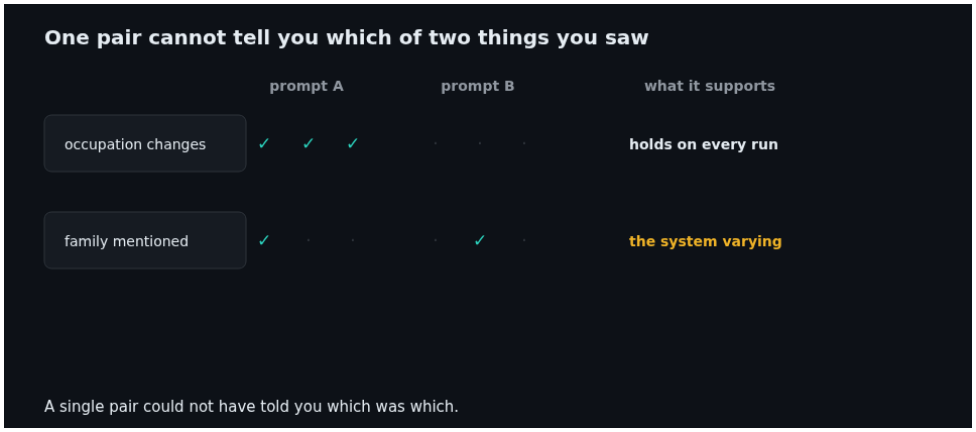

Figure 17: One pair, then three. A tally builds across three matched pairs; one feature holds every run and one turns out to be variation.

Reading time: about 8 minutes. Activity: about 20 minutes.

15.1.1 Checking without an oracle, one more time

Unit 9 ran the module's method on a case with no computable answer, and the reference was what the sources said. This unit runs it on a harder object still, where the reference is not a document at all.

The question is not "is this output biased", which invites a yes-or-no answer to something that does not have one. It is the question this module always asks:

> What would I compare this against, and what would that comparison establish?

That has real answers here. They are less comfortable than a value in a table, and they are the same kind of work.

15.1.2 Where the distributions come from

A model's output reflects the statistical patterns of its training data, shaped further by post-training (Unit 3). Two consequences, stated at the level this module can defend.

What is common in the data tends to be common in the output. Ask for "a nurse" and "a surgeon" and the outputs will tend towards whatever was common in the material — in pronouns, in names, in described appearance. "Tends" is doing real work in that sentence: post-training, system instructions and filtering all sit between the training distribution and what you see, and any of them can damp a pattern or introduce one. So the training data is where the tendency comes from and not a formula for the output. Nothing in the system decided this, and describing it as a decision would be the anthropomorphism the module rules out.

What is rare in the data is thin in the output. This is the less-discussed half and often the more consequential. Material about smaller languages, smaller places, minority practices and less-documented conditions is thinner, so output about them is more likely to be generic, out of date, or wrong in ways that are harder to catch, because there is also less to check it against.

Those two are why "bias" is a poor single word for it. One is a skew you can sometimes see in a single output. The other is an absence, and absences do not announce themselves.

15.1.3 Three places it shows up in your work

In summaries. A summary reflects what is prominent in what it summarised. If you ask for "the main criticisms of X" and one strand of criticism is well-represented online and another is not, the shape of the answer is a fact about the corpus rather than about the criticisms.

In examples. Generated examples, names, scenarios and case studies carry the distribution. In teaching material, recruitment material or anything with an audience, that becomes a fact about who sees themselves in your work.

In descriptions of people. Anything that classifies, ranks, or predicts something about people is the highest-consequence case, and Unit 12's escalation row usually applies.

15.1.4 What a check looks like

You cannot compare against ground truth, because there is no table of what the answer should have been. There are three things you can do, and they establish different things.

1. Vary one thing. Produce the same output with a single attribute changed — a name, a pronoun, a place, a language, and compare. This is the closest thing to a controlled comparison available, and it is the one to reach for.

It is worth being exact about what one such pair shows, because the obvious reading of it is wrong. Two fresh conversations differ in **two** ways, not one: the attribute you changed, and the ordinary run-to-run variation of a system that does not produce the same words twice. So a difference between them is not yet evidence about the attribute. It is one observation with two candidate explanations.

What separates them is repetition. Run the pair several times, in both orders, and look at whether the difference *persists*. A feature that appears in four runs out of five on one side and once out of five on the other is telling you something; a feature that appears once is telling you the system varies.

What a repeated comparison establishes: that outputs differed in these specific ways, at this frequency, on this system, on this date. What it does not establish: that no difference exists elsewhere, that the size of the difference is stable, or that you have found the mechanism. You have found a pattern in some outputs, which is a real finding and a modest one.

2. Compare against a real distribution. Where a published statistic exists — who actually works in a profession, which languages are spoken in a place — you can compare the output's spread against it.

What it establishes: a specific, citable mismatch. What it does not establish is more subtle: a match is not a licence, because the real distribution may be the thing you were trying not to reproduce. A recruitment description matching current demographics may be exactly the problem.

3. Ask who is missing. Not a comparison but a prompt to yourself. Who is not in this output, and would I notice if they were absent? Absence is the failure mode that no comparison catches, because you cannot compare against something you did not think of.

15.1.5 Accessibility is the checkable part

Some of this is genuinely hard to establish. Accessibility mostly is not, and it is where a check produces a definite answer.

If you use generated material in anything others read, the ordinary requirements apply and are testable:

- **Alt text** that conveys what the image conveys *for the purpose it is used for*. Generated alt text is a description of an image by something that cannot know why you included it, so it tends to describe contents rather than what the image is doing in your document. A purely decorative image takes *empty* alt text, so a screen reader skips it — describing it is the wrong answer, not a thorough one. A chart needs the figure's point in the alt text and its numbers reachable some other way, in a table or a text summary.
- **Headings that reflect the structure**, without skipping levels where the structure does not skip. This is a check on whether the outline is real, not a rule that a jumped level is automatically a failure.
- **Contrast** that meets the standard, checkable with a tool in seconds.

- **Meaningful link text** — not “click here”, which tells a screen-reader user nothing when links are listed out of context.
- **Colour never used alone** to carry meaning.
- **Keyboard access**, with a focus indicator you can see, in an order that matches the reading order.

The point is not that generated material is worse at this. It is that the requirements are checkable and often unchecked, and that unlike most of this unit you can settle them today, with a tool, for free.

This module runs those checks on itself. `scripts/check_accessibility.py` is in `make test`, and it fails the build, and its output ends with a list of the things a machine cannot decide, which is the honest shape for this whole subject.

15.1.6 What this unit is not saying

It is not saying generated material is uniquely biased. Human-written material carries distributions too, and the module makes no comparative claim. What is different is scale and invisibility: an output arrives without the provenance a human author’s work usually carries.

It is not saying a varied output is a fair one. Varying an output changes the output. Whether the underlying task was appropriate is a separate question, and it is the one worth asking first when the task is about people.

It is not asking you to solve this. It is asking you to notice it, check what can be checked, and escalate what cannot, which is what the module asks everywhere.

15.1.7 Summary

- Output tends to reflect the distribution of its training data, as later training and system design allow: what was common tends to be common, and what was rare is thin. The second is harder to catch because absences do not announce themselves.
- It shows up in summaries, in examples, and most consequentially in anything describing or classifying people.
- Three checks: vary one attribute and compare *over repeated runs*, compare against a published distribution, and ask who is missing. Each establishes something narrow, and none establishes fairness. A single pair differs in the attribute and in ordinary run-to-run variation, so it settles nothing on its own.
- A match to the real distribution is not a licence — sometimes the real distribution is the problem.
- Accessibility is the part with definite answers. Check it, because you can.

Then answer `cp15-what-a-check-shows` and `cp15-alt-text` at the foot of this page.

15.2 Unit 15 activity — vary one thing, then check what can be checked

Time: about 20 minutes. Read `page.md` first.

15.2.1 Part 1 — the one-attribute comparison (12 minutes)

Pick a prompt that asks for something about a person or a role — a short character description, a worked example about a professional, a scenario involving a named individual.

Use a **fictional** person throughout. Do not build this on a real name: you would be inferring things about someone who did not ask to be in your exercise, and the exercise works just as well without it.

1. Write the two versions of the prompt first, differing in exactly one attribute — a name, a pronoun, a country, a language. Decide before you run anything which features you are going to record, so that you are not choosing them after seeing what came back.
2. Run **three pairs**, each in **separate fresh conversations** (Unit 5 — the same conversation is not a second trial). Alternate which version you run first, because order is one more thing that can differ.
3. Record, for each of your predefined features, how many of the three runs on each side showed it. Specifics rather than impressions: which nouns, which adjectives, which details appeared, and what was assumed without being asked.

Six runs rather than two, because two fresh conversations differ in two ways — the attribute you changed and the system's own run-to-run variation — and a single pair cannot tell you which one you are looking at. Three each is not a study. It is the smallest number that lets you see whether a difference persists, and seeing that is the point of the exercise.

Then write two sentences, and be careful about both:

What this comparison established: …

What it did not establish: …

The second sentence is the harder one and the one that matters. Six runs are six runs. If a feature showed up on one side every time and never on the other, you have a pattern worth reporting; if it showed up twice on one side and once on the other, you have watched a system vary. Either way you have not measured a rate, you have not shown that no difference exists elsewhere, and you have not found the cause.

Report it as what it is: *on this system, on this date, across three pairs, these features appeared this often.* That sentence is defensible. "The model is biased against X" is not, on this evidence, and Unit 17 is about the difference.

If the outputs were near-identical throughout, say so. That is a result, with the same limits attached.

15.2.2 Part 2 — who is missing (3 minutes)

Take one of the outputs and answer three questions.

Who or what is **not** in this output?
Would I have noticed if I had not been asked?
Is there a version of this task where that absence would matter?

The middle row is the point. Absence is the failure mode that no comparison catches — you cannot compare against something you did not think of — which is why it needs a prompt rather than a procedure.

15.2.3 Part 3 — the checkable part (5 minutes)

Take a piece of generated material you might actually publish or hand in: a figure with a caption, a page with headings, a slide, a short document.

Run the checks that have definite answers:

- ☐ **Alt text** on every image. Does it say what the image conveys *for the purpose you are using it for*, not just what is in it?
- ☐ **Headings in order**, with no level jumps.
- ☐ **Contrast** meets the standard. Use a contrast checker; it takes seconds.
- ☐ **Link text** that means something out of context.
- ☐ **No meaning carried by colour alone.**

Record what failed. Then rewrite one piece of alt text yourself and note the difference between the two versions — the generated one will usually describe the contents, and yours will describe the function.

15.2.4 Now take the checkpoints

`cp15-what-a-check-shows` is about Part 1's second sentence. **`cp15-alt-text`** is about Part 3.

```
python3 selfcheck.py run cp15-what-a-check-shows
python3 selfcheck.py run cp15-alt-text  # or in the browser
```

Run the Python route from `course/lab/`, or use the browser self-check.

15.2.5 Optional extensions

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

Test the thin half (about 6 minutes). Ask for something specific about a smaller place, a smaller language, or a less-documented practice that you happen to know well. Then check it. This is the half of the unit that gets less attention and it is often where the clearest errors are — and you are one of the few people positioned to catch them, which is itself the finding.

Read this module's own accessibility output (about 4 minutes). Run `make check-accessibility` and read what it prints, including the list at the end of things it cannot decide. Compare that list with the checks you ran in Part 3. A green result from a machine and a conformance claim are different things, and the script says so itself.

15.2.6 Activity Answers

15.3 Unit 15 answers and guidance

Read this after attempting the activity.

15.3.1 Part 1 — the two sentences

What it established. A good answer is narrow, specific and counted: "changing the name from one associated with one country to one associated with another produced a different described occupation in three runs out of three, and added a sentence about family circumstances in two runs out of three that appeared in none of the others". Frequencies, on that system, on that day.

What it did not establish. This is where most answers overreach, in one of three ways:

Overreach	Why it is wrong
"This shows the system is biased"	A general claim from six observations on one system on one day. You have a pattern, not a rate.
"This shows it treats X worse"	"Worse" is a judgement the comparison does not make. You observed a difference; whether it is a harm depends on the use.
"The difference was small, so there is no problem"	Sampling a little and finding little is weak evidence for absence, and you have not looked at what is missing from both.
"The attribute caused this"	Three pairs can show that a difference persists. Persistence is not a mechanism, and you did not look inside anything.

The honest version sounds thinner than any of those, and it is the only one you can defend: *across three pairs, these features appeared this often on each side, on this system, on this date.*

If a feature appeared once on one side and not the other, that is the run-to-run variation the repetition exists to expose, and the answer is that you found nothing about the attribute — which is a real result and not a failed exercise.

If your outputs were near-identical throughout, the same discipline applies in the other direction. "No difference across three pairs" is a result; "there is no bias here" is not something six runs can support.

15.3.2 Part 2 — the middle row

The question is deliberately about you, not the output.

The most common honest answer is no — and that is the finding. Absence is invisible by construction: nothing in an output flags what it left out, and nothing in your reading of it will either, unless you go looking. This is the same structure as the unstated assumption in Unit 2, where the gap is invisible to the person holding it.

