Writing for teaching

Last modified: 2026-11-07 21:34:25 (PST)

> **AI-Assisted Authorship**
>
> This chapter was written with the assistance of GitHub Copilot, which expanded on outlined ideas and draft notes provided by the author. The content represents the author's perspective and has been reviewed for accuracy, but the detailed prose was generated through AI assistance.

Lecture notes, slides, and problem sets teach rather than report. The general rules for scientific writing in the other chapters still apply. This chapter adds the rules that apply to teaching materials in particular.

Teaching writing and writing for peers

A teaching document and a paper have different readers, so they follow different rules. The reader of a teaching document is learning the material. The reader of a paper is a researcher who already knows the field and wants to judge new work.

	Teaching materials	Writing for peers (a paper)
Reader	A learner who needs each term defined before it is used.	A researcher in the field who knows the standard terms.
Purpose	The reader can do the method afterwards.	The reader can judge and build on the new result.
Standard material	Stated in full, with its definitions, notation, and worked examples.	Cited, not restated.
Order	An exercise first, then the definition, the result, and its proof.	The standard paper structure.

	Teaching materials	Writing for peers (a paper)
Examples	Every definition is followed by an example, and every example by an exercise.	Examples appear where they support a claim.
Repetition	One written version of each result is reused wherever it is needed.	Each result is stated once, where the argument needs it.

Four consequences follow for how you write.

- **Define every term that the reader might not know.** A paper may use a standard term of its field without a definition. Teaching materials define the term, or link to the page that does.
- **Show each step of a derivation.** A paper may skip a routine step. Teaching materials give one operation per line, so the learner can check each step.
- **Test the reader's understanding.** Teaching materials include exercises and solutions. A paper does not.
- **Use the same notation throughout, and state its alternatives.** A paper fixes one notation for its own argument. A learner also meets other notations in other textbooks and papers.

The rest of this chapter gives the rules for teaching materials. The chapters on paper organization and manuscript submission give the rules for papers.

Build the lecture from exercises and solutions

Active learning, in which students work through tasks themselves during class, improves exam performance over lecture alone (Freeman et al. 2014). The exercise-first structure is this guide's own convention for putting that result into practice. Pose each task as an exercise, give students time to try it, and then show the solution.

- Pose every data manipulation or analysis as an exercise, and show the code in the solution. Do not show the code first and explain it afterwards.
- Pose every calculation, derivation, or proof as an exercise, and put the worked steps in the solution.
- Ask the question that a definition or theorem answers before stating it. The solution answers informally, in plain words, and the formal `#def-` or `#thm-` div follows the solution div.

- Restate every result that a solution first derives in its own theorem-type div (`#thm-`, `#lem-`, `#cor-` or `#prp-`) after the exercise (Ezra, 2026-10-29). The solution shows the working, and the theorem div states the result by itself, so a reader can find and cite it. The `check-restated-results` workflow in `gha` warns about a derivation solution that a pull request edits and that no theorem-type div follows (gha#1309). The check never sweeps existing content: restate the result when you edit the passage anyway. `<!-- restated-ok: reason -->` in the solution exempts a result that is stated elsewhere. A restated theorem passes review only when it meets these four conditions:
  - **The proof stands alone.** It derives the result step by step, or cites earlier results, and does not only point back at the solution that first derived it.
  - **The statement uses only defined symbols.** Every symbol in it is defined in the statement itself or in an earlier div.
  - **The first use of each term is linked** to the div that defines it.
  - **Each source line holds one clause**, as the semantic line break rule requires.
- Give each bottom-level prompt its own `#exr-` div and its own `#sol-` div, in order, rather than one exercise with parts (a) through (e). Any exercise with several parts is split, however the parts are marked: letters, numbers or bullets. Put the shared setup in the first exercise, and begin each later exercise by citing it, as in "Make the assumptions of the first exercise." Later exercises can use earlier results through cross-references.
- Split the solution the same way as its exercise (Ezra, 2026-10-28). Each `#sol-` of a split exercise answers only its own `#exr-`, takes its slug from that exercise (see the slug rule below), and sits in its own subfile after its exercise. This covers code: a lab solution puts each step's code, output and figure in the solution of the exercise that asks for it, not all in the first solution. A solution does not label parts such as "(b)" or "Part 2", and does not draw a figure that a later exercise asks for. Code that a later part needs stays in the earliest solution that needs it, because the page shares one session per language. The `check-split-solutions` workflow in `gha` enforces this (gha#1290).
- Make each exercise ask for one thing (Ezra, 2026-10-27). A prompt that says "compute the slope, and compare it with the fitted value" asks for two things, so split it into two exercises, each with its own solution. Split a list item or step inside an exercise the same way, into separate exercises. Phrases such as "and then", "and also", and "and compare" mark a second ask.
- Put the instruction or question of an exercise in its own paragraph, apart from the scenario information (Ezra, 2026-11-02). The scenario paragraphs give the data, the setting and the definitions that the exercise needs, and the last paragraph asks the question or gives the instruction ("Show that …", "Which model …?"). A reader scanning for the task then finds it without rereading the setup. A paragraph that runs from the scenario into the task hides the task. The `check-exercise-question-paragraph` check fails a pull request that adds such a paragraph. Put `<!-- question-paragraph-ok: reason -->` inside an exercise that it misreads.

