Data Wrangling with the Tidyverse

Paul Kieffaber

09/02/2025

Introduction: What is Data Wrangling?

Before you can build a predictive model, create an insightful plot, or find a meaningful pattern, you must first confront the raw material of your work: the data itself. Data in its original form is rarely, if ever, ready for analysis. It might be structured inconveniently, contain errors, or be missing critical information. The process of taking this raw, messy data and transforming it into a clean, organized, and usable format is known as **data wrangling** (or sometimes *data munging*).

Think of it like being a chef. You cannot prepare a gourmet meal until you've washed the vegetables, trimmed the fat, and organized your ingredients. Data wrangling is the essential preparation that makes all the exciting parts of data science possible.

The 80/20 Rule: The Importance of Data Preparation

There is a well-known adage that governs the daily life of nearly every data analyst, scientist, and researcher, often referred to as the "80/20 Rule of Data Science." The rule states that a data professional will spend approximately **80% of their time discovering, cleaning, and reorganizing data**, and only **20% of their time on the "core" tasks of analysis and modeling**. While this might seem surprising, it's a reality born from a simple truth: data in the wild is almost never ready for a statistical model or a visualization.

The 80% is the foundational, often unglamorous, work of data wrangling. It's the process of transforming raw data into a pristine, structured dataset. The 20% is the payoff—the part where you uncover insights and generate value. Without a solid foundation, any analysis you build will be unreliable at best and dangerously misleading at worst. This principle is famously captured in the phrase *"garbage in, garbage out."* The skills you learn in this chapter are designed to ensure you work with "gold in, gold out."

Raw data comes with a wide array of potential problems, including:

- **Missing values** that need to be handled or removed.
- **Inconsistent formatting**, such as dates written in different ways ("10-03-2025" vs. "Oct 3, 2025") or categorical data with typos ("Williamsburg", "williamsburg", "Wmsbg").
- **Incorrect data types**, like numbers stored as text that prevent mathematical operations.
- **A structure that is not "tidy"**, where data is organized for human readability (like a wide spreadsheet) but is unsuitable for R's analytical functions.

Mastering the tools to efficiently tackle these issues is what turns the frustrating 80% of your time into a powerful and controlled part of your workflow, paving the way for the insightful 20% that follows.

Chapter Goals and Dataset

In this chapter, you will learn a complete data wrangling workflow using the core packages of the Tidyverse. By the end, you will be able to:

- Import data cleanly using `readr`.

- Select, filter, and create variables using `dplyr`.
- Reshape your data into a tidy format using `tidyr`.
- Summarize your data by groups.
- Create an insightful plot from your wrangled data using `ggplot2`.

Throughout this chapter, we will be working with a dataset containing historical weather and demographic data for our own location of Williamsburg, Virginia, exploring patterns and preparing the data for a hypothetical analysis.

The Tidyverse

While base R has many powerful tools for data manipulation, the **Tidyverse** offers an ecosystem of packages designed specifically for the daily tasks of a data scientist. The packages share an underlying design philosophy and grammar, making the code you write more consistent, readable, and effective.

At the heart of this philosophy is the concept of stringing commands together using the **pipe operator** (`%>%` or `|>`), which allows you to read a sequence of data transformations like a sentence rather than a series of nested, hard-to-read functions.

The Principle of Tidy Data

The ultimate goal of data wrangling within the Tidyverse is to produce **tidy data**. A dataset is considered "tidy" when it conforms to three simple rules:

1. **Every column is a variable.**
2. **Every row is an observation.**
3. **Every cell contains a single value.**

Consider the difference. A "messy" dataset might be structured for easy data entry, like this:

City	Temp_2024	Temp_2025
Williamsburg	62	64
Richmond	61	63

This format is difficult for a program to analyze. The tidy version of this data would be:

City	Year	Temperature
Williamsburg	2024	62
Williamsburg	2025	64
Richmond	2024	61
Richmond	2025	63

This "long" format is far more flexible and is the standard that Tidyverse tools are designed to work with.

The Foundation: Pipes and Data Ingestion

Before we can start wrangling, we need two fundamental tools: a way to write clean, readable code and a way to reliably get data into R. In this section, we'll cover the pipe operator, which is the syntax that makes the Tidyverse so intuitive, and the `readr` package, our tool for importing data.

The Key to Readability: The Pipe Operator (`%>%` and `|>`)

Imagine you want to perform a series of operations on some data. In base R, you might end up nesting functions, like this: `function_c(function_b(function_a(data)))`

This code is difficult to read because you have to work from the inside out. It's also hard to debug; if something goes wrong, it's not immediately clear at which step the error occurred.

The Tidyverse solves this problem with the **pipe operator**. The most common pipe, from the `magrittr` package (which is included in the core `tidyverse`), is `%>%`.

The pipe takes the result of the expression on its left and passes it as the **first argument** to the function on its right. This allows you to rewrite the nested code above as a linear chain of operations:

`data %>% function_a() %>% function_b() %>% function_c()`

This is revolutionary for readability. You can now read your code from left to right, like a sentence: "Take the data, then do function A, then do function B, then do function C."

A Simple, Non-Data Example Let's see this in action with a simple character string. Suppose we want to take a messy string, convert it to lowercase, replace a word, and then remove leading/trailing whitespace.

The nested approach would be confusing:

```r
trimws(gsub("world", "R", tolower("  Hello World!  ")))
```

```
## [1] "hello R!"
```

The piped approach is clear and sequential:

```r
# The tidyverse package loads the pipe for us
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.2     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

```r
"  Hello World!  " %>%
  tolower() %>%
  gsub(pattern = "world", replacement = "R") %>%
  trimws()
```

```
## [1] "hello R!"
```

> **A Note on the Native R Pipe:** Since R version 4.1.0, a native pipe, `|>`, has been built into R. It functions very similarly for simple cases. Throughout this chapter, we will primarily use the Tidyverse pipe, `%>%`, as it is the historical standard for these packages and offers some additional features we may see later.

Importing Your Data with `readr`

The `readr` package, part of the core Tidyverse, provides a fast and friendly way to read rectangular data from delimited files, like comma-separated value (.csv) files.

For our examples to be self-contained and reproducible, we will first create our own sample CSV file directly from our R script using the `writeLines()` function. In a real-world scenario, this file would already exist on your computer.

```r
# Define the content for our example CSV file in a multi-line string
csv_content <- "year,visitors_millions,revenue_millions
2022,1.5,50.2
2023,1.7,55.8
2024,1.6,53.1"

# Write the content to a file named 'williamsburg_visitors.csv'
writeLines(csv_content, "williamsburg_visitors.csv")
```

Now that we have simulated the existence of `williamsburg_visitors.csv`, we can read it with a single function, `read_csv()`:

```r
visitors <- read_csv("williamsburg_visitors.csv")
```

```
## Rows: 3 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): year, visitors_millions, revenue_millions
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
print(visitors)
```

```
## # A tibble: 3 x 3
##    year visitors_millions revenue_millions
##   <dbl>             <dbl>            <dbl>
## 1  2022               1.5             50.2
## 2  2023               1.7             55.8
## 3  2024               1.6             53.1
```

When we run `read_csv()`, it prints a helpful message listing the column specifications. It intelligently "guessed" that `year` is a double (a type of number), `visitors_millions` is a double, and `revenue_millions` is a double.

Understanding the Output: Tibbles

The `readr` functions don't return a standard `data.frame`; they return a **tibble**. Tibbles are the Tidyverse's modern take on data frames. They work like data frames but are tweaked to be more user-friendly:

- **Printing:** They only print the first 10 rows and as many columns as fit on screen, preventing you from accidentally printing thousands of rows in your console. They also display the data type under each column name.
- **Consistency:** They are more consistent with subsetting. For example, using `[` to select columns will always return another tibble, whereas a data frame might sometimes return a vector.

Inspecting Your Import with `glimpse()`

A great way to get a quick overview of your newly imported data is with the `glimpse()` function from `dplyr`. It provides a transposed view of your data, making it easy to see every column, its data type, and the first few values.

