EC7422: Data Science for Economic Analysis, Stockholm University
Instructor: Adam Altmejd, adam.altmejd@su.se

Lecture 5 · 2026-04-23

Catching Bugs and Coding Assistance

Today

- Why code breaks and what to do about it
- How to fail early
- Getting coding assistance from your IDE and agents

Why this lecture now

- The hard part is no longer just writing code
- The hard part is keeping track of objects, files, assumptions
- This is where people start wasting hours
- Good workflow habits are cheap

> Once a project is more than a few expressions long, most of the time goes into understanding what is happening rather than writing new code. The tools and habits in this lecture are what make that time shorter and the surprises smaller.

Positron

- We will be using Positron today, because of its built-in debugger.
- Positron is a VS Code fork, so it will look familiar.
- If you prefer to stay with VS Code you can use the R Debugger extension but it can be a bit finicky to get going.

Errors everywhere

Debugging

Pre-empt the bugs: fail usefully

IDE code assistance

Agentic development

Errors everywhere

Programming is a process of constant errors.

> Errors are not a sign that something is off about how the code was written. They are the normal feedback loop. The question is how quickly they can be located and understood.

Getting unstuck

```
Error in if (x) { : missing value where TRUE/FALSE needed
```

```
Error: object 'panel_2023' not found
```

```
Error in library(httr2) : there is no package called 'httr2'
```

```
Error in [ : subscript out of bounds
```

Your job: debug the code and find the *bad assumption*.

> Error messages usually name two things: what went wrong, and where. Reading them in that order is the first step. The *bad assumption* is whatever the code took for granted that turned out not to hold — a missing object, a wrong type, a file that is not where it used to be.

Running code is not necessarily *working* code

- Errors stop the script (good!)
- But bad assumptions do not always cause errors
- A silent bug is much worse
- In data work, incorrect output can still look great

> Errors are the easy case. A silent bug passes every check, produces a clean-looking table, and often only surfaces later when someone else tries to reuse the output. Much of the rest of this lecture is about making bugs loud.

Debugging is a loop

1. Reproduce the problem
2. Isolate the first bad step
3. Inspect what exists now
4. Compare expected versus actual
5. Change one thing
6. Retry

> The common mistake is to change several things at once when frustrated. Changing one thing per iteration keeps the signal clean: if behavior changes, the last edit caused it; if not, undo and try something else.

Make it repeatable and small

- Get to one command or one script that fails every time
- Strip away code that is not part of the problem
- Keep the failing example fast to rerun
- Pay attention to inputs that fail and inputs that do not
- Small, repeatable bugs are much easier to debug and share

> A minimal reproducible example is worth the time it takes to build. Stripping away unrelated code often reveals the bug on its own, and a small example is easy to share — with a colleague or with a language model.

What to inspect first

- What file or object is this line using?
- What class does the object have?
- What values do I actually have?
- Which line first creates something surprising?
- What changed since the last time this worked?

> The shift from “what is the code supposed to do” to “what does the data actually look like right now” is where most bugs get caught. The commands on the next slide are how to answer these questions in R.

Useful inspection commands

```
str(x)                # structure: type, dimensions, first values
class(x)              # what type of object?
head(x)               # first rows
names(x)              # column names
dim(x)                # rows × columns
```

`str()` is probably the most useful inspection command in R.

When inspecting large objects with `str()` use its second argument `max.level` to control how deep it goes. For example, `str(x, max.level = 1)` will show the top-level structure without going into nested details.

Remember to start a clean session

- Restart R to clean the slate
- Rerun the script from the top
- If it works in a clean session, it is more likely to be truly fixed

> Hidden state from an earlier session is the single most common cause of scripts that seem to work and then do not. A restart forces the script to stand on its own, which is also the condition under which any collaborator — human or otherwise — will run it.

Don’t use `.RData`

R offers to save your workspace on exit and reload it on startup

- Objects from last week’s session silently reappear
- This is the top cause of “huh, this worked before?”

Turn it off.