If you answered yes, check whether the absence was of something you happen to know about. That is usually why absences become visible, and it is an argument for the thing this unit ends on: the people positioned to notice a particular absence are usually the people who know that area, which is a reason to ask rather than to check alone.

15.3.3 Part 3 — alt text is the tell

The pattern almost everyone finds: generated alt text describes **contents**, because contents are what is in the image. Useful alt text describes **function** — what the image is doing in your document, which the system cannot know because it depends on why you put it there.

A worked comparison:

> *Generated:* "A line graph with a blue line rising from left to right, with the x-axis labelled 2019 to 2024."
>
> *Yours:* "Participation rose steadily from 2019 to 2024, with no interruption during the years the scheme was suspended."

The first describes the picture. The second says what the reader is meant to take from it — and a reader using a screen reader needs the second, because they cannot supply it from the picture.

Note that this is not a criticism of the generated version. It is an accurate description. It is answering "what is in this image", which is the only question available to something that does not know why you included it.

If everything passed, check that you ran the contrast test with a tool rather than by eye. Contrast is the one that looks fine and fails, particularly for grey text on white, which is the most common failure on the web and the easiest to introduce without noticing.

15.3.4 The checkpoints

`cp15-what-a-check-shows` tests the scoping of a finding, which is the same skill as Unit 9's scoped statement and Unit 17's defence. The tempting wrong answers generalise from one comparison — in both the accusing and the exonerating direction.

`cp15-alt-text` tests the contents-versus-function distinction, which is the one practical thing in this unit that produces a definite right answer.

15.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp15-what-a-check-shows` — What one comparison establishes

Kind: choice.

You asked for the same short professional profile twice in separate fresh conversations, changing only the name. The two outputs assigned different seniority and one added unrequested detail about family circumstances. What can you defensibly say?

Options, numbered from zero:

0. That in this instance, changing that one attribute changed these specific things – and that this is one comparison, not a measurement of how often it happens or of what happens elsewhere.
1. That the system is biased with respect to names of that kind.
2. That the system treats one group worse than another.
3. Nothing, since two runs prove nothing and the variation could be sampling.

Answer: `0`

Hint: You changed one thing and observed a difference, twice. Ask what that licenses you to say about this instance, and what it would take to say anything about how often it happens.

Diagnostics, by the answer given:

- 1 — A general claim from two observations. You have found a real difference in one instance, which is worth recording and worth pursuing – but ‘is biased’ is a claim about a rate, and a rate needs many trials. The finding is not nothing; it is narrower than the sentence.
- 2 — ‘Worse’ is a judgement your comparison did not make. You observed that the outputs differed in seniority and in an unrequested addition. Whether that constitutes a harm depends on what the output is for, which is a separate question and often a more serious one.
- 3 — This overcorrects into saying nothing, and it is wrong about what you have. You controlled the input and varied exactly one attribute, so the difference you observed is attributable to that attribute on this occasion. That is a real, if narrow, finding – and dismissing it because it is not a measurement discards the evidence you actually have.

Explanation: One controlled comparison establishes something narrow and real: this attribute changed, and these specific things changed with it, this time. It does not establish a rate, does not establish harm, and does not establish that no difference exists elsewhere. The discipline is the same as the scoped statement in Unit 9 and the defence in Unit 17 – say what you did, say what you found, and stop. Both the accusing and the exonerating generalisations go beyond the evidence, and the exonerating one is the easier to make without noticing.

Checkpoint `cp15-alt-text` — Alt text describes function

Kind: choice.

You are including a chart in a report to make the point that participation rose without interruption. Which alt text does the job?

Options, numbered from zero:

0. Participation rose steadily from 2019 to 2024, with no interruption during the years the scheme was suspended.
1. A line graph with a blue line rising from left to right, with the x-axis labelled 2019 to 2024 and the y-axis labelled participation.
2. Chart showing participation data over time.
3. See the chart below for participation figures.

Answer: 0

Hint: Ask what a reader who cannot see the chart needs in order to follow your argument. It is not what the chart looks like.

Diagnostics, by the answer given:

- 1 — An accurate description of the picture, and the one a system will usually produce – because contents are the only thing available to something that does not know why you included the image. A reader using a screen reader now knows there is a rising blue line and still does not know the point you were making with it.
- 2 — Too thin to do any work. It tells a reader that a chart exists and nothing about what it shows, which leaves them strictly worse off than a sighted reader rather than equivalently informed.
- 3 — This directs a reader to the very thing they cannot see. It is the alt-text equivalent of ‘click here’ – a pointer where the content should be.

Explanation: Alt text carries the function of an image, not its contents: what a reader is meant to take from it, given why it is there. That is precisely the part a generative system cannot supply, because it depends on your purpose rather than on the picture – which makes generated alt text a reasonable starting point and never a finished one. It is also the accessibility check most often skipped, and unlike most of this unit it has a definite right answer that you can produce in a sentence.

16 Unit 16 — The historical bug and historical honesty

16.1 Reading

Media for this unit:

- **Animation**: *Two indexings, one term* — this unit has an interactive version on the course website (`ada-indexing.html`); the still below is one moment from it.

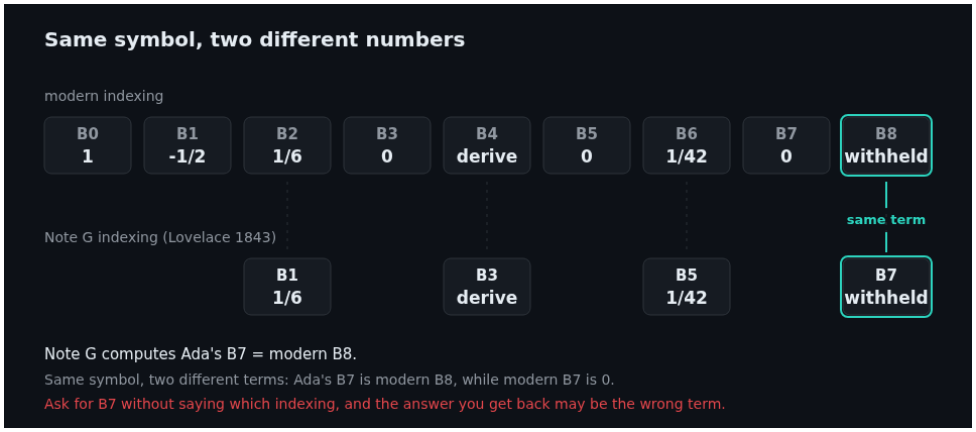

Figure 18: Two indexings, one term. Modern Bernoulli indexing and Note G's indexing start aligned and slide out of alignment, until Ada's B7 sits under the modern B8.

- **Animation**: *Three Buckets* — this unit has an interactive version on the course website (`three-buckets.html`); the still below is one moment from it.

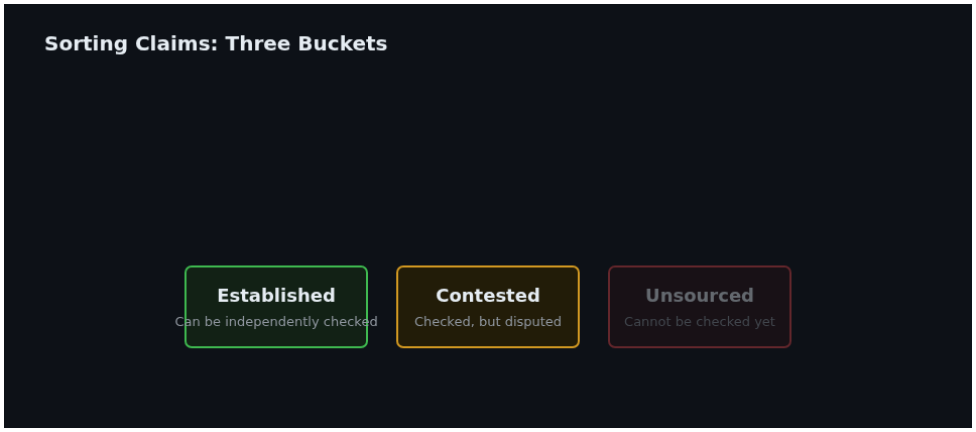

Figure 19: Three Buckets. Three claims are sequentially sorted into Established, Contested, and Unsourced buckets.

> **A required unit, and the one where this module checks itself.** It works through a published error in Note G's table, the module's own mistaken account of that error, and the difference between a claim that is established and one that is merely repeated. The lesson — that a claim's *status* is part of the claim — is also recorded in `course/misconceptions.md` entry 12, "Published means true"; this unit is where you do it rather than read it.

Reading time: about 11 minutes. **Activity:** about 12 minutes. **Checkpoint:** `cp16-note-g`. **Before this unit:** Unit 6 (the Bernoulli numbers and Ada's indexing) and Unit 8 (the verification lab).

16.1.1 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.

16.1.2 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
B5	B6
B7	B8

You met this mapping in Unit 6. The values are tabulated in `course/historical-notes.md`, deliberately not repeated here, because working the value out for yourself is the checkpoint.

16.1.3 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.

16.1.4 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_5 and V_4 in that order, while producing its stated result `(2n−1)/(2n+1)` requires dividing V_4 by V_5. 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.

16.1.5 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, and the clearest example is this module’s own. Across the web, the Note G table “as printed” is said to compute −25621/630. This module recomputed it, got 139/630, and published that as a refutation. Both figures are reproducible; neither claim was complete. Note G’s table carries more than one printed fault: operation 4’s inverted division, and operation 24, which as printed omits the sign inversion the computation needs, and it consumes Bernoulli numbers that may be supplied or generated by the routine itself. Those choices give at least four different answers, set out in `course/historical-notes.md` §4. 139/630 is what you get having repaired operation 24 without saying so; −25621/630 is what you get when the faulty routine feeds its own output back in. Neither the web accounts nor this module had stated which object was being executed.

So the failure is not that someone repeated a wrong number. It is that a confident, checkable-looking claim was made without saying what it was a claim about — inside finished, internally consistent, already-vouched-for accounts, this module’s included. The fix is not more care. It is stating the interpretation,

then recomputing under it, and treating a bare number with no stated interpretation as unfinished rather than as evidence.

16.1.6 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.

16.1.7 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, where they differ;
- surviving manuscript or correspondence material from the period;
- a published study that states which copy it worked from.

That last group is *unsourced*: what manuscript material exists, and where it is held, is recorded as an open question in `course/historical-notes.md`.

16.1.8 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.

16.1.9 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 18 teaches from; nothing on this page, in the activity, or in `cp16-note-g` depends on it, and Unit 18 in turn does not assume you have read this one.

16.1.10 Checkpoint

`cp16-note-g` asks you to state the value Note G’s example should produce, from the oracle and the Unit 6 mapping, before comparing it with anything historical. Work through the activity first, then:

```
python3 selfcheck.py run cp16-note-g  # or in the browser
```

Both routes ask the same question and accept the same answers. Neither is a fallback for the other.

16.1.11 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 17 — turning “I checked this” into a written defence someone else can read. It does not require this unit.

16.2 Unit 16 activity — check the table, then sort the claim

Time: about 12 minutes. **Checkpoint:** `cp16-note-g`. **You will need:** the index mapping from Unit 6, and either the table you built in Unit 6 or the lab oracle from Unit 8. Either is sufficient.

A required unit, and the one where the module examines its own mistake. The first task is the checkpoint; the second is the one worth protecting if you are short of time.

Two tasks. The first takes about four minutes; the second takes about eight.

16.2.1 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.

16.2.1.1 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 6. Write the modern index down before going on.
2. Look that modern index up in the Bernoulli table you built in Unit 6, 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.

16.2.1.2 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.

16.2.1.3 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.

16.2.1.4 Now run the checkpoint

```
python3 selfcheck.py run cp16-note-g  # or in the browser
```

Both routes ask the same question, accept the same forms of the answer, and give the same diagnostics. Pick whichever you have to hand.

16.2.2 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.

16.2.2.1 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.

16.2.2.2 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 17 — 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".

16.2.3 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.

16.2.4 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
- `cp16-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 17, as an appendix marked optional. Unit 17 does not require them.

16.2.5 Activity Answers

16.3 Unit 16 answers and guidance

Read this after attempting the activity.

This key does not restate the value asked for at `cp16-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`.

16.3.1 Task 1 — the three written questions

16.3.1.1 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 `cp08-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.

16.3.1.2 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 you connected those two things unprompted, you have understood the unit.

16.3.1.3 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.

16.3.2 Task 2 — the seven statements

The bucket assignments below are this module's positions, stated as positions. If you placed a statement differently but gave a defensible reason and a concrete "what would move it", you have done the exercise correctly. The reasoning is the part that counts, not the label.

16.3.2.1 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.

16.3.2.2 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. Marking this "contested" confuses it with statement C, which is the confusion the whole unit is built around — a productive mistake, and one worth sitting with 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.

16.3.2.3 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_5 and V_4 in that order, where producing the stated `(2n−1)/(2n+1)` needs $V_4 \div V_5$. 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. **If you marked this "contested", you gave the answer that was correct on the older evidence; section 3 is where the evidence changed.** The reasoning counts either way, and the tell is whether it rests on the facsimile check or on the claim's popularity. What stays contested is *who* introduced it — items D and E — because no facsimile can show that.

16.3.2.4 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`).

