# Building Computational and Data Skills in the First Year of a Social Science PhD

A Guide to a More Pleasant Grad School Experience

Draft v1.0 | 2026-08-21 | Working Paper

**Edwin Alvarado-Mena, PhD**

Policy Data Science Consulting LLC

AlvaradoCSS.com ↗ | PolicyDataScience.com ↗

## ABSTRACT

Graduate students in the social sciences increasingly need computational and data skills, yet many begin doctoral training with limited experience in programming, quantitative reasoning, or reproducible research. This guide proposes a practical first-year approach to building those capabilities without treating computational training as a race to master a long list of technologies. It emphasizes programming fundamentals, algorithmic thinking, data inspection and validation, quantitative reasoning, project organization, version control, reproducibility, and the responsible use of generative AI. The central argument is that these skills are most effectively developed as mutually reinforcing habits through sustained research practice. Rather than aiming for technical mastery during the first year, students should establish sound mental models, traceable workflows, and the ability to diagnose problems, evaluate computational outputs, and continue learning as their research demands evolve.

**Keywords:** computational social science; programming; data literacy; reproducible research; quantitative reasoning; graduate education; generative AI

# Contents

# 1 Introduction

Starting a PhD program without a deep background in statistics, mathematics, or programming can feel overwhelming. But if you find yourself in that position, rest assured: there is no need to panic. The first year of a PhD program is a perfect opportunity to address gaps and establish a solid foundation for your research.

The goal should not be to become a mathematician, statistician, and computer scientist all at once. During your first year, focus on steadily developing your ability to wrangle and analyze data, reason quantitatively, and use computational tools to investigate important questions about social phenomena. Deliberately developing these skills will make your grad school experience more pleasant.

**AI Use Disclosure**

The prompt I use to proofread technical manuscripts, including this guide, is publicly available here: review-technical-manuscript.

You can also explore my broader collection of project settings and prompts and reuse or adapt anything you find useful: my-prompts (GitHub repository).

This outline offers a practical approach to building essential skills during the first year of a social science PhD. It focuses on key areas that reinforce one another and become more valuable as your research develops, including data management, quantitative reasoning, programming, reproducible research, and scientific communication.

Code examples are intentionally kept to a minimum so that the focus remains on principles and good habits rather than technicalities.

The sequence developed in this guide begins with a primary programming language and algorithmic thinking, then builds toward confident work with data, quantitative reasoning, and reproducible project organization. Once this foundation is in place, further development should be guided by the demands of the research. Version control, software environments, computational documents, databases, and large-scale workflow tools become useful as projects create a need for them. More specialized languages and computing infrastructure should generally follow as doctoral students take on more advanced computational work.

# 2 High-level advice

The guide begins with general advice on developing the habits and mindsets that support computational and data literacy.

## 2.1 Give yourself time to learn

The breadth of this guide should not be mistaken for a prescription to move quickly. There is no need to rush through the material, learn everything at once, or follow a predetermined pace. You

may well finish your first year of the PhD program without having mastered everything covered here.

Developing the skills to succeed in grad school takes time, and progress is rarely linear. Some concepts will click immediately, while others may remain confusing for months, or even years, before becoming intuitive. You will also discover gaps in your knowledge that send you back to more fundamental material.

That nonlinear trajectory is perfectly fine. Avoid comparing your progress with that of students who seem to learn faster. People enter PhD programs from very different starting points. Someone who has been programming since college will naturally progress differently from someone who has just downloaded R for the first time.

The goal is not to meet some imaginary benchmark. What matters is continuing to build from your own starting point and continue learning along the way.

You also do not need to master one topic before moving on to the next. Learning is inherently iterative. Working in R might expose a gap in your statistical understanding, prompting you to revisit the relevant mathematics and then return to your code better prepared. New projects will reveal other gaps and provide reasons to address them. Over time, you will revisit earlier material with greater depth and understanding.

So give yourself permission to learn slowly, revisit what you have learned, and build from there. This is not a race from beginner to senior programmer. Your first-year objective is to develop the habits and mental models that will help you keep improving throughout your PhD.

## 2.2 Take the mystery out of programming

At a basic level, a programming language provides a way to express operations in a form that a computer can interpret and execute. Programming may seem intimidating at first, but with time and sustained effort, you can become a resourceful programmer.

Try not to think of programming as either *hard* or *easy*. Think of it as a skill that improves with practice. Spend enough time programming and things that once seemed inaccessible will gradually become familiar.

Importantly, you will make progress even when it does not feel like it. Programming has a peculiar learning curve: during your first few weeks or months, seemingly small tasks can consume far more time than you expect. It is key that you approach the learning process with a positive mindset.

Expect your patience to be tested. You might spend an entire afternoon trying to import a file, only to discover that the real problem is how your computer is organized. Another day, a mysterious error might send you searching for how a function actually works. Some frustration is inevitable, especially when you are just getting started.

But the time you spend struggling with code is also time spent learning. Every small problem teaches you something. Figuring out why your code failed or why your data set is not behaving as expected gradually builds your understanding of how the language interacts with your computer and, crucially, teaches you how to find your way out when you get stuck.

Over time, problems that once took hours begin to take minutes. The goal, therefore, is not to memorize everything or expect your code to run without errors, but to become increasingly resourceful. When something goes wrong, learn to diagnose the problem, use the information available to work toward a solution, and verify that the result makes sense.

That is what programmers actually do: they get better at working through errors. The challenge is to treat the frustration that accompanies them as part of the learning process.

## 2.3 Learn to read code before expecting yourself to write it

Beginners often judge their programming ability by whether they could start from a blank screen and write a program on their own. That sets an unnecessarily high bar. Fluency in writing code comes later; first, you need to become comfortable reading and understanding it.

Make a habit of reading good code. Explore well-organized GitHub repositories, work through programming books and documentation, or ask ChatGPT or Claude to generate a program and explain it line by line. Early on, understanding what a script does is already meaningful progress. Some scholars maintain extensive public repositories that let you learn by seeing how experienced researchers actually write and organize their code. Professor Andrew Heiss at Georgia State University is an excellent example of a scholar who chooses to publish very polished repositories.

As you read code, try to trace how information moves through the program. If you can identify what the code starts with, how it transforms that information, and what it ultimately produces, you are already developing computational understanding. Then experiment with working examples. Take a useful program, modify something, predict what will happen before you run it. Small changes will gradually reveal how the pieces fit together.

Eventually, the patterns become familiar. The strange symbols begin to resolve into instructions and transformations, revealing the underlying logic of the programming language:

```
survey |>
  filter(age >= 18) |>
  group_by(region) |>
  summarise(mean_income = mean(income, na.rm = TRUE))
```

You will begin to read code almost as if it were a sentence: *take the survey data*, *retain the adults*, *group them by region*, *compute the average income for each region*. At that point, the syntax begins to recede, and you can see more directly what the program is doing. This shift marks an important transition in learning to program.

## 2.4 Use generative AI without hurting your learning process

Used deliberately, generative AI can be a valuable part of both learning and computational work. That said, generating code is not the same as understanding what it does or verifying that it works correctly. The key, then, is to use AI wisely.

Tools such as ChatGPT or Claude are especially useful when you get stuck. They can help you interpret errors, understand unfamiliar code, explore alternative approaches, and make sense of

technical documentation. This support is particularly valuable early on, when you may not yet know the vocabulary needed to search effectively.

You can even ask the model to surface that vocabulary directly:

> "Based on this chat, what programming terms and concepts should I make sure I understand? Summarize them in a table with three columns: concept or term, definition, and relevance to our discussion."

Generative AI has become remarkably convenient to use. That same convenience, however, creates risks you must manage. Generative AI can produce plausible-looking code that runs perfectly well while being inappropriate for your data or research design. Conversely, it can produce excellent code that you simply do not understand. In either case, successful execution tells you little about whether you have learned anything or whether the result is correct.

Do not settle into a workflow where you describe the desired result, copy whatever code AI generates, run it once, and move on because nothing seems to have gone wrong. That routine will keep you from developing your own computational judgment.

A good principle to follow is:

> **Never submit, publish, or rely on code whose behavior and output you cannot adequately explain and validate.**

This does not mean that you need to write every line yourself. It means something more fundamental: you must take responsibility for the code you incorporate into your work.

If AI introduces an unfamiliar function or package, consult the documentation. When the code matters, test it on a case for which you already know the answer. Make sure you understand what the code is doing and why. Your responsibility extends beyond making the script run: you need to understand the assumptions that connect its inputs to its outputs.