In RStudio, go to Tools → Global Options → General, and set “Save workspace to .RData on exit” to “Never”. Positron already defaults to off. To disable globally (including in VS Code), you could replace `q()` in your `.Rprofile` with:

```
q <- function(save = "no", ...) quit(save = save, ...)
```

Example 1: Selecting a data frame column

```r
col_subset <- mini_panel[mini_panel$year == 2023, "unemployment_rate"]
```

Let’s inspect:

```r
names(col_subset)
```

```
NULL
```

```r
class(col_subset)
```

```
[1] "numeric"
```

```r
str(col_subset)
```

```
 num [1:4] 12.1 7.8 10 10.6
```

- Selecting a single column returns a vector
- An issue if you expected a data frame

> The `[` operator on a data frame drops to a vector when exactly one column is selected, and stays a data frame when two or more are. That default saves a few keystrokes in interactive use but regularly surprises downstream code that expected a table.

Example 1: Selecting a data frame column (cont.)

`drop = FALSE` forces R to keep the data frame structure:

```r
col_subset_df <- mini_panel[
  mini_panel$year == 2023,
  "unemployment_rate",
  drop = FALSE
]
```

```r
names(col_subset_df)
```

```
[1] "unemployment_rate"
```

```r
is.data.frame(col_subset)
```

```
[1] FALSE
```

```r
is.data.frame(col_subset_df)
```

```
[1] TRUE
```

Asking for help

Sometimes you cannot figure out or resolve the error on your own.

I was planning on talking about Stack Overflow here, but...

Asking for help (from robots)

> Stack Overflow's question volume has fallen sharply since capable LLMs became widely available. The same community of answers now lives inside the model, and asking a language model directly is often faster and more specific to the exact code at hand.

https://blog.pragmaticengineer.com/stack-overflow-is-almost-dead/

Errors everywhere

Debugging

Pre-empt the bugs: fail usefully

IDE code assistance

Agentic development

Debugging tools

- Simple issues can be debugged by printing and inspecting
- For more complex problems, R provides powerful debugging tools:
  - → `traceback()` to see the call stack after an error
  - → `browser()` to step through code interactively
  - → `debug(FUN)` to run `browser()` on `FUN()` calls

We use `traceback()` to find *where* the error happened and `browser()` (or `debug()`) to inspect *what* is going on at that point.

Example 2: A simple `traceback()`

```r
# in_class_examples/lecture_5/01_traceback_test.R
prepare_series <- function(x) growth_rates(trimws(x))
growth_rates <- function(x) log_change(x)
log_change <- function(x) diff(log(x))
average_growth <- function(x) mean(prepare_series(x))

average_growth(c("100", "120", "oops", "150"))


f <- function(x) {
  apply(x, 1, mean)
}
```

```
Error in log(x) : non-numeric argument to mathematical function
```

Reading a traceback

- `traceback()` prints the *call stack*
- One row per function call, starting with the most recent

The error message tells you **what** went wrong; the traceback helps you locate **where** the error occurred.

> The bottom of the trace is the top-level call from the script; the top is where the error actually surfaced. Reading upward from the bottom shows the path of calls; reading downward from the top points to where to look first.

Example 2: A simple `traceback()` (cont.)

```
1 traceback()
```

```
Calls: source ... prepare_series -> growth_rates -> log_change -> diff
11: (function ()
    traceback(2))()
10: diff(log(x))
9: log_change(x)
8: growth_rates(trimws(x))
7: prepare_series(x)
6: mean(prepare_series(x))
5: average_growth(c("100", "120", "oops", "150"))
4: eval(ei, envir)
3: eval(ei, envir)
2: withVisible(eval(ei, envir))
1: source("/home/runner/work/datascience-course/datascience-
course/in_class_examples/lecture_5/01_traceback_test.R",
       chdir = TRUE)
```

Example 3: A more realistic `traceback()`

```r
# in_class_examples/lecture_5/02_traceback_realistic.R
get_municipality_history <- function(panel, municipality_code) {
  panel[panel$municipality_code == municipality_code, , drop = FALSE]
}

get_latest_rate <- function(municipality_panel) {
  latest_year <- max(municipality_panel$year)

  municipality_panel[
    municipality_panel$year == latest_year,
    "unemployment_rate"
  ][[1]]
}

get_reference_rate <- function(municipality_panel, reference_year) {
  reference_rate <- municipality_panel[
    municipality_panel$year == reference_year,
    "unemployment_rate"
  ]

  reference_rate[[1]]
}
```

⋯ continues below

```
Error in reference_rate[[1]] : subscript out of bounds
```

This example shows how `traceback()` helps find the bad assumption when the error message alone is uninformative. The failing call sits near the top of the stack; tracing back through its callers is what reveals where the surprising value first came from.