- On slides, put a slide break between an exercise and its solution, so the prompt is on screen alone while students work.
- Give each solution the slug of its exercise, with `sol-` in place of `exr-`.
- Put a hint in a callout titled "Hint" inside the exercise div, not in italic text.
- Give every exercise a hint that links to the sections of the notes it draws on (Ezra, 2026-11-04), in a lab as in a homework. Where an exercise asks for code, link the section of the programming primer that introduces each command it needs, and link each definition or result the exercise rests on. Use a direct anchor link on the longest phrase that names the thing, as Link the words, not a trailing citation describes. The hint points the reader to the material and does not give the answer.
- Where the solution has a tab for each language, a hint that names a command of one language names the other language's counterpart too (Ezra, 2026-11-04). Say which language each command belongs to, as in "`.predict` in Python, `predict()` in R", and link the primer example whose tabs show both. A reader working in the other language otherwise finds only a name that does not exist in their language.
- Give each exercise the background that its title implies, and say why the reader should care (Ezra, 2026-10-29). Whatever the title tells the reader about the object, the body also says: name the object, link its definition, and say where it arises and what it is used for. For a title such as "Maximizing a gamma-family kernel", the body says what a gamma distribution and its kernel are, and that maximizing the kernel finds the mode without computing the normalizing constant. A body that only restates the formula leaves a reader who skipped the title with a formula and no reason to care. The first exercise of a split series carries the background, and each later exercise begins by citing it, as the splitting rule above says.

Example 0.1 (Asking before defining)**.**

✗ **Generalization** is the ability of a model to perform well on new data.

✓

```
::: {#exr-generalization}
What does it mean for a model to generalize?
:::

::: {#sol-generalization}
A model generalizes when it predicts well on data it was not fit to.
:::

::: {#def-generalization}
(the formal definition)
:::
```

The revised version lets students attempt the idea before they see the formal definition.

Build every unit from smaller atomic units

Make every unit of analysis as concise and minimal as possible. The units are mathematical expressions and divs of every kind: definitions, theorems, lemmas, corollaries, examples, remarks and proofs. When a unit grows complicated, do not write the complexity into it. Abstract the complexity out into smaller, atomic definitions, results and expressions, each with its own name and identifier, and glue the pieces together with hyperlinks.

This is the compositional approach of modular programming: a program is built from small, single-purpose modules that call one another, and a maintainer changes one module without reading the rest. The notes work the same way. A theorem links to the definitions it uses, a proof links to the lemmas it uses, and an expression is written with the named pieces it is built from.

- A display holds one idea. Name each meaningful sub-expression, give it a macro, and build the display from the names (see Making expressions concise).
- A div holds one term or one result. Move a second term, a side property or a long argument into its own div, and link to it from the place that needs it (see Give each defined term its own div).
- A theorem-type div states the result and its hypotheses, and no reasoning. Words that justify the claim ("so", "because", "hence", "therefore", "since") belong in the proof.
- A proof states the lemmas it needs and links to them, instead of deriving them in the middle of the argument.
- Extract a lemma wherever feasible. When a proof, solution or example states a general fact that does not depend on its setting, such as $\mathbb{E}[Z^2] = \mathrm{Var}(Z) + (\mathbb{E}[Z])^2$ for any random variable Z, give the fact its own lemma div before its first use, and link to it from each place that uses it. Put the lemma on the prerequisite site that owns the topic

(mds, pds or sds), not in the course page that needs it. Extract a lemma from a passage when you edit that passage anyway, and do not sweep existing content.
- A link replaces a repeated explanation. If a piece is already written, link to it instead of writing it a second time.

When a unit still looks long, ask what could be named and moved out.

Example 0.1 (Splitting one long definition)**.**

✗ One div defines the ridge estimator, defines its penalty, and proves that it shrinks the coefficients.

✓ Three divs: a definition of the penalty, a definition of the ridge estimator that links to the penalty, and a theorem on shrinkage that links to both.

The ✓ version lets a later chapter link to the penalty alone, and each piece fits on a slide.

Say what a yes-or-no answer means

Follow a "Yes" or "No" with what the answer means in the words of the question, as in "No: it's not possible that the advanced network has a test error of 0.25," not a bare "No." (Ezra, 2026-10-28). A reader who sees only the first line then still knows what the answer claims, and a bare "No." leaves open what is being denied. The rule covers solutions, worked examples and any other answer to a question.

Example 0.1 (Saying what a "No" means)**.**

✗ No. The irreducible error is a lower bound on the expected squared prediction error.

✓ No: it's not possible that the advanced network has a test error of 0.25. The irreducible error is a lower bound on the expected squared prediction error.

The ✓ version states the conclusion in the question's own words, so the reasons that follow support a claim the reader has already seen.

Give each question's answer its own paragraph

When a prompt asks several questions, answer each one in its own paragraph, or in several paragraphs if it needs them (Ezra, 2026-10-28). A solution to "Which model underfits, which overfits most, and which would you choose?" has one paragraph for the underfitting model,

one for the overfitting model and one for the choice. A reader looking for one answer then finds it without reading the others. No automated check enforces this rule, because a script cannot tell how many questions a prompt asks.