```r
glimpse(visitors)
```

```
## Rows: 3
## Columns: 3
## $ year              <dbl> 2022, 2023, 2024
## $ visitors_millions <dbl> 1.5, 1.7, 1.6
## $ revenue_millions  <dbl> 50.2, 55.8, 53.1
```

Troubleshooting Common Import Problems

Real-world data is rarely perfect. Let's create another, messier file to demonstrate how to handle common issues.

```r
# Define the content for our messy example file
messy_csv_content <- "# Data collected on: 2025-10-03
# Source: Colonial Williamsburg Archives
#
Jamestown,1607,104
Williamsburg,1699,500
Richmond,1737,250"

# Write the content to a file named 'messy_data.csv'
writeLines(messy_csv_content, "messy_data.csv")
```

If we try to read this directly, it will fail. However, we can use arguments within `read_csv()` to fix it: * `skip`: Skips a specified number of lines at the start of the file. * `comment`: Specifies a character that indicates a line is a comment and should be ignored. * `col_names`: Allows you to provide a character vector of column names.

```r
historic_sites <- read_csv(
  "messy_data.csv",
  skip = 3,
  col_names = c("settlement", "year_founded", "initial_population")
)
```

```
## Rows: 3 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): settlement
## dbl (2): year_founded, initial_population
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
print(historic_sites)
```

```
## # A tibble: 3 x 3
##   settlement   year_founded initial_population
##   <chr>               <dbl>              <dbl>
## 1 Jamestown            1607                104
## 2 Williamsburg         1699                500
## 3 Richmond             1737                250
```

With just a few arguments, we have successfully tamed a messy file and imported it into a clean, tidy tibble, ready for the next steps in our analysis.

The Core Verbs of `dplyr`: Transforming Your Data

Now that we know how to get data into R, we can begin the real work of wrangling. Our primary tool for this is the `dplyr` package, the workhorse of the Tidyverse. The power of `dplyr` lies in its consistent and intuitive design. It provides a set of functions, or "verbs," where each verb corresponds to a common data manipulation task.

These verbs are designed to be chained together using the pipe operator (`%>%`), allowing you to write a clean, step-by-step recipe for transforming your data. For the examples in this section, we will use the `starwars` dataset, which is included with `dplyr`.

```r
# Load the full tidyverse, which includes dplyr
library(tidyverse)

# Let's take a glimpse at our dataset
glimpse(starwars)
```

```
## Rows: 87
## Columns: 14
## $ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia Or~
## $ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2~
## $ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77.~
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown", N~
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light", "~
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue",~
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0, ~
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "female",~
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femini~
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan", "T~
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma~
## $ films      <list> <"A New Hope", "The Empire Strikes Back", "Return of the J~
## $ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "Imp~
## $ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1",~
```

As you can see, the `starwars` dataset contains information about characters from the Star Wars films, with a nice mix of character, numeric, and list columns.

Subsetting Your Data: `select()` and `filter()`

Two of the most fundamental data wrangling tasks are subsetting by columns and by rows.

Choosing Columns with `select()`

The `select()` verb is used to choose which columns you want to keep. You can specify columns by name.

```r
# Select only the name, height, and mass columns
starwars %>%
  select(name, height, mass)
```

```
## # A tibble: 87 x 3
##    name               height  mass
##    <chr>               <int> <dbl>
##  1 Luke Skywalker        172    77
##  2 C-3PO                 167    75
##  3 R2-D2                  96    32
##  4 Darth Vader           202   136
##  5 Leia Organa           150    49
##  6 Owen Lars             178   120
##  7 Beru Whitesun Lars    165    75
##  8 R5-D4                  97    32
##  9 Biggs Darklighter     183    84
## 10 Obi-Wan Kenobi        182    77
## # i 77 more rows
```

You can also use a range of helper functions for more powerful selections: * Use - to exclude columns: `select(-films, -vehicles, -starships)` * Use a range with `:` `select(name:mass)` * Use helper functions like `starts_with()`, `ends_with()`, and `contains()`:

```r
# Select all columns that contain the word "color"
starwars %>%
  select(name, contains("color"))
```

```
## # A tibble: 87 x 4
##    name               hair_color    skin_color  eye_color
##    <chr>              <chr>         <chr>       <chr>
##  1 Luke Skywalker     blond         fair        blue
##  2 C-3PO              <NA>          gold        yellow
##  3 R2-D2              <NA>          white, blue red
##  4 Darth Vader        none          white       yellow
##  5 Leia Organa        brown         light       brown
##  6 Owen Lars          brown, grey   light       blue
##  7 Beru Whitesun Lars brown         light       blue
##  8 R5-D4              <NA>          white, red  red
##  9 Biggs Darklighter  black         light       brown
## 10 Obi-Wan Kenobi     auburn, white fair        blue-gray
## # i 77 more rows
```

Choosing Rows with `filter()`

The `filter()` verb is used to choose which rows you want to keep based on logical conditions.

```r
# Find all droids
starwars %>%
  filter(species == "Droid")
```

```
## # A tibble: 6 x 14
##   name   height  mass hair_color skin_color  eye_color birth_year sex   gender
##   <chr>   <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
## 1 C-3PO     167    75 <NA>       gold        yellow           112 none  masculi~
## 2 R2-D2      96    32 <NA>       white, blue red               33 none  masculi~
## 3 R5-D4      97    32 <NA>       white, red  red               NA none  masculi~
## 4 IG-88     200   140 none       metal       red               15 none  masculi~
## 5 R4-P17     96    NA none       silver, red red, blue         NA none  feminine
## 6 BB8        NA    NA none       none        black             NA none  masculi~
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```r
# Find all characters from the planet Tatooine with a height greater than 100
starwars %>%
  filter(homeworld == "Tatooine", height > 100)
```

```
## # A tibble: 9 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 Luke Sky~    172    77 blond      fair       blue            19   male  mascu~
## 2 C-3PO        167    75 <NA>       gold       yellow         112   none  mascu~
## 3 Darth Va~    202   136 none       white      yellow          41.9 male  mascu~
## 4 Owen Lars    178   120 brown, gr~ light      blue            52   male  mascu~
## 5 Beru Whi~    165    75 brown      light      blue            47   fema~ femin~
## 6 Biggs Da~    183    84 black      light      brown           24   male  mascu~
## 7 Anakin S~    188    84 blond      fair       blue            41.9 male  mascu~
## 8 Shmi Sky~    163    NA black      fair       brown           72   fema~ femin~
## 9 Cliegg L~    183    NA brown      fair       blue            82   male  mascu~
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Inside `filter()`, you can use standard logical operators: * `&` or `,`: AND * `|`: OR * `!`: NOT * `%in%`: is in a set of values. This is a useful shortcut for multiple `|` conditions.

```
# Find all characters whose eye color is blue or red
starwars %>%
  filter(eye_color %in% c("blue", "red"))
```

```
## # A tibble: 24 x 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Luke Sk~    172    77 blond      fair       blue            19   male  mascu~
##  2 R2-D2        96    32 <NA>       white, bl~ red             33   none  mascu~
##  3 Owen La~    178   120 brown, gr~ light      blue            52   male  mascu~
##  4 Beru Wh~    165    75 brown      light      blue            47   fema~ femin~
##  5 R5-D4        97    32 <NA>       white, red red             NA   none  mascu~
##  6 Anakin ~    188    84 blond      fair       blue            41.9 male  mascu~
##  7 Wilhuff~    180    NA auburn, g~ fair       blue            64   male  mascu~
##  8 Chewbac~    228   112 brown      unknown    blue           200   male  mascu~
##  9 Jek Ton~    180   110 brown      fair       blue            NA   <NA>  <NA>
## 10 IG-88       200   140 none       metal      red             15   none  mascu~
## # i 14 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Ordering and Finding Uniques: `arrange()` and `distinct()`

Sorting Rows with `arrange()`

The `arrange()` verb reorders the rows of your data frame based on the values in one or more columns. By default, it sorts in ascending order.