16.3.2.5 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 you noticed 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.

16.3.2.6 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. "Contested" is a reasonable answer here; "established" is not, 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.

16.3.2.7 G. The as-printed table computes −25621/630 Unsourced as stated — and it stays unsourced after checking, because the statement does not say what it is about.

Before any check, this is a textbook bucket-two-or-three claim: the figure is everywhere online and cited to no scholarly source, and repetition is not corroboration. Unlike D and E it looks checkable from your desk, and this module checked it — and got the check wrong in an instructive way.

Note G's table has more than one printed fault, and it consumes earlier Bernoulli numbers that may be supplied or generated. Running `python3 noteg.py` in `course/lab/` prints what each reading produces: `−1/30` intended, `139/630` with operation 4 as printed and operation 24 repaired, `−139/630` with both faults literal, and `−25621/630` when the faulty routine generates the earlier values it then consumes. The last of those *is* the repeated figure, reproduced exactly.

So the honest verdict is not "refuted". It is that the phrase "the as-printed table" does not pick out one object, so the claim cannot be true or false until its interpretation is stated. An earlier version of this module marked it refuted on the strength of a run that had silently repaired operation 24 — the same failure, committed by the people warning about it.

Filing this under "established" because the number is widely repeated is the misfiling the unit warns about. Filing it under "refuted" because a recomputation disagreed is the subtler one.

What would move it: stating the interpretation. "Operation 4 as printed, operation 24 repaired, earlier values supplied" is a claim that can be checked, and checks out at 139/630. "The routine run from the start with the operation-4 inversion in place" checks out at −25621/630. A high-resolution facsimile confirming operation 24 independently of [S7] would firm up the whole family.

16.3.3 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.

16.3.4 Why this unit is written the way it is

- It is deliberately short and deliberately under-coloured. If it feels thin next to the rest of the module, that is the design: this is the unit where invented historical detail is most likely to creep in, from any author, human or otherwise. Restraint here is the same discipline the unit is teaching.

- The question worth carrying out of it is not “which bucket” but **“what would move it”**. That one generalises to essentially every claim you will meet in the rest of your degree.
- If you found the sorting easy, statement F is the one to sit with. It is quantitative, memorable, entirely uncontroversial in tone, and unsupported — which describes a large fraction of confident writing.
- Everything historical here defers to `course/historical-notes.md`. If the evidence changes, that file changes first; this unit deliberately holds no independent historical position that could drift out of step with it.

16.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp16-note-g` — Comparing against the historical table

Kind: fraction.

Note G’s worked example computes Lovelace’s B7. Using the modern oracle and the index mapping from Unit 6, what exact value should that be? Enter it as a fraction.

Answer: `-1/30`

Hint: Two steps, in this order. First: which modern index does Ada’s B7 correspond to? (You answered this in cp06-ada-index.) Second: what is the oracle’s value at that index? You can run ‘python3 selfcheck.py explore’ to look it up.

Diagnostics, by the answer given:

- `-1/42` — Close to the right region of the table but one term off. Map her B7 to its modern index first, then look up that value – do not map and look up in one step.
- `1/30` — The magnitude is right and the sign is not. Check the modern value at the index you mapped to; the even Bernoulli numbers alternate in sign.
- `1/42` — That is the modern B6, which is Lovelace’s B5 – one label too early. Recheck the mapping for B7 specifically.

Explanation: Being able to state the correct value independently is what makes the historical table checkable at all. Note G’s published table does contain printed errors – see course/historical-notes.md for what is established, what is contested, and why ‘what the faulty table computes’ has more than one answer. Verification is what lets you say that carefully instead of repeating a story.

17 Unit 17 — Disclosure, defence, and responsible use

17.1 Reading

Media for this unit:

- **Animation**: *Verification Evidence & Disclosure Pipeline* — this unit has an interactive version on the course website (`evidence-pack.html`); the still below is one moment from it.

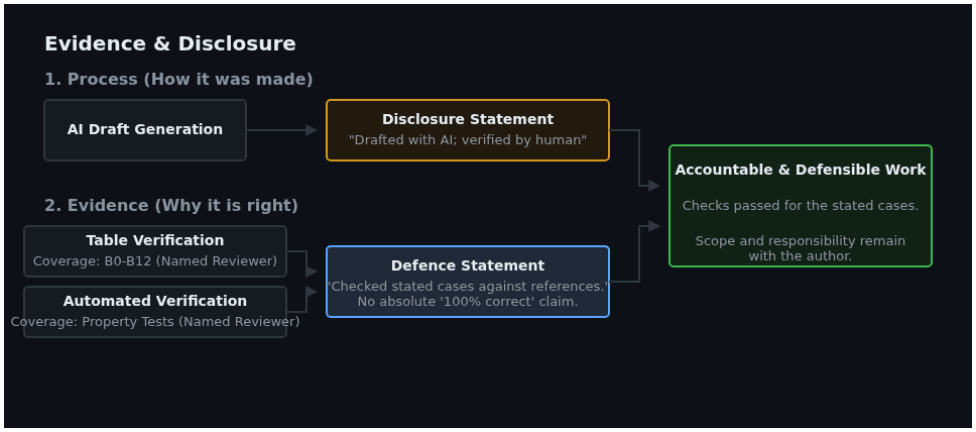

Figure 20: Verification Evidence & Disclosure Pipeline. Distinguishes provenance disclosure from warrant defence, showing equal-status verification branches converging on accountable work.

Reading time: about 19 minutes. **Activity:** about 14 minutes. **Checkpoints:** `cp17-disclosure` and `cp17-scope-the-claim`. **Before this unit:** Unit 10 (re-prompting and specification). Unit 16 (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 18 closes the module afterwards by naming the general question all of this has been circling.

17.1.1 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.

17.1.1.1 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.

17.1.1.2 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 a faulty reading of Note G's table well enough to predict what it computes — `139/630` — and then reported that number as what "the table as printed" produces, which it is not: that run had silently repaired a second printed fault (`course/historical-notes.md` §4). A good map, of an object we had not said we were mapping. Understanding without a correctness check is not yet a defence; and a defence that does not say what it is a claim *about* is not one either. 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 at the level your claim needs, 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.

17.1.2 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.

17.1.2.1 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 / explain>
<which part of the work> at the <stage> stage.

The output was <not used verbatim / revised after verification / used verbatim
after checking>. Specifically: <what you changed and why>.

All <values / claims / code> were verified by <table-based comparison against a
reference I derived / running the module's checker / consulting the cited
source>, and I am responsible for the final content.

<Tool and version, when required or materially useful.>
```

17.1.2.2 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 link/hash to the retained prompt].
- **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 above
    `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 change.
```

17.1.2.3 Worked academic example

```
AI-use disclosure
-----------------
AI assistance was used to draft the initial version of the bernoulli() routine
in section 3, at the first-draft stage.

The output was revised after verification: the first version used the second
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 every n from 0
to 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.

17.1.3 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 6.

17.1.3.1 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.

17.1.3.2 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>, which I
resolved by <what you did>.
```

17.1.3.3 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 convention
(B1 = −1/2). I derived B4 from the recurrence by hand and took the remaining
values from the module's published table, then compared the generated
table against them term by term.

The discrepancy I found was at B1, where the generated version gave +1/2; I
traced this to the second convention and corrected the specification.
```

Automated verification:

```
The routine agreed with the oracle under the interface contract in Unit 10 for
every n from 0 to 30, returned exact Fraction values on those inputs, and
satisfied the odd-term property over that range. That is what the evidence
supports: it is not a claim about n > 30.

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.

17.1.4 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 7
2	Generated output or code	Unit 7
3	Verification result	Unit 8
4	Discrepancy notes	Unit 8
5	Revised prompt	Unit 10
6	Final result	Unit 10
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.

17.1.5 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.

Unit 13 teaches this boundary rather than only signposting it: what leaves your device, what is retained, and how to work with material you are not free to share. Sustainability is signposted, and the resource section in Unit 4 is as far as this module takes it.

17.1.5.1 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. If you are unsure, ask whoever set the work, and if you are studying this on your own, with no assessed work attached, the question does not arise until it does.

17.1.5.2 What a disclosure can and cannot do One more thing about disclosure, and it is the sort of thing this module would be inconsistent to leave out.

A disclosure cannot be verified from the work. Nobody can establish, by inspecting a finished piece, that the disclosure attached to it is complete or that its absence is a lie. Text does not carry a reliable mark of how it was made. The tools sold as detectors do not settle it either: they return likelihoods over surface features, they are wrong in both directions, and being wrong in the direction of accusing someone is a serious thing to be wrong about.

Follow that where it goes. A disclosure is not a control. It records something that no inspection can confirm, so nothing should be designed as though it could — and a rule that depends on detecting undisclosed use is asking for evidence that does not exist. If a piece of work genuinely has to be produced without AI assistance, that condition has to be created by the setting it is produced in, not asserted by a rule about the artefact afterwards.

So what does carry the weight? Two things, and both can actually be established:

- **The work.** Whether a claim is correct is checkable, by the methods this whole module is about. That does not depend on how the work was made, which is why the check is worth more than the declaration.
- **Who is answerable.** If you put your name to something, you are answerable for it being right, and that holds whether a model drafted it, a colleague drafted it, or you wrote every word. Assistance does not move responsibility, and no disclosure discharges it.

None of which makes disclosure optional. You still owe a reader an honest account of how work was made, and in many settings you are required to give one. The point is narrower and worth being exact about: **disclosure is a duty you discharge, not a fact anyone can audit.** Treat it as the first, easy half, and put your effort into the half that can be checked.

17.1.6 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 your own submission, you remain 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 your responsibility, and the module does not claim that nobody else has duties.

17.1.7 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, from Unit 16: 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.

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. Use Unit 16's three claim-status buckets when deciding what still needs checking.

17.1.8 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 18 interprets that distinction through the second-order material set out in `course/second-order.md`; neither this page, the evidence pack, nor `cp17-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.

17.1.9 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 cp17-disclosure  # or in the browser
```

Both routes ask the same question and give the same diagnostics.

17.2 Unit 17 activity — assemble the pack, write the defence

Time: about 14 minutes. **Checkpoints:** `cp17-disclosure` and `cp17-scope-the-claim`. **You will need:** your prompts and outputs from Units 7 and 10, your verification results from Unit 8, your claim sorting from Unit 16, and the eleventh row you wrote in the Unit 3 activity — the real piece of your own work you said you would or would not check.

Four short tasks, then the checkpoints. The optional CS extension at the end sits outside the core time budget.

17.2.1 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.

That file opens with a table of everything the pack holds and which unit each part came from, generated from `course/evidence-pack.yml` so that it cannot drift from what the units actually ask you to keep. Work down it rather than from memory.

Two things go in before item 1, and they are easy to forget because you wrote them before the pack existed: the **verification sentence** from Unit 0, and the **task classification table** from Unit 1. Unit 18 asks you to look back at the first of them.

Which unit item 3 and item 4 come from depends on your route. On the table-based and code routes they come from Unit 8, the Bernoulli comparison. On the route without the mathematics they come from Unit 9, the citation comparison — the same two items, established the same way, against sources rather than against a table.

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.

17.2.2 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.

17.2.3 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.

17.2.3.1 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.

17.2.4 Task 4 — turn it on your own work (about 4 minutes)

Everything so far has been about Bernoulli numbers, which is a case chosen so that “correct” is decidable. Your own work is not usually like that, and a method you can only run on the tidy case is not a method you have.

Go back to the eleventh row of your Unit 3 triage: the thing you personally use an AI system for, or are about to. If you wrote “I would not have checked that”, this is where that becomes useful rather than embarrassing.

Write five short lines about **one** artefact from your own subject or work — one you have generated, or one you are about to:

1. **What it is**, in a sentence, and what it would be used for.
2. **Which kind of claim it makes.** Exact and checkable, source-dependent, a judgement, or a low-risk draft you will rewrite anyway. More than one may apply; privacy and consequence override “low risk”.
3. **What could fail, and how you would notice.** Not “it might be wrong” — the specific failure, and its symptom.

4. **The check you would actually run**, with the reference it would use, and how long it would take. If no independent reference exists, say that: it is a finding, not a gap in your answer.
5. **What you could then honestly claim**, and what would remain unchecked.

Add it to your pack as item 9. Nobody collects it. The point is that the method has now been run twice: once where the answer was decidable, and once where you had to decide what "checked" would even mean.

If you have someone to show it to — a tutor, a colleague, a study partner — a pack is a much better thing to discuss than a finished piece of work, because it shows what you did rather than what you produced.

17.2.5 Task 5 — the checkpoints