Give each defined term its own div

Defining terms clearly covers what a good definition contains. For teaching materials, four further rules keep definitions findable.

- Replace a bolded keyword in running prose with a `#def-` div. A bolded keyword usually marks an inline definition.
- Define one term per `#def-` div. A term defined inside another term's div, or two terms joined in one title ("Regression and classification"), has no identifier of its own, so nothing can link to it.
- Never nest one theorem-type div inside another. The rule covers `#def-`, `#thm-`, `#lem-`, `#cor-`, `#prp-`, `#cnj-`, `#exm-`, `#rem-`, `#exr-`, and `#sol-`. A special case, such as a linear evaluation function, gets its own div after the general case, and the two link to each other.
- Keep only the definition in a `#def-` div. Put the properties of the defined object, such as the symmetry and idempotence of a matrix, in a separate `#thm-`, `#lem-` or `#prp-` div after it, with a proof where one is known. Each abstract div then gets its own example, as the rule below requires.

Every alternative name for the same concept stays in the div of the term it names and is treated like the primary name: it appears in the div title and is bold where it first appears in the definition. Name synonyms and near-synonyms where the term is defined, and say which one the notes use. A student who encounters one name here and another in a textbook cannot tell whether the difference matters. For a near-synonym, name the seam: critical points include the stationary points and also the points where the derivative does not exist.

State the notation and its alternatives

When a symbol or convention has more than one common form, state the one the notes use and name the others a reader will encounter elsewhere. Put the alternatives in the definition div or in a callout beside it. For example, some sources include 0 in the natural numbers and others exclude it, so write $\mathbb{N}_0$ or $\mathbb{N}^+$ to make the choice explicit.

Lead with the simplest correct form of a definition

Define a quantity in the form that needs the fewest symbols, and state the textbook form as a theorem that follows. For example, define variance as the mean squared deviation $\mathrm{E}\left[\mathrm{devn}(X)^2\right]$, where $\mathrm{devn}(X) = X - \mathrm{E}[X]$, and then prove that it equals $\mathrm{E}[X^2] - (\mathrm{E}[X])^2$.

Parsimony outranks convention when choosing a primary definition. The conventional form still appears in the notes, as a named result.

Reuse one written version of each result

When several courses or repositories have already written the same definition, theorem, or exercise, compare the versions on notation, decomposition, parsimony, and clarity. Use the best one, or merge them, and make it the single copy. Other places include or link to that copy instead of keeping a second wording that can drift.

Match datasets and assignments to the reference texts

Students can ask an AI assistant for a complete solution to any homework problem, so a problem is no more secure for never having been published (Ezra, 2026-11-04). Choose assignments for their fit with the course, not for novelty.

- Use as an in-class example the dataset that the reference texts and the other instructors of the course use for that topic. Depart from it only for a stated reason.
- Assign as homework the kinds of exercise that the reference texts and the other instructors generally assign for that topic, and use the same datasets they do.
- Keep assessment that must be secure, such as an exam, in a setting the instructor controls, because a take-home problem cannot be.

Check the reference text itself before naming its datasets. Open the chapter and list the dataset each section uses, rather than recalling them.

Use real data and compute every number

- Where an example needs data, use a real dataset instead of invented round numbers. Name its original source, link to where it is published, and cite it.
- Show a dataset before analyzing it, with a printed look at its first rows and its structure.

- Show the data where it is first mentioned. Put a table of a few rows, or a plot of the variables, directly after the sentence that names the dataset, before any discussion of its features, the methods to apply to it, or the questions it raises. A reader who has seen the data can picture what the next slides describe, while a reader who has not must hold an abstract description in mind until the data appears. If the data are too large to show, show a sample or a summary figure. Link later mentions of the dataset back to this first display.
- Never type or paste output. The page shows what the code printed when the page was rendered, and each number in the prose comes from inline code. A number typed into prose goes stale when the data or the code changes.

Example 0.1 (Showing the data at first mention)**.**

> ✗ Slides 3 to 9 discuss the spam data's 57 predictors, its class imbalance and the loss we will minimize. Slide 10 shows a table of the first rows.
>
> ✓ Slide 3 introduces the spam data and shows a table of its first rows and a plot of one predictor by class. Slides 4 to 10 discuss the predictors, the class imbalance and the loss, and refer back to that table and plot.

The ✓ version lets every later slide refer to something the reader has already seen.

Store structured data in structured formats

This rule applies to all teaching materials, in every course and every teaching repository.

- Keep a list or table of like items, such as courses, readings, channels or glossary terms, in a CSV, YAML or JSON file with one record per item. The page reads the file and builds the table or list with a short function, instead of holding a hand-written Markdown table.
- A data file can be counted, filtered, sorted and reused by another page, and a new row is a one-line change that a reviewer recognizes as a data change.
- List the file under the page's `resources:` when the site publishes it.
- Convert an existing hand-written table in its own pull request, and render the page before and after to confirm that the output text is identical.
- A short list that no other page reads can stay as prose.

Pair exploratory data analysis with plots

- Make each exploratory data analysis exercise or lab include graphical methods, not only numerical summaries (Ezra, 2026-11-04). Pair the summary with a plot of the same quantity:

  - **A distribution:** a histogram or boxplot.
  - **A trend or a relationship:** a scatterplot or time plot.