As your skills improve, AI should extend your computational judgment rather than stand in for it. Apply the same discipline to errors. Before asking AI for a fix, investigate the problem yourself. Read the console message, locate the failure, inspect the relevant object. Importantly, form a hypothesis. Then bring in AI and test your hypothesis. That brief period of friction is often where much of the learning happens.

Once you understand the task, however, there is no virtue in doing everything the slow way. Use AI wherever it removes unnecessary friction from your work. Let it handle routine assistance with syntax, boilerplate, debugging, testing, and documentation. Let it accelerate work you already know how to evaluate. But never equate generated code with reviewed code. Whatever AI contributes still requires your judgment.

The same principle of responsibility applies to research integrity. Never send restricted data, code, or unpublished material to an external AI service unless the relevant institutional rules, agreements, and service terms clearly permit it.

As code becomes easier to generate, the ability to determine whether LLM-generated code makes sense scientifically and computationally becomes more valuable. Generative AI does not replace

sound programming knowledge. If anything, it makes that knowledge more consequential for several reasons:

- First and foremost, you should never rely on code you do not understand well enough to take responsibility for. Professional ethics come before computational convenience.
- You are also a graduate student, and part of your training involves becoming capable of deeply understanding and independently producing computational work. Some courses will therefore require you to complete programming assignments without generative AI.
- Most programming problems have multiple workable solutions, and chatbots are increasingly good at producing one that gets the job done. But code that works is not necessarily good code. A proposed solution may be unnecessarily expensive or inefficient, difficult to maintain, or poorly suited to your broader project and its anticipated needs. The better you understand programming, the better equipped you are to evaluate these tradeoffs rather than simply accepting the first solution an LLM produces.
- The more programming knowledge you have, the more precisely you can communicate with chatbots and the more ambitious you can be about what you ask them to help you build. Knowing the relevant concepts and terminology allows you to describe problems with greater precision, recognize useful approaches, and ask better follow-up questions when the first solution falls short. It also expands the range of possibilities you can imagine in the first place. Strong programming skills combined with generative AI can take you remarkably far.

## 2.5 Choose a primary programming language

Beginners generally benefit from staying with one programming language long enough to become comfortable using it. For social science students pursuing quantitative research, R is an excellent choice.

R can accompany you from your first programming exercises into substantive research, where the same language can be used to work with data, conduct statistical analyses, and produce reproducible results. Paired with an integrated development environment such as RStudio or Positron, it also provides an approachable setting in which to begin programming.

That said, which programming language you choose should also reflect your research environment. If your lab, advisor, methods sequence, or collaborators work primarily in Python, Stata, MATLAB, Julia, or something else, learning that language first may be more useful than choosing R in isolation.

Whatever you choose, stay with it long enough for the underlying programming concepts to become familiar. Once you understand how one language represents data, applies functions, controls execution, and helps you diagnose problems, much of that conceptual understanding will carry over when you learn another language.

Eventually, you will likely use several languages anyway. Real-world research workflows often cross these boundaries as different tasks call for different languages and computational tools. An R-centered project, for example, might rely on SQL for accessing a database, Python for a specialized library, or LaTeXand Quarto for producing the final document. There is no reason

to remain faithful to one programming language or set of tools when your research calls for something else.

Nor is there much value in collecting programming languages as credentials. Learn a new one when your research gives you a reason to use it.

## 2.6 Identify the best books and resources

Internet searches and chatbot conversations are excellent tools for solving specific programming problems, but they are a poor substitute for a structured introduction to a language. Used alone, they can leave you with fragments of knowledge rather than a coherent understanding of how the language works.

A good textbook or structured, open-access online course provides something that isolated, piecemeal learning rarely does: a carefully designed sequence. The material reflects deliberate choices about what to introduce first, how later concepts build on earlier ones, and what can wait until later. You benefit from the experience of people who have already worked through the problem of teaching programming effectively.

In the R ecosystem, books associated with Hadley Wickham and the `tidyverse` provide a common entry point. *R for Data Science* (Wickham, Çetinkaya-Rundel, and Grolemund 2023), for example, takes you through many of the operations you will eventually perform in your own research, from working with raw data to analyzing, visualizing, and reporting the results reproducibly. Kelsey Gonzalez's R Programming with the tidyverse YouTube course can help you get started on solid footing.

You do not need to finish an entire programming book or online course before putting what you learn to use. Read a chapter, watch a few videos, work through the exercises, and then apply those ideas to a small project of your own. These projects do not have to be academic. If you want to practice retrieving and manipulating data, pull information about your favorite artist or public figure from Wikipedia using `rvest` and turn it into a visualization. A few ideas for web scraping are available here.

Sustained practice matters enormously. The subject matters less than having a reason to use what you are learning.

## 2.7 Learn to use documentation

As you become more comfortable programming, gradually shift from relying on isolated examples toward consulting documentation and understanding how things really work.

In R, when you encounter an unfamiliar function, try:

```
?mean
```

Or:

```
help(mean)
```

Those commands will take you to R's documentation. Treat it as something to understand, not merely a place to find code to copy. Identify what a function expects, how its arguments shape its behavior, and what it returns, then use the examples to see those relationships in practice. Although it might feel overwhelming at first, documentation is one of R's core strengths.

Package websites and reference manuals may look dense at first. Keep using them. As their structure becomes familiar, documentation will shift from something intimidating into one of your most useful programming resources. The ability to navigate technical documentation matters even more in the age of generative AI. Chatbots can confidently recommend code that is outdated, inappropriate, or simply nonexistent. Documentation gives you a reliable way to check what you have been told.

Over time, aim for a workflow like this:

1. Define the problem at hand, that is, what you are trying to accomplish.
2. Identify a promising function or package.
3. Consult the documentation.
4. Test your understanding with a minimal example.
5. Apply the solution to your own data.
6. Verify that the result makes sense.

Computational judgment also means knowing how to find authoritative technical information when you need it.

## 2.8 Work within a dedicated computational project

Before worrying about sophisticated statistical analysis, learn to organize the computational environment in which your research lives. At first, this can be as simple as organizing your computer for data work.

You do not want a research project scattered across files with names like:

```
analysis_final.R

analysis_final2.R

analysis_final_REALLY_FINAL.R

data_clean_new.csv

data_clean_new2.csv
```

A simple, consistent directory structure will save you a great deal of frustration later; for example:

```
research-project/
   data/
       raw/
       processed/
   R/
   scripts/
   output/
   figures/
   documents/
   README.md
   .gitignore
```

The exact structure is not sacred. Conventions vary across disciplines, labs, and teams, and your directory structure will likely become more elaborate as your projects grow and source code, configuration files, tests, logs, workflow definitions, manuscripts, and generated artifacts begin to accumulate. The underlying principle remains the same: you should know where files belong, what they are for, and where they came from.

Whenever feasible, preserve source data exactly as you received them. Treat the original files as immutable inputs and perform transformations through code rather than manual editing. Scripts should transform inputs into derived data and results through steps that can be inspected and repeated. Generate figures and tables directly from code when possible. If a step requires manual intervention, document it explicitly.

Preserving source data unchanged does not imply that those data must be stored inside the project directory. Security requirements, data size, or computing infrastructure may necessitate that the data reside elsewhere. In some research environments, copying a database locally may be impractical, technically unavailable, or prohibited by access and security requirements. In those cases, your project should preserve the information needed to locate and identify the data, not the data sets themselves.

Document every important data set well enough that you can understand and use it correctly later. A README file, data dictionary, codebook, or metadata file should record what the observations and variables represent, where the data came from, and any important restrictions or limitations.

When working with a third-party data set, take the time to locate and review the accompanying documentation. If the underlying data can change, record enough information to identify the exact version used in your analysis, such as a date, a snapshot, a checksum, or another persistent identifier.

Organize your work around project directories and avoid scattering machine-specific absolute paths throughout your scripts. RStudio projects and Positron workspaces are particularly useful for this purpose because they establish a project directory as a common reference point for your files.

**Blog**

R projects are discussed in more detail on the accompanying data literacy blog.

A project directory gives your files a common reference point, allowing you to use project-relative paths instead of machine-specific paths such as:

```
C:/Users/YourName/Desktop/Random Folder/Dissertation/New Analysis/data.csv
```

For files within the project, prefer project-relative paths:

```
data/raw/data.csv
```

Those path references will then remain valid when the project is placed in a different location, provided the project's internal structure is preserved. Other requirements, such as software dependencies, permissions, and external resources, still need to be satisfied for the code itself to run successfully.

