Table of contents

Programming for Analytics

Student Edition

By Kimberly Seefeld, EdD

Published under **DataJoyAI** imprint

Companion resources for this book are available at:

https://datajoyai.com

Edition: First Edition
ISBN: 978-1-969233-36-4

1 Why Code

1.1 Learning Outcomes

1-1. Explain why programming is valuable for data analysis beyond spreadsheet-based workflows.
1-2. Distinguish programming for data analysis from computer science and software development.
1-3. Describe how programming supports reproducibility, automation, and transparency in data work.
1-4. Recognize how AI tools are changing what it means for analysts to "know how to code."

1.2 Introduction

If you've picked up this book, chances are you're feeling a mix of curiosity and anxiety about programming. Maybe you've heard that "everyone needs to learn to code" but you're not entirely sure why. Perhaps you've been getting by just fine with Excel and are wondering what all the fuss is about. Or maybe you've tried programming before and found it frustrating, intimidating, or just plain confusing.

Let me start with some good news: you don't need to become a computer scientist to use programming effectively in data analysis. You don't need to memorize syntax, build complex algorithms, or understand the inner workings of computers. What you do need is to understand why programming has become such a powerful tool for working with data—and how recent advances in artificial intelligence have fundamentally changed what it means to "know how to code."

This chapter is about removing the fear and mystery around programming by explaining exactly why it matters for data work. We'll explore what changes when you move from spreadsheets to scripts, why reproducibility and automation matter, and how AI tools are revolutionizing the way analysts approach programming. By the end, you'll understand that programming isn't about becoming a technical wizard—it's about thinking more clearly and working more effectively with data.

1.3 From Spreadsheets to Scripts

Most of us start our data analysis journey with spreadsheets. Excel, Google Sheets, or similar tools feel natural and intuitive. You can see your data, click on cells, drag formulas, and create charts with a few mouse clicks. So why would anyone want to give up this visual, immediate approach for typing cryptic commands into a blank screen?

The answer becomes clear when you start working with real data analysis challenges. Let's consider a typical scenario that many students and analysts face.

Imagine you're analyzing survey data from 500 students about their study habits. In Excel, you might:

1. Open the data file
2. Create a new column for calculated values
3. Use formulas to clean up inconsistent entries
4. Create pivot tables to summarize results
5. Generate charts to visualize patterns
6. Copy and paste results into a report

This works fine for a one-time analysis. But what happens when:

- Your advisor asks you to update the analysis with 200 more survey responses?
- You discover an error in your data cleaning and need to redo everything?
- You want to apply the same analysis to survey data from a different semester?
- Someone else needs to verify your results or build on your work?

With the spreadsheet approach, you're back to square one. You'll need to manually repeat every step, hoping you remember exactly what you did and in what order. This is where the fundamental difference between spreadsheets and scripts becomes apparent.

A script is simply a written record of the steps you want the computer to perform. Instead of clicking and dragging, you write instructions like "read this data file," "calculate the average for each group," and "create a bar chart showing the results." The magic isn't in the complexity of these instructions—it's in the fact that they're written down and can be run repeatedly.

Think of it this way: a spreadsheet analysis is like giving someone verbal directions to your house each time they visit. A script is like writing down those directions once so anyone can follow them perfectly, every time. The directions themselves aren't complicated—the value is in having them recorded and repeatable.

This shift from manual steps to written instructions is the core difference between spreadsheet work and programming. You're not learning to think like a computer; you're learning to document your thinking in a way that computers can follow.

1.4 Reproducibility

The word "reproducibility" might sound academic, but it addresses a very practical problem that every data analyst faces: being able to recreate your own work. How many times have you looked at a spreadsheet you created weeks ago and struggled to remember exactly what you did? How often have you needed to explain your analysis to someone else and found yourself saying, "Well, I think what I did was…"?

Reproducibility in data analysis means that you (or someone else) can take your raw data and your analysis instructions and arrive at exactly the same results. This isn't just about being thorough—it's about being able to trust your work and build on it over time.