```
# Arrange characters from shortest to tallest
starwars %>%
  arrange(height)
```

```
## # A tibble: 87 x 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Yoda         66    17 white      green      brown            896 male  mascu~
##  2 Ratts T~     79    15 none       grey, blue unknown           NA male  mascu~
##  3 Wicket ~     88    20 brown      brown      brown              8 male  mascu~
##  4 Dud Bolt     94    45 none       blue, grey yellow            NA male  mascu~
##  5 R2-D2        96    32 <NA>       white, bl~ red               33 none  mascu~
##  6 R4-P17       96    NA none       silver, r~ red, blue         NA none  femin~
##  7 R5-D4        97    32 <NA>       white, red red               NA none  mascu~
##  8 Sebulba     112    40 none       grey, red  orange            NA male  mascu~
##  9 Gasgano     122    NA none       white, bl~ black             NA male  mascu~
## 10 Watto       137    NA black      blue, grey yellow            NA male  mascu~
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
# To sort in descending order, use the desc() helper
starwars %>%
  arrange(desc(height))
```

```
## # A tibble: 87 x 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Yarael ~    264    NA none       white      yellow            NA male  mascu~
##  2 Tarfful     234   136 brown      brown      blue              NA male  mascu~
##  3 Lama Su     229    88 none       grey       black             NA male  mascu~
##  4 Chewbac~    228   112 brown      unknown    blue             200 male  mascu~
##  5 Roos Ta~    224    82 none       grey       orange            NA male  mascu~
##  6 Grievous    216   159 none       brown, wh~ green, y~         NA male  mascu~
##  7 Taun We     213    NA none       grey       black             NA fema~ femin~
##  8 Rugor N~    206    NA none       green      orange            NA male  mascu~
##  9 Tion Me~    206    80 none       grey       black             NA male  mascu~
## 10 Darth V~    202   136 none       white      yellow          41.9 male  mascu~
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
# You can sort by multiple columns to break ties
starwars %>%
  arrange(homeworld, name)
```

```
## # A tibble: 87 x 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Bail Pr~    191    NA black      tan        brown             67 male  mascu~
##  2 Leia Or~    150    49 brown      light      brown             19 fema~ femin~
##  3 Raymus ~    188    79 brown      light      brown             NA male  mascu~
##  4 Ratts T~     79    15 none       grey, blue unknown           NA male  mascu~
##  5 Lobot       175    79 none       light      blue              37 male  mascu~
##  6 Jek Ton~    180   110 brown      fair       blue              NA <NA>  <NA>
##  7 Nute Gu~    191    90 none       mottled g~ red               NA male  mascu~
##  8 Ki-Adi-~    198    82 white      pale       yellow            92 male  mascu~
##  9 Mas Ame~    196    NA none       blue       blue              NA male  mascu~
## 10 Mon Mot~    150    NA auburn     fair       blue              48 fema~ femin~
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Finding Unique Values with `distinct()`

The `distinct()` verb is used to find unique values or unique combinations of values across columns.

```
# Find all unique species in the dataset
starwars %>%
  distinct(species)
```

```
## # A tibble: 38 x 1
##    species
##    <chr>
##  1 Human
##  2 Droid
##  3 Wookiee
##  4 Rodian
##  5 Hutt
##  6 <NA>
##  7 Yoda's species
##  8 Trandoshan
##  9 Mon Calamari
## 10 Ewok
## # i 28 more rows
```

Creating New Information: `mutate()`

Perhaps the most powerful verb is `mutate()`, which allows you to add new columns or modify existing ones.

```
# The 'height' is in cm. Let's create a new column with height in meters.
starwars %>%
  select(name, height, mass) %>%
  mutate(
    height_m = height / 100,
    bmi = mass / (height_m ^ 2) # You can use a newly created column immediately!
  )
```

```
## # A tibble: 87 x 5
##    name               height  mass height_m   bmi
##    <chr>               <int> <dbl>    <dbl> <dbl>
##  1 Luke Skywalker        172    77     1.72  26.0
##  2 C-3PO                 167    75     1.67  26.9
##  3 R2-D2                  96    32     0.96  34.7
##  4 Darth Vader           202   136     2.02  33.3
##  5 Leia Organa           150    49     1.5   21.8
##  6 Owen Lars             178   120     1.78  37.9
##  7 Beru Whitesun Lars    165    75     1.65  27.5
##  8 R5-D4                  97    32     0.97  34.0
##  9 Biggs Darklighter     183    84     1.83  25.1
## 10 Obi-Wan Kenobi        182    77     1.82  23.2
## # i 77 more rows
```

`mutate()` is often combined with the `if_else()` function to create conditional columns.

```
# Create a category for characters based on their mass
starwars %>%
  select(name, mass) %>%
  mutate(
    mass_category = if_else(mass > 100, "Heavy", "Not Heavy")
  )
```

```
## # A tibble: 87 x 3
##    name                mass mass_category
##    <chr>              <dbl> <chr>
##  1 Luke Skywalker        77 Not Heavy
##  2 C-3PO                 75 Not Heavy
##  3 R2-D2                 32 Not Heavy
##  4 Darth Vader          136 Heavy
##  5 Leia Organa           49 Not Heavy
##  6 Owen Lars            120 Heavy
##   7 Beru Whitesun Lars    75 Not Heavy
##   8 R5-D4                 32 Not Heavy
##   9 Biggs Darklighter     84 Not Heavy
##  10 Obi-Wan Kenobi        77 Not Heavy
## # i 77 more rows
```

Managing Columns: `rename()`, `relocate()`, and `transmute()`

These are helpful "housekeeping" verbs for cleaning up your final dataset.

- `rename()`: Renames columns. The syntax is `rename(new_name = old_name)`.
- `relocate()`: Changes the order of columns.
- `transmute()`: A special version of `mutate()` that *only* keeps the columns you create.

```r
starwars %>%
  # Rename the 'name' column to be more descriptive
  rename(character = name) %>%
  # Create a new bmi column
  mutate(
    height_m = height / 100,
    bmi = mass / (height_m ^ 2)
  ) %>%
  # Move the new bmi column to be right after mass
  relocate(bmi, .after = mass) %>%
  # Select a few columns to show the result
  select(character, mass, bmi, height_m)
```

```
## # A tibble: 87 x 4
##    character           mass   bmi height_m
##    <chr>              <dbl> <dbl>    <dbl>
##  1 Luke Skywalker        77  26.0     1.72
##  2 C-3PO                 75  26.9     1.67
##  3 R2-D2                 32  34.7     0.96
##  4 Darth Vader          136  33.3     2.02
##  5 Leia Organa           49  21.8     1.5
##  6 Owen Lars            120  37.9     1.78
##  7 Beru Whitesun Lars    75  27.5     1.65
##  8 R5-D4                 32  34.0     0.97
##  9 Biggs Darklighter     84  25.1     1.83
## 10 Obi-Wan Kenobi        77  23.2     1.82
## # i 77 more rows
```

And here is `transmute()` in action, which is useful when you only care about the new variables you've created.

```r
# Calculate BMI but only keep the character name and the result
starwars %>%
  transmute(
    character = name,
    bmi = mass / ((height / 100) ^ 2)
  )
```

```
## # A tibble: 87 x 2
##    character            bmi
##    <chr>              <dbl>
##  1 Luke Skywalker      26.0
##  2 C-3PO               26.9
##   3 R2-D2                 34.7
##   4 Darth Vader           33.3
##   5 Leia Organa           21.8
##   6 Owen Lars             37.9
##   7 Beru Whitesun Lars    27.5
##   8 R5-D4                 34.0
##   9 Biggs Darklighter     25.1
##  10 Obi-Wan Kenobi        23.2
## # i 77 more rows
```

By combining these core verbs, you can construct powerful and readable data wrangling pipelines to prepare any dataset for the next stages of analysis.

Aggregating Data: The Split-Apply-Combine Strategy

So far, we have learned how to transform our data on a row-by-row or column-by-column basis. We can select, filter, and create new variables. However, some of the most powerful insights in data analysis come from **aggregating** data—that is, collapsing many values down into a single summary statistic, like a count, an average, or a maximum.