Resources outside the project require a slightly different approach. Store machine-specific locations for external data, shared storage, databases, or computing clusters in configuration files or environment variables so that the scripts themselves remain portable.

A computational project should be portable to the extent that its data, software, permissions, and computing requirements allow. That does not necessarily mean being able to copy a working directory to another machine and run it immediately. A project is portable when another authorized researcher, or you in six months, can determine what it requires, reconstruct its working environment, and rerun the relevant workflow without having to guess how everything fits together.

Develop these habits early. They will make your later computational work more reliable and make you a stronger collaborator and research assistant.

# 3 Programming in R

This section focuses not on introducing R itself, but on how beginners can learn it effectively. Learning *how to learn R* is a different problem from learning the language itself. The ideas that follow are intended to help with the former.

## 3.1 Build a mental model

People learn by gradually refining mental models of how a subject works. This idea is one of the principles emphasized in Carpentries Instructor Training:

> "Understanding is never a mirror of reality, even for an expert; rather, it is an internal representation based on our experience with a subject. This internal representation is often described as a mental model. A mental model allows us to extrapolate, or

> make predictions beyond and between the narrow limits of experience and memory, filling in gaps to the point that things *make sense*.
>
> As we learn, our mental model evolves to become more complex and, most importantly, more useful. A useful model makes reasonable predictions and fits well within the range of things we are likely to encounter. While there will always be inaccuracies – or *misconceptions* – these do not interfere with day-to-day functioning. A useful model does not seize up or break down entirely as new concepts are added." (The Carpentries 2025).

At first, R may appear to be an enormous collection of unrelated commands. As you develop a simple mental model of how its pieces fit together and gradually refine it, the language begins to make much more sense.

You do not need to understand the entire language before you can use it productively. Start with a few basic ideas and refine your understanding through practice.

The concepts below provide reference points for developing an initial mental model of how R works.

## 3.2 Applying functions to objects

A surprisingly large amount of R programming can be understood as creating objects and applying functions that do something to them.

You first create an object; for example, a list of numbers:

```
x <- c(2, 4, 6, 8)
```

Then you apply a function to it:

```
mean(x)
```

```
[1] 5
```

The function takes the object as input and returns a result. In this case, that result is the mean of the values.

This simple object-and-function mental model will take you surprisingly far in R. A data set, a fitted statistical model, and even a `ggplot` visualization can all be understood as objects that functions create, transform, inspect, or otherwise act upon.

A function typically takes one or more inputs, performs an internal operation, and returns a value. Some functions also produce side effects, such as writing a file, printing output, modifying an external resource, or creating a plot, in which case the returned value may not be the main result you care about.

Once you begin seeing R this way, complicated-looking code becomes more intuitive. You can work through it by repeatedly asking:

1. What object or input am I working with?
2. What function am I calling?
3. What arguments control the behavior of this function?
4. What value, file, plot, model, or other effect should result?

Even a simple object-and-function mental model will take you much further than trying to learn R without a conceptual framework.

## 3.3 Develop algorithmic thinking

Programming is less about knowing clever commands than about learning to decompose a problem into a sequence of sensible operations a computer can execute. This approach to problem-solving is called algorithmic thinking.

Suppose you have thousands of survey responses and want to calculate the average income of respondents over 30 for each state. Before thinking about R syntax, reason through the sequence of operations needed to get there:

1. Start with the survey data.
2. Identify the variable containing age.
3. Keep observations where age is greater than 30.
4. Identify the variable containing states.
5. Divide the remaining observations by state.
6. Identify the income variable.
7. Calculate the mean income within each group.
8. Store or display the result.

Only after you conceptualize the sequence should you worry about translating it into R:

```
result <- survey |>
  filter(age > 30) |>
  group_by(state) |>
  summarise(
    mean_income = mean(income, na.rm = TRUE)
  )
```

This distinction is fundamental. If you cannot work out the solution conceptually, knowing more R functions will not necessarily bring you any closer to implementing it.

When a computational task feels overwhelming, decompose it. What looks like a large computational problem is often just a sequence of smaller problems that you can solve one at a time.

Do not ask yourself:

> How do I program this entire analysis?

Ask:

What is the first thing that needs to happen?

Then:

Given that result, what needs to happen next?

For example, suppose your eventual goal is to analyze the text of 10,000 government documents. The problem sounds complicated when described in its entirety. So you can begin decomposing it:

```
locate the files
    ↓
read each file
    ↓
extract the text
    ↓
clean the text
    ↓
identify the unit of analysis
    ↓
apply the same operation to each unit
    ↓
store the results
    ↓
validate the results
    ↓
analyze them
```

Each step can then be decomposed further. Critically, the very act of reasoning through the conceptual structure of the task will help you explore alternative ways to solve it before committing to a particular implementation.

Algorithmic thinking extends far beyond programming. Whether you are cleaning data, building a simulation, scraping the web, or designing a dissertation, complex problems become more manageable when you learn to decompose them into smaller problems.

Likewise, algorithmic thinking helps you recognize repetition. If you find yourself performing the same operation manually several times, ask whether you can express it as a general rule and let the computer handle the repetition. Functions are one of the main ways to do this: define a generalizable operation once and reuse it whenever you need it.

One advantage of algorithmic thinking is that it prevents you from overextending yourself. There is little reason to worry about statistical modeling, API calls to proprietary models, or fancy Quarto reports while you are still figuring out how to wrangle the data you need. Keep your attention on the problem immediately in front of you. Once you have solved and understood it, move to the next one. Complex workflows become manageable when you build them incrementally rather than jumping ahead to more advanced tasks.

### 3.3.1 Use pseudocode before writing code

A practical method for developing algorithmic thinking is pseudocode: a plain-language description of what a program should do, organized as a sequence of logical steps without worrying about programming syntax.

Return to the problem of comparing average income across states among survey respondents over age 30. Before translating the solution into R, write the underlying logic in pseudocode:

```
load the survey data

inspect the variables

keep respondents older than 30

group respondents by state

calculate average income for each state

save the resulting table
```

No executable code has been written yet, but much of the conceptual work is already done: you know what needs to happen and in what order. The remaining task is to translate that logic into R, one step at a time.

Pseudocode is most valuable when a task begins to feel too complicated to hold in your head at once. Start by describing the solution in ordinary language, then decompose each step until you are left with operations small enough to understand and implement on their own.

For example:

```
for every file in the data directory:

    read the file

    check that the required variables exist

    if the variables are present:
        clean the data
        add the filename as an identifier
        store the result

    otherwise:
        record the file as an error

combine all successfully processed files
```

```
validate the combined data set

save the final data set
```

Notice that the pseudocode already captures important programming ideas such as sequence, iteration, conditions, validation, inputs, and outputs. You can work through this logic before translating it into actual code and often save time by identifying missing steps, faulty assumptions, or unnecessary complexity before they become buried in syntax and debugging.

Pseudocode can also become part of your documentation. Before implementing a complicated section of a script, describe the intended logic in comments. As you proceed, replace or supplement those comments with working code:

```
# Load all survey files.

# Check that each file contains the required variables.

# Clean each valid file using the same procedure.

# Record files that fail validation.

# Combine the valid data sets.

# Validate the combined data set.

# Save the processed data.
```

Pseudocode helps you separate two questions that beginners may conflate: *What should the computer do?* and *How do I express those instructions in R?*

The lesson can be summarized as:

> **Get the logic right first. Only then worry about the syntax.**

With experience, familiar or routine tasks may no longer require explicit pseudocode. But when a research pipeline becomes complex, writing down the logic before implementing it remains a valuable discipline, regardless of your level of experience.

Pseudocode is also useful when working with generative AI because it gives you a concrete specification to provide to the model rather than leaving the model to infer what you want from a vague request. It also gives you a clearer basis for evaluating whether the generated code actually solves the problem you intended to solve.

### 3.3.2 Algorithmic thinking in the age of generative AI

Generative AI makes algorithmic thinking more important, not less. You can now ask a chatbot to "write code that analyzes these files" and receive plausible-looking code almost immediately. But the quality of the response depends heavily on how clearly you can specify the problem.

Instead of asking AI to solve an entire vaguely defined task, get used to describing the workflow explicitly:

> Read every CSV file in this directory.
>
> Verify that each contains these five variables.
>
> Combine them into one data set.
>
> Add a variable containing the original filename.
>
> Report files that do not match the expected structure rather than silently dropping them.