- Do not limit an exploratory analysis to counts and correlations. A plot shows the shape, the outliers, and the trend that one number hides.
- Give each plot the figure treatment in Make figures accessible.

Show each idea applied and in a figure

- Give every definition, method and result at least one applied example that computes it on real data, so the reader sees what the idea does before or beside the algebra.
- Follow every abstract div directly with a concrete `#exm-` div. The rule covers each `#def-`, `#thm-`, `#lem-`, `#cor-` and similar div. The example of a related result does not exempt a definition: a theorem that uses a definition gets its own example, and so does the definition. The ai-config rule links here.
- Give nearly every `#exm-` div a figure of its data, model or results (Ezra, 2026-10-23). When an example has no figure, state the reason in that div. Put the figure inside the div, or include a subfile whose name ends in `_fig` inside the div. When an example has nothing to plot, such as a single number, add `<!-- no-figure: reason -->` inside the div, replacing `reason` with why there is nothing to plot. The `check-exm-figures` workflow in `gha` reports, in a pull request, each example that has no figure and no `no-figure` comment (gha#1193).
- Give every div whose title mentions visualization (*visualize*, *visualizing*, *visualization*) a corresponding figure (Ezra, 2026-10-26). For an exercise, the figure may sit in the exercise or in its solution. The rule also covers any div that asks for a picture: a title or an exercise prompt that says *plot*, *sketch*, *draw*, *diagram*, or *contour* (in any inflection, such as *plotting* or *sketching*), or that asks the reader to mark points on a surface. The `check-visualization-figures` workflow in `gha` reports, in a pull request, each such div that has no figure in it or in its solution, and a `<!-- no-figure: reason -->` comment inside the div suppresses that report, as it does for an example (gha#1262).
- Give every topic at least one figure that shows the idea, such as the data with a fitted curve, a distribution, or a diagram of the procedure. A page of definitions with no figure and no data is incomplete.
- Prerequisite notes follow the same rule as course lectures. A prerequisite page that is only definitions and algebra needs an applied example and a figure.
- Place each figure directly after the sentence that first refers to it. Do not refer to a figure, table, or result that appears later in the document, because the reader encounters the reference before the thing it points to. This rule is for teaching materials, which are reading copies. A journal submission may collect its figures at the end, as Figures and tables describes.
- Make each figure follow the accessibility rules below.

- Discuss and cross-reference every figure in the prose (Ezra, 2026-11-02). Cite it with `@fig-id`, and say what the reader should see in it. A figure that no sentence cites or interprets is decoration.
- Nest a figure that belongs to one example or exercise inside that `#exm-`, `#exr-` or `#sol-` div (Ezra, 2026-11-02). A figure outside the div leaves the example or exercise incomplete when the reader meets it alone, and moves with the wrong text when the page is reorganized. Leave a figure that several examples share outside every div, and cite it from each. The `check-figure-placement` workflow in `gha` reports, in a pull request, each figure on a changed line that no prose cites and each figure placed directly after an example, exercise or solution div without being inside it. A `<!-- figure-placement: allow -->` comment on the figure or inside its div suppresses the report for a shared figure (gha#1378).

Follow every example with an exercise and solution

Reading a worked example is passive. A reader who has just seen one method applied learns more by applying it to a new case than by reading a second example. So each example gets a practice exercise, with its solution, right after it.

- After every `#exm-` div, add an `#exr-` div that asks the reader to do the same thing for a different case, followed by its `#sol-` div. The exercise changes the data, the function or the question, so the solution is not a copy of the example.
- The full sequence for a new idea is therefore: the exercise that poses the question and its solution, the definition or theorem, an example, and a practice exercise with its solution.
- More generally, give readers an opportunity to think at every stage: a question they can answer before the next paragraph, a calculation to try, a prediction to make before the code runs.
- A counterexample (see Defining terms clearly) can stand in for the practice exercise when the exercise would be "decide whether this case fits the definition".
- Give the practice exercise its own slug and follow the numbering and slide-break rules above.

Example 0.1 (An example followed by a practice exercise)**.**

✓

```
::: {#exm-sample-mean}
(the sample mean of one variable in a real dataset, computed by a code chunk)
:::

::: {#exr-sample-mean-new-variable}
Compute the sample mean of a different variable in the same dataset.
:::

::: {#sol-sample-mean-new-variable}
(the code chunk and its output)
:::
```

The example shows the method once. The exercise asks the reader to repeat it on a new variable before moving on, and the solution's numbers still come from running code (see Use real data and compute every number).

Write labs for both languages and any computer

These rules apply to all teaching materials, in every course and every teaching repository, not only to one course.

- Give labs and code exercises in both R and Python, side by side.
- Put each step's Python and R pair in its own tabset with `group="language"`. Start a new tabset whenever prose or a new step sits between code chunks, so the reader sees one step's code in one language at a time. The shared group makes one choice of language apply to every tabset on the page, as long as the tab headings use identical text in every tabset, such as always "R" and "Python".
- Write the R code with tidymodels, or show tidymodels and base R side by side. Do not give base R alone. ISLR-tidymodels-labs shows the style.
- When a lab follows a textbook's lab, use the book's exercises, data and fitting functions, in the book's order. For the ISL labs, fit Python models with `statsmodels`' `sm.OLS`, as ISLP_labs does. If an exercise departs from the book, say why in the lab.
- Never throw away an exercise you replace. Either keep both exercises, or keep the replaced one as an outtake: leave its file in place and stop including it.
- Tell students where to get the data.
- Point students only to places they can open. Students read a course's published site, not its repository, which is often private. So never send them to a file in the repository, to a `github.com` link into a private repository, or to "the copy in this repository". Link a data file from the published site, from a public repository or package, or from the data's original public source.

