# Muddy Points

## Exploratory Data Analysis and Data Visualization

### Thanks for the feedback

Thank you to everyone who submitted post-class feedback. I'm pleased to see that so many of you found the live coding helpful, and I appreciate the specific questions about R syntax, packages, and workflow.

### A note on pacing

Looking at the post-class survey for this lecture:

- 77.8% felt the pace was about right
- 14.8% felt it was slightly too fast
- 7.4% felt it was slightly too slow

This is excellent feedback overall. The strong central tendency around "about right" suggests the material was well-calibrated for the class level. For those who found the pace slightly too fast, remember that:

- The lectures are recorded and you can rewatch sections
- The .qmd files are available for you to work through at your own pace
- The homework gives you extended practice time with these concepts
- **The in-class examples directly mirror the homework structure** — if you can follow the class examples, you have the scaffolding you need

### Connection between class examples and homework

Several of you mentioned wanting examples that follow exactly what you'll need for homework. Good news: **that's exactly what the in-class examples are designed to do**.

The in-class .qmd file is a template for the homework. When you're working through homework problems, refer back to the class examples — you'll see the same patterns (filter → select → summarize, building ggplots step-by-step, using pipes to chain operations).

Going forward, I'll be more explicit about this connection:

- "This pattern of piping filter → select → summarize? You'll use this exact structure in homework problem X."
- "Notice how we're building this ggplot layer by layer — you'll follow this same approach for your visualizations."

## Packages: what you need and when

Several questions came up about R packages. Let me clarify:

### What are packages?

Packages are collections of functions that extend R's capabilities. Think of base R as a smartphone, and packages as apps you install to do specific tasks.

### The packages we're using in this course

For this course, you need these packages installed:

- **tidyverse** (this actually installs several packages including dplyr, ggplot2, and others)
- **janitor** (for the `adorn_` functions)

That's it for now. I'll let you know when we need additional packages.

### Install once, load each session

This is a common source of confusion:

**Installation** (do this once per computer):

```
install.packages("tidyverse")
install.packages("janitor")
```

**Loading** (do this once per R session):

```r
library(tidyverse)
library(janitor)
```

Think of it like this:

- **Installing** is like downloading an app to your phone (do once)
- **Loading with `library()`** is like opening that app (do every time you restart R)

#### Why do we need `library()` every session?

Yes, you need to run `library(package_name)` at the start of every R session (or at the top of every script/Quarto document). This tells R which packages you want to use for this particular analysis.

It might seem redundant, but it's actually helpful because:

- It makes your code more reproducible (others know exactly which packages you're using)
- It avoids conflicts when different packages have functions with the same name
- It keeps R's memory usage lower by only loading what you need

### R syntax: commas, quotes, and spacing

Several of you mentioned confusion with R syntax, especially around commas and quotes. This is completely normal — these small details take practice.

#### Common syntax patterns

**Function arguments are separated by commas:**

```r
# Correct
summarize(data, mean_value = mean(x), median_value = median(x))

# Incorrect - missing comma between arguments
summarize(data, mean_value = mean(x) median_value = median(x))
```

**Column names usually don't need quotes in tidyverse:**

```r
# Both work in tidyverse functions
select(data, Species, Sepal.Length)
select(data, "Species", "Sepal.Length")

# But unquoted is more common and easier to type
```

**Spacing doesn't usually matter:**

```r
# These are equivalent
x <- mean(y)
x<-mean(y)
x  <-  mean(  y  )
```

### When you get an "unexpected" error

If you get an error like "unexpected ',' in..." or "unexpected ')' in...", it usually means:

- Missing or extra comma
- Missing or extra parenthesis
- Unmatched quotes

**Debugging tip:** Look at the line number in the error message, then carefully check that line for:

1. Equal number of opening `(` and closing `)`
2. Commas between function arguments
3. Matching quotes (every `"` has a closing `"`)

### Code that was missing from class notes

One comment noted that some code pieces were missing from the class notes (like needing to reload packages). You're absolutely right — for people new to R, it's confusing when code doesn't run because a package wasn't loaded.

Going forward, I'll make sure the .qmd files are complete and runnable from top to bottom, including all necessary `library()` calls at the beginning.

## Understanding pipes: more than just readability

One question was: "Pipes. Do they just help the readability and flow of your code?"

Great question! The short answer is: **primarily yes, but they also help you avoid creating intermediate objects**.

### What pipes do

The pipe operator `%>%` takes the output from one function and passes it as the first argument to the next function.