To do this, we use a powerful mental model called the **Split-Apply-Combine** strategy:

1. **Split:** Break a large dataset into smaller groups based on the values of a categorical variable.
2. **Apply:** Apply a summary function (e.g., `mean()`, `sum()`) to each group independently.
3. **Combine:** Combine the results from each group into a new, smaller, and cleaner summary data frame.

This strategy is remarkably intuitive to implement in `dplyr` using just two key verbs: `group_by()` and `summarise()`.

Splitting Data into Groups with `group_by()`

The `group_by()` verb is the first step in our strategy. Its job is to take a data frame and add metadata to it that marks it as "grouped" by one or more columns. Visually, the data doesn't change much, but this grouping information fundamentally changes how subsequent `dplyr` verbs operate.

Let's group the `starwars` data by the `species` column.

```r
# We continue to use the tidyverse library and the starwars dataset
library(tidyverse)

starwars %>%
  group_by(species)
```

```
## # A tibble: 87 x 14
## # Groups:   species [38]
##    name      height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Luke Sk~     172    77 blond      fair       blue            19   male  mascu~
##  2 C-3PO        167    75 <NA>       gold       yellow         112   none  mascu~
##  3 R2-D2         96    32 <NA>       white, bl~ red             33   none  mascu~
##  4 Darth V~     202   136 none       white      yellow          41.9 male  mascu~
##  5 Leia Or~     150    49 brown      light      brown           19   fema~ femin~
##  6 Owen La~     178   120 brown, gr~ light      blue            52   male  mascu~
##  7 Beru Wh~     165    75 brown      light      blue            47   fema~ femin~
##  8 R5-D4         97    32 <NA>       white, red red             NA   none  mascu~
##  9 Biggs D~     183    84 black      light      brown           24   male  mascu~
## 10 Obi-Wan~     182    77 auburn, w~ fair       blue-gray       57   male  mascu~
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Notice the output. The data looks the same, but the header now says `Groups:   species [38]`. This tells us that any `dplyr` verb we apply next will be performed independently on each of the 38 species groups. You can also group by multiple variables, for example, `group_by(homeworld, species)`.

Applying Summaries with `summarise()`

Once the data is grouped, we can use `summarise()` (you may also see its alias, `summarize()`) to apply our summary functions. The `summarise()` verb collapses all the rows within each group into a single summary row.

Let's start with an ungrouped summary to see how it works on the entire dataset.

```r
starwars %>%
  summarise(
    avg_height = mean(height, na.rm = TRUE),
    min_mass = min(mass, na.rm = TRUE),
    character_count = n()
  )
```

```
## # A tibble: 1 x 3
##   avg_height min_mass character_count
##        <dbl>    <dbl>           <int>
## 1       175.       15              87
```

Here, we created a new data frame with a single row containing three summary values. * We used `mean()` and `min()` to calculate statistics. The argument `na.rm = TRUE` is crucial; it tells R to ignore missing values (`NA`) when calculating the summary. Without it, any calculation involving an `NA` will result in `NA`. * We also used `n()`, a special `dplyr` function that counts the number of rows in the current group. Here, since there were no groups, it counted all the rows in the dataset.

Now for the magic: let's combine `group_by()` and `summarise()`.

```r
# For each species, calculate the average mass and the number of characters.
species_summary <- starwars %>%
  group_by(species) %>%
  summarise(
    avg_mass = mean(mass, na.rm = TRUE),
    character_count = n()
  ) %>%
  arrange(desc(character_count))

print(species_summary)
```

```
## # A tibble: 38 x 3
##    species  avg_mass character_count
##    <chr>       <dbl>           <int>
##  1 Human        81.3              35
##  2 Droid        69.8               6
##  3 <NA>         81                 4
##  4 Gungan       74                 3
##  5 Kaminoan     88                 2
##  6 Mirialan     53.1               2
##  7 Twi'lek      55                 2
##  8 Wookiee     124                 2
##  9 Zabrak       80                 2
## 10 Aleena       15                 1
## # i 28 more rows
```

This is the Split-Apply-Combine strategy in action. `dplyr` automatically: 1. **Split** the `starwars` data into groups by `species`. 2. **Applied** the `mean()` and `n()` functions to each group. 3. **Combined** the results into the new `species_summary` tibble.

We even piped the result into `arrange()` to immediately see the most numerous species!

The Importance of `ungroup()`

After you perform a grouped operation, the data frame often remains grouped. You can see this in the output of our `species_summary` if we just ran the `summarise` step—it would show `Groups: [species]`.

This can be problematic if you intend to perform further manipulations. For example, a subsequent `mutate()` would try to operate on a per-group basis, which might not be what you want.

It is considered good practice to explicitly remove the grouping information with `ungroup()` once you are finished with your summary calculations.

```
# This pipeline is now "safe" because the final output is a regular, ungrouped tibble.
species_summary_safe <- starwars %>%
  group_by(species) %>%
  summarise(
    character_count = n(),
    avg_height = mean(height, na.rm = TRUE)
  ) %>%
  ungroup()

# Because we ungrouped, this mutate now works on the whole summary table,
# not on a per-species basis.
species_summary_safe %>%
  mutate(rank = min_rank(desc(character_count))) %>%
  arrange(rank)
```

```
## # A tibble: 38 x 4
##    species  character_count avg_height  rank
##    <chr>              <int>      <dbl> <int>
##  1 Human                 35       178      1
##  2 Droid                  6       131.     2
##  3 <NA>                   4       175      3
##  4 Gungan                 3       209.     4
##  5 Kaminoan               2       221      5
##  6 Mirialan               2       168      5
##  7 Twi'lek                2       179      5
##  8 Wookiee                2       231      5
##  9 Zabrak                 2       173      5
## 10 Aleena                 1        79     10
## # i 28 more rows
```

By adding `ungroup()` to the end of your aggregation pipelines, you ensure that your summary tables are clean, simple, and ready for the next steps of analysis or plotting without any surprising group-wise behavior.

Reshaping Your Data with `tidyr`

We've now mastered the `dplyr` verbs for transforming the *values* within our data—selecting, filtering, arranging, and summarizing. But sometimes, the problem isn't with the values themselves, but with the *layout* of the data. This is where the `tidyr` package comes in. `tidyr` is designed to change the shape or layout of your data, helping you create the "tidy" data structure that is the ultimate goal of the wrangling process.

As a reminder, the three rules of tidy data are: 1. Every column is a variable. 2. Every row is an observation. 3. Every cell is a single value.

The two most important functions in `tidyr` for achieving this are `pivot_longer()` and `pivot_wider()`.

When Data Isn't Tidy: Wide vs. Long Formats

Data often comes in one of two formats: wide or long.

- **Wide Format:** This format is often found in spreadsheets and reports. It's easy for humans to scan across, but it violates the principles of tidy data. In wide data, you'll often find columns that are not variables, but rather *values* of a variable (e.g., columns named `2023`, `2024`, `2025`). This format is difficult to use in R for plotting and modeling.
- **Long Format:** This is the tidy format. Every row is a single observation. This format is machine-readable and is what the Tidyverse is designed to work with. It's much easier to group, summarize, and plot data in a long format.

For example, imagine we have data on visitor numbers for historical sites in Williamsburg. The wide (messy) format might look like this:

site	2023	2024	2025
Governor's Palace	305k	312k	315k
Capitol Building	280k	295k	302k

The tidy (long) version of the same data would be:

site	year	visits
Governor's Palace	2023	305k
Governor's Palace	2024	312k
...	...	...

From Wide to Long: `pivot_longer()`

`pivot_longer()` is the workhorse function you will use most often. It takes multiple columns and "lengthens" them by collapsing them into two new columns: one for the names of the original columns and one for their values.

Let's create the wide-format tibble from our example and then tidy it.