- Do not assume that students own a laptop. Assume only that they can use some computer, so setup steps must work on a shared or hosted machine.

Phrase optional steps as conditions

An instruction that applies to only some readers reads as an order to all of them. Put the condition first, then the instruction, as in "If you use a language other than R or Python, …".

- State the condition at the start of the sentence, as "If you …, do …". The reader who is not in that case can stop reading at the comma.
- Do not open with a bare imperative that only some readers should follow.
- Where a step can be skipped, say what the reader loses or what to do instead.
- Give each case its own sentence.

The labs in the reference text write optional work this way. They state the condition first and then the consequence: "If you have not installed ggplot2, then the code below will still run, but the plot will be less attractive" (James et al. 2021, sec. 10.9, p. 445). They give an alternative to the reader who is short of time: "If you are in a rush, then you could set B equal to a smaller value (e.g. B = 500)" (James et al. 2021, sec. 13.6, p. 587). They label a step that has no condition and say when to do it: "this step is optional and should be performed only if we want each gene to be on the same scale" (James et al. 2021, sec. 12.5, p. 543).

> ✗ In another language, use a port of `SyncRNG` to get the same data; without one, draw your own data from the same model.
>
> ✓ If you use a language other than R or Python, use a port of `SyncRNG` to get the same data. If no port exists, draw your own data from the same model; your numbers will differ slightly from the key's.

Introduce a specific case as a specific case

A sentence with an indefinite subject and a bare verb states a general fact: "A logistic regression classifier has a decision boundary" holds for every such classifier. Used for one particular case, the same form claims something false, and the reader must guess that you meant only one.

- Use "a" or "any" with a bare verb only for a statement that holds in every case.
- Introduce a specific case with "Consider a …", "Suppose a …", or "Let … be …", and give its particular values in the same sentence.
- Refer back to it with "the" or "this", and not with "a" again.

This applies Grice's maxims of Quality, because the sentence is not true as written, and of Manner, because it is ambiguous between one case and all of them (Grice 1975).

✗ A binary logistic regression classifier in two dimensions has parameter vector $[-2, 1, 0.5]'$.

✓ Consider a binary logistic regression classifier in two dimensions with parameter vector $[-2, 1, 0.5]'$.

Keep each step's code in one chunk, and put parallel chunks in a language tabset

This rule applies to every executed code chunk in teaching materials, in every course and every teaching repository, in every language.

- Put parallel versions of one computation in one language tabset, never one chunk after another. A reader should see one language at a time.
- Write one chunk for each step of the computation. A figure's code is one chunk, even when it takes ten lines.
- Split a chunk only for a reason that the reader can see:
  - **Own label or options**: the second part needs its own label, caption or chunk options.
  - **Separate output**: the second part shows output that the first part should not.
  - **Dependent prose**: prose between the parts needs the first part's output.
- Do not split a chunk to name its parts, or to keep it short. Comments name the parts (see Comment every code chunk to say what and why).
- Give a chunk that only stores values for later code or inline code `#| include: false`. When the chunk that computes those values is already hidden, merge the storing code into that chunk.
- Check for output that splits a chunk without a reason. In Python, reticulate prints the value of each top-level call that returns something other than `None` and shows the chunk as one code block per printed value, such as a `<matplotlib.lines.Line2D object at 0x...>` line after every `ax.plot(...)`. End each such call with a semicolon, or assign its value.

Present R results with modern packages

This rule applies to all R code in teaching materials, in every course and every teaching repository.

- Even when you fit a model with a base R function such as `lm()` or `glm()`, show its results with modern packages, not with base R output.

- Show coefficient tables with parameters (`parameters() |> print_md()`) or gtsummary (`tbl_regression()`), not with `summary()` or `coef(summary())` printouts.
- Draw figures with ggplot2, not with base graphics such as `plot()`, `lines()` and `abline()`.
- Draw model plots with packages built on ggplot2: sjPlot or ggeffects for predictions, and performance's `check_model()` for diagnostic plots.
- Show model fit statistics with performance's `model_performance()` or broom's `glance()`.
- Ezra's rme notes show the style.

Manipulate data with dplyr, not base R

This rule applies to all R code in teaching materials, in every course and every teaching repository.

- Select columns with dplyr's `select()`, not with `df[, c("a", "b")]` or `df[c("a", "b")]`.
- Filter rows with `filter()`, not with `df[df$x > 0, ]` or `subset()`.
- Add or change columns with `mutate()`, not with `df$new <- ...`.
- Drop rows with missing values with tidyr's `drop_na()`, not with `df[complete.cases(df), ]`.
- Stack data frames with `bind_rows()`, not with `rbind()`.
- Build data frames with tibble's `tibble()`, not with `data.frame()`.
- Chain steps with the native pipe `|>`.
- Do not nest function calls. Write `tibble(x, y) |> pander()`, not `pander(tibble(x, y))`, and `sizes |> lapply(f) |> bind_rows()`, not `bind_rows(lapply(sizes, f))`. Give a call that would sit inside an argument, such as `data = filter(df, train)` in a model call, a name on its own line first.
- Reading one value out of a data frame for inline code is fine, as is indexing a matrix or a vector. The rule is about selecting, filtering and building data frames.