Example 3: A more realistic `traceback()` (cont.)

```
1 traceback()
```

```
Calls: source ... compute_change_from_reference -> get_reference_rate
8: (function ()
   traceback(2))()
7: get_reference_rate(municipality_panel, reference_year)
6: compute_change_from_reference(municipality_panel, reference_year)
5: build_municipality_report(panel, municipality_code = "0180",
       reference_year = 2020)
4: eval(ei, envir)
3: eval(ei, envir)
2: withVisible(eval(ei, envir))
1: source("/home/runner/work/datascience-course/datascience-
course/in_class_examples/lecture_5/02_traceback_realistic.R",
       chdir = TRUE)
```

After traceback: inspection

- `traceback()` shows you where the error happened
- Especially useful with nested (custom) function calls
- Next step: inspect the state at that point
- Simple version: `print()` objects at critical lines inside functions
- Advanced version: use `browser()` to step through the code interactively

Both `print()` and `browser()` have their use! `print()` is quick and good for a few checks, while `browser()` is more powerful for stepping through complex code. You can start with `print()` and switch to `browser()` when you need more control.

A broken function

```r
classify_unemployment <- function(panel, threshold = 10) {
  latest_year <- max(panel$year)
  current <- panel[panel$year == latest_year, ]

  current$risk_group <- ifelse(
    current$unemployment_rate <= threshold,
    "high",
    "low"
  )
  current
}
classify_unemployment(mini_panel, threshold = 10)
```

```
   municipality_code municipality_name year unemployment_rate risk_group
3               0114    Upplands Väsby 2023               12.1        low
6               0180         Stockholm 2023                7.8       high
9               0184             Solna 2023               10.0       high
12              0380           Uppsala 2023               10.6        low
```

Notice anything wrong?

Silent bugs are much harder to detect. Most bugs are silent, but in a few slides we will talk about fail loudly to catch them.

The comparison is inverted: rates at or below the threshold are labelled `"high"`, so municipalities with *low* unemployment end up flagged as the high-risk group. The code runs, the output looks clean, and nothing warns that the labels are swapped. This is the prototype of a silent bug.

Use `browser()` to step through the code interactively

Drop a call to `browser()` into the function where things look suspicious. When R hits it, execution pauses and the prompt changes to `Browse[1]>`. From there:

- `n`: next expression
- `s`: step into (function)
- `f`: finish current function
- `c`: continue until the end or the next breakpoint
- `Q`: quit

```
  > classify_unemployment(mini_panel, threshold = 10)
Browse[1]> ls()
[1] "current"     "latest_year" "panel"       "threshold"
Browse[1]> threshold
[1] 10
Browse[1]> latest_year
```

> The prompt changes to `Browse[1]>` whenever R enters the browser. At that prompt, typing a variable name prints its current value; the letter commands above move execution one step at a time. The same shortcuts drive the GUI debugger on the next slide.