That is a more robust computational specification.

Notice that the important intellectual work happened before any code was generated. You defined where the workflow should start and end, the logic connecting the two, and how the program should handle problems along the way.

This discussion points to an increasingly important principle for programming in the age of generative AI:

> Knowing how to write code is not enough. You also need to know how to specify a computational process.

AI can assist with syntax, implementation choices, and overlooked cases. Nonetheless, you remain responsible for specifying the computational process clearly and determining whether the resulting code actually implements it.

So ask yourself:

- What information do I have, and what result do I want?
- What sequence of transformations connects the input to the output?
- Which operations need to be repeated?
- What assumptions does the workflow make about the data?
- What could go wrong, and how should the workflow respond?
- How will I verify that the result is correct?

If you can reason through those questions clearly, the code is no longer the starting point. It becomes the implementation of a deliberate solution you have already begun to design.

The technologies you use will change throughout your career. What will remain valuable is the ability to turn a problem into a sequence of precise, testable computational steps.

## 3.4 Know what kind of object you are working with

The earlier mental model of R, applying functions to objects to obtain a result, becomes more useful once you recognize that the kind of object you are working with determines what you can do with it.

Objects are fundamental to programming. Their structure, class, and underlying type govern how they behave and how functions operate on them. Understanding these distinctions helps you anticipate and diagnose errors.

For now, think of an object as a container for information. The following examples illustrate what that can look like in R:

```
age <- 25

name <- "Maria"

completed <- TRUE
```

These three objects, `age`, `name`, and `completed`, contain different kinds of information: a number, a string (text), and a logical value. Beginners, however, are often more familiar with tabular containers such as spreadsheets. In R, the corresponding structure is called a data frame.

A data frame organizes data into columns, which are themselves vectors. Their types and classes affect how R represents their values, which operations are appropriate, and how functions behave. R also represents missing values explicitly as `NA`, which requires deliberate handling because missingness can affect calculations.

As you become more familiar with R, develop a working understanding of its fundamental data structures:

- Vectors.
- Data frames.
- Lists.
- Matrices.

You do not need to memorize R's internal architecture. But you do need to establish the habit of asking:

> What kind of object is this, and what assumptions is my code making about it?

Many programming problems that initially seem mysterious arise from a mismatch between what the code assumes about an object and what that object actually is:

- A number was imported as text.
- A date is actually a character string.
- A function expects a vector but received a data frame.
- A categorical code such as `1`, `2`, `3` has been imported as numeric even though arithmetic on those values would be meaningless.
- A factor has levels that should not exist.
- A missing value (`NA`) is propagating through a calculation.

Unexpected behavior in R is often a signal to inspect the object itself. Before changing your code, use functions such as these to determine what R is actually working with:

```
class(x)

typeof(x)

str(x)

attributes(x)
```

For `tidyverse` workflows, you can also invoke:

```
glimpse(x)
```

Understanding the objects moving through your program removes much of the mystery from programming. It allows you to diagnose failures more precisely, verify that transformations produced the intended result, and provide chatbots with better information when asking for help.

## 3.5 Loading and inspecting data

Early in your work with R, you should become comfortable importing data while keeping track of what was loaded and how R interpreted it.

Start with simple tabular data, such as a familiar spreadsheet, typically stored as a CSV or Excel file. As you progress to databases, JSON, geospatial data, APIs, and other sources, you will encounter different ways of accessing, representing, and importing data. These differences shape how data arrive in R, including how variables are interpreted and whether dates, missing values, text, and numerical precision are preserved correctly.

Importing data is only the first step. You should always inspect what R actually loaded. So begin by loading the data:

```
library(readr)

survey <- read_csv("data/raw/survey.csv")
```

After importing the data, check the object R actually created:

```
library(dplyr)

glimpse(survey)
```

Before running models or performing substantial transformations, make sure you understand the object you are working with. For a data frame, a few questions you should answer early are:

- How many rows does it contain?
- What does each row represent?
- What does each column represent?
- Which variables are numeric?
- Which are categorical?
- Are missing values represented correctly?
- Are dates actually stored as dates?
- Are identifiers unique when they should be?
- Are the ranges of your variables plausible?
- Did the import function guess any column types incorrectly?

Whenever the data set comes with documentation, use it as your reference point. Compare what R imported against the documented structure rather than assuming that data that look right were necessarily interpreted correctly.

For important workflows, move beyond informal inspection and introduce explicit checks. Verify the assumptions your analysis depends on, such as the presence of required columns, the uniqueness of identifiers, and the plausibility of values. Over time, these checks should become assertions or automated validation tests.

Data provenance matters as much as reliable import. Record where the data came from and which version you used. Being able to reproduce `read_csv("data.csv")` is of limited scientific value if the identity and origin of `data.csv` cannot be established.

Systematic inspection and validation should therefore become habits early in your training. These practices also reinforce the importance of project organization. When data have a clear place in the project and their provenance is documented, importing them becomes a reproducible part of the analysis rather than an undocumented preliminary step.

## 3.6 Tidying data

Proficiency in data wrangling expands the range of research that feels feasible. When preparing data becomes less of an obstacle, projects that once seemed too cumbersome become realistic undertakings. The payoff also arrives relatively early: efficient data wrangling can save considerable time during coursework.

Beginners often imagine that quantitative research begins with statistical modeling. In practice, much of the work happens before a model is ever estimated. Raw data rarely arrive ready for analysis, so observations may need to be selected, variables constructed, information combined or reshaped, and inconsistencies resolved before modeling can begin.

Data wrangling operations are not peripheral to data analysis. They are part of the analytical process because decisions made during data preparation can shape the cases, variables, and relationships that ultimately enter the analysis. Cleaning and tidying also force you to become intimately familiar with your data. Think twice before delegating this work entirely to your chatbot of choice. You may miss something important.

**Tidy data**

Since the purpose of this guide is to emphasize principles rather than provide extensive code, the essential reference for this topic is Hadley Wickham's *Tidy Data* (Wickham 2014).

You will likely do much of your data wrangling with `dplyr` and `tidyr`, so their core functions should become part of your every-day vocabulary:

```
select()

filter()

mutate()

arrange()

summarise()

group_by()

left_join()

pivot_longer()

pivot_wider()
```

Do not worry about memorizing function names and arguments. Focus instead on understanding how each operation transforms the data.

For example:

```
library(dplyr)

survey_summary <- survey |>
  filter(age >= 18) |>
  group_by(region) |>
  summarise(
    mean_income = mean(income, na.rm = TRUE)
  )
```

You can read the code above almost like a sentence:

- Take `survey`.
- Keep respondents age 18 or older.
- Group them by region.

- Calculate average income.

This readable style is one reason R works particularly well for social scientists who are beginning to program.

## 3.7 Debug and distinguish errors from wrong results

Your code will break. In fact, it will break several times a day. This is not a phase you eventually outgrow. Experienced researchers encounter errors constantly as well. Programming is, in large part, the practice of identifying why something failed and deciding what to try next.

What improves with experience is your ability to diagnose errors systematically. Debugging is not an interruption of programming but one of its central practices. So you need to learn from your errors, quite literally, not as a self-help metaphor.

When something fails, resist the temptation to modify code at random until the error disappears. Inspect the problem first, form a hypothesis about what went wrong, and only then change the code.

Start with the error message:

- Which line failed?
- What function was being called?
- What objects were passed to it?
- What did you expect those objects to contain?

R error messages vary considerably in clarity, but ChatGPT or Claude can help you decipher what the console is telling you. Read errors closely and keep track of the kinds you encounter repeatedly. If the problem is buried inside a long pipeline, break the pipeline into smaller pieces and run them separately.

For example:

```
result <- survey |>
  filter(age >= 18) |>
  mutate(income_log = log(income)) |>
  group_by(region) |>
  summarise(mean_income = mean(income_log))
```

Instead of debugging this entire operation, break the pipeline into individual transformations and inspect each result. The aim is to identify the earliest step at which the program stops behaving as expected.

A useful debugging sequence is as follows:

1. Read the error or warning message.
2. Identify where the unexpected behavior begins.
3. Inspect the relevant objects.

4. Check their classes, dimensions, names, and values.
5. Reduce the problem to a small reproducible example when possible.
6. Test one explanation at a time.
7. Consult documentation, search authoritative sources, or ask AI when necessary.
8. Verify that the fix solves the underlying problem rather than merely suppressing the symptom.