```
python3 selfcheck.py run --unit 17  # or in the browser
```

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.

17.2.6 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.

17.2.7 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 8 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.

17.2.8 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 18 closes the module by naming the general question behind it, and your pack is what you take into it.

17.2.9 Activity Answers

17.3 Unit 17 answers and guidance

Read this after attempting the activity.

`cp17-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`.

17.3.1 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.

17.3.2 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. 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.

17.3.3 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 10 interface contract, verified by running the checker over every n from 0 to 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 B4 as I derived it from the recurrence by hand and the rest as published in the module's reference table, 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 instinct treats the second as the weaker piece of work, that is the scheme failing, not the work. 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."

Your own confidence is not the evidence either. Ask yourself what "thoroughly" consisted of: what you compared, observed, sourced or reproduced. The answer is often good and simply was not written down.

17.3.4 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 you pasted the record in place of item 8, the tool has already said so.

17.3.5 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.

17.3.6 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. Reporting only step 3 produces 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, which is worth noticing when it happens to you.
- **Reported defects that are not defects** are as informative as missed ones, and they are easy to under-record because they feel like non-events. Record them anyway.

Step 5 is where this lands: you add tests you did not previously have, because the exercise revealed which parts of your own program were never really being checked. If that happens, the extension has done its job regardless of how the AI performed.

17.3.7 What this unit is not

- **It states no assessment policy.** The rules for AI use in assessed work are set by your institution or your course, not by this module. Nothing here overrides them, and nothing here is a substitute for reading them. If you are unsure what your rules are, that is the question to take away from this unit.
- **Your evidence pack is private and is not collected.** `course/learner-evidence-template.md` carries the completion criteria and you check your own pack against them. It is not submitted, not marked and not reviewed. It evidences a verification habit, not programming ability — so a pack from the table route and a pack from the Python route are read against the same four defence criteria. The routes differ in medium, not in standard.
- **The privacy paragraph is a boundary, not a lesson.** It marks where the question starts; it does not answer it.
- **The Ada section is left unresolved on purpose.** It is the place where a reader's own view most easily hardens into a settled position. It should read as open, because it is.

17.3.8 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp17-disclosure` — What a defence has to contain

Kind: multi.

You submit work containing AI-generated code. Which of these belong in your correctness defence? Select all that apply.

Options, numbered from zero:

0. The specific checks you ran and what they showed.
1. Which convention or assumption you fixed, and why.
2. The name and version of the AI tool you used.
3. That the model stated the answer was correct.

Answer: `0, 1`

Hint: A defence answers 'why is this right'. Ask of each item whether it is evidence about the work, or a statement about how the work was made, or about how you feel about it.

Diagnostics, by the answer given:

- `0,1,2` — The tool name belongs in your *disclosure* – it says how the work was made. Your *defence* says why the work is right, and the identity of the tool is no evidence of that. Keep the two statements separate; Unit 17 gives a template for each.
- `0,1,2,3` — Two of these are evidence and two are not. A tool name describes provenance; ordinary model confidence wording has not been calibrated for this task. Neither tells a reader why this output is correct.
- `0` — Test results are the backbone, but they are only meaningful against a stated specification. If you never say which convention you chose, a reader cannot tell whether the passing tests tested the right thing.

Explanation: Disclosure and defence answer different questions: 'how was this made' and 'why is this right'. An AI system may help draft either statement, but you must gather the evidence, check the wording, and remain able to defend it.

Checkpoint `cp17-scope-the-claim` — Say what the evidence supports, and no more

Kind: choice.

Your routine agreed with the oracle for every index from 0 to 30, returned exact Fraction values on those inputs, and satisfied the odd-term property over that range. You fixed the first Bernoulli convention (B1 = -1/2) in the prompt. Which sentence should go in your defence?

Options, numbered from zero:

0. The routine is correct: it passed every test I ran.
1. The routine agrees with the oracle for n = 0 to 30 under the first convention (B1 = -1/2), returning exact Fractions and zero odd terms above B1. It is not tested beyond n = 30.
2. The routine was verified using an independent oracle and is suitable for use.
3. The routine agrees with the oracle for n = 0 to 30 under the first convention, and I inspected the source to confirm it uses no floating-point arithmetic internally.

Answer: `1`

Hint: Two things make a defence checkable: the scope of the evidence, and the specification it was checked against. Which option states both, and claims nothing the checker could not have shown?

Diagnostics, by the answer given:

- `0` — Passing every test you ran is not correctness – it is agreement over the inputs you tried. This is the overclaim the unit exists to prevent: it states a general property from finite evidence, and it names neither the range nor the convention, so a reader cannot tell what was actually established.
- `2` — Nothing here is false, and nothing here is checkable either. 'Verified using an independent oracle' does not say over which inputs, under which convention, or what the check showed, and 'suitable for use' is a judgement the evidence does not reach. A defence a sceptical reader cannot test is not a defence.
- `3` — The first half is right and the second half claims evidence you do not have. The checker reads returned values; it cannot see inside the routine, so it establishes exact Fraction *output*, not the absence of floating-point arithmetic in the computation. Unit 10 makes that distinction under clause C6.

Explanation: That is the shape of a defensible claim: what was compared, over which inputs, under which convention, with what result, and what remains untested. It is a smaller claim than 'it is correct', and it is one you can stand behind – which is the whole trade this module has been making. Notice it also recalls two things from earlier units: the convention distinction from Unit 6, and the range the checker actually covers from Unit 8.

18 Unit 18 — Alles Lüge! First- and second-order reasoning

18.1 Reading

Media for this unit:

- **Animation**: *Accountability Handover* — this unit has an interactive version on the course website (`accountability-handover.html`); the still below is one moment from it.

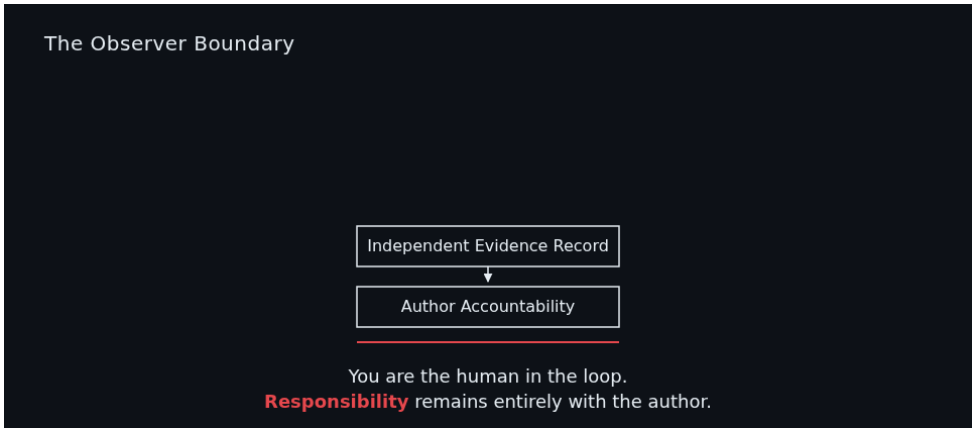

Figure 21: Accountability Handover. The system generation loop fades out, leaving the evidence record and author's accountability in sharp focus.

- **Animation**: *Formal Systems Leap* — this unit has an interactive version on the course website (`formal-systems-leap.html`); the still below is one moment from it.

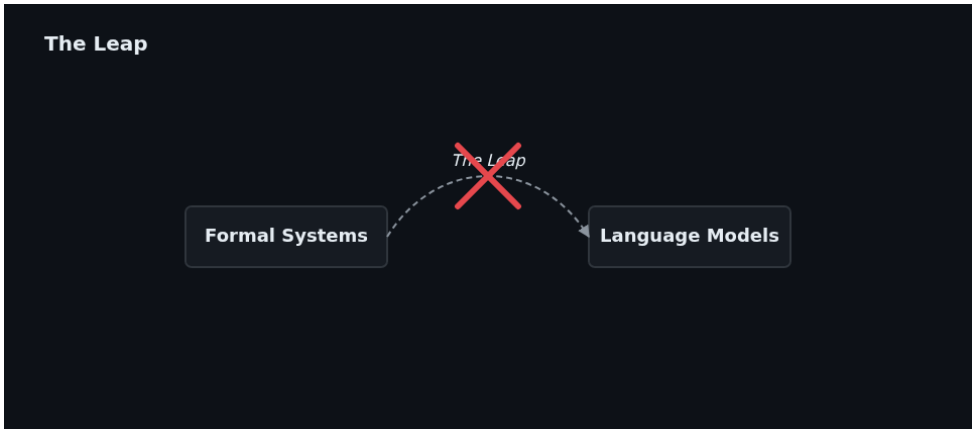

Figure 22: Formal Systems Leap. Displays Formal Systems and Language Models. A dashed leap connects them, which is then firmly crossed out.

- **Animation**: *Lyrics Context* — this unit has an interactive version on the course website (`lyrics-context.html`); the still below is one moment from 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 15 minutes. **Activity:** about 12 minutes. **Checkpoint:** `cp18-sort-the-claims`. **Before this unit:** Unit 17 (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.

Figure 23: Lyrics Context. Three songs as three positions on doubt: despair that everything is a lie, the realisation that no view from outside is available, and the practical acceptance that follows.

18.1.1 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` carries what this page leaves out — the structural account of understanding, and the constraints the unit is written under.

18.1.2 The title, and the movement behind it

The unit is called *Alles Lüge!* — "all lies!". That is the **first step of a movement, not its conclusion**, and the page walks all three steps.

1. **Everything sounds like a lie.** If every record was made by some observer, from inside the system, nothing arrives already certified, and the short way to read that is: trust none of it. Lacrimosa's line, quoted in section 5, is that position at full volume [Q1].
2. **The standpoint above is gone.** Harder, and more accurate. The trouble is not that everyone is lying; it is that there is no outside position from which a claim could be settled once and for all. "Gott hat sich erschossen / ein Dachgeschoss wird ausgebaut" — *God has shot himself; a loft is being converted* (Einstürzende Neubauten, "Haus der Lüge") [Q3]. Both halves of that matter: no architect, and the building work carrying on regardless. Sections 2 and 4 are the same step in published form — the observer is inside the system it describes, and a formal system cannot define a truth predicate for its own sentences.
3. **What is left is a method, and you have used it since Unit 8.** Warrant does not descend from above; it is assembled locally. Section 5 is that step.

Despair, then the missing god's-eye view, then acceptance — practical rather than consoling. If you take one step of the three from the title, take the third.

18.1.3 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.

18.1.4 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].

An observer is a thing with a reach and a cost, not a viewpoint. Drawing by the module.

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.

18.1.4.1 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.

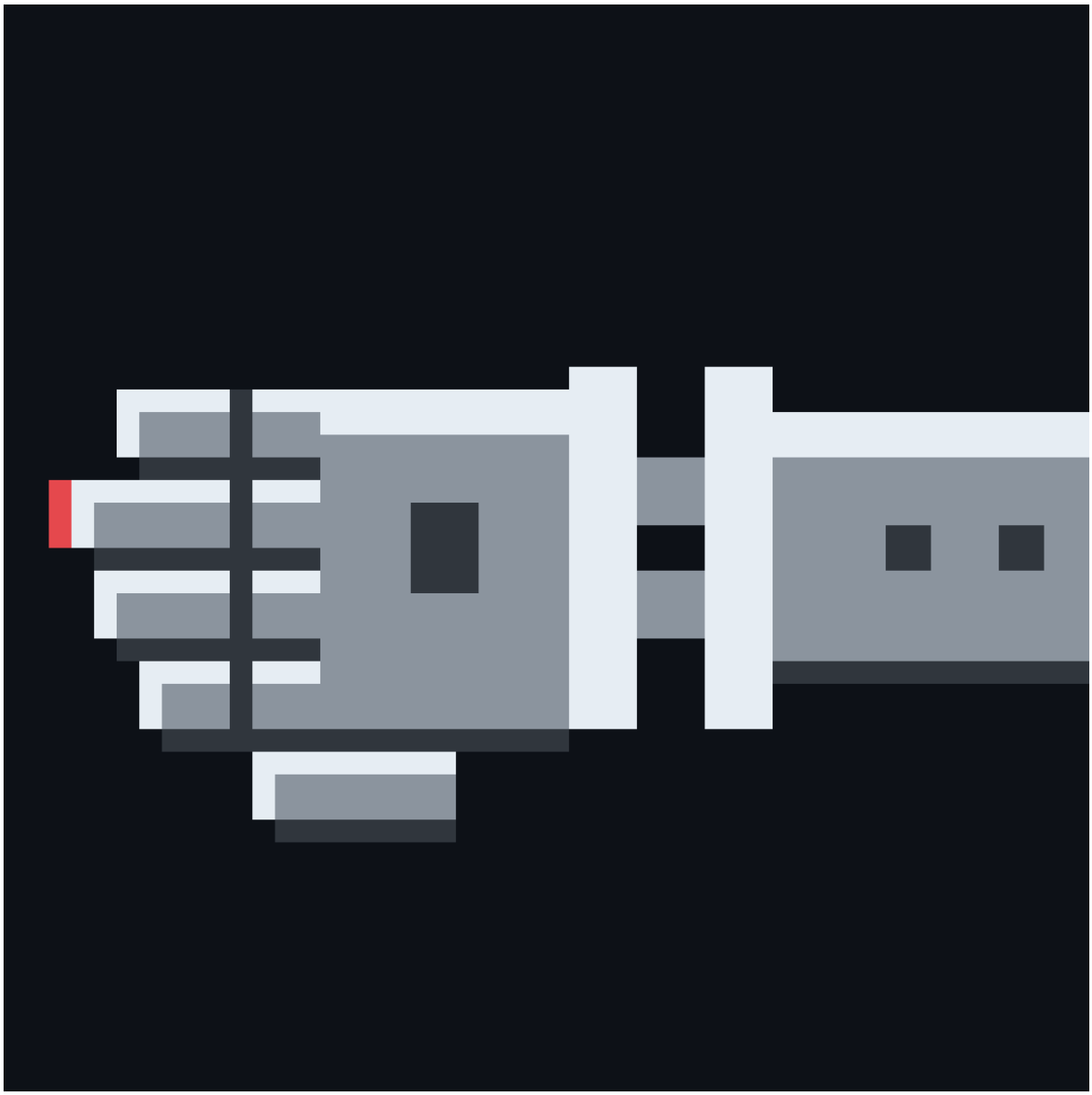

Figure 24: A mechanical hand and forearm, drawn in pixel art, reaching into an otherwise empty dark field.

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.

18.1.5 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 operation-4 inversion is established; its attribution is contested; and "what the faulty table computes" has as many answers as there are readings of it.	Saying which part is which — and which object a claim is about — 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 one faulty reading of Note G's table — while misnaming which reading it was. 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.

18.1.6 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 first incompleteness theorem** (1931): any formal system that is **consistent**, **effectively axiomatised** — its axioms can be listed by a procedure — and strong enough to express elementary arithmetic contains a sentence it can neither prove nor refute [C6].
- **Tarski's undefinability theorem** (1936): for such a system, arithmetical truth for its own language is not definable **within that same language** [C7].

Those qualifications are not decoration, and this unit would be doing the thing it warns against if it dropped them. "Consistent" and "effectively axiomatised" are load-bearing: remove the second and the theorem is false, because the set of all true arithmetical sentences is consistent and complete and simply cannot be listed by any procedure. The popular one-line versions omit exactly these conditions, which is what makes them so easy to carry somewhere they do not go.

The usual gloss adds that the unprovable sentence is *true*. That is a further step: it holds under the intended reading of the symbols, and saying so means saying which interpretation you mean. Provability is a property of the system; truth is a property relative to a model.

18.1.6.1 What they do not say This subsection matters more than the two lines above it, so read it slowly.

Both are mathematical results about **formal systems of a specific kind**: abstract entities with explicit symbols and deductive rules. A language model, however, is not a formal theory; it is a **physical system** — software running on hardware — studied by physics, engineering and experiment.

Applying a theorem about a formal, mathematical system directly to a physical, empirical one is a category error. They do **not** straightforwardly transfer to language models, to institutions, or to people, and this unit does not claim they do. Their role here is to establish that limits on self-reference are real and provable *somewhere* in logic — much weaker, and much safer, than asserting a mathematical limit on a particular piece of physical AI software.

A claim that Gödel settles something about machine intelligence is missing its middle. Not because software and formal systems are unrelated — a computer can implement a formal system, and results about formal systems can bear on computation, which is how computability theory works at all. The gap is specific: to carry the theorem across you have to say which component you are treating as a formal theory, show that the theorem's conditions actually hold of it, and argue that what follows for the formal object also follows for the running system. Each of those is a real step and each can fail.

So the fault is not the analogy. It is asserting the conclusion without the argument, and the conclusion may well be true for other reasons. Noticing that missing middle — in a paragraph that is fluent, confident, and cites a real theorem correctly — is the skill this module teaches.

18.1.7 5. What this does *not* license

Three overclaims are easy to reach from here. This unit closes each one off.

18.1.7.1 "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 8 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 method rather than exempted from it: AI-assisted, then held to external checks before it reached you, with its AI-use disclosure recorded once at module level as the honest record of how, which is the difference between trusting it and taking it on faith.

Figure 25: A face seen through shattered glass: every fragment shows a piece of it, and no fragment shows the whole.

Step one, and the position this section refuses. Illustration, AI-generated.

"Und doch ich hör' nur Lügen" — *and yet I hear only lies* — (Lacrimosa, "Alles Lüge") [Q1]. That is step one of the movement above, and it 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.

18.1.7.2 "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.

18.1.7.3 "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 7 trained you to catch.

18.1.7.4 What is left once the god's-eye view goes Step three, and where the movement the title opens actually ends. Look under the system for the bedrock and there is none: "It is as black as Malevich's Square" (Einstürzende Neubauten, "Sabrina") [Q2] — a square you can stare into, holding nothing. The work goes on anyway, in the same house, on the same floor.

Kazimir Malevich, Black Square, *1915, Tretyakov Gallery, Moscow (public domain).*

Warrant is built from below, out of the three details section 1 named — observer, context, record — plus the agreement of records produced independently of each other. That is what the oracle gave you in Unit 8, and what naming the Bernoulli convention did to an unresolvable disagreement. Less than certainty, and checkable: the trade this module has been making throughout.

"It's not the red of which we bleed" [Q2]. The red accent in these materials means divergence, never authority, 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.

18.1.8 6. Further context (optional)

Nothing in this section is required for the activity or checkpoint.

18.1.8.1 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. `course/second-order.md` holds the structural account and the design constraints.

18.1.8.2 The register, and the rule that governs it This is the only unit that uses the module's gothic register at full strength, including three 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 still records the date of the ghost-story challenge as contested — a range, not a night. So: connection supported, dating conventional rather than documented to the day, influence claim not made.

Figure 26: Kazimir Malevich's Black Square: a black square on a white ground, its paint cracked with age.

The three songs mark the three steps at the top of this page: [Q1] the despair, [Q3] the missing standpoint, [Q2] what is left. Take all three out and the argument is unchanged, which is the test they had to pass. Each quotation is short, attributed, and used for commentary; entries and rights position are in `course/references.md`.

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.

18.1.9 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.

18.1.10 Checkpoint

Work through the activity 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 cp18-sort-the-claims  # or in the browser
```