Include code from files instead of copying it

This rule applies to code that appears in more than one place in teaching materials.

- When the same code appears in several pages, or in a page and a script, keep one source file and include it where it is shown. A copy in each place drifts when one copy is fixed and the others are not (see Reuse one written version of each result).
- Code that the page runs should be an executed chunk, as above. For code that is displayed but not executed on the page, such as a standalone script or a configuration file, Quarto's include-code-files extension fills a code block from a file. Add `filters: [include-code-files]` to the front matter, then write a code block with an `include` attribute, such as ```` ```{.python include="script.py"} ````.

- The extension also takes `start-line` and `end-line`, to show part of a file. Prefer a range that names the lines you want over a copy of those lines.
- The extension is a new dependency of the site: install it with `quarto add quarto-ext/include-code-files`, and commit the `_extensions` directory it creates.

Comment every code chunk to say what and why

This rule applies to all reader-facing code in teaching materials, in every course and every teaching repository, in every language.

- Put comments in each code chunk that explain both what the code is doing and why it does it.
- Leave out a comment that would give away the solution in an exercise prompt. Add the comments to the solution's code instead.
- A comment that only restates the code, such as `# add 1 to x` above `x <- x + 1`, gives the *what* without the *why*. Say what the step accomplishes and why the analysis needs it, for example `# center age so the intercept is the mean outcome at the average age`.
- Comment each step that a student could not reconstruct from its name, such as a transformation, a modeling choice, a tuning value or a random seed.

Set `code-fold` by what the code does and whether the prose mentions it

This rule applies to every executed code chunk in teaching materials, in every course and every teaching repository.

Set `#| code-fold:` on each chunk to `show` or `true`. Never set `false`: it removes the toggle and gains nothing, because `show` also keeps the code and its output together. The choice depends on one question: does the prose mention the code?

- `#| code-fold: show` when the prose mentions the code. The choice of `show` holds whether the code produces output outside the chunk or not, and a chunk that defines functions or objects that later chunks use counts as producing output. The code starts open, and a reader who wants to focus on the output can hide it.
- `#| code-fold: true` when the prose does not mention the code. The code starts hidden, and a reader who wants it can open it.

For example:

- A chunk that fits a model the text walks through and prints its coefficients is `show`.

- A chunk that draws a figure the text discusses is `show`.
- A chunk that defines the helper functions a section introduces is `show`.
- A chunk that loads a package the text never mentions is `true`.

Put a full command in a code block, not in inline code

This rule applies to all teaching materials that name commands, in every course and every teaching repository.

- Put a full command that a reader will run, such as an install command, in its own fenced code block with a language, such as `r` or `python`. A rendered code block has a copy button, and inline code does not.
- Do not run the block when its output is not the point. Open the fence with the language alone, as in `r`, and not in braces, as in `{r}`, so Quarto shows the code and does not evaluate it.
- Keep inline code for a name inside a sentence, such as the name of a package, a variable or an argument.
- Introduce the block with a sentence that ends in a colon. Do not follow a semicolon or a colon in the sentence with inline code that the reader has to select by hand.

Example 0.1 (Moving an install command out of a sentence)**.**

✗ The R tabs need `readr`, `dplyr`, `ggplot2` and `purrr`: `install.packages(c("readr", "dplyr", "ggplot2", "purrr"))`.

✓ The R tabs need `readr`, `dplyr`, `ggplot2` and `purrr`:

```r
install.packages(c("readr", "dplyr", "ggplot2", "purrr"))
```

The package names stay inline because they sit in the sentence. The install command moves to a block, so the reader copies it with one click.

Introduce every function in a programming primer

This rule applies to all teaching materials that run code, in every course and every teaching repository.

- Give the course a programming primer that introduces each package function the course's chapters call. A student encounters the function there first, in a short note, a worked example and an exercise, and then sees it used in the chapters.

- Keep the primer complete by computing the list of functions, not by remembering it. Scan the code chunks of the chapters for the functions they call, and compare the list with the functions the primer mentions. Run the scan when a chapter adds a function, and again before a term starts.
- Add the missing functions in separate pull requests, one per package. A pull request that adds a function to a chapter adds it to the primer too, or links the primer entry that already introduces it.
- Functions that a chapter defines for itself are not package functions. A chapter introduces those where it defines them.

Make figures accessible

- Wrap each figure, including an interactive one, in a `#fig-` div with a caption.
- Give every plot alt text. For Observable Plot, set `ariaLabel`.
- Show the code that makes each figure, and let readers fold it away. Code folding does nothing for a cell that does not echo its code, so do not set `echo: false` on a cell you want students to read.
- Do not cross-reference an interactive figure from prose, because the PDF format renders no figure for it, so the reference has no target there (observed when rendering lecture notes to PDF).