**Without pipes** (nested functions, read inside-out):

```
summarize(
  filter(
    select(iris, Species, Sepal.Length),
    Sepal.Length > 5
  ),
  mean_length = mean(Sepal.Length)
)
```

**With pipes** (read top to bottom):

```
iris %>%
  select(Species, Sepal.Length) %>%
  filter(Sepal.Length > 5) %>%
  summarize(mean_length = mean(Sepal.Length))
```

### Why pipes matter

1. **Readability**: You can read the code in the order operations happen (like reading a recipe)
2. **Avoid intermediate objects**: You don't need to save temporary results
3. **Easier to modify**: You can add or remove steps without restructuring everything

**Without pipes, you'd need intermediate objects:**

```
step1 <- select(iris, Species, Sepal.Length)
step2 <- filter(step1, Sepal.Length > 5)
step3 <- summarize(step2, mean_length = mean(Sepal.Length))
```

**With pipes, it's one continuous flow:**

```
result <- iris %>%
  select(Species, Sepal.Length) %>%
  filter(Sepal.Length > 5) %>%
  summarize(mean_length = mean(Sepal.Length))
```

So while readability is the main benefit, pipes also make your code more concise and easier to maintain.

## Chaining dplyr verbs: when and how

One comment: "I'm a little confused on how to correctly chain dplyr verbs and when it is necessary to chain something"

### When to chain

You chain dplyr verbs (using pipes) when you want to perform multiple operations in sequence on the same dataset.

**Chain when operations are related:**

```
# Good use of chaining - logical sequence
iris %>%
  filter(Species == "setosa") %>%        # First, filter to one species
  select(Sepal.Length, Sepal.Width) %>%  # Then, select specific columns
  summarize(mean_length = mean(Sepal.Length))  # Finally, calculate summary
```

**Don't chain when operations are independent:**

```
# These are separate analyses, don't chain them
setosa_summary <- iris %>%
  filter(Species == "setosa") %>%
  summarize(mean_length = mean(Sepal.Length))

versicolor_summary <- iris %>%
  filter(Species == "versicolor") %>%
  summarize(mean_length = mean(Sepal.Length))
```

### How to chain correctly

Each verb in the chain:

1. Takes the result from the previous step as its first argument (automatically via `%>%`)
2. Returns a modified data frame
3. Passes that data frame to the next verb

**The pattern:**

```
data %>%
  verb1(arguments) %>%
  verb2(arguments) %>%
  verb3(arguments)
```

### Common chaining patterns you'll use

**Filter → Select → Summarize:**

```
data %>%
  filter(condition) %>%      # Reduce rows
  select(columns) %>%        # Reduce columns
  summarize(statistic)       # Calculate summary
```

**Group → Summarize:**

```
data %>%
  group_by(category) %>%     # Define groups
  summarize(mean = mean(x))  # Calculate within each group
```

**Filter → Mutate:**

```
data %>%
  filter(condition) %>%      # Subset the data
  mutate(new_col = x + y)    # Add a new column
```

You'll practice these patterns extensively in the homework.

## Understanding adorn functions

One comment: "I got a little lost with how Adorn works but I think I figured out my issue after reviewing the notes."

I'm glad you figured it out! For others who might still be unclear:

### What adorn functions do

The `adorn_` functions from the `janitor` package add nice formatting to tables, especially frequency tables created with `tabyl()` or `count()`.

**Common adorn functions:**

- `adorn_totals()` — adds row/column totals
- `adorn_percentages()` — converts counts to proportions
- `adorn_pct_formatting()` — formats proportions as percentages (e.g., "25.0%" instead of 0.25)
- `adorn_ns()` — adds counts alongside percentages

### How to use them

Adorn functions are meant to be **chained** (using pipes) after you create a table:

```
iris %>%
  count(Species) %>%                # Create frequency table
  adorn_totals("row") %>%           # Add total row
  adorn_percentages("col") %>%      # Convert to proportions
  adorn_pct_formatting(digits = 1) # Format as percentages
```

### The order matters

Apply adorn functions in this sequence:

1. `adorn_totals()` — add totals first
2. `adorn_percentages()` — then convert to proportions
3. `adorn_pct_formatting()` — then format the display
4. `adorn_ns()` — optionally add counts in parentheses

If you do them out of order (like formatting before calculating percentages), you'll get errors or unexpected results.

### Too many options vs. one clear path

One insightful comment: "It is confusing for people who are starting out to hear multiple options of how to do something. It may be more helpful to have examples that we work through that follow exactly what we will need to be able to do to complete the homework and master tasks."

This is excellent feedback. I want to address this directly:

#### My teaching approach and why

I tend to mention alternative approaches because:

- I want you to recognize different coding styles when you see them online or in others' code
- Some of you have prior R experience and benefit from seeing connections
- It helps build conceptual understanding of why we do things a certain way

#### But I hear you

For those new to R, multiple options can be overwhelming when you're just trying to learn **one** way that works.