Both routes ask the same question and give the same diagnostics.

18.1.11 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 three song quotations, and the source policy.

18.2 Unit 18 activity — sort the claims, then sort your own

Time: about 12 minutes. **Checkpoint:** `cp18-sort-the-claims`. **You will need:** the student page for this unit, and your evidence pack from Unit 17.

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.

18.2.1 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 Unit 16 works through, and one you can apply here whether or not you did that unit first. 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.

18.2.1.1 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.

18.2.1.2 Write it up like this

```
Statement:        <letter>
Bucket:  proved theorem | established research programme | overclaim | none
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.

18.2.2 Task 2 — now sort a claim of your own

Open the evidence pack you assembled in Unit 17. You are going to turn the same method on your own work, which is the only version of it that costs you anything.

18.2.2.1 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.

18.2.2.2 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*
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.

18.2.2.3 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?**

18.2.2.4 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.

18.2.3 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 cp18-sort-the-claims  # or in the browser
```

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.

18.2.4 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 7.

18.2.5 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
- `cp18-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.

18.2.6 Activity Answers

18.3 Unit 18 answers and guidance

Read this after attempting the activity.

This key does not restate the answer to `cp18-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. `course/lab/checkpoints.py` is the canonical machine-readable source of checkpoint answers, and this key is not a second copy of one. Answers do appear elsewhere in the materials — the guidance edition of the handbook carries the checkpoint data and its explanations — so the point is that they are generated from that one source rather than restated by hand here. 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]*, *[Q2]* and *[Q3]* are the three songs. Fuller treatment of this unit's argument: `course/second-order.md`.

The part that counts is the reasoning, not the label. Placing a statement differently but naming a real citation, a real tradition or a real missing step is doing the exercise. Landing on the intended bucket and writing "it sounds like a theorem" is not.

18.3.1 Task 1 — the six statements

18.3.1.1 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. Marking it "established research programme" because it is unfamiliar confuses *unfamiliar* with *unsettled* — the mirror image of the error waiting in statement B, and worth naming as such.

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.

18.3.1.2 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 asking it is finding the seam.

18.3.1.3 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 Unit 16 works through, and named in the activity itself so that it is available whether or not you did that unit first.

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. Doing only the first still counts, because recognising that you are holding the wrong instrument is most of the skill.

The trap is marking 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.

18.3.1.4 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.

"The theorem is proved, the transfer is not, and the sentence hides the seam" is worth much more than "overclaim" with nothing behind it.

18.3.1.5 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 8. 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.

Describe behaviour, not intent. "The system reported X" is the right form; "the system believed X" is not, and slipping into the second is drifting away from what you can actually evidence.

18.3.1.6 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. Saying so, and then saying the remark is checkable against the theorem statements, understands it better than the bucket alone shows.

What must not happen is marking F an overclaim. F is the caution, not the leap. Reading the caution as itself an overreach inverts the section you most needed.

18.3.2 Task 2 — a claim of your own

18.3.2.1 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 17'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. Marking all three as stated almost certainly means you read your own sentence generously, which is the one reading a sceptical reader will not give it.

18.3.2.2 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 outcome is finding an **overclaim** in your 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 you by name, not dressed up as something more.

18.3.2.3 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. Concluding "so my defence is weak" misreads the exercise. The point is to be able to say which part is which.

18.3.3 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 you leave saying "so it cannot be trusted", the second overclaim has not been closed properly. Nothing here establishes a base rate for correct output. What it teaches is how 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.

18.3.4 Why this unit is written the way it is

- **It comes last, and nothing before it depends on it.** No checkpoint, activity or definition in Units 0, 2, 3, 6, 7, 8, 10, 16 and 17 needs this unit; those units carry a pointer to it and nothing more. You can work through the module without reaching here and lose none of the method. This unit is the part that asks what the method cannot reach — which is a different question, and one you are entitled to leave until you want it.
- **It raises the issue; it does not settle it.** The scope is fixed to established, published material. Where the argument could be extended — however interesting the extension — the unit stops, because past that line it would be doing the thing it warns about.
- **Section 4's second half is the unit.** "What they do not say" is the point, not a caveat attached to one. If you read only one part of this unit, read that.
- **Statement D is the one to sit with.** 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.
- **The atmosphere never replaces the citation.** Every claim in this unit keeps its source or its uncertainty label, including the ones on the darkest pages. A unit about unwarranted certainty that asked for your trust on mood would be refuting itself.
- 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.

18.3.5 Checkpoint data

The self-check questions for this unit, as structured data. Both routes read this block, so the questions and their feedback cannot differ between them. Editing it changes both.

Checkpoint `cp18-sort-the-claims` — Sort the claims

Kind: match.

Unit 18 makes claims of three different kinds, and its own argument says you should not accept them on its authority. Match each claim to what it actually is.

Observations, in order:

1. No sufficiently expressive formal system can define a truth predicate for its own sentences.
2. Descriptions should include the observer, because an observer inside a system cannot step outside it to check.
3. Because no system can fully account for itself, nothing can really be known.

Options, numbered from zero:

0. A proved theorem, with a date and a standard citation.
1. An established research programme: published, influential, decades old, and still not a theorem.
2. An overclaim: it does not follow, and your own lab work contradicts it.

Answer: `0, 1, 2`

Hint: One has a date and a name attached to it in any logic textbook. One is a way of framing problems that a research community has developed since the 1940s. One sounds profound and contradicts something you personally verified in Unit 8.

Diagnostics, by the answer given:

- `1,0,2` — You have the first two swapped. Tarski's undefinability theorem is a result from 1936 that appears in logic textbooks – it is proved. Including the observer in the description is the founding move of second-order cybernetics: published since the Macy Conferences, widely built on, and a way of framing problems rather than something anyone proved.

- `0, 2, 1` — The third claim is the one that does not follow. In Unit 8 you established agreement with the oracle over tested indices and checked properties, under a stated convention. That scoped knowledge, based on a record rather than a feeling, contradicts ‘nothing can be known’.
- `2, 1, 0` — Reversed. Ask which of these you could look up in a logic textbook, which names a research tradition whose history you could read, and which contradicts something you verified with your own hands.

Explanation: This is the module’s last exercise and it turns the method on the module itself. A proved theorem, an established research programme and an overclaim are three different kinds of thing, and telling them apart is the whole skill – not scepticism about everything, but knowing what kind of support each claim actually has. Note that the middle category is not a criticism: a research programme can be decades old, influential and worth taking seriously while still not being a theorem.

19 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.

19.1 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]

19.2 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 2 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.

19.3 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_1	B_2	1/6
B_3	B_4	-1/30
B_5	B_6	1/42
B_7	B_8	-1/30

Note G's worked example computes her B_7, which is the modern B_8 = -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_bernoulli.py`.

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`.

19.4 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 ${}^2V_5$ and ${}^2V_4$, in that order, and its stated result is `(2n−1)/(2n+1)`. Since V_4 holds `2n−1` and V_5 holds `2n+1`, producing that result requires dividing V_4 by V_5. 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 $V_4 \div V_5$; the published $V_5 \div V_4$ 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_7, the modern B_8.

Established (facsimile plus [S7] §5.2): operation 4 is not the only printed fault. Glaschick's errata list for the same table also records operation 24: "In line 24 the negative value of V_{13} must be transferred to V_{24}, i.e. $V_{24} - V_{13}$ must be calculated. Also, in the 6th column it must read = − B_7." As printed, operation 24 transfers the value without that sign inversion. Operation 21 carries further printed defects of the same kind. So "the table as printed" is not one faulty object with one result.