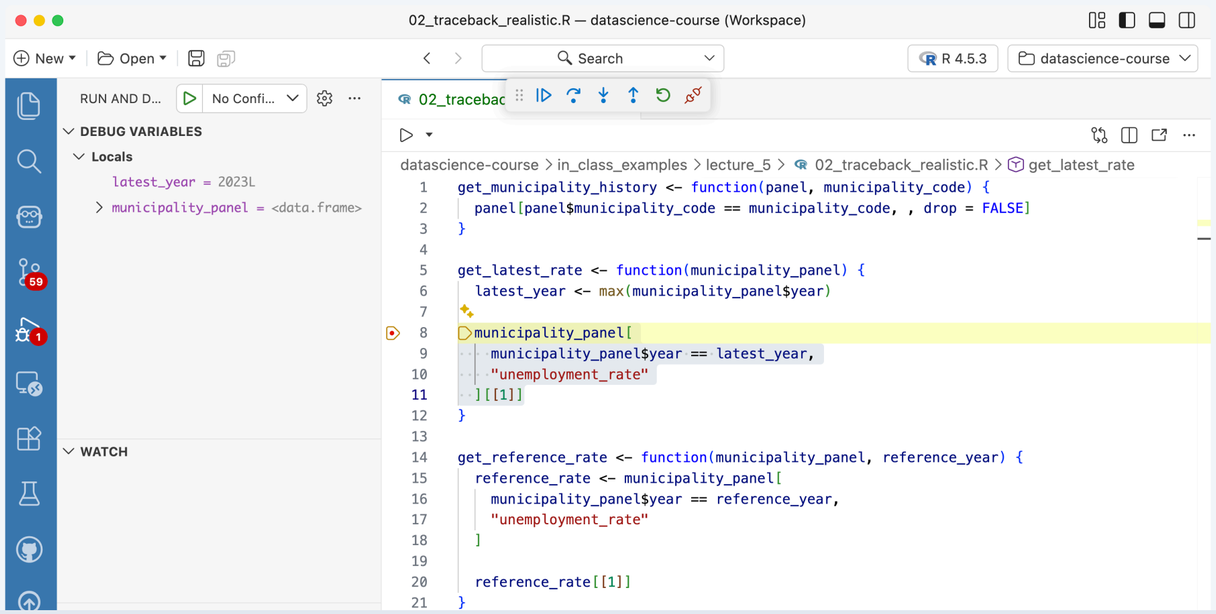
Enter the browser on a function call: `debug()`

- `debug(<function_name>)` browses the function every time it's called
- `debugonce(<function_name>)` enters `browser()` only on the next call
- `undebug(<function_name>)` turns debugging off

> Forgetting `undebug()` is a common footgun, default to `debugonce()` if you just want an interactive pass through the function.

IDE debugger

- VS Code, Positron, and RStudio all have built-in debuggers
- GUI + breakpoints instead of `browser()` calls. Same underlying tools.



Positron and RStudio come with built-in R debugging. To get debugging in VS Code, you need to install the R Debugger extension. It can be a bit finicky which is why I recommend you use Positron for this lecture.

The GUI debugger uses the same underlying engine as `browser()`, with breakpoints placed by clicking the editor gutter instead of editing the source. Breakpoints persist across reruns, which becomes convenient once the same bug has been chased more than once.

A useful fallback: `recover`

```
1 options(error = recover)
```

- Good when you are unsure where the error is happening
- On error, R shows the call stack and lets you inspect

When you are done, set `options(error = NULL)` to reset the error behavior to the default.

Errors everywhere

Debugging

Pre-empt the bugs: fail usefully

IDE code assistance

Agentic development

Fail early and loudly

Do you know what the code is supposed to do? Why not verify that it actually does?

- Adding checks, errors, messages, assertions, and tests is a cheap way to catch problems early.

> We'll focus on `stop()`, `warning()`, and `message()` here, but there are also packages like `assertr`/`validate` for more formal checks and `testthat` for unit testing. See the extra slides for a brief introduction to these tools.

`stop()`, `warning()`, `message()`

Use:

- `stop()` to generate error
- `warning()` to warn about risky state
- `message()` to inform about progress

```
panel_path <- file.path("data", "mini_panel.csv")
message("Reading ", panel_path)
```

```
Reading data/mini_panel.csv
```

```
if (!file.exists(panel_path)) {
  stop("Can't find ", panel_path)
}
```

```
Error:
! Can't find data/mini_panel.csv
```

```
if (anyNA(c(mini_panel$municipality_code, NA))) {
  warning("Some municipality codes are missing")
}
```

```
Warning: Some municipality codes are missing
```

Turn warnings into errors

```
1 options(warn = 2)
```

- Temporarily turns warnings into errors
- Useful when a warning is the first visible sign of a real problem
- Then you can use the usual traceback and debugger tools

Reset with `options(warn = 0)` afterwards.