Let's consider what reproducibility looks like in practice. Suppose you're analyzing data about plant growth under different light conditions for a biology class. In a spreadsheet, your analysis might involve:

- Manually removing outliers that "look wrong"
- Creating calculated columns with formulas
- Generating summary statistics
- Making charts with specific formatting

Six months later, when you want to include this analysis in a larger project, you face a problem. Which data points did you remove and why? What exactly were those formulas? How did you create those charts? Even if you kept the original spreadsheet, the steps you took to create it are invisible.

With a script-based approach, every step is documented. Your code shows exactly which data points were removed and why. The formulas are written out explicitly. The chart creation process is recorded step by step. Most importantly, you can run the entire analysis again with a single command, producing identical results.

This documentation isn't just helpful for your future self—it's essential for collaboration and verification. When you share your analysis with others, they can see not just your results but exactly how you arrived at them. They can check your work, suggest improvements, or apply your methods to their own data.

The reproducibility that programming provides isn't about following rigid academic standards—it's about creating work that you can trust, build upon, and share with confidence.

1.5 Automation: Saving Time on Repetitive Tasks

One of the most immediate benefits of programming becomes apparent when you find yourself doing the same analysis repeatedly. Maybe you need to process monthly sales reports, analyze weekly survey data, or generate regular updates for a research project. The first time you do this analysis, programming might actually take longer than using a spreadsheet. But the second time, third time, and every time after that, the time savings become dramatic.

Automation in data analysis isn't about replacing human judgment—it's about eliminating the tedious, error-prone parts of analysis so you can focus on interpretation and insight. Consider the difference between these two approaches to a recurring analysis task:

Manual approach (each month): 1. Download new data file 2. Open Excel and import the data 3. Clean up formatting issues 4. Apply the same calculations as last month 5. Update charts and tables 6. Copy results into a report template 7. Double-check everything for errors

Automated approach (after initial setup): 1. Run the analysis script 2. Review the generated report

The automated approach doesn't eliminate your role as an analyst—it eliminates the repetitive mechanics so you can spend your time on the parts that actually require human insight: interpreting results, identifying patterns, and making decisions based on the analysis.

This automation becomes even more powerful when you're working with larger datasets or more complex analyses. Tasks that would take hours of manual work can be completed in minutes. More importantly, automated analyses are consistent—they apply exactly the same logic every time, reducing the risk of human error that comes with repetitive manual work.

The key insight here is that automation isn't about being lazy—it's about being strategic with your time and mental energy. By automating the routine parts of analysis, you free yourself to focus on the creative and interpretive aspects that actually add value.

1.6 Transparency

In our current era of data-driven decision making, the ability to show your work has become increasingly important. Whether you're a student submitting an assignment, a researcher publishing findings, or an analyst making business recommendations, people want to understand not just what your results are, but how you arrived at them.

Transparency in data analysis serves multiple purposes. First, it builds trust. When others can see exactly what steps you took, they can evaluate the appropriateness of your methods and the validity of your conclusions. Second, it enables learning and improvement. When your analysis process is visible, others can suggest better approaches or catch potential errors. Third, it facilitates collaboration and knowledge sharing within teams and organizations.

Programming naturally creates this transparency because your analysis process is written down as code. Every decision you made, every calculation you performed, and every assumption you built into your analysis is explicitly

documented. This is fundamentally different from spreadsheet-based analysis, where much of your process remains hidden in the sequence of clicks and manual operations you performed.

Consider the difference in transparency between these two approaches to analyzing student performance data:

Feature	Spreadsheet Approach	Programming Approach
Visibility of results	Final results are visible in the spreadsheet.	Final results are produced by running the analysis code.
Documentation of calculations	Formulas can be examined cell by cell.	All calculations are explicitly written in code.
Data cleaning steps	Manual data cleaning steps are not visible or recorded.	Data cleaning decisions are explicitly documented in code.
Chart creation	The chart creation process is not documented.	Chart creation steps are reproducible and recorded.
Assumptions and reasoning	Assumptions and decisions are not formally recorded.	Comments in the code explain reasoning and assumptions.

This transparency isn't just about meeting academic or professional standards about creating work that can be understood, verified, and improved upon. When your analysis process is transparent, you're not just sharing results; you're sharing knowledge and methodology that others can learn from and build upon.