Established (recomputed in this module): the result depends on which question is asked, and there are at least four. `course/lab/noteg.py` reconstructs the 25 operations and runs them with exact arithmetic. The run that repairs both faults reproduces the documented `−1/30`, which is what earns trust in the reconstruction. The others then answer different questions:

What is executed	Operation 4	Operation 24	Earlier B values	Result
The intended computation	repaired	repaired	supplied	`−1/30`
One printed fault, in operation 4	as printed	repaired	supplied	`139/630`

What is executed	Operation 4	Operation 24	Earlier B values	Result
The literal single table	as printed	as printed	supplied	`−139/630`
One printed fault, in operation 24	repaired	as printed	supplied	`1/30`
The faulty routine run from the start	as printed	repaired	generated by itself	`−25621/630`

Withdrawn: an earlier version of this module stated that "the as-printed table computes `139/630`" and that the widely repeated `−25621/630` "does not survive recomputation". Both statements were wrong, in the same way and for the same reason: they did not say which object was being executed. The `139/630` run repairs operation 24 without saying so, and `−25621/630` reproduces exactly under the reading where the faulty routine generates the earlier Bernoulli numbers it then consumes — a reading Note G's operation 25 invites, since it increments *n* so the Engine can go round again. Those accounts do not state their interpretation; neither did this module. The error is retained here rather than quietly repaired, because it is the module's own instance of the failure it teaches.

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_5 before V_4) 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 each 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, and is needed before the operation-24 reading is treated as anything more than Glaschick's. What is settled now is the location of the operation-4 inversion, the operand order, and — by computation, under stated interpretations — what each reading of the table produces.

Why this restraint is the lesson, not a limitation. Unit 16'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 it is one this module has to model rather than only describe, which section 4 records it failing to do and then repairing.

19.5 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 17 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.

19.6 6. Claim register

Every historical claim the module makes should appear here, with its status split into the three things "established" used to run together: how well it is supported, whether anyone disagrees, and whether this module has actually read the source. Those are independent — a claim can be well supported and disputed, or uncontested and known only through somebody else's citation — and collapsing them is how "established" came to mean both "we checked it" and "nobody argues with it".

`scripts/check_claims.py` validates this table: ids unique, every status drawn from the closed vocabulary below, and every source key resolving to `course/references.md`.

There is no "cited but not read" status, and that is deliberate. A source is either checked or the claim does not go in. Naming a source you have not opened tells a reader that the claim rests on something, while leaving unexamined the only question that matters about it — and it is the same move as a generated citation that points at a real work saying something else. The vocabulary therefore records *which kind* of source was checked, primary or secondary, and `none` for a claim with no source at all. `none` is honest; it means unknown.

dimension	values
support	`verified` (checked against the source, or recomputed here) · `supported` (a source states it) · `inferred` (follows from the sources without being stated by them) · `unverified` (no check yet) · `refuted`
disagreement	`uncontested` · `disputed` (sources disagree) · `unresolved` (the record cannot settle it)
source	`primary-checked` (the original examined) · `secondary-checked` (a later work reporting it, read and located) · `none` (no source; the claim is unknown)

id	claim	support	disagreement	source	sources
H1	Babbage designed the Analytical Engine; never completed in his lifetime	supported	uncontested	secondary-checked	[S1], [S4]
H2	Store and mill architecture, punched-card input	supported	uncontested	secondary-checked	[S1]
H3	Menabrea published a French account in 1842	supported	uncontested	primary-checked	[P1]
H4	Lovelace translated it and added Notes A to G, published 1843	supported	uncontested	primary-checked	[P2]
H5	The Notes are about three times the length of the article	inferred	uncontested	primary-checked	[P2]

id	claim	support	disagreement	source	sources
H6	Note G contains an operation-by-operation Bernoulli computation	verified	uncontested	primary-checked	[P2]
H7	Note G's example computes Ada's B7 = modern B8 = -1/30	verified	uncontested	primary-checked	[P2]
H8	Ada's B1, B3, B5, B7 are the modern B2, B4, B6, B8	verified	uncontested	primary-checked	[P2]
H9	"The first program ever written"	inferred	disputed	secondary-checked	[S2], [S3], [S5]
H10	Extent of Lovelace's own contribution to the Notes	unverified	disputed	secondary-checked	[S2], [S3], [S5]
H11	The published table contains an error, in operation 4	verified	uncontested	primary-checked	[P2a]
H12	The error is an inverted division, V5 divided by V4 where V4 divided by V5 is correct	verified	uncontested	primary-checked	[P2a], [S7]
H13	Operation 24 is also printed in a form the intended computation does not use	supported	uncontested	secondary-checked	[S7]
H14	With operation 4 as printed, operation 24 repaired and earlier values supplied, the table computes 139/630	verified	uncontested	primary-checked	[S7]
H15	Taking both printed faults literally gives -139/630; repairing only operation 4 gives 1/30	verified	uncontested	primary-checked	[S7]
H16	The repeated -25621/630 reproduces when the faulty routine generates its own earlier values	verified	uncontested	primary-checked	[S6], [S7]

id	claim	support	disagreement	source	sources
H17	Which of those readings a given published account means	unverified	unresolved	none	
H18	The error was a typesetting slip rather than authorial	inferred	disputed	secondary-checked	[S8]
H19	“No pretensions whatever to originate anything” appears in Note G	verified	uncontested	primary-checked	[P2]
H20	What that passage implies about language models	unverified	disputed	none	

H14, H15 and H16 are recomputed by `course/lab/noteg.py`, whose tests assert each value; [S7] is cited for the operation sequence they run, and the arithmetic is this module’s own. H13 rests on [S7] alone, which is why `TODO.md` Stage 6 carries a task for confirming operation 24 against a high-resolution facsimile.

20 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** (each unit page, and `build/site/progress.html` for all of them at once) 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 6, 8 and 16 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.

20.1 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 and no virtual environment. The tools supplied here make no network requests. That is a statement about this code, not a restriction on yours: `check.py` runs the file you hand it, and nothing in the sandbox prevents that file from opening a connection. The safety section below is about exactly that gap.

Tested on Python 3.13.14. Nothing here uses a feature newer than Python 3.9, so earlier 3.x versions are expected to work – expected, not verified: no other version has been run against the test suite.

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
```

20.2 Running generated code: what protects you, and what does not

`check.py` **executes** the file you hand it, because there is no way to find out what a routine returns without running it. It does not run it here: `sandbox.py` starts it as a separate process, and only data comes back.

What that process gets. A wall-clock timeout (10 seconds), a CPU-time limit, an address-space limit, a file-size limit, a cap on how many processes it may create, its own session so nothing lingers on your terminal, and an environment stripped to `PATH` — so API keys and tokens sitting in your shell are not handed to it. Whatever it returns crosses back as text and integers, never as objects to be unpickled or code to be run.

What it is. A blast-radius reduction, and a way to turn a hang or a runaway allocation into a named finding instead of a frozen terminal.

What it is not. A security boundary. The child runs as you: it can read the files you can read, and if the platform provides no resource limits (Windows) only the timeout applies. Nothing here stops deliberate exfiltration by code written to do it.

So the judgement is still yours, and it is a real one:

- **Read the whole file first.** Look for anything that touches the filesystem, the network, or the shell. A Bernoulli routine needs `fractions` and perhaps `math`, and nothing else. Reading is an educational control, not a security one: obfuscated or simply misunderstood code passes a careless read.
- **For code you have not read, from a source you have no reason to trust**, run the whole lab inside a container or a virtual machine with no network and no access to your home directory — or take the table route, which executes nothing and completes the unit.

- **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.

The threat model is deliberately modest and stated rather than implied: this lab protects you from *accidents* — infinite loops, runaway memory, a stray file write in your working directory — and from handing your environment to a stranger. It does not protect you from an adversary.

20.3 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.
`noteg.py`	A reconstruction of Ada's Note G table, run with exact arithmetic under each stated interpretation. Repairing both printed faults gives $-1/30$; operation 4 as printed with 24 repaired gives $139/630$; both faults literal gives $-139/630$; letting the faulty routine generate its own earlier values gives $-25621/630$, the figure repeated online. Run it: `python3 noteg.py`. Used by Unit 16 to show that a bare number without a stated interpretation is not yet a claim.
`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 36 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.

20.3.1 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, `B1 = +1/2`.	`b1-sign`
`examples/wrong_odd_terms.py`	Uses Lovelace's Note G labels: her `B1, B3, B5, B7` are the modern `B2, B4, B6, B8`.	`odd-terms`, `ada-indexing`, `one-observation`

file	what is wrong with it	code
`examples/float_output.py`	Returns `float`. Convincing at small indices, wrong everywhere.	`not-fraction`, `odd-terms`
`examples/off_by_one_range.py`	Every value is a genuine Bernoulli number, filed one index too early.	`off-by-one`, `odd-terms`, `one-observation`

20.3.2 The exercises

file	checkpoint	unit	status
`exercises/ex04_bernoulli.py`	`cp08-bernoulli`	8	core
`exercises/ex05_write_a_test.py`	—	8	optional CS extension
`exercises/ex06_ada_translation.py`	—	8	optional CS extension

The optional CS extensions are labelled optional in the module because they are optional. Skipping them costs you no core content.

20.4 The interface contract

This is the contract `check.py` enforces, and the same words appear in the Unit 10 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 what **the checker enforces**, and it is short enough to read and precise enough to test against — a specification precise enough to test against is a specification precise enough to prompt with.

It is not the whole specification. Unit 10 writes the full version out as nine clauses, C1 to C9, which add what happens on a negative index, that importing the module must have no side effects, and what may not be asserted in a comment. The four lines above are the part a program can decide; the rest is still part of the contract, and Unit 10 is where it lives.

20.5 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`	Lovelace's Note G labels: only the non-zero terms are numbered, and with the odd integers, so her `B1, B3, B5, B7` are the modern `B2, B4, B6, B8` and there is no `B2`.
`one-observation`	`ada-indexing` and `off-by-one` both matched. They agree at every index except `B0`, so this is one observation with two readings rather than two faults.
`off-by-one`	Every value shifted one index.
`not-fraction`	Returns a type other than `fractions.Fraction`. A `float` is additionally inexact; an `int` is exact and still wrong.
`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.

20.6 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.

20.6.1 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.

20.6.2 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 8 (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 6 (the convention trap), Unit 10 (state it in the prompt)

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 17 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.

20.6.3 List the checkpoints

```
$ python3 selfcheck.py list
Self-check checkpoints
----------------------------------------------------------------------

Unit 0
  [x] cp00-contract  (choice)
      The learning contract
  [x] cp00-what-this-needs  (choice)
      What this module assumes of you

Unit 1
  [x] cp01-where-the-reference-lives  (match)
      Where the reference lives
  [x] cp01-changing-class  (choice)
      Moving a task into a cheaper class

Unit 2
  [x] cp02-ordered-steps  (multi)
      What the procedure left unsaid
```

```
Unit 3
  [x] cp03-risk-triage  (match)
      Triage by how it can be wrong
  [x] cp03-system-layers  (match)
      Separate the assistant's system layers

Unit 4
  [ ] cp04-what-changes-the-check  (match)
      What each kind of system changes about the check
  [ ] cp04-cheapest-adequate  (choice)
      When the cheap option is the right one

Unit 5
  [ ] cp05-what-reaches-the-system  (multi)
      What actually reaches the system
  [ ] cp05-locators  (choice)
      What a locator is worth

Unit 6
  [ ] cp06-b4-by-hand  (fraction)
      B4 by hand
  [ ] cp06-convention  (choice)
      Which convention is this?
  [ ] cp06-ada-index  (integer)
      Reading Ada's index

Unit 7
  [ ] cp07-trust-triage  (choice)
      What you can trust before testing

Unit 8
  [ ] cp08-bernoulli  (code)
      Make the checker pass
  [ ] cp08-what-it-shows  (multi)
      What the check established, and what it did not
  [ ] cp08-table-evidence  (match)
      State what a table comparison establishes
  [ ] cp08-diagnose  (choice)
      Name the failure mode

Unit 9
  [ ] cp09-diagnose-source  (match)
      Naming the way a citation failed
  [ ] cp09-scope-the-claim-source  (choice)
      Scoping what the check established

Unit 10
  [ ] cp10-spec-clauses  (match)
      Which clause kills which bug

Unit 11
  [ ] cp11-reversibility  (choice)
      Sorting an action by what it costs to be wrong
  [ ] cp11-trace-evidence  (multi)
      What in a trace is evidence

Unit 12
  [ ] cp12-two-axes  (choice)
      Which question does consequence answer
  [ ] cp12-escalation-trigger  (multi)
      What licenses stopping
```

```
Unit 13
  [ ] cp13-what-leaves  (multi)
      What actually leaves your device
  [ ] cp13-whose-material  (match)
      Whose material is it

Unit 14
  [ ] cp14-three-obligations  (match)
      Three obligations, separately owed
  [ ] cp14-settled-or-not  (choice)
      Which questions have answers

Unit 15
  [ ] cp15-what-a-check-shows  (choice)
      What one comparison establishes
  [ ] cp15-alt-text  (choice)
      Alt text describes function

Unit 16
  [ ] cp16-note-g  (fraction)
      Comparing against the historical table

Unit 17
  [ ] cp17-disclosure  (multi)
      What a defence has to contain
  [ ] cp17-scope-the-claim  (choice)
      Say what the evidence supports, and no more

Unit 18
  [ ] cp18-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.