```r
library(tidyverse)

# Create a "wide" tibble
cw_visits_wide <- tribble(
  ~site, ~"2023", ~"2024", ~"2025",
  "Governor's Palace", 305000, 312000, 315000,
  "Capitol Building",    280000, 295000, 302000,
  "Art Museums",          150000, 165000, 180000
)

print(cw_visits_wide)
```

```
## # A tibble: 3 x 4
##   site              `2023` `2024` `2025`
##   <chr>              <dbl>  <dbl>  <dbl>
## 1 Governor's Palace 305000 312000 315000
## 2 Capitol Building  280000 295000 302000
## 3 Art Museums       150000 165000 180000
```

Now, let's use `pivot_longer()` to tidy it. We need to tell the function: * `cols`: Which columns to pivot. Here, it's everything *except* the `site` column. * `names_to`: The name of the new column that will contain the former column headers (in our case, `"year"`). * `values_to`: The name of the new column that will contain the values that were in the pivoted columns (in our case, `"visits"`).

```
cw_visits_long <- cw_visits_wide %>%
  pivot_longer(
    cols = -site,
    names_to = "year",
    values_to = "visits"
  )

print(cw_visits_long)
```

```
## # A tibble: 9 x 3
##   site              year  visits
##   <chr>             <chr>  <dbl>
## 1 Governor's Palace 2023  305000
## 2 Governor's Palace 2024  312000
## 3 Governor's Palace 2025  315000
## 4 Capitol Building  2023  280000
## 5 Capitol Building  2024  295000
## 6 Capitol Building  2025  302000
## 7 Art Museums       2023  150000
## 8 Art Museums       2024  165000
## 9 Art Museums       2025  180000
```

Notice that the new `year` column is a character string. We can easily fix this by telling `pivot_longer` to parse the new column as a number.

```
# A more advanced pivot with type conversion
cw_visits_long_clean <- cw_visits_wide %>%
  pivot_longer(
    cols = -site,
    names_to = "year",
    values_to = "visits",
    names_transform = as.integer # Convert the new 'year' column to an integer
  )

glimpse(cw_visits_long_clean)
```

```
## Rows: 9
## Columns: 3
## $ site   <chr> "Governor's Palace", "Governor's Palace", "Governor's Palace", ~
## $ year   <int> 2023, 2024, 2025, 2023, 2024, 2025, 2023, 2024, 2025
## $ visits <dbl> 305000, 312000, 315000, 280000, 295000, 302000, 150000, 165000,~
```

This tidy format is now much easier to work with. For example, plotting visits over time is now trivial.

From Long to Wide: `pivot_wider()`

Sometimes, you need to do the reverse operation. `pivot_wider()` is the inverse of `pivot_longer()`. While you'll use it less often for analysis, it can be very useful for creating summary tables for reports.

Let's use the `cw_visits_long` tibble we just created to turn it back into its original wide format. We need to tell the function: * `names_from`: The column that contains the values that will become the new column *headers* (in our case, `year`). * `values_from`: The column that contains the values that will fill the cells of the new columns (in our case, `visits`).

```r
cw_visits_long %>%
  pivot_wider(
    names_from = year,
    values_from = visits
  )
```

```
## # A tibble: 3 x 4
##   site              `2023` `2024` `2025`
##   <chr>              <dbl>  <dbl>  <dbl>
## 1 Governor's Palace 305000 312000 315000
## 2 Capitol Building  280000 295000 302000
## 3 Art Museums       150000 165000 180000
```

As you can see, we have perfectly reconstructed the original wide-format table. Understanding how to move data between these two formats is a critical skill for effective data wrangling.

Advanced Reshaping: Handling Complex Layouts

While simple pivots are common, real-world data is often more complex. You might have multiple identifier columns that need to stay fixed, and the column headers themselves might contain several pieces of information that you need to separate.

For this example, let's imagine we have water quality data from two monitoring sites on local rivers, "Jamestown" on the James River and "Yorktown" on the York River. For each site and date, both the mean and standard deviation (`sd`) were recorded for two different pollutants.

This "wide" format is very common for reporting experimental results:

```r
library(tidyverse)

# A more complex "wide" tibble
water_quality_wide <- tribble(
  ~site,       ~collection_date, ~nitrogen_mean, ~nitrogen_sd, ~phosphorus_mean, ~phosphorus_sd,
  "Jamestown", "2025-10-01",     1.25,           0.15,         0.08,             0.02,
  "Yorktown",  "2025-10-01",     0.95,           0.11,         0.06,             0.01,
  "Jamestown", "2025-10-02",     1.31,           0.12,         0.09,             0.02,
  "Yorktown",  "2025-10-02",     0.99,           0.09,         0.07,             0.01
)

print(water_quality_wide)
```

```
## # A tibble: 4 x 6
##   site   collection_date nitrogen_mean nitrogen_sd phosphorus_mean phosphorus_sd
##   <chr>  <chr>                   <dbl>       <dbl>            <dbl>         <dbl>
## 1 James~ 2025-10-01               1.25        0.15             0.08          0.02
## 2 Yorkt~ 2025-10-01               0.95        0.11             0.06          0.01
## 3 James~ 2025-10-02               1.31        0.12             0.09          0.02
## 4 Yorkt~ 2025-10-02               0.99        0.09             0.07          0.01
```

Our goal is to tidy this data. A tidy version would have columns for `site`, `collection_date`, `pollutant`, `mean`, and `sd`.

More Complex `pivot_longer()`

The problem here is that the column names like `nitrogen_mean` contain two pieces of information: the pollutant ("nitrogen") and the statistic measured ("mean"). `pivot_longer()` can handle this elegantly using a special name, `.value`.

By including `.value` in the `names_to` argument, you tell `tidyr` that part of the column name should be treated as the *name* for a new column that will contain the cell values.

- `cols`: We select all columns that contain an underscore `_`.
- `names_to`: We provide a vector, `c("pollutant", ".value")`. This tells `pivot_longer` to split the old column names into two parts. The part before the separator will go into a new column called `pollutant`, and the part after the separator (which corresponds to `.value`) will become the names of the new *value* columns (in this case, `mean` and `sd`).
- `names_sep`: We specify the character to split the names by, which is `_`.

```r
water_quality_long <- water_quality_wide %>%
  pivot_longer(
    cols = contains(c("nitrogen_","phosphorus_")),
    names_to = c("pollutant", ".value"),
    names_sep = "_"
  )

# Let's print the result to confirm the column names
print(water_quality_long)
```

```
## # A tibble: 8 x 5
##   site      collection_date pollutant   mean    sd
##   <chr>     <chr>           <chr>      <dbl> <dbl>
## 1 Jamestown 2025-10-01      nitrogen    1.25  0.15
## 2 Jamestown 2025-10-01      phosphorus  0.08  0.02
## 3 Yorktown  2025-10-01      nitrogen    0.95  0.11
## 4 Yorktown  2025-10-01      phosphorus  0.06  0.01
## 5 Jamestown 2025-10-02      nitrogen    1.31  0.12
## 6 Jamestown 2025-10-02      phosphorus  0.09  0.02
## 7 Yorktown  2025-10-02      nitrogen    0.99  0.09
## 8 Yorktown  2025-10-02      phosphorus  0.07  0.01
```

This single, powerful command has perfectly reshaped our complex wide data into a tidy format. We have our two identifier variables (`site` and `collection_date`) preserved, and we've successfully created new columns for the pollutant and its corresponding mean and standard deviation.

More Complex `pivot_wider()`

Now, let's perform the inverse operation to demonstrate a more complex `pivot_wider()` call. Our goal is to take the tidy `water_quality_long` tibble and reshape it back to its original complex wide format.

Here, we have multiple value columns (`mean` and `sd`) that we want to spread out. `pivot_wider()` can handle this by accepting a vector for the `values_from` argument. It will automatically combine the names from `names_from` and `values_from` to create the new column headers.

- `id_cols`: We can explicitly state which columns are identifiers and should not be pivoted.
- `names_from`: The column whose values will become *part* of the new column names (in our case, `pollutant`).
- `values_from`: The columns whose values will fill the cells of the new columns. We provide a vector: `c(mean, sd)`.