Conditional `browser()`

```
classify_unemployment <- function(panel, threshold = 10) {
  latest_year <- max(panel$year)
  current <- panel[panel$year == latest_year, ]

  if (threshold < 0) {
    browser()
  }

  current$risk_group <- ifelse(
    current$unemployment_rate <= threshold, "high", "low"
  )
  current
}
```

- Useful when the bad case is rare
- Like in a loop where only certain iterations are problematic

Put checks in the code

```
required <- c("municipality_code", "year", "unemployment_rate")
missing <- setdiff(required, names(panel))
if (length(missing) > 0) {
  stop("Missing columns: ", paste(missing, collapse = ", "))
}
```

Issue `stop()` or `warning()` when:

- Required columns are missing
- Key columns have duplicates
- Values are not in a plausible range
- Missing values appear where they should not
- Row counts collapse or explode
- Input file paths are wrong

> Each item on this list corresponds to a mistake that is easy to make and hard to spot after the fact. Writing the checks directly into the script — rather than remembering to run them by hand — makes a pipeline that keeps working as the input data changes over time.

Municipality panel checks

- Municipality code should not disappear
- Year should stay in the expected range
- Unemployment rate should not be negative
- Keys should not duplicate by accident
- Filters should not silently return zero rows unless that is expected

Debugging versus testing

Debugging

- Manual
- Find out why this failed now
- Trace the first bad assumption
- Inspect current state
- Fix the immediate bug

Testing

- Automatic
- Catch the same bug earlier
- Use known inputs
- Verify expected outputs

After a bug fix, add the smallest check that would have caught it earlier.

> The two activities are complementary, not alternatives. Debugging catches a bug the first time; a test captures the fix so the same bug cannot come back quietly. Together they bound both ends of the feedback loop.

Errors everywhere

Debugging

Pre-empt the bugs: fail usefully

IDE code assistance

Agentic development

Code formatting with Air

- Air cleans up your code on save.
- Consistent indentation, spacing, line breaks
- Air extension in Positron or VS Code

```
# before
df<-read.csv( "data.csv" );summary(df[ ,c("x","y") ])

# after
df <- read.csv("data.csv")
summary(df[, c("x", "y")])
```

Air is really new. There is an older (and much slower) code formatter called `styler` that you can use if Air does not work.

Code linting with Jarl

- Flags suspicious patterns *before* you run the code.
- Catches what errors and tests won't:
  - → `any(is.na(x))` → suggests the faster `anyNA(x)`
  - → Unreachable code after `return()`, `stop()`, or `break`
- Jarl extension in Positron or VS Code
- Air makes code *look* consistent, Jarl makes it *behave* better

```
f <- function(x) {
  `apply(x, 1, mean)` is inefficient. Use `rowMeans(x)` instead. Jarl(matrix_apply)
  View Problem (⌥F8)   Quick Fix... (⌘.)   Fix (⌘I)
  apply(x, 1, mean)
}
```

Jarl is even newer, and in active development. If you want a more robust option (that is a lot slower), use `lintr` instead.

AI-assisted code completions

- Start writing code and `Copilot` suggests the rest in grey
- `Tab` to accept, `Esc` to dismiss, or keep typing and the suggestion adapts
- Suggestions use the whole file as context, not just the current line
- Comments steer what you get:

```
# read your_file.csv from raw_data
```

gives you a suggestion like:

```
read.csv("raw_data/your_file.csv")
```

> Completions are graded autocomplete: the suggestion updates as the next few characters are typed, and the surrounding file shapes what gets offered. That makes small, descriptive comments effective as a steering signal — the comment is not just documentation, it is part of the prompt.

Inline Copilot interactions

- Highlight code and click the 💡 (or press `Cmd/Ctrl + I`) to ask Copilot to **modify** or **review**
- On R console errors, the Positron assistant can suggest a fix:

```
Error in `file()`:
! cannot open the connection
▶ Show Traceback  ✧Fix  ✧Explain
```

From completions to agents

- **Completions** suggest the next line. You stay in control of every keystroke.
- **Agents** run multi-step tasks: read files, write files, run commands, loop until done.
- Same underlying models — the difference is how much rope you give them.