Errors that stop execution are only one kind of failure. More dangerous are cases in which the code runs successfully but produces the wrong result (or produces the correct result through a process different from the one you think is occurring, which can be even harder to detect).

Here are some cases where this could happen:

- A join can unexpectedly duplicate observations when the key relationships are not what you assumed.
- A filter can remove the wrong cases.
- A recode can turn an unexpected category into missing data.
- A model can fit perfectly well to a data frame whose unit of analysis is wrong.

That is why debugging must be paired with validation. The specific checks will depend on what your code is doing, but the principle is constant: look for opportunities to verify that each important operation did what you intended. This might mean checking row counts around a transformation, examining unmatched keys after a join, comparing summaries against known benchmarks, spot-checking records, or constructing a small input with a known expected result.

Above all, treat errors and warnings as information. Each one can teach you something about how the language behaves and sharpen your ability to diagnose future problems. At the same time, remember that silent success deserves scrutiny too: code can run perfectly and still produce a scientifically invalid result.

## 3.8 Plot your data early and often

Data visualization is not something you add after the "real" analysis is finished. Plotting is itself a form of analysis: it helps you explore your data, recognize patterns, detect problems, and discover questions worth pursuing. As a bonus, visualization is often an enjoyable entry point into programming.

The better you understand the data set, the better equipped you are to decide how to model it. Before fitting a statistical model, use visualization to become familiar with your data. Examine distributions and group differences, explore relationships between variables, and, when the data have a temporal dimension, look at how observations change over time. A good book, such as Wilke's *Fundamentals of Data Visualization* (Wilke 2019), can introduce you to the range of options available for different situations.

A key point is that visualization can expose structure that summary statistics hide. An unusual value may be a coding error or a genuine outlier; a gap, cluster, or nonlinear pattern may reveal something important about the phenomenon you are studying. Visualization is therefore useful

not only for finding problems or communicating what you have already found, but also for discovering what deserves closer investigation.

Learning `ggplot2` early is therefore a worthwhile investment. The package follows a layered approach to building visualizations.

For example:

```
library(ggplot2)

survey |>
  ggplot(aes(x = age, y = income)) +
  geom_point()
```

The code above first establishes the plot and maps `age` and `income` to the `x`- and `y`-axes using `ggplot()`, and then represents each observation as a point using `geom_point()`.

Prioritize informative plots over beautiful ones. Learn first to choose an appropriate visualization, map variables correctly, and make the pattern of interest visible. Data visualization offers endless possibilities for customization, but visual polish should remain secondary to the analytical purpose of the plot.

## 4 Build quantitative reasoning as you learn to program

Being good at programming does not automatically make you a good quantitative social scientist. Computational proficiency and quantitative reasoning are distinct skills, and understanding that distinction is especially important when you are starting graduate school.

You can become very good at manipulating data in R while having only a superficial understanding of the statistical analysis you are performing. Modern statistical software, and now generative AI, makes sophisticated methods remarkably easy to execute. The risk is that the code may be under your control while the analysis is not.

If you run a linear regression, understand what it estimates. If you calculate a standard error, understand where the uncertainty comes from. If you conduct a hypothesis test, understand the null hypothesis. If you use a causal inference design, understand the assumptions that make a causal interpretation possible. If you impute missing data, understand the process that produced the missingness.

Whatever the situation, develop the habit of separating at least three questions:

1. Can I make the computer perform this analysis?
2. Do I understand what the analysis does?
3. Is this analysis appropriate for my research question and data?

Being able to answer the first question does not guarantee that you can answer the other two. Programming lets you execute an analysis; it does not tell you whether the analysis makes sense.

That requires a different kind of practice. As your computational skills grow, make sure your methodological judgment grows with them.

## 4.1 Strengthen your mathematical foundations strategically

If your mathematical background is still limited, the notation you encounter in statistics courses, methods textbooks, and applied papers may initially feel like another language. That is not a bad analogy: mathematical notation is a language you have to learn to read.

Do not mistake unfamiliarity with mathematical notation for evidence that quantitative research is beyond you. Mathematics is learned through sustained practice. Whether you call it hard or easy matters less than recognizing that it takes time. Give it that time, and your ability to read and use mathematics will improve.

Improvement will be substantial if you keep practicing, even when progress is difficult to perceive from one day to the next. Periods that feel stagnant may still be periods of accumulating knowledge, forming connections, and becoming more familiar with complex ideas. With sustained effort, those incremental gains eventually become visible and actionable.

None of this means that empirical research must wait until you have mastered an entire mathematics curriculum. Build your mathematical foundations in parallel with your methodological training.

At minimum, become increasingly comfortable with:

- Algebra.
- Functions.
- Exponents and logarithms.
- Summation notation.
- Basic probability.
- Random variables.
- Probability distributions.
- Expectation.
- Variance and covariance.

Depending on your methodological interests, you may eventually need more linear algebra, calculus, optimization, or probability theory. A little combinatorics never hurts either.

Learn these subjects strategically. For instance, if a regression course introduces matrix notation, that is a good reason to learn enough linear algebra to understand what the notation is doing. If you begin studying maximum likelihood estimation, that may be the moment to strengthen your understanding of derivatives and optimization.

A few useful readings include:

- *Thinking Clearly with Data* (De Mesquita and Fowler 2021) is an excellent prose-based introduction to many of the core ideas you will encounter throughout a typical PhD methods sequence in the United States.

- *A First Course in Machine Learning* (Rogers and Girolami 2016) offers an accessible walk-through of the mathematics behind linear regression, along with beginner-friendly introductions to maximum likelihood and Bayesian statistics.
- *Quantitative Social Science* (Imai and Williams 2022) and *Data Analysis for Social Science* (Llaudet and Imai 2023) are useful starting points for social scientists who are still building their technical foundations.
- *Introductory Econometrics* (Wooldridge 2019) and *Regression and Other Stories* (Gelman, Hill, and Vehtari 2020) are hard-to-beat textbooks in econometrics and statistics, respectively.

The objective is not to accumulate mathematics for its own sake. You want to reach the point where mathematical notation helps you understand a method rather than preventing you from understanding it. Eventually, you may even find yourself using mathematics to express your own ideas.

Some people learn best by working through the mathematics directly. Others benefit from programming with real data, where abstract concepts become concrete operations they can observe and manipulate. There is no reason not to combine the two. But there is also a particular joy in grabbing a good book, working through the mathematics, and gradually becoming fluent in the language itself.

## 5 Build (reproducible) projects, not just exercises

Tutorials and occasional queries to ChatGPT or Claude can help you learn, but projects are where that knowledge becomes yours. Choose data related to something you genuinely care about and build a small analysis around it. Kaggle or Google's Dataset Search may help you find a data set you will enjoy working with.

Import the data frame. Inspect it, validate it, and clean it. Document where it came from and what each row represents. Make a plot. Ask a question and run a simple analysis. Produce a table and write a short interpretation. Nobody is watching or grading you. Give yourself low-stakes spaces where you can experiment without worrying about the outcome.

A project gives programming a purpose. It forces different skills to interact and exposes weaknesses that isolated exercises often conceal. Projects also grow with you. Return to one a few years later, and you may be surprised by how much your skills have developed. It is a satisfying reminder of how much work it took to get there.

Aim to make your projects computationally reproducible from beginning to end. A useful test is to imagine deleting every derived data set, figure, table, and model output. Assuming you still have authorized access to the source data and required software, could you regenerate everything from your workflow?

While reproducibility is a critical target to work toward, do not interpret this as a requirement to regenerate every expensive result every time you open the project. Real scientific workflows often cache or checkpoint expensive intermediate results.

The important point is that those intermediate results have a documented computational chain of production: you know which inputs and code produced them, and you can deliberately rebuild them when necessary.

## 5.1 Distinguish exploratory code from durable research code

Not every line of code needs to be polished, generalized, or preserved forever. Exploratory work is often messy by design. You may try several transformations, make temporary plots, inspect unusual cases, fit models you later discard, or write a few awkward lines simply to understand what is happening.

Exploratory code is a normal part of scientific reasoning. A console, notebook, or script can be exactly the right place for this type of work. The important transition occurs when a computation begins to support a result you intend to rely on, share, publish, or build upon.

At that point, move your code into a more durable workflow. Give important steps stable inputs and outputs. Remove accidental dependencies on your interactive session. Replace unexplained manual edits with code or documentation. Add validation steps. Record parameters and software dependencies. In sum, make it possible for you or a collaborator to rerun the analysis deliberately.