```r
water_quality_long %>%
  pivot_wider(
    # The id_cols must match the exact column names from the input tibble.
    id_cols = c(site, collection_date),
    names_from = pollutant,
    values_from = c(mean, sd)
  )
```

```
## # A tibble: 4 x 6
##   site   collection_date mean_nitrogen mean_phosphorus sd_nitrogen sd_phosphorus
##   <chr>  <chr>                   <dbl>           <dbl>       <dbl>         <dbl>
## 1 James~ 2025-10-01               1.25            0.08        0.15          0.02
## 2 Yorkt~ 2025-10-01               0.95            0.06        0.11          0.01
## 3 James~ 2025-10-02               1.31            0.09        0.12          0.02
## 4 Yorkt~ 2025-10-02               0.99            0.07        0.09          0.01
```

The result is identical to our original `water_quality_wide` tibble. The `pivot_wider()` function automatically created the combined column names (e.g., `mean_nitrogen`, `sd_nitrogen`) for us. These advanced features of `tidyr` provide the flexibility to handle nearly any data reshaping task you might encounter.

Case Study: From Messy Report to Final Plot

In this section, we will bring together everything we have learned. Data wrangling is not about applying functions in isolation; it's about building a sequential pipeline of operations that turns messy, raw data into a clean, tidy dataset ready for analysis and visualization.

Our Scenario: Imagine we are environmental analysts based in Williamsburg, Virginia, and our team is preparing to launch a new, local air quality monitoring project. Before we begin collecting our own data, our task is to develop and validate a data wrangling workflow in R. To do this, we will use a famous, publicly available historical dataset as a stand-in: the built-in `airquality` dataset, which contains records from New York City in 1973.

Our goal is to create a reproducible script that takes a "messy report" based on this historical data, tidies it completely, and produces a summary plot. This validated script can then be easily adapted for our real Williamsburg data once it becomes available.

Step 1: Assessing the "Messy Report"

First, let's load the Tidyverse. Then, to simulate the kind of messy, wide-format report we might receive from a monitoring station, we will programmatically reformat the built-in `airquality` data.

```r
library(tidyverse)

# --- This code simulates the messy report based on the NYC data ---
messy_report <- airquality %>%
  # We're interested in the summer months for our pilot study
  filter(Month %in% c(5, 6, 7)) %>%
  # Select a few relevant columns
  select(Day, Month, Wind, Temp) %>%
  # Pivot the data so each month's temperature is a different column
  pivot_wider(names_from = Month, values_from = Temp, names_prefix = "Temp_Month_")

# Now, let's look at the "report" we need to clean
print(messy_report)
```

```
## # A tibble: 89 x 5
##      Day  Wind Temp_Month_5 Temp_Month_6 Temp_Month_7
##    <int> <dbl>        <int>        <int>        <int>
##  1     1   7.4           67           NA           NA
##  2     2   8             72           NA           NA
##  3     3  12.6           74           NA           NA
##  4     4  11.5           62           NA           NA
##  5     5  14.3           56           NA           NA
##  6     6  14.9           66           NA           NA
##  7     7   8.6           65           NA           NA
##  8     8  13.8           59           NA           NA
##  9     9  20.1           61           NA           NA
## 10    10   8.6           69           NA           NA
## # i 79 more rows
```

Let's use `glimpse()` to diagnose the problems:

```r
glimpse(messy_report)
```

```
## Rows: 89
## Columns: 5
## $ Day          <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17~
## $ Wind         <dbl> 7.4, 8.0, 12.6, 11.5, 14.3, 14.9, 8.6, 13.8, 20.1, 8.6, 6~
## $ Temp_Month_5 <int> 67, 72, 74, 62, 56, 66, 65, 59, 61, 69, 74, 69, 66, 68, 5~
## $ Temp_Month_6 <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
## $ Temp_Month_7 <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
```

Diagnosis: This report format is not suitable for analysis. We can identify several key problems: 1. **It's "Wide" Data:** The columns `Temp_Month_5`, `Temp_Month_6`, and `Temp_Month_7` are not variables; they are *values* of a "month" variable. 2. **Redundant Rows and `NA`s:** The wide format creates rows with many `NA` (missing) values, as a given day only exists in one month. 3. **Clunky Column Names:** The column names are not ideal for coding.

Step 2: Reshaping with `pivot_longer()`

Our first priority is to fix the fundamental structural problem by converting the data from a wide to a long format. We will use `pivot_longer()` to collapse the temperature columns into a "month" column and a "temperature" column.

```r
airquality_long <- messy_report %>%
  pivot_longer(
    cols = starts_with("Temp_Month_"), # Select all columns that start with "Temp_Month_"
    names_to = "Month",               # The new column for the old column names
    values_to = "Temperature"         # The new column for the cell values
  )
```

```
# Let's see the result
head(airquality_long, 10)
```

```
## # A tibble: 10 x 4
##      Day  Wind Month        Temperature
##    <int> <dbl> <chr>              <int>
##  1     1   7.4 Temp_Month_5          67
##  2     1   7.4 Temp_Month_6          NA
##  3     1   7.4 Temp_Month_7          NA
##  4     2   8   Temp_Month_5          72
##  5     2   8   Temp_Month_6          NA
##  6     2   8   Temp_Month_7          NA
##  7     3  12.6 Temp_Month_5          74
##  8     3  12.6 Temp_Month_6          NA
##  9     3  12.6 Temp_Month_7          NA
## 10     4  11.5 Temp_Month_5          62
```

This is a huge improvement! Our data is now in a much longer, tidier format, which will be easier to clean and analyze.

Step 3: Cleaning and Transforming with `dplyr`

Now that the data has the correct shape, we can build a `dplyr` pipeline to finish the cleaning process.

```
airquality_tidy <- airquality_long %>%
  # 1. Remove rows where the Temperature is NA. These were artifacts of the pivot.
  filter(!is.na(Temperature)) %>%

  # 2. Clean the 'Month' column by removing the prefix
  mutate(Month = str_replace(Month, "Temp_Month_", "")) %>%

  # 3. Rename columns to be cleaner and more consistent (snake_case)
  rename(
    day = Day,
    wind_speed = Wind,
    temperature_F = Temperature,
    month = Month
  ) %>%

  # 4. Convert month from a character to a number for easier sorting later
  mutate(month = as.integer(month)) %>%

  # 5. Arrange the data logically
  arrange(month, day)

# Inspect our final, tidy tibble
glimpse(airquality_tidy)
```

```
## Rows: 92
## Columns: 4
## $ day           <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~
## $ wind_speed    <dbl> 7.4, 8.0, 12.6, 11.5, 14.3, 14.9, 8.6, 13.8, 20.1, 8.6, ~
## $ month         <int> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,~
## $ temperature_F <int> 67, 72, 74, 62, 56, 66, 65, 59, 61, 69, 74, 69, 66, 68, ~
```

This is our final, analysis-ready dataset. Every column is a variable, every row is a unique observation, and every cell has a value.

Step 4: Summarizing the Data

With our clean data, the final analysis step is easy. We want to find the average wind speed for each month to test our summary logic.

```r
monthly_summary <- airquality_tidy %>%
  group_by(month) %>%
  summarise(
    avg_wind_speed = mean(wind_speed)
  ) %>%
  ungroup()