> The tradeoff scales with the amount of rope. Autocomplete is cheap to review one line at a time; an agent session can touch many files at once and needs a correspondingly larger review effort on the diff it produces.

Errors everywhere

Debugging

Pre-empt the bugs: fail usefully

IDE code assistance

Agentic development

What is an AI agent?

A language model given **tools**: read files, write files, run commands, read the output — then loop.

- Copilot agent mode in VS Code / Positron (free with GitHub Education)
- Paid standalones: Claude Code, OpenAI Codex, Cursor
- Recent revolution: models have not changed; their *harness* has

Harness = the code that lets the model call tools and read their results. A model that can run your tests is qualitatively different from one that cannot.

Working with agents: what changes

- **Commit more often.** Every agent session starts from a clean diff you can throw away
- **Scope each task.** One file, one fix — not “refactor the project”
- **Read the diff.** Every line. Don’t merge what you can’t explain
- **Close the loop with checks.** Tests, `air`, `jarl`, CI — the agent sees failures and corrects itself

Context is everything

Vague

```
Fix this traceback.
```

Scoped

```
In 02_traceback_realistic.R, explain
why get_reference_rate() fails for
reference_year = 2020 and propose the
smallest fix. Don't rewrite the script
or add packages.
```

Good context has four parts:

- the file
- the error
- the expected output
- a stop point

> Agents can only act on what the context contains. Stating the file, the error, the expected output, and where to stop turns a vague request into something with a concrete success criterion — closer to writing a well-formed issue for a colleague than to giving a command.

`AGENTS.md`: house rules for the repo

Agents read instruction files before acting. Tell them which packages you prefer, how the code is structured, how to run the tests.

```
<!-- in_class_examples/lecture_5/agent-example/AGENTS.md -->
# Agent guide

A small playground for practicing Copilot Chat in **agent mode** inside VS Code
or Positron. Each file in `scripts/` has a comment block at the top with the
task and a suggested prompt.

## How this repo steers the agent

Three files tell the agent how to behave here. Open them, read them, edit them
and watch the behaviour change:

- `AGENTS.md` — this file. Read by most agents (Copilot, Claude, Codex, Cursor).
- `.github/copilot-instructions.md` — included automatically in every Copilot chat.
- `.github/instructions/r-scripts.instructions.md` — scoped to `scripts/**/*.R` via `applyTo`.

## Rules for the agent

- Stay in base R unless the script already imports another package.
```

`AGENTS.md` at the repo root is the emerging convention — different tools read slightly different filenames (Claude uses `CLAUDE.md`), but the shape is the same. Short and opinionated works better than exhaustive: the file is read on every turn, so length competes with the actual task for model attention.

Point the agent at the right files

`AGENTS.md` doesn't have to say everything — it can point.

```
- Data dictionary: `data/README.md`
- Style conventions: `docs/style.md`
- Run the full pipeline: see `Makefile`
```

- Keeps the root file short and scannable
- Agent loads detail only when the task needs it
- Think of it as a table of contents for the repo

Agents read text like humans do

- No special format required — READMEs, source, CSV headers, `.Rprofile`
- A good comment or docstring helps both your teammate *and* the agent
- The flip side: outdated or misleading docs mislead the agent too
- Write for humans first; agents come along for free

Skills: packaging repeatable tasks

A **skill** is a reusable bundle of instructions (± helper scripts) for a specific workflow the agent should run the same way every time.

- “Open a PR the way we like it”
- “Check register metadata before writing a query”
- “Format, lint, run tests, then commit”

The agent picks up the skill when the task matches — you stop re-explaining the same workflow in every prompt.

> Copilot and Codex both support putting skills in a `.agents/skills` folder in the project repo; Claude requires `.claude/skills/`. The structure follows an open standard with a subfolder for each skill, at least containing a `SKILL.md` witht basic instructions.