Split a page before it grows too long

A page should load and finish rendering its math in under 5 seconds, and its scroll bar should not shrink to its smallest length (Ezra, 2026-10-31). The browser typesets every formula on a page, so a page with hundreds of formulas takes seconds to finish. A page tall enough to shrink the scroll bar to its minimum leaves the reader no way to see where they are in it.

- Split a page that fails either test into several pages, one coherent topic each. The math prerequisites page on measures, for example, became six pages for this reason.
- Split along the headings that already divide the page, and give each new page its own title and a short introduction.
- Keep the old section ids as stubs that link to the new pages, so links from other pages and other sites still reach the content.
- Link between the new pages where one page uses a concept that another page defines.
- Do not delete anything to shorten a page: move it to the page where it belongs.

The `check-page-length` check in `gha` (gha#1377) warns when a pull request adds to a page that, with its includes expanded, has more than 600 math expressions or 5000 source lines. Those limits are proxies, since the check does not run a browser. It warns only on pages a pull request touches, so split a long page when you next edit it, and do not sweep the others.

Keep sibling headings to about five

Aim for at most about five sibling headings under one parent heading, and split or group the headings when there are more than seven (Ezra, 2026-11-02). Sibling headings are headings of the same level under the same parent. The top-level headings of a page or book chapter count as siblings under the page.

A reader who scans a list of headings holds the whole list in mind while choosing among them. Miller's "magical number seven, plus or minus two" is the classic estimate of how many items a person can hold at once. Cowan's reconsideration puts the capacity nearer four chunks. Those estimates concern memory, and a reader can look at a table of contents again, so the limit is a guideline and not a hard rule. Aim for five so that the guideline still holds when a few headings are added later.

- Group related sibling headings under a new parent heading that names what they share.
- Split a page with too many top-level headings into several pages, as in Split a page before it grows too long.
- Leave a list of parallel parts, such as one heading per lab or per week, alone when the parts have no natural grouping, and say in the introduction how many there are.
- Do not delete a heading to meet the guideline: move its content under the heading it belongs to.

The `check-heading-siblings` check in `gha` (gha#1390) warns when a pull request adds a heading to a group of more than seven siblings. It warns only on pages a pull request touches, so regroup a page when you next edit it, and do not sweep the others.

Fold a one-paragraph section into its neighbor

Give a heading to a point that can stand alone, and fold a section of a single short paragraph into the section it qualifies or follows (Ezra, 2026-11-02). A heading that opens a one-paragraph section adds an entry to the list of headings and gives the reader nothing to find there. This is a guideline, not a minimum length: a definition, a distinct step or a named result can be short.

- Merge a caveat into the section it qualifies. If it qualifies a whole list, put it in the closing section on how to use the list.
- Use a callout for an aside to one nearby claim.
- Use a footnote for a minor qualification.
- If a point is one of several parallel points, make it an item in the parent's list.
- Do not delete the content: move it to the section it belongs to.

Structure slides for the reader

- Check the rendered slides, not only the web page. Slides need the same boxes and colors on theorem-type and callout divs that the web page has, so a reader can see where a definition ends and the commentary begins.
- Put a list of three or more phrases in a bullet list, as Put lists in bullet points describes.
- Show every proof on the slides. Do not wrap a proof in `{.content-hidden when-format="revealjs"}`. Remove an existing wrapper from a proof when you edit the file for another reason, and do not sweep the others (Ezra, 2026-10-29).
- Put each named example in an `#exm-` div.
- Put on a slide only what you want the audience to look at. Put an example in an `#exm-` div and short commentary in a `#rem-` div, because a `::: notes` div is plain text without a box on the web page.
- Put a long paragraph of commentary in a remark that is also speaker notes: `::: {#rem-name .remark .notes}`. Reveal.js shows it only in speaker view, so the audience does not have to read it, and the web page shows it as a boxed remark. Readers who did not attend the lecture use the web page, so nothing is lost, and nothing is said twice. Give a long paragraph a separate short `#rem-` div only when the slide needs a takeaway to look at.
- Add structure only where content has a shape. Do not put a heading over a single item, or over the same topic as the heading just before it.
- Keep body text off section and title slides. Reveal.js centers those slides vertically and does not scroll them, so long text is clipped at the top and bottom of the screen (observed in lecture slides rendered with Quarto). Put a slide break after the heading and move the text onto content slides.

Give each concept in a note sheet its own row

A note sheet is a reference table of formulas with brief labels and no prose. This is one case of building every unit from smaller atomic units. Give each concept its own row. A row holds one formula and one short label.

- Never put two definitions in one row.
- Never merge rows to save space. Find the room another way, such as a smaller table font or a second table beside the first.
- Never put notation or assumptions inside a label. Give each assumption its own row, and state the notation once in its own row.
- Write each formula in its most compact form, built from pieces that have their own rows (see Making expressions concise).

The same rule applies to every other reference table, such as a table of notation.

Example 0.1 (Splitting a merged row)**.**

✗ Row: `Dot product, sample mean`, with both formulas in one cell.

✓ Rows: `Dot product` with its formula, then `Sample mean` with its formula.

The ✗ row makes the reader find where one concept ends and the next begins. The ✓ rows let the reader find one formula by one label.

Talk about the ideas, not the notes

When a term is defined on another page or in another site, state the idea in the sentence and link the term. Do not write about where the definition lives.