Think of project development as a spectrum rather than a binary distinction:

```
exploration → working analysis → durable research workflow
```

The structure of your code should reflect its development stage and expected reuse. Do not spend an hour engineering a reusable function for a calculation you will perform once and discard ten minutes later—unless, of course, writing that function is itself the learning exercise, in which case that hour may be very well spent.

Premature abstraction can slow research just as surely as disorganized code can. First understand the problem. Then decide which parts deserve to become durable.

A useful principle is:

> **Explore freely, but formalize the computations that become part of the scientific record.**

Likewise, make clean, readable, well-documented code a habit from the beginning. The Google Style Guides provide a useful reference for consistent style.

### A prompt of mine

If useful, a prompt for refactoring, organizing, and commenting R code is available here: pretty-r-scripts.

Even if you are just practicing coding drills, get used to writing good code. Use meaningful object names, keep formatting consistent, and organize your logic so that you can understand it when you return later.

## 5.2 Design your research for reproducibility

Reproducible research has appeared several times throughout this guide for a reason. Reproducibility is not something to add once the analysis is finished. It should shape how you work from the very beginning.

Here, it is useful to define the term precisely. Computational reproducibility means obtaining consistent results using the same input data, computational steps, methods, code, and conditions of analysis(National Academies of Sciences, Engineering, and Medicine 2019). That is different from replicability, which the National Academies defines as obtaining consistent results across studies aimed at answering the same scientific question, each using its own data.

For computational work, reproducibility requires more than sharing a final data set or a folder of scripts. Ideally, you should be able to identify:

- The source and version of the input data.
- The code used for each transformation and analysis.
- Important configuration settings and parameters.
- The software and package versions.
- Random seeds or random-number settings when stochastic procedures are used.
- The sequence and dependencies among computational steps.
- Manual interventions that could not be automated.
- The provenance of generated figures, tables, and model outputs.

Think of reproducibility as a chain:

```
source data
    ↓
import and validation
    ↓
cleaning and transformation
    ↓
derived data
    ↓
models or analyses
    ↓
figures and tables
    ↓
manuscript or report
```

The fewer unexplained manual steps between those stages, the stronger your reproducible workflow becomes.

Record the broader computational environment when exact regeneration matters. Random seeds deserve one qualification: setting a seed is often necessary to make stochastic analyses reproducible, but a seed alone does not guarantee identical results across software versions, algorithms, parallelization strategies, or hardware.

On the other hand, reproducibility does not mean that every data set belongs in version control or must be made public. Ethical, legal, contractual, licensing, privacy, and security restrictions may constrain what you can share. When they do, document those constraints and explain as clearly as possible how authorized researchers can obtain or reconstruct the required inputs.

For long or expensive analyses, it is reasonable to save intermediate results rather than recomputing everything constantly. The scientific requirement is that those artifacts have traceable provenance and can be deliberately regenerated from known inputs and code.

While all these practices have scientific benefits, they also have an extremely practical one: reproducibility protects you from yourself. Graduate projects become complicated, and after a few months away, you will inevitably forget some of what you did. A well-organized, reproducible workflow serves as an external memory, preserving the logic and history of your research process.

# 6 Build the surrounding toolkit

Once you are reasonably comfortable with R and the fundamentals of working with data, start expanding your computational toolkit. The areas below are worth exploring, but not all at once. Add new tools as your research creates a reason to learn them.

## 6.1 Interacting with computational systems: Bash, shells, and the terminal

Learn how to communicate with your computer through a terminal. Many computer users first learn to work through graphical interfaces, clicking icons, opening folders, navigating menus. The terminal lets you perform many of the same operations by issuing commands directly.

At first, the terminal may seem like a primitive alternative to clicking through folders. It is more useful to think of it as a programmable interface to your operating system and the software running on it. Before long, you may be surprised by how often you choose to use the terminal.

You do not need to master the command line. Learn enough to navigate directories, manage files, and run programs without the terminal feeling like unfamiliar territory. You will begin combining commands, automating repetitive operations, and, importantly, using command-line tools that have no convenient graphical equivalent.

On macOS and Linux, you will commonly encounter shells such as Bash or Zsh. On Windows, you may use PowerShell, Git Bash, Windows Subsystem for Linux, or a remote Linux shell. The exact commands vary across environments, so focus first on the concepts rather than memorizing a single operating system's command set.

Some basic Unix-like commands worth learning include:

```
pwd

ls

cd
```

```
mkdir

cp

mv
```

You do not need the command-line expertise of a systems administrator, but you should become comfortable with the basic concepts that make the terminal intelligible: working directories, paths, standard output, environment variables, and passing arguments to programs.

The terminal becomes especially useful when your work leaves your personal computer. Remote servers, computing clusters, and cloud systems often rely on command-line interaction. Rather than keeping an analysis running in an open session, for example, you can submit it to a cluster scheduler such as Slurm and let the system execute it for you.

Treat destructive commands with respect. The terminal gives you considerable control over your system, often without the safeguards and confirmation dialogs you may expect from a graphical interface.

## 6.2 Git and collaborative version control

Version control is one of the highest-return skills in computational research because your work will never remain still for long. You will revise code, reconsider analytical decisions, and experiment with alternatives, sometimes only to discover that an earlier approach was better. Version control lets you experiment without losing the path back.

Git is the standard tool for giving your computational work a history. At its core is the commit: a recorded snapshot of your tracked files, accompanied by metadata and a brief message describing what changed.

A version-control system records the history of your project, allowing you to inspect changes, compare versions, and recover earlier states without accumulating duplicated files. That history also makes experimentation safer and collaboration much easier.

If something breaks, you can inspect earlier versions. When collaborating, you can review changes instead of emailing files back and forth. And when you return to a project months later, the commit history helps you reconstruct what happened and pick up where you left off.

At some point, you will hear about GitHub. Git and GitHub are not the same thing. GitHub is a popular platform for hosting Git repositories remotely, making it useful for collaboration, code review, project documentation, and sharing software. Depending on your research team's needs, you may instead encounter GitLab, Bitbucket, or a self-hosted Git service.

You do not need to master Git immediately. Start with a small workflow:

```
git status

git add

git commit

git push
```

Then gradually learn pulling, branches, merging, `.gitignore` files, pull and merge requests, and conflict resolution as you need them. GitHub also provides useful tools beyond version control, such as GitHub Classroom for managing coursework and GitHub Pages for publishing websites with little deployment work.

Do not worry if Git and GitHub feel elusive at first. Their value often becomes clearer as your projects grow and you make a few mistakes. Generative AI can make version control much more approachable: tell the model what you want to accomplish with your repository, and let it translate that objective into an appropriate sequence of commands while explaining each step along the way.

In collaborative research, version control can serve as a form of scientific quality control. Small, coherent commits make changes easier to inspect, while pull or merge requests allow collaborators to review both the code and the analytical decisions it implements.

Not everything in a research project belongs in GitHub. Keep large data sets and generated outputs elsewhere when appropriate, and never commit credentials, API keys, personally identifiable information, restricted research data, or other sensitive files. Use institutional storage, data repositories, or specialized data-versioning tools for materials that require them, and configure `.gitignore` (a dedicated file where you specify which files and folders Git must ignore) so that material that should remain outside version control stays there.

Start getting familiar with version control as soon as possible. This is one useful exception to the general principle of learning new tools as research needs arise: version control becomes more valuable when you begin using it before you need to recover an earlier version, trace a change, or coordinate work with collaborators.

Practice with a toy working directory containing unimportant files where you can experiment freely without worrying about the consequences of mistakes. Learning Git alongside a small, evolving project will make its logic much clearer than introducing it only after your work has become complicated.

## 6.3 SQL

At some point, learn the basics of SQL, or Structured Query Language. SQL lets you communicate with relational databases, where information is typically distributed across tables connected through shared fields or keys.

CSV files and spreadsheets dominate many graduate courses because they are convenient teaching tools, but they represent only part of the data environments you will encounter. In many organizations, the data you need will live in relational databases.

You do not need advanced database engineering skills. Begin with the core ideas:

- Tables.
- Rows and columns.
- Primary and foreign keys.
- Filtering.
- Aggregation.
- Joins.

Beyond SQL, the way you organize information into tables is far more consequential than it may seem. Broman and Woo's *Data Organization in Spreadsheets* (Broman and Woo 2018) is an instructive resource for learning good practices.