GitHub Issues and PRs: a systematic review surface

- **Issue** = contract: what “done” looks like, and a good home for agent instructions
- **Pull request** = diff: line-by-line review, the place to reject bad edits

Never paste secrets into AI tools

- API keys, passwords, credentials
- Private or unpublished data
- Anything you wouldn’t send on Slack or push to GitHub

Assume anything you paste may be logged or used for training.

> Some tools claim to redact sensitive values, but none make firm guarantees about training or logging. Treating anything pasted into an AI product as potentially public is the only rule that stays safe as terms of service change.

Before you accept the diff

1. Commit your own work first — cheap rollback
2. Open the diff and read every line
3. Run the code
4. Confirm package and function names actually exist (hallucination check)
5. Reject large edits that solve the wrong problem
6. If you can’t explain it, don’t submit it

A reviewable diff beats a vague memory of what changed, even when you’re working alone.

Working alone with an agent is the situation where review is easiest to skip and hardest to recover from. Every accepted change eventually becomes code that has to be read anyway when something breaks — reviewing upfront is the cheaper version of that reading.

Live AI demo

- Ask Copilot to explain the traceback in `01_traceback_test.R`
- Ask for the smallest fix, not a rewrite
- Ask for one guard that prevents the same mistake next time
- Review the diff before accepting anything

Next lecture: Data Wrangling I

Extra: data validation with `assertr`

Fail-fast style. Halts the pipeline on the first bad row.

```
library(assertr)

mini_panel |>
  verify(has_all_names("municipality_code", "year", "unemployment_rate")) |>
  assert(within_bounds(0, 100), unemployment_rate) |>
  assert(not_na, year)
```

- `verify()`: one logical expression — row counts, column names
- `assert()` / `insist()`: predicate / data-driven bounds per column
- Error points at the offending rows

Same idea as the `if (!all(...)) stop(...)` pattern from earlier, but pipe-native.

Extra: data validation with `validate`

Report style. Define rules, confront the data, inspect — nothing crashes.

```
library(validate)

rules <- validator(
  year_range   = year >= 2000 & year <= 2025,
  unemp_bounds = unemployment_rate >= 0 & unemployment_rate <= 100,
  no_dupes     = is_unique(municipality_code, year)
)
summary(confront(mini_panel, rules))
```

- Rules are objects: store, reuse, keep in a YAML file
- Pass/fail report per rule instead of stopping

`assertr` when bad data should *break* the pipeline. `validate` when you want to *measure* data quality — better for recurring audits on register data.

Extra: unit tests and `testthat`

Validation checks *data*. Tests check *code*. Remember the silent bug in `classify_unemployment()`?

```
library(testthat)

test_that("high unemployment is classified as high risk", {
  panel  <- data.frame(year = 2023, unemployment_rate = 15)
  result <- classify_unemployment(panel, threshold = 10)
  expect_equal(result$risk_group, "high")
})
```

```
── Failure: high unemployment is classified as high risk ──────────────────────
Expected `result$risk_group` to equal "high".
Differences:
1/1 mismatches
x[1]: "low"
y[1]: "high"
```

```
Error:
! Test failed with 1 failure and 0 successes.
```

Extra: unit tests and `testthat` (cont.)

- After each bug fix, add the smallest test that would have caught it
- Run with `testthat::test_file()` — works in plain projects, not just packages
- Tests are documentation: they show how the function is supposed to be used and what it should return
- Really useful when you have an agent making edits: ask the agent to write a test that would have caught the bug before accepting the fix

`usethis::use_testthat()` scaffolds `tests/testthat/`. Common expectations: `expect_equal()`, `expect_true()`, `expect_error()`.

Extra: continuous integration (CI)

Automatic checks on every change. Like a spell-checker for your code, but it can run tests and linters too.

- GitHub Actions is the standard for R — free for public repos
- You actually already experienced the auto-grading CI in PS1
- Configure it to run on every push or pull request
- `usethis::use_github_action("check-standard")` for packages

> Another way for agents to self-correct and to catch errors before they reach production: a failing check blocks deployment and is a signal for the agent to fix something.