- Write "the bias of an estimator", linked to its definition, not "the statistics notes define the bias of an estimator".
- Do not write "see the probability notes" or "as in our notes". The link already says where the term lives.
- Name a remark by the idea it holds, such as "Bias of a prediction", not "Bias of a prediction in the statistics notes".

Students read concepts, not the structure of the site. A sentence about which notes define a term adds no content, and it goes stale when the definition moves.

The same holds inside a page. Focus on the content of the notes, not on their structure (Ezra, 2026-11-02). When a part refers back to an earlier exercise, example, or result, name the specific objects it reuses instead of pointing at the exercise as a whole. "Use the setup of Exercise 1" makes the reader reread the exercise to learn which data, variables, or model the part needs.

ℹ Name what the back-reference reuses

- ✗ "Use the setup of Exercise 1."
- ✓ "Use the WCGS data, with outcome Y (coronary heart disease) and predictor X (behavior pattern) from Exercise 1."

Keep the cross-reference so that the reader can look up the full statement, and add the names of the reused data, variables, parameters, or model. When the notes define a dataset in its own callout, link the dataset by that callout's id instead of describing it again, and put the link on the dataset's name. Quarto renders `@dat-wcgs` as "Dataset 1" in the link text, so write the dataset name as the link text, as in `[WCGS data](#dat-wcgs)`, not "the WCGS data (`@dat-wcgs`)" (Ezra, 2026-11-04; see "Link the words, not a trailing citation" in

`chapters/citations-evidence.qmd`). Prefer a self-contained sentence when the part needs only one or two of those objects.

Write grading rubrics as checklists

Structure the rubric for most STEM questions as a checklist (Ezra, 2026-10-26). A grader then marks each item as earned or not, and the points are the sum of the earned items. A rubric written as a paragraph makes the grader reconstruct the items, and two graders reconstruct different items.

- Write one checklist item per observable criterion, and put its points first.
- Make the points of the items sum to the question's points, and state that total in the line that introduces the checklist.
- Write each item so that it is either met or not met. Name the answer or the evidence that meets the criterion, as in "cites the gap between training and CV MSE".
- List an acceptable alternative answer as a sub-bullet without points under the item it also satisfies, so the items still sum to the question's points. Do not put the alternative in a separate paragraph.
- Keep the rubric in the solution, after the answer, so a course that withholds solutions from students withholds the rubric too.
- Use another structure when a question has no separable criteria, such as a single holistic judgment of a proof's rigor.

Example 0.1 (A rubric as a checklist)**.**

> ✗ Grading (6 points): two for $\vec{r}$, one for $\underset{\sim}{x} \cdot \vec{r} = 0$, three for "orthogonal" with its link to the normal equation.
>
> ✓

```
Grading (6 points):

- [ ] 2 points: gives $\vresid = \tp{[-1, -1, 1]}$.
- [ ] 1 point: computes $\dprod{\vx}{\vresid} = 0$.
- [ ] 3 points: says $\vresid$ is orthogonal to $\vx$
  and links this orthogonality to the normal equation.
```

The checklist shows the grader each item to mark and lets a script check that the points sum to the total.

Credit other courses without summarizing them

- Write your own version of another course's material, and credit the source in a collapsed `.callout-note` titled "Source", not in running prose. Adapting another course's material gives the details, including when a `::: notes` div is a safe alternative.
- Quote the primary source directly, and check the wording and page against the original. Quote the original explains why.
- Title a section by its topic, not by its author or course. Put the content first explains why.
- Where the book is the subject rather than its authors, name the book by its title, because a bare citation key renders as the author list.

Finish each unit completely

Complete each sentence, step, term, result and exercise, and do not leave part of it implied, merged into a neighbor, or half done. A reader should never have to finish a unit for you. Several rules in this guide are cases of this one:

- **Sentence.** Write each sentence in words or in math, not both, so a hybrid becomes one complete sentence of each kind.
- **Step.** Give each operation its own line and justification, and do not "cancel" several operations as one step.
- **Term.** Give each defined term its own div, and keep equalities and equivalences out of the definition by stating them as results.
- **Result.** Restate a result first derived in a solution as a theorem after the exercise, so the result stands alone.
- **Exercise.** Split an exercise with several parts, so each exercise asks for one thing.

When you catch yourself taking a shortcut, finish the unit instead. A shortcut saves the writer a few lines and costs every reader the work of rebuilding them.

Rules that apply to all writing

These rules from the other chapters apply to teaching materials as to any other writing:

- start a sentence with its subject;
- name the referent instead of a bare demonstrative;
- say which thing;
- state the literal claim instead of an idiom;
- avoid clichés.

Freeman, Scott, Sarah L. Eddy, Miles McDonough, et al. 2014. "Active Learning Increases Student Performance in Science, Engineering, and Mathematics." *Proceedings of the National Academy of Sciences* 111 (23): 8410–15. https://doi.org/10.1073/pnas.1319030111.

Grice, H. Paul. 1975. "Logic and Conversation." In *Syntax and Semantics 3: Speech Acts*, edited by Peter Cole and Jerry L. Morgan. Academic Press.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning: With Applications in r*. 2nd ed. Springer. https://doi.org/10.1007/978-1-0716-1418-1.