You do not need to become a database engineer. A few useful SQL commands are worth knowing:

```
SELECT

FROM

WHERE

GROUP BY

ORDER BY

JOIN
```

Learning SQL is valuable even if you rarely write SQL. Its deeper contribution is developing a relational mindset: understanding which data belong together, which should be stored separately, and how different pieces connect. That perspective carries into your work in R and, more broadly, into how you translate analytical problems into data requirements.

Another valuable database concept is the distinction among conceptual, logical, and physical data models. There are good online materials to learn them (IBM n.d.). Together, these levels trace the path from identifying the entities and relationships that matter, to organizing them into a data structure, to implementing that structure in a particular database system.

For applied social scientists who regularly work with relational databases, learning some SQL may be more useful than immediately adding another general-purpose programming language.

## 6.4 Develop intuition for computational scale

Most first-year exercises are small enough that you rarely need to think about computational resources. As your research grows, however, you will discover that data and algorithms have

costs. Your projects consume memory, occupy storage, and take time to process. Designing them with computational efficiency in mind therefore becomes increasingly important.

A script that works comfortably on a 10,000-row data set may behave very differently on 100 million rows. Reading an entire data set into memory may be reasonable on your laptop for one project and impossible for another. A model that takes five seconds on a small sample may take hours, days, or far longer when repeated across many simulations, bootstrap samples, input files, parameter combinations, or model specifications.

Fortunately, you do not need a formal education in algorithmic complexity to use computational resources sensibly. A few well-chosen habits will serve you well:

- Test unfamiliar workflows on small subsets before launching large jobs.
- Identify which steps dominate runtime or memory use.
- Avoid recomputing expensive results unnecessarily.
- Save well-defined intermediate outputs when recomputation is costly.
- Profile your code before attempting to optimize it.
- Move to databases, chunked processing, parallel computing, remote servers, GPUs, or clusters only when your computational needs justify the additional complexity.

Resist the instinct to *vectorize* or *parallelize* everything automatically. Vectorized R operations are often efficient and expressive, but they can create large temporary objects and increase memory use. Parallel computation can accelerate suitable workloads, but it also introduces coordination overhead, memory pressure, potential nondeterminism, and additional debugging complexity. You probably do not need to worry much about these tradeoffs in your first year, but remember that they exist. They may matter sooner than you expect.

Some research problems will eventually push you toward larger-than-memory tools, distributed systems, GPUs, computing clusters, or cloud infrastructure. When that happens, expand your toolkit accordingly. Throughout this guide, the principle is the same: let the demands of the research determine what you learn next.

Performance matters, but not at the expense of correctness and transparency. First make the computation correct and inspectable. Then identify where the bottleneck actually lies before optimizing it.

## 6.5 Manage software environments and dependencies

Code does not run in isolation. Every project operates within a computational environment shaped by the software, packages, and versions installed on your machine, including dependencies that may predate the project itself. Those dependencies affect whether the code runs as expected and whether someone else can reproduce the analysis on another machine.

As discussed earlier, preserving source code is only part of reproducibility. Your scripts run within an environment that includes particular versions of R or Python, packages, system libraries, and external programs. If that environment changes, the same code may behave differently or fail altogether.

Reproducibility therefore extends to the computational environment in which your code runs. For an RStudio project, tools such as `renv` can record package versions in a lockfile and create a project-local library, allowing you or a collaborator to reinstall the same package versions later.

In Python, tools such as `venv` create isolated environments so that each project can have its own dependencies, while `conda` can manage both environments and packages, including some dependencies outside Python itself. Newer tools such as `uv` can manage environments, install packages, and record project dependencies in a reproducible way.

The particular language or tool matters less than the underlying principle: your project should explicitly record and isolate its dependencies instead of relying on whatever happens to be installed on your machine.

For more demanding projects, you may want to package the computational environment itself. Container technologies such as Docker or Apptainer (formerly known as Singularity) make that environment more portable, which is particularly useful when the same analysis must move between your laptop, a remote server, and a computing cluster.

Environment management is not something you need to master during your first year. The principle behind it, however, is worth learning early:

> **Reproducibility requires information about software dependencies, not just source code.**

For important analyses, leave behind enough information to reconstruct the computational environment, beginning with the versions of the programming language and packages you used.

## 6.6 Add validation, assertions, and tests as projects grow

You do not need to engineer every analysis like production software. But once a project becomes difficult to verify manually, especially as it grows or involves repeated runs and collaborators, automated checks become an important safeguard.

A check can be as small as:

```
stopifnot(nrow(survey) > 0)
```

Or:

```
stopifnot(!anyDuplicated(survey$id))
```

A few conditions are particularly useful to check:

- Required columns are present.
- Identifiers are unique when they should be.
- Values stay inside valid ranges.
- Joins have the expected number of matched and unmatched cases.
- A function returns the expected answer for a small known example.
- An important transformation does not, by itself, change the number of observations.

Scale your testing practices to the work at hand. A reusable function or research software project may benefit from formal unit testing with a framework such as `testthat` in R. For a one-off analysis, a few targeted assertions and validation summaries may capture most of the benefit.

The point, as throughout this guide, is not to turn your research into a software-engineering exercise. It is to make silent scientific errors less likely and easier to detect.

## 6.7 Think in workflows, not just scripts

A small analysis may be perfectly manageable with scripts numbered in execution order:

```
01_import.R

02_clean.R

03_analyze.R

04_visualize.R
```

As a project becomes more complex, the relationships among computations become more important than the file names. You may have a workflow in which several data sets feed a cleaning step, several models depend on the cleaned data, and many figures depend on those models. Re-running everything manually becomes slow and error-prone.

Tools such as `targets` in R, `make`, Snakemake, and Nextflow allow you to describe a project in terms of computational dependencies rather than a sequence of scripts. The workflow management system can then determine what has become outdated after a change and recompute only what is necessary, a major advantage when analyses are expensive or highly interconnected.

But you do not need a workflow manager for every first-year project. For now, recognizing the underlying idea is enough: many scientific workflows can be represented as dependency graphs; later computations depend on particular inputs and earlier computational steps. Mature computational workflows make those dependencies explicit.

## 6.8 Distinguish analysis code from research software

This distinction parallels the one made earlier between exploratory and durable research code. Not all scientific code needs the same degree of engineering. A script written for a single dissertation chapter serves a different purpose from functions maintained by a research lab, and both differ from a software package used by hundreds of researchers.

As the expected reuse and consequences of failure increase, so should the investment in documentation, testing, interface design, maintenance, and review. Recognizing where your project falls along this spectrum helps you invest engineering effort where it actually pays off.

Analysis code is usually written to answer a particular scientific question with particular data. It can legitimately contain project-specific assumptions, variable names, file structures, and em-

pirical choices. It should still be readable, validated, version-controlled, and reproducible when it supports scientific claims, but it does not need to become a general-purpose software product.

Research software is built for reuse across data sets, projects, users, or institutions. The broader that reuse becomes, the greater the need for stable interfaces, testing, documentation, error handling, systematic releases, and continued maintenance. A tool that supports the work of an entire lab therefore warrants substantially more engineering than a script written for a single exploratory analysis.

There is plenty of middle ground between one-off analysis code and full research software. A dissertation project may begin as a collection of scripts, gradually develop reusable functions, and eventually warrant an internal package. Let that evolution follow a practical need: abstraction should reduce duplication or clarify scientific logic, not add machinery for its own sake.

The standard should therefore be proportional to the consequences of failure and the expected reuse of the code:

> **The level of engineering should match the scientific role of the code, neither falling short nor adding complexity without purpose.**

Always be clear about what kind of project you are building and what level of engineering it requires. Matching engineering effort to a project's role does not mean abandoning good research practices. Even a script intended only for your own computer should be documented and reproducible.

## 6.9 Connecting computation to research outputs

Bringing computation and writing into the same workflow can make research substantially more reproducible. Quarto provides an excellent environment for integrating code, results, and prose in a single document.

**Blog**

Quarto is discussed in more detail on the accompanying data literacy blog.

Quarto allows you to combine prose and computation in a single source document, including citations, mathematical notation, executable R, Python, or Julia code, figures, tables, and generated results. The same source can then be rendered to formats such as HTML, PDF, or Word.

For the kinds of analyses you will develop in graduate school, Quarto can spare you a familiar cycle: copy a regression table into Word, change the model, forget to update the table, revise the analysis again, update one figure but not another, and eventually lose track of which output belongs to which analysis.