print(monthly_summary)
```

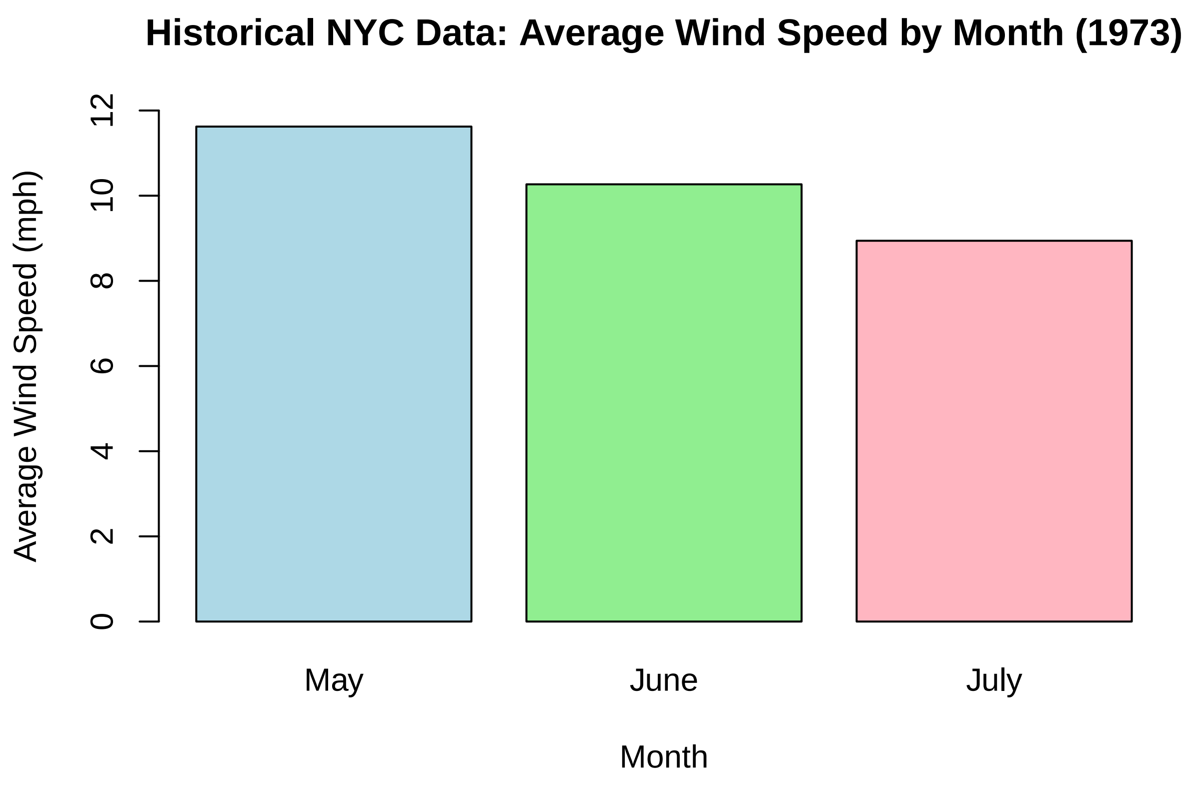
```
## # A tibble: 3 x 2
##   month avg_wind_speed
##   <int>          <dbl>
## 1     5          11.6
## 2     6          10.3
## 3     7           8.94
```

This summary tibble contains the exact information we need to create our final plot.

Step 5: The Payoff – The Final Plot

Now we will visualize our summary. The title and labels will be precise, clearly stating the source and content of the data, which is a critical part of any good analysis.

```r
# We use the barplot() function from base R
barplot(
  height = monthly_summary$avg_wind_speed,
  names.arg = c("May", "June", "July"),
  main = "Historical NYC Data: Average Wind Speed by Month (1973)",
  xlab = "Month",
  ylab = "Average Wind Speed (mph)",
  col = c("lightblue", "lightgreen", "lightpink"),
  ylim = c(0, 12)
)
```



Conclusion

We have successfully completed our goal. We started with a messy, report-style dataset and built a reproducible R script that transformed it into a clean, tidy format. From this tidy data, we calculated a summary statistic and created a clear visualization.

The key takeaway is that we now have a validated workflow. When we receive our real air quality data from Williamsburg, we can confidently adapt this script to quickly produce similar analyses. This is the power of a well-designed data wrangling pipeline.

A Quick Tour of Base R Plots

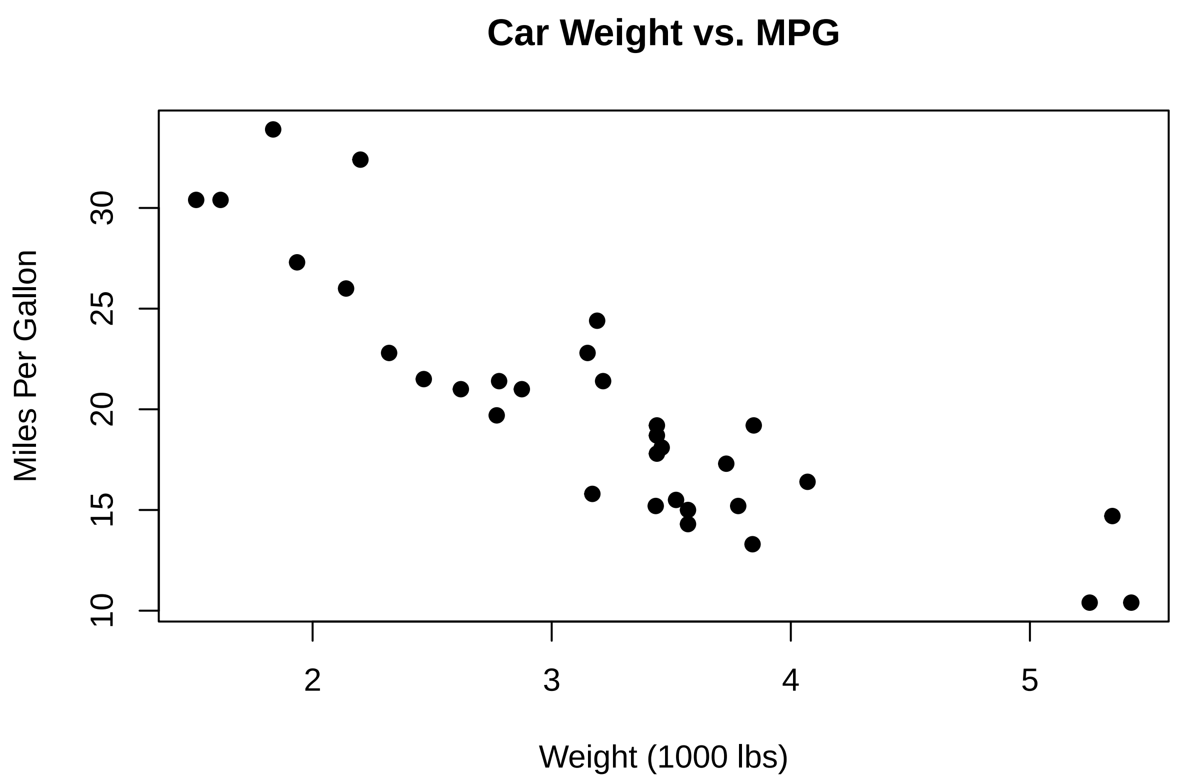
While ggplot2 is the modern standard, R's built-in plotting functions are fast and great for quick exploratory work without loading any additional libraries.

Base R Plot: Scatterplot (plot)

Purpose: To visualize the relationship between two continuous variables.

Example: Plot car weight vs. miles per gallon from the built-in 'mtcars' dataset