20.6.4 Run one checkpoint

```
$ python3 selfcheck.py run cp06-ada-index
------------------------------------------------------------------------
cp06-ada-index  (Unit 6)  -  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 cp06-b4-by-hand
------------------------------------------------------------------------
```

```
cp06-b4-by-hand  (Unit 6)  -  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 `cp06-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 cp08-bernoulli
----------------------------------------------------------------------
cp08-bernoulli  (Unit 8)  -  Make the checker pass
----------------------------------------------------------------------
Complete course/lab/exercises/ex04_bernoulli.py so that
bernoulli(n) meets the contract, then run: python3 selfcheck.py run
cp08-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.

20.6.5 Run every checkpoint in a unit

```
$ python3 selfcheck.py run --unit 6
----------------------------------------------------------------------
cp06-b4-by-hand  (Unit 6)  -  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 8.

----------------------------------------------------------------------
cp06-convention  (Unit 6)  -  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 10 puts that
sentence in the prompt.

----------------------------------------------------------------------
cp06-ada-index  (Unit 6)  -  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.
----------------------------------------------------------------------
```

20.6.6 See how far you have got

```
$ python3 selfcheck.py progress
Passed 7 of 36 checkpoints.

  Unit 0  [##]  2/2
  Unit 1  [##]  2/2
  Unit 2  [#]  1/1
  Unit 3  [##]  2/2
  Unit 4  [..]  0/2
  Unit 5  [..]  0/2
  Unit 6  [...]  0/3
  Unit 7  [.]  0/1
  Unit 8  [....]  0/4
  Unit 9  [..]  0/2
  Unit 10  [.]  0/1
  Unit 11  [..]  0/2
  Unit 12  [..]  0/2
  Unit 13  [..]  0/2
  Unit 14  [..]  0/2
  Unit 15  [..]  0/2
  Unit 16  [.]  0/1
  Unit 17  [..]  0/2
  Unit 18  [.]  0/1

Next up: cp04-what-changes-the-check (Unit 4)
```

20.6.7 Export a record for your evidence pack

```
$ python3 selfcheck.py progress --evidence
# Self-check record

Generated 2026-07-27 09:00 UTC.

Passed 7 of 36 checkpoints.

| checkpoint | unit | passed | attempts |
|---|---|---|---|
| cp00-contract | 0 | yes | 1 |
| cp00-what-this-needs | 0 | yes | 1 |
| cp01-where-the-reference-lives | 1 | yes | 1 |
| cp01-changing-class | 1 | yes | 2 |
| cp02-ordered-steps | 2 | yes | 1 |
| cp03-risk-triage | 3 | yes | 1 |
| cp03-system-layers | 3 | yes | 1 |
| cp04-what-changes-the-check | 4 | no | 0 |
| cp04-cheapest-adequate | 4 | no | 0 |
| cp05-what-reaches-the-system | 5 | no | 0 |
| cp05-locators | 5 | no | 0 |
| cp06-b4-by-hand | 6 | no | 0 |
| cp06-convention | 6 | no | 0 |
| cp06-ada-index | 6 | no | 0 |
| cp07-trust-triage | 7 | no | 0 |
| cp08-bernoulli | 8 | no | 0 |
| cp08-what-it-shows | 8 | no | 0 |
| cp08-table-evidence | 8 | no | 0 |
| cp08-diagnose | 8 | no | 0 |
| cp09-diagnose-source | 9 | no | 0 |
| cp09-scope-the-claim-source | 9 | no | 0 |
| cp10-spec-clauses | 10 | no | 0 |
| cp11-reversibility | 11 | no | 0 |
| cp11-trace-evidence | 11 | no | 0 |
| cp12-two-axes | 12 | no | 0 |
| cp12-escalation-trigger | 12 | no | 0 |
| cp13-what-leaves | 13 | no | 0 |
| cp13-whose-material | 13 | no | 0 |
| cp14-three-obligations | 14 | no | 0 |
| cp14-settled-or-not | 14 | no | 0 |
| cp15-what-a-check-shows | 15 | no | 0 |
| cp15-alt-text | 15 | no | 0 |
| cp16-note-g | 16 | no | 0 |
| cp17-disclosure | 17 | no | 0 |
| cp17-scope-the-claim | 17 | no | 0 |
| cp18-sort-the-claims | 18 | 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 17 defence is for.
```

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.

20.6.8 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.

20.6.9 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.
```