A computational document can instead make the relationship explicit:

```
data → code → analysis → figure/table → document
```

Integrating computation and writing does not require putting the entire analysis inside the manuscript. In larger projects, it is often better to let a separate analytical pipeline produce validated, versioned results and use Quarto as the reporting layer that reads validated outputs from the analytical pipeline and generates the manuscript's tables, figures, and references.

The point is not to squeeze an entire research pipeline into one document. It is to maintain a traceable connection between the computations you perform and the results you report, without relying on manual copy-and-paste steps. Quarto extends that integration to citations as well, allowing you to manage references through a BibTeX bibliography.

LaTeXwill probably enter your workflow eventually, particularly if your field relies on mathematical notation or journal templates built around it. Quarto can handle much of the interaction for you, so there is little reason to master LaTeXimmediately. But being able to read and make basic modifications to LaTeXis a useful skill to develop over time.

Collaborative writing introduces its own version-control and review requirements. Quarto or LaTeXunder Git may work well for some teams, while others depend on Word and tracked changes. The important skill is not loyalty to a particular tool, but preserving provenance while adapting to how your collaborators and co-authors actually work.

## 7 What you do not need yet

A guide of this depth can accidentally make computational research sound as though you need to learn an entire technology stack before you are ready to analyze data. That is not the case.

During your first year, not every script needs to become a package, every analysis a managed pipeline, every calculation a function, or every project an elaborate repository. These practices exist to solve particular problems. Learn and adopt them as those problems emerge, or use lower-stakes projects to practice them when you have the time.

You probably do not need to master Docker, Kubernetes, cloud architecture, distributed computing, GPU programming, workflow managers, continuous integration, package development, high-performance computing, database administration, multiple programming languages, or advanced software testing. Some researchers will eventually need several of these tools. Others will conduct substantial computational research without touching most of them.

What you do need is enough awareness to recognize when your current way of working is becoming fragile:

- If you repeatedly lose track of file versions, learn Git.
- If collaborators cannot recreate your environment, learn dependency management.
- If a pipeline has many interdependent, expensive steps, consider using a workflow manager.
- If the data no longer fit comfortably in memory, learn techniques appropriate to that scale.
- If the same code is reused across projects, consider turning it into reusable research software.

If you really need those tools, the need will emerge naturally from your research. Your first-year objective is much more modest:

> **Become competent with a core set of tools, develop sound computational habits, and learn to recognize gaps in your mental models and address them through continued learning.**

And expect some friction along the way. It is often a sign that you are extending your expertise beyond familiar territory.

## 8 What to prioritize during the first year

A practical first-year priority order is:

1. **R fundamentals (or the primary language used in your research environment):** develop a programmer's mindset through regular practice; get used to reading, writing, and modifying code rather than relying on code copied and pasted from a chatbot.
2. **Algorithmic thinking and pseudocode:** learn to translate problems into a sequence of manageable steps before coding, then follow that logic systematically as you implement the solution.
3. **Objects, data structures, and data types:** know what you are manipulating; many beginner errors come from applying the right function to the wrong kind of object, or the wrong function to the right object.
4. **Project organization, paths, and data provenance:** understand where files live in your machine, how code finds them, and where outputs come from; keep the path from data to reported results traceable.
5. **Data import, inspection, manipulation, and validation:** become comfortable getting data sets in, examining them, transforming them, and checking your work; cleaning data frames yourself is also how you learn what they contain.
6. **Data visualization:** learn how an object's structure determines how it can be visualized; use visualization to investigate data, not just to present results.
7. **Debugging, documentation, assertions, and basic testing:** spend time diagnosing failures rather than just copy-pasting code until it runs; leave enough documentation and checks to catch mistakes early.
8. **Basic statistics and quantitative reasoning:** build intuition about uncertainty, comparison, variation, association, and estimation; understand what your computations mean, not just how to code them.
9. **Mathematics as needed to understand your methods:** master the mathematics your methods require as those needs arise; fill conceptual gaps rather than trying to master everything in advance.
10. **Reproducible computational documents with Quarto:** connect prose, code, figures, and results so that reported claims remain tied to their computations; the goal is traceability, not putting everything in one file.

Priorities depending on your program or institutional role:

11. **Git and collaborative version control:** learn to track changes, recover previous work, and collaborate without proliferating copies; preserve provenance while adapting to the tools your collaborators use.
12. **Command-line basics:** learn to navigate directories, inspect files, and run programs from the terminal; you do not need to live there, but you should know your way around.
13. **Software environments and dependency management:** understand that reproducibility depends on packages and versions as well as code; learn to make those dependencies explicit.
14. **SQL and relational data:** learn to think in terms of tables, keys, and relationships; relational thinking becomes increasingly useful even if you never use SQL.
15. **Workflow and pipeline concepts for larger projects:** organize research projects as connected computational stages with explicit inputs and outputs; adopt this structure as projects outgrow particular scripts.
16. **More advanced statistical or computational methods:** expand your methodological toolkit once the foundations are in place; advanced methods are more intuitive when you can understand and inspect what less advanced methods do.
17. **Additional programming languages when a research need emerges:** learn another language when it solves a real problem; do not collect languages simply because they appear on lists of desirable skills.
18. **Remote computing, clusters, containers, or parallel computing when your work actually requires them:** adopt these tools when your research begins to demand them, or when you have bandwidth to practice; they are useful capabilities, not first-year prerequisites.

The ordering is not rigid. You will learn many of these things simultaneously, and your lab or field may reasonably change the sequence and expectations. More importantly, these are not eighteen subjects you need to "finish" during your first year.

Do not collect tools in search of problems. Let research problems, and the computational environment in which your collaborators actually work, pull you toward new tools.

## 9 Treat your first year as an investment

The first year of a PhD can create pressure to produce immediately. But time spent building foundational computational skills is not time taken away from research. In fact, it is an investment in every subsequent research project. You will become faster and more resourceful, and the range of social science questions you can pursue will expand with you.

The returns compound. An operation that takes you three hours today may take you ten minutes next year. A workflow that initially feels cumbersome may eventually allow you to analyze hundreds of files systematically. A programming concept that seems abstract today may become the foundation of your dissertation several years from now.

Above all, learn how to keep learning. Your first year is not about becoming an expert programmer or accumulating technologies. It is about establishing a trajectory: developing sound

computational habits, building better mental models, and becoming increasingly capable of figuring out unfamiliar problems on your own.

You are building a way of thinking and working that can support reliable, transparent, and increasingly ambitious research throughout your career. By the end of the year, you will still have plenty to learn. That is precisely the point. What should have changed is your ability to approach new computational problems systematically and continue expanding what you can do.

# 10 References

Broman, Karl W, and Kara H Woo. 2018. "Data Organization in Spreadsheets." *The American Statistician* 72 (1): 2–10.

De Mesquita, Ethan Bueno, and Anthony Fowler. 2021. *Thinking Clearly with Data: A Guide to Quantitative Reasoning and Analysis*. Princeton University Press.

Gelman, Andrew, Jennifer Hill, and Aki Vehtari. 2020. *Regression and Other Stories*. Analytical Methods for Social Research. Cambridge, United Kingdom: Cambridge University Press.

IBM. n.d. "What Is Data Modeling?" Accessed August 21, 2026. https://www.ibm.com/think/topics/data-modeling.

Imai, Kosuke, and Nora Webb Williams. 2022. *Quantitative Social Science: An Introduction in Tidyverse*. Princeton University Press.

Llaudet, Elena, and Kosuke Imai. 2023. *Data Analysis for Social Science: A Friendly and Practical Introduction*. Princeton University Press.

National Academies of Sciences, Engineering, and Medicine. 2019. *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press. https://doi.org/10.17226/25303.

Rogers, Simon, and Mark Girolami. 2016. *A First Course in Machine Learning*. CRC press.

The Carpentries. 2025. "Building Skill with Practice." The Carpentries; https://preview.carpentries.org/instructor-training/02-practice-learning.html#building-a-mental-model.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software* 59: 1–23.

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. *R for Data Science*. 2nd ed. O'Reilly Media. https://r4ds.hadley.nz/.

Wilke, Claus O. 2019. *Fundamentals of Data Visualization*. O'Reilly Media, Incorporated.

Wooldridge, Jeffrey M. 2019. *Introductory Econometrics: A Modern Approach*. 7th ed. Boston, MA: Cengage Learning.