#### What I'll do differently

Going forward, I'll use a **core path + alternatives** approach:

1. **Teach one clear method first** — this is the "homework way"
2. **Practice that method** with multiple examples
3. **Briefly mention alternatives** only after the core method is solid, clearly labeling them as "you might also see…" rather than "you could also do…"

For example:

> "For this course, we'll use `library(tidyverse)` at the start of our scripts. You might also see people use `require()` or load packages individually like `library(dplyr)`, but we'll stick with `library(tidyverse)` for consistency."

This way, you have a clear, reliable approach for homework while still being aware that other methods exist.

## Live coding pace and practical adjustments

Several comments mentioned the pace of live coding:

- "The class just felt fast and if there were any errors in my code, we had already moved on before I could see what the issue was."
- "The R stuff is still a little fast for me, especially with how small the screen is"
- "We just moved very quickly through everything but still didn't finish the entire lecture"

These are all valid concerns. Here's what I'll adjust for our next R-heavy session:

### Pacing changes

1. **Built-in debug time**: After complex examples, I'll pause for 1-2 minutes and ask "Anyone getting an error? Let me know in chat or raise your hand"
2. **Slower typing**: I'll be more deliberate about typing speed, especially for syntax-heavy sections
3. **Finish or cut strategically**: Rather than rushing to finish all slides, I'll plan to end at a logical stopping point, even if we don't cover everything

### Visual improvements

1. **Larger font in RStudio**: I'll increase the font size for better visibility
2. **Clear transitions**: I'll announce when switching between slides and RStudio (e.g., "Now I'm opening RStudio…")
3. **Zoom on key areas**: When typing complex syntax, I'll zoom in on the code

### Managing package installation delays

One comment: "Installing data packages always takes a few minutes or longer if there's an error. It seems instantaneous on your screen, and I'd hate to fall behind while waiting for something to load."

You're absolutely right. Here's what we'll do:

1. **Pre-class package list**: I'll post which packages to install before class
2. **Buffer time in class**: When we do install packages during class, I'll build in 3-5 minutes of buffer time
3. **Acknowledge the wait**: I'll explain that package installation takes time and that it's normal

## Practice makes perfect: the role of homework

One comment really captured the learning process well: "I think I just need some practice with all of the R code that we went over in class, it all feels pretty muddy at this point but I think working with it and doing more examples and making mistakes will help me make sense of it."

This is exactly right! Another comment echoed this: "The lecture material is helpful but I have trouble following the in class coding examples. I have to work through the homework in order to fully grasp concepts."

### This is normal and expected

Live coding in class is designed to:

- Show you what's possible
- Demonstrate the workflow
- Give you examples to reference
- Provide scaffolding for homework

It is **not** expected that you'll master everything during the live demo. Real learning happens when you:

- Work through homework problems at your own pace
- Make mistakes and debug them
- Apply concepts to different datasets
- Rewatch recorded sections when needed

### The class-homework connection

The in-class examples are **templates** for the homework. When you're stuck on homework, go back to the .qmd file from class and look for a similar pattern.

For example:

- **Homework asks**: Calculate mean Sepal.Length by Species
- **Class example showed**: Calculate mean bill_length by species (for penguins data)
- **Pattern is the same**: `group_by(category) %>% summarize(mean = mean(variable))`

#### If you're feeling lost during class

That's okay! Your strategies should be:

1. **Watch and absorb** rather than trying to type everything perfectly in real-time
2. **Use the .qmd file** as your reference when doing homework
3. **Rewatch recordings** for sections that went too fast
4. **Come to office hours** with specific questions from homework

One student noted: "doing the homework was far more informative" — this isn't a criticism of lecture, it's recognition that **doing** is how we learn coding. Lecture provides the framework; homework provides the practice.

### What's working well

Many of you identified clear points that I want to acknowledge and continue:

#### The .qmd file for following along

Multiple comments mentioned how helpful it was to have the Quarto document to code along with. I'll continue providing complete, runnable .qmd files for every R session.

#### Real-world examples and explanations

Several of you appreciated:

- The pipe operator analogy (keys, driving to work)
- Learning the history and context of packages
- Understanding how functions answer specific questions

I'll continue using concrete analogies and providing the "why" behind our tools.

#### Step-by-step building

You valued seeing how to construct things incrementally (especially ggplots). This scaffolded approach works well, so I'll maintain this pattern.

## Looking ahead

Based on your feedback, I'll:

1. Start with a clear list of required packages (posted before class)
2. Make the connection between class examples and homework more explicit
3. Use the "core path + alternatives" approach to avoid overwhelming you with options
4. Build in debug/catch-up time during live coding
5. Focus on finishing a coherent chunk of material rather than rushing through all slides

Thank you again for the thoughtful feedback. These muddy points help me teach more effectively, and your willingness to identify what's confusing is exactly what makes this process work.