20.6.10 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 10.
```

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.

20.6.11 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.

20.7 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.

20.8 The browser route

The browser route runs the same 36 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`, so the **questions** cannot differ between the routes.

That is narrower than it sounds, and worth stating precisely. What is generated from one source is the checkpoint data. The prose around it — the instructions in each unit, the commands printed on the page, the counts quoted in this guide — is written by hand and can drift, and has: six commands here once told you to run a unit number that no longer existed. `make test` now runs every command the course prints and regenerates the transcripts below, because the shared source protects the questions and nothing else.

The browser route 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 8, which reaches the same conclusion by hand.

20.9 Where this fits

you are working on	start with
Unit 6, grounding yourself in the numbers	`python3 reference_bernoulli.py`, then `selfcheck.py run --unit 6`
Unit 7, judging generated code	read the file, then `python3 check.py <file>`
Unit 8, the verification lab	`exercises/ex04_bernoulli.py` and `selfcheck.py run cp08-bernoulli`
Unit 10, writing the precise prompt	the interface contract above, quoted verbatim
Unit 16, the historical table	`selfcheck.py explore`, then `map`
Unit 17, disclosure and defence	`selfcheck.py progress --evidence`

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.

21 Glossary

Terms used across this module, with the unit that introduces each one. The units restate them where they are first needed.

Where a term has a loose everyday meaning and a precise one here, the precise one is what the module means.

21.1 Index

term	introduced in
algorithm	Unit 2
ordered procedure	Unit 2
determinism	Unit 2
state	Unit 2
language model	Unit 3
next-token prediction	Unit 3
attention	Unit 3
pre-training	Unit 3
post-training	Unit 3
retrieval	Unit 3
plausibility	Unit 3
hallucination	Unit 3
independent verification	Unit 3
convention	Unit 6
exact rational arithmetic	Unit 6
floating point	Unit 6
ground truth	Unit 6
off-by-one	Unit 7
verification	Unit 8
test oracle	Unit 8
differential testing	Unit 8
property-based check	Unit 8
first divergence	Unit 8
overreliance	Unit 10
established, contested, unsourced	Unit 16
disclosure	Unit 17
defence	Unit 17
provenance	Unit 17
warrant	Unit 17
formative and summative	Unit 0
first-order and second-order reasoning	Unit 18
observer	Unit 18
Macy Conferences	Unit 18
understanding	Unit 17, Unit 18
task class	Unit 1
context window	Unit 5
attachment	Unit 5
tool call	Unit 4, Unit 11
agent	Unit 11
trace and log	Unit 11
reversibility and recovery	Unit 11
escalation	Unit 12
competence and authority	Unit 12
data minimisation	Unit 13
redaction and re-identification	Unit 13
locator	Unit 9
source-checking failure modes	Unit 9
representation	Unit 15
accessibility	Unit 15

21.2 Definitions

21.2.1 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 2. 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.

21.2.2 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 2.

21.2.3 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.

Determinism is a property of the *procedure as specified*. A real implementation of it can still vary, because it runs somewhere: a search returns what the index holds today, a file operation depends on what is on disk and who may write to it, and a network call can time out. Unit 11 turns on this distinction — the tool an agent calls is not the safe step simply because the procedure it implements is deterministic.

Introduced in Unit 2. Contrast with a language model, whose output for the same prompt may differ between runs.

21.2.4 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 2.

21.2.5 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 3.

21.2.6 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 3.

21.2.7 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 3.

21.2.8 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 3.

21.2.9 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 3.

21.2.10 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 3.

21.2.11 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 3.

21.2.12 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.

Unsupported generation is the more useful working category, because falsity is not the only problem and not the easiest one to detect. Output can be unsupported and accidentally true, more specific than its

source, attributed to the wrong source, or simply unverifiable. Truth and evidential support are different questions, and this module is mostly about the second.

Introduced in Unit 3. Used consistently in this sense throughout the module.

21.2.13 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 3; practised in Unit 8.

21.2.14 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 6. Failure-mode code: `b1-sign`.

21.2.15 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 6.

21.2.16 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 6. Failure-mode code: `not-fraction`.

21.2.17 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 6 — not from the program being tested, and not from a second generated answer that agrees with the first.

The ordering matters: establish ground truth *before* you look at the output you are judging, or you will find yourself deciding that whatever you got must have been right. This is a discipline rather than a law — a reference sometimes has to be revised once you understand the problem better, and Unit 16 does exactly that. What makes a revision honest is that it leaves a record: what changed, when, and on what evidence.

Some questions have no ground truth of this kind. Unit 15's representation questions have relevant empirical distributions but no table of what an output should have said, and fairness is a normative question rather than a factual one. "No ground truth" there means no single correct answer to compare against, not that nothing can be checked.

Introduced in Unit 6.

21.2.18 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 7. Failure-mode code: `off-by-one`.

21.2.19 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 8.

21.2.20 test oracle

A trusted source of correct answers, used to judge a piece of code under test. `course/lab/reference_bernoulli.py` 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 8.

21.2.21 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 8.

21.2.22 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`.

A property check catches faults a hand-picked table of examples misses, because the property is stated for the whole domain rather than for the rows someone happened to write down. It still only *tests* the inputs it runs: this module checks the property over its stated range, so it is an invariant check over that range, not a proof for every `k`. Elsewhere "property-based testing" usually means a tool that generates many inputs to attack the property; the difference is how the inputs are chosen, not what is being asserted.

Introduced in Unit 8. Failure-mode code: `odd-terms`.

21.2.23 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 8.

21.2.24 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 10.

21.2.25 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 16, §6 of its reading; the same status labels are used throughout `course/references.md`.

21.2.26 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 17. See `course/learner-evidence-template.md`.

21.2.27 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 17.

21.2.28 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 17.

21.2.29 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 17; taken up again in Unit 18, where a warranted claim carries its index — who observed, in what context, with what record.

21.2.30 first-order and second-order reasoning

In Unit 18, **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 18.

21.2.31 observer

In Unit 18’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 18.

21.2.32 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 18.

21.2.33 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 one faulty reading of Note G's table and predicted exactly what it computes, while misnaming which reading they had mapped. The capacity to map (understanding) and the empirical accuracy of the map (correctness) are two different things, and a defence needs both — plus a statement of what the map is a map *of*.

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 17; formalised through the second-order material in Unit 18.

21.2.34 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.

21.2.35 task class

Where the reference for a claim lives, which is what decides how it can be checked. Unit 1 sorts work into **retrieval** (the reference is a source you can open), **transformation** (the reference is your own input), **computation** (the reference is a rule you can apply), and **judgement** (there is no single correct answer).

The class gives the *starting estimate* of what checking will cost. Consequence, access to sources, the size of the task, your own expertise and how current the evidence has to be all move it from there.

Introduced in Unit 1.

21.2.36 context window

Everything a system has in front of it when it produces the next piece of output: your prompt, instructions the product supplied, earlier turns, attachments, and anything a tool fetched. It is bounded, and what happens when the bound is reached — truncation, summarising, selective retrieval — varies by product.

Introduced in Unit 5.

21.2.37 attachment

A file supplied to a system alongside a prompt. For privacy purposes the whole file has left your device; whether all of it reaches the model is a separate question, since products may index, chunk or extract from it.

Introduced in Unit 5; the privacy consequences are Unit 13.

21.2.38 tool call

A system invoking something outside itself — a search, a calculation, a file operation, an API — and receiving a result. The **selection** of tool and arguments is generated; the **execution** runs under the tool's own semantics and against the state of the world at that moment, so it can fail, half-succeed, or succeed while doing the wrong thing.

Introduced in Unit 4; central to Unit 11.

21.2.39 agent

A system that acts rather than answers: it calls tools in a loop, and some of those calls change something outside the conversation. What changes for you is that the world may move before you have looked at anything.

Introduced in Unit 11.

21.2.40 trace and log

A **log** is the record a system wrote as it ran: calls, arguments, results. A **trace** as displayed in a product may be that log, a condensed rendering of it, or a generated account of what happened — and the three look alike on screen. A log is evidence; a generated account of a run is generated text.

Introduced in Unit 11.

21.2.41 reversibility and recovery

Four different things that "undo" can mean, costing different amounts. **Undo** reverses the action. A **compensating action** leaves it standing and offsets it — a refund, a correction. **Recovery** rebuilds state from something kept for the purpose, and only exists if someone arranged it beforehand. **Containment** limits what follows when none of the others is available.

Introduced in Unit 11; generalised in Unit 12.

21.2.42 escalation

Handing a decision to someone qualified or authorised to make it. It is a routine professional move rather than an admission, and it is triggered by the situation — consequence, irreversibility, conflicting evidence, whose decision it is — not by the subject matter.

Introduced in Unit 12.

21.2.43 competence and authority

Competence is whether you could recognise a wrong answer. **Authority** is whether the decision is yours to make. They are independent: you can understand a clause perfectly and still not be the person who may accept it, and either one missing is a reason to escalate.

Introduced in Unit 12.

21.2.44 data minimisation

Sending the least that will do. Usually the practical move is not "do not use the tool" but "change what you send" — generalise, abstract the question, or supply a rewritten fragment rather than the document.

Introduced in Unit 13.

21.2.45 redaction and re-identification

Redaction removes identifying details. **Re-identification** is recovering a person from what is left, which a combination of ordinary details often permits — a role, a date and a place may be enough. Redaction reduces risk; it does not by itself make material anonymous, nor make its use authorised.

Introduced in Unit 13.

21.2.46 locator

The part of a citation that says *where* in a source to look: page, section, clause, timestamp. A locator is what makes a citation checkable in bounded time, which is why Unit 9 asks for one and why a fabricated locator is its own failure mode.

Introduced in Unit 9.

21.2.47 source-checking failure modes

The six named ways a generated citation fails, in decision order: `fabricated-locator`, `stale-version`, `misattributed`, `over-specified`, `unsupported-inference`, `right-source-wrong-claim`. No program assigns them — there is no oracle for whether a source supports a claim — so the discipline is the closed vocabulary and the order in which you ask.

Introduced in Unit 9; defined in `docs/AUTHORING.md` *§3.2.*

21.2.48 representation

Whose lives, names, places and practices appear in generated material, and in what proportions. Output tends to reflect the distribution of its training data as later training and system design allow. Two failures matter: a visible skew, and an absence — and absences do not announce themselves.

Introduced in Unit 15.

21.2.49 accessibility

Whether material can be used by people with different vision, hearing, motor control and attention. Much of it is checkable today with a tool: alt text that conveys the image's purpose, decorative images marked as such, headings that follow the structure, sufficient contrast, meaningful link text, keyboard access with visible focus. Automated checks are necessary and not sufficient, and a green result is not a conformance claim.

Introduced in Unit 15.

22 Prompts for producing checkable work

This module's method is to compare a generated artefact against something established independently of it. Nothing on this page changes that. What a prompt can do is change the *shape* of what comes back, so that comparing it costs you less: a result with its convention stated is quicker to check than the same result without one, and a claim with a locator is quicker to check than a claim without.

That is the whole of what these prompts are for.

They do not verify anything. A prompt cannot make a system truthful, cannot stop it producing a confident wrong answer, and cannot make it report reliably on which of its own statements are well supported. An output that arrives neatly laid out, with its steps numbered and its sources listed, is still an output: the layout is generated too. If a system tells you a claim is established, that is one more claim to check, not a check.

So read this page as a way of lowering the cost of the work in Units 8 and 9, not as a way of skipping it.

22.1 Facts, and the things that are not facts

When you check a claim, first ask what kind of claim it is.

A **factual** claim can be settled against evidence, a computation, or a record. `B8 = -1/30` is settled by the recurrence. Whether a paper says what it is cited as saying is settled by reading the paper.

A claim about **values or ethics** is not settled that way. Whether a use of AI in assessed work is acceptable is a question about rules and about what people owe each other, and no amount of text prediction settles it.

The distinction is not that one is checkable and the other is not. It is that they take *different* checks. A value claim still rests on factual premises, and those premises can be wrong: an argument that a policy is unfair may depend on a claim about what the policy says, and that part you can look up. What you cannot do is treat a system's fluency about ethics as evidence about ethics.

How to handle it. Attribute rather than assert — "one argument runs…", "the guidance says…". Check the premises. Say whose judgement the conclusion is.

22.2 Two prompts

Neither is a persona, and the three things they are sometimes called are not the same: a **system prompt** is instruction text a product puts before your conversation, **custom instructions** are a setting that inserts your own text the same way, and a **persona** is a description of a character to write as. These are the first two. If your tool has a custom-instructions box, they go there; otherwise put them at the top of the conversation.

22.2.1 1. For mathematics, code and anything with a procedure

Ask for the parts you would need in order to reproduce the result, and no more. Long chains of stated reasoning are not an execution trace — nothing ran — and a longer explanation is not a better-supported one.

> State the input and output contract, the domain, the convention in use, and any assumption you are making.
>
> Then give:
>
> 1. the result;
> 2. the formula, algorithm or rule you applied;
> 3. only the intermediate values needed to reproduce the result;
> 4. the edge cases, and test cases that would distinguish a correct implementation from a plausible wrong one;
> 5. the result in a machine-readable form where that is possible;
> 6. what your own method does not establish.

Do not treat the explanation as evidence that the result is correct. It will be checked against an independent reference.

The point of item 4 is that it hands you the comparison. The point of item 6 is that a stated limit is easier to check than an implied one.

22.2.2 2. For prose, summaries and anything with sources

Ask for claims separated from each other and tied to locators, because that is what makes Unit 9's check possible at all.

Break the answer into individual claims, one per line.

For each claim that states a fact, give:

- the source you are relying on;
- its date or version;
- a precise locator — page, section, or clause;
- a short quotation or a faithful paraphrase of the supporting passage;
- whether the claim reports the source or draws an inference from it.

List separately anything you could not retrieve or could not support.

Do not supply a source or a locator you are not relying on. Mark an item "not retrieved" instead. The citations will be checked against the sources.

A citation that comes back this way is not verified. It is *checkable*: you know which source to open and where to look, which is the difference between a check that takes a minute and one you never do. Unit 9's six failure modes are exactly the ways this can still be wrong.

22.3 What to expect when you use them

Expect the second prompt to produce fewer claims, and expect some of them to be marked as unsupported. That is the prompt working. A system asked to supply a locator for everything can supply a locator for everything, and some of those locators will not exist — which is `fabricated-locator`, and is why Unit 9 has a name for it.

Expect neither prompt to help with a question whose difficulty is judgement rather than retrieval.

22.4 Practise on something you can check

Use the first prompt on the Bernoulli routine from Unit 7, then run the checker in Unit 8. Compare where the stated steps diverge from the reference against where `check.py` reports the first divergence. They are not always the same place, and noticing that is the exercise: the narrated steps are generated, and the divergence is measured.

Use the second prompt on a short question in your own subject, then run Unit 9's comparison on two of its claims. Count how many of the locators exist, and how many of the ones that exist say what they were cited as saying. Those are two different questions, and Unit 9's decision order keeps them apart.

23 References and source policy

23.1 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 that blocks release. 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".

23.1.1 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 16, for a claim deliberately left open together with the evidence that could move it.

23.1.2 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 task in `TODO.md`, Stage 6.

23.2 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.

23.2.1 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-16-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 ${}^{2}V_{5}$ and ${}^{2}V_{4}$ 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.

23.3 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 16’s distinction between established and contested claims.

[S6] The printed errors, and what each reading computes. Operation 4 is printed as $V_5 \div V_4$ where $V_4 \div V_5$ is correct, commonly read as a typesetting rather than an authorial error; its *location and operand inversion* are established by the facsimile [P2a] and the reconstructions [S7], [S8]. It is not the only printed fault: [S7] §5.2 also lists operation 24, where “the negative value of V_{13} must be transferred to V_{24}”, and defects in operation 21. A phrase like “the table as printed” therefore does not name one object.

The module reconstructed the 25-operation table and ran it with exact arithmetic (`course/lab/noteg.py`), which reproduces the documented $-1/30$ when both faults are repaired and then reports each other reading: $139/630$ (operation 4 as printed, 24 repaired, earlier

values supplied), `−139/630` (both faults literal), `1/30` (only 24 as printed), and `−25621/630` when the faulty routine generates the earlier Bernoulli numbers it consumes.

The last of those is the value repeated in encyclopaedia and blog sources (https://en.wikipedia.org/wiki/Note_G, accessed 2026-07-23). **Correction of record:** this module previously stated that `139/630` was what the as-printed table computes and that `−25621/630` "does not survive recomputation". Both were wrong for the same reason — neither said which object was executed — and the `139/630` run silently repaired operation 24. What those sources can still be faulted for is not stating their interpretation; so could this module, until 2026-07-26.

[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.pdf (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_4 must be divided by V_5") and groups it with typographical-domain errors; independently confirms the target value `B_7 = −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.

23.3.1 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 16 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.

23.4 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.py` deliberately uses a different method (the Akiyama–Tanigawa transform) so that agreement between them is evidence rather than duplication.

23.5 Language models and current AI systems

These references support the compact account in Unit 3. They describe distinct layers of a modern assistant: transformer architecture, pre-training and post-training, retrieval, and calibrated self-evaluation.

Unit 3 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 3, 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 3’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 17 says ordinary confident wording is not a correctness defence while avoiding the stronger, false claim that every model-derived confidence measure carries no information.

23.6 Second-order reasoning

Background for `course/second-order.md` and Unit 18. 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 18 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 (accessed 2026-07-24).

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 18'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.

23.7 Second-order cybernetics

Background for `course/second-order.md` and Unit 18, 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_laboratory/, 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/2016/12/06/heinz-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 16 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 18, 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 18 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 18 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.

23.8 Gothic and Romantic context

Supporting the visual-register decision recorded for the module and the short passage in Unit 2. 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/frankenstein-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_Register/Volume_11/Number_63/The_Vampyre, 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.

23.9 Music quoted

Unit 18 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.

The three entries below mark the three steps of that unit's argument, in the order the unit takes them: [Q1] the position that everything is a lie, [Q3] the realisation that no standpoint outside the system is available, [Q2] what is left once that is accepted. Each work is quoted to at most two short lines, and the argument at every step is stated in the module's own words beside the quotation.

Rights position. The position is to **reference the song and quote a minimal set of lyrics**, 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.

This module is produced in the UK, so the framework is **fair dealing** under the Copyright, Designs and Patents Act 1988, section 30 — which permits fair dealing for criticism or review and, since 2014, for quotation. It is not US fair use, which is a different test, and this module should not be read as relying on one.

The conditions that framework attaches, and how each is met:

- **the work has been made available to the public** — each is a commercially released recording, with label, year and the independent transcriptions each wording was checked against recorded in the entries below;
- **genuinely for criticism, review or quotation** — each line carries a step of Unit 18's argument and the unit discusses the line it quotes, never decoration;
- **no more than the purpose requires** — two lines at most from any single work;
- **sufficient acknowledgement** — title and artist inline, and a full entry on this page.

A quotation that fails any of them does not go in, and the argument never depends on a lyric, so removing one would cost the module nothing. By the module's own rule the register may carry an argument and may never substitute for one.

The licence does not extend to them. The AGPL covers this module's original work only:

> Third-party material — in particular the song lyrics quoted in Unit 18 — remains the copyright of its respective owners, is included strictly for criticism, review and educational illustration, and is **excluded from the AGPL grant**. Reusing this module does not give you any right to reuse that quoted material.

Where this stands. On 2026-07-27 the director judged the three quotations below to meet the conditions above, these being short quotations from lyrics. He said so in those terms, including the qualification "as far as I can tell", and that is what is recorded: the judgement of the person responsible for the module, not settled legal advice. On 2026-07-28, having considered a reviewer's argument that an open worldwide release carries a more complicated rights context than a closed educational presentation, he confirmed the decision and paired it with the carve-out above. Employer ownership and the copyright status of generated output remain open for a dedicated copyright review. A reader following the chain from a quotation to its basis should be able to see exactly how far the basis goes, which is why the qualification is quoted rather than smoothed away.

[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. Two lines are quoted in Unit 18: "It's not the red of which we bleed" and "It is as black as Malevich's Square". Both were checked against two independent lyric transcriptions (https://readdork.com/lyrics/einsturzende-neubauten-sabrina and https://www.letras.com/einstuerzende-neubauten/675817/, both accessed 2026-07-25). The transcriptions agree on the wording and differ on the spelling of the painter's name (*Malevich* and *Malevitch*);

the module prints *Malevich*, the usual English form, and the picture referred to is Kazimir Malevich's *Black Square*. As with [Q3], lyric sites are not authoritative editions and the wording is a review item for the sleeve text.

[Q3] Einstürzende Neubauten. "Haus der Lüge." Title track of the album *Haus der Lüge*, Some Bizzare (UK, `Bart 333`) / Rough Trade (Germany, `RTD 126`), 1989. Referenced, and quoted minimally under the educational quotation exception: Unit 18 quotes the refrain couplet "Gott hat sich erschossen / ein Dachgeschoss wird ausgebaut" and glosses it in English, as the step where the standpoint above is gone and the building work carries on regardless. On the German release the title track is track 4, listed as "Haus der Lüge / Epilog" (MusicBrainz release group https://musicbrainz.org/release-group/f30172ff-a6de-341f-9871-8614c4ccec66 and release https://musicbrainz.org/release/bf441775-8718-4e4d-bae7-7616cbe088c8, giving date 1989-09-04, country DE, labels Some Bizzare and Rough Trade with those catalogue numbers, both accessed 2026-07-25). The 1989 year is corroborated across the fifteen releases in that release group; the exact 4 September date rests on the single release entry and is recorded here as single-sourced. The couplet's wording was checked against two independent lyric transcriptions (https://readdork.com/lyrics/einsturzende-neubauten-haus-der-luge and https://www.songtexte.com/songtext/einsturzende-neubauten/haus-der-luge-2bd2a476.html, both accessed 2026-07-25), which agree; both record it as a repeated refrain rather than a single line. Lyric sites are not authoritative editions, so the wording is recorded as transcription-sourced and is a review item for the sleeve text.

23.10 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.