1.7 Programming vs. Computer Science

One of the biggest misconceptions that keeps people away from programming is the belief that you need to understand computer science to write useful code. This confusion is understandable—after all, programming is a fundamental tool

```
Example 2: One-time analysis (2 hours, once)
```

```
should_i_automate(2, 1, 6) # 2 hours, 1 time, 6 hours to automate
```

```
Manual time per year: 2 hours
Automation setup time: 6 hours
DO MANUALLY. Automation would cost 4 extra hours
```

```
print("\nExample 3: Monthly report (1 hour, 12 times/year)")
```

```
Example 3: Monthly report (1 hour, 12 times/year)
```

```
should_i_automate(1, 12, 8) # 1 hour, 12 times, 8 hours to automate
```

```
Manual time per year: 12 hours
Automation setup time: 8 hours
AUTOMATE! You'll save 4 hours per year
```

Automation has an upfront cost, so it's not always the right choice. Tasks that happen frequently or take significant time are usually strong candidates for automation, while one-off tasks are often better done manually. You should also consider how error-prone the manual process is and how long you expect to repeat the task. A thoughtful decision about when to automate ensures that your time is spent where it has the greatest impact.

AI Integration

AI tools can help you evaluate whether automation is worth the effort. If you describe the task — how long it takes, how often you do it, and what steps are involved — the AI can estimate the potential time savings and suggest whether automation is a good investment. It can also help you break down the automation work into smaller steps, identify parts that are easy to automate, or point out hidden complexity you might not have considered. This makes the decision process faster, clearer, and more grounded in real numbers.

9.10.2 Practice

Use the code below for all questions. You may choose either the R version or the Python version. The questions are the same for both.

9.10.2.1 R Version

```r
# Repetition example
math_grades <- c(85, 92, 78, 88, 95)
english_grades <- c(82, 89, 91, 76, 88)
science_grades <- c(90, 85, 83, 92, 87)

math_avg <- mean(math_grades)
english_avg <- mean(english_grades)
science_avg <- mean(science_grades)

# Automated version
check_grades <- function(grades) {
  avg <- mean(grades)
  pass <- sum(grades >= 70)
  return(list(average = avg, passing = pass))
}

# If-then logic
assign_letter <- function(score) {
  if (score >= 90) return("A")
  else if (score >= 80) return("B")
  else if (score >= 70) return("C")
  else return("F")
}

# Loop example
months <- list(
  jan = c(1200, 1500, 1800),
```

```
  feb = c(1400, 1700, 1200),
  mar = c(1600, 1300, 2000)
)

for (m in names(months)) {
  total <- sum(months[[m]])
  avg <- mean(months[[m]])
}
```

9.10.2.2 Python Version

```
# Repetition example
math_grades = [85, 92, 78, 88, 95]
english_grades = [82, 89, 91, 76, 88]
science_grades = [90, 85, 83, 92, 87]

math_avg = sum(math_grades) / len(math_grades)
english_avg = sum(english_grades) / len(english_grades)
science_avg = sum(science_grades) / len(science_grades)

# Automated version
def check_grades(grades):
    avg = sum(grades) / len(grades)
    passing = sum(1 for g in grades if g >= 70)
    return avg, passing

# If-then logic
def assign_letter(score):
    if score >= 90: return "A"
    elif score >= 80: return "B"
    elif score >= 70: return "C"
    else: return "F"
```

```
# Loop example
months = {
    "jan": [1200, 1500, 1800],
    "feb": [1400, 1700, 1200],
    "mar": [1600, 1300, 2000]
}

for name, sales in months.items():
    total = sum(sales)
    avg = sum(sales) / len(sales)
```

9.10.3 Questions

1. Identify one place in the code where the same logic is repeated manually. Quote the specific lines.
2. Rewrite that repeated logic using a function. Show the function definition.
3. Run your function on at least two different grade lists and show the output.
4. Modify the assign_letter function so that scores below 60 return “F” and scores between 60–69 return “D”. Show your updated code.
5. Use a loop to print the letter grade for each score in this list: c(95, 72, 61, 88) or [95, 72, 61, 88]. Show the loop and the output.
6. Identify one task in your own workflow (outside this lab) that could be automated. Describe the repeated pattern.
7. Estimate the time tradeoff: How long does the task take manually, how often do you do it, and how long would automation take?
8. Ask an AI assistant to suggest an automation strategy for the task you identified. Summarize one concrete suggestion it gave you.