```r
plot(mtcars$wt, mtcars$mpg,
     main = "Car Weight vs. MPG",
     xlab = "Weight (1000 lbs)",
     ylab = "Miles Per Gallon",
     pch = 19) # pch changes the point shape
```

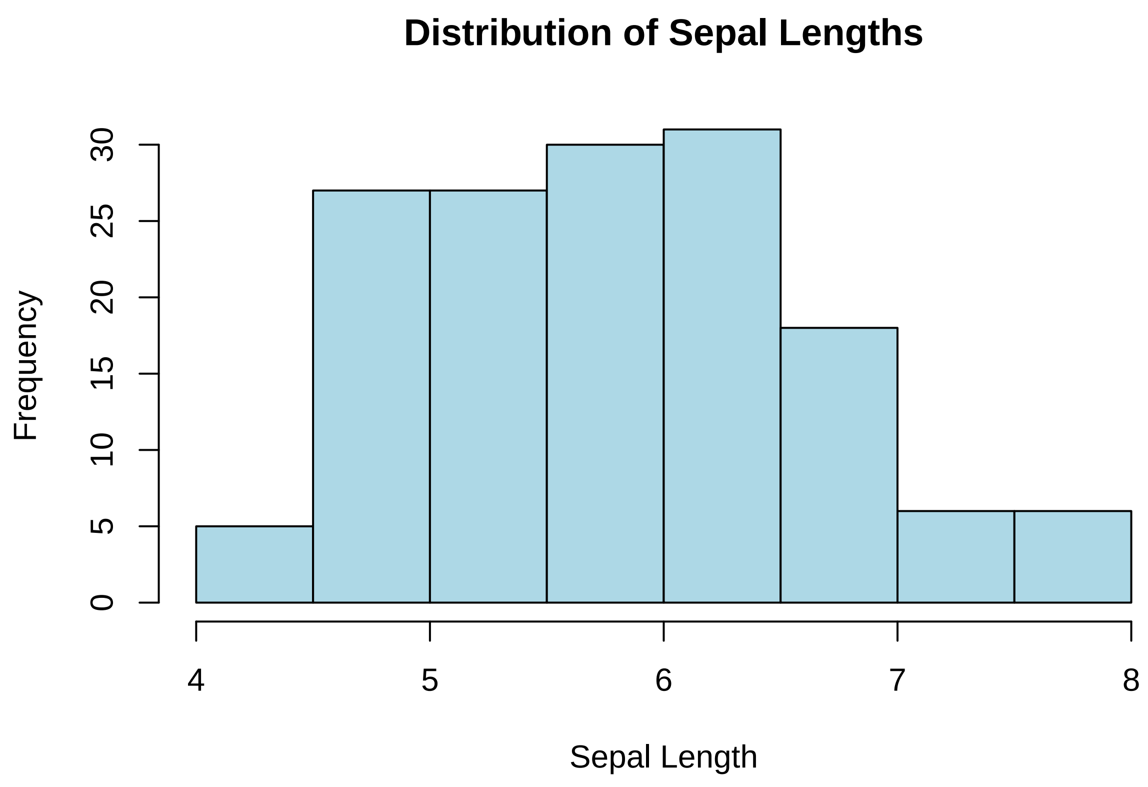


Base R Plot: Histogram (hist)

Purpose: To understand the distribution of a single continuous variable.

Example: Plot the distribution of sepal lengths from the built-in 'iris' dataset

```r
hist(iris$Sepal.Length,
     main = "Distribution of Sepal Lengths",
     xlab = "Sepal Length",
     col = "lightblue") # col sets the color
```

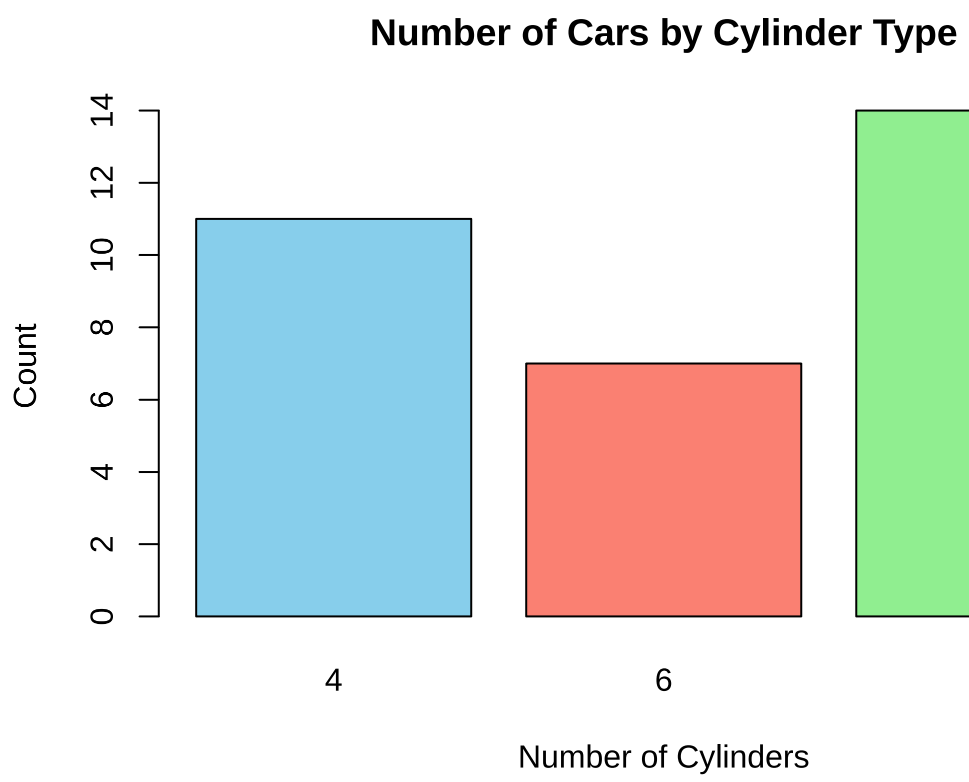


Base R Plot: Bar Plot (barplot)

Purpose: To compare the size of different categories. Note: barplot usually needs pre-summarized data (like averages across conditions).

```r
# First, create a summary table of counts for each cylinder type
cylinder_counts <- table(mtcars$cyl)

# Now, create the bar plot of the counts
barplot(cylinder_counts,
        main = "Number of Cars by Cylinder Type",
        xlab = "Number of Cylinders",
        ylab = "Count",
        col = c("skyblue", "salmon", "lightgreen"))
```

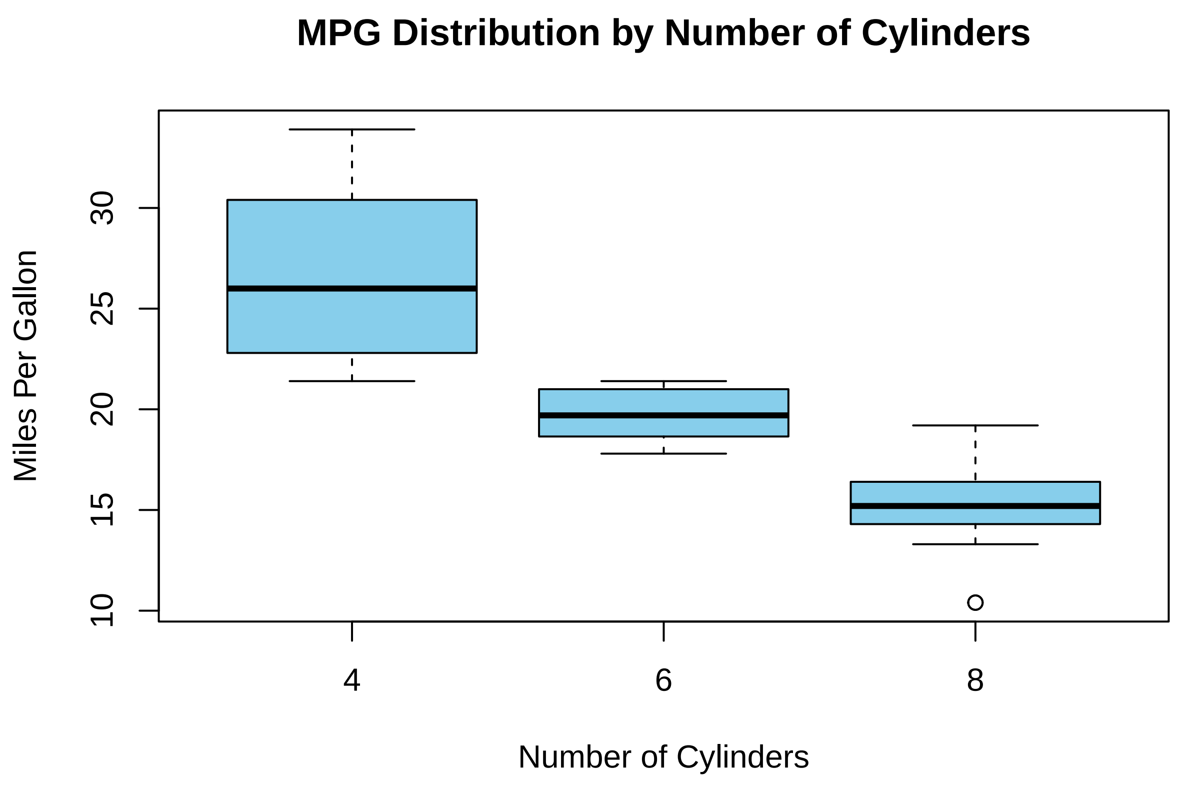


Base R Plot: Box Plot (boxplot)

Purpose: To compare the distribution of a continuous variable across several groups.

Example: Compare MPG distribution for each cylinder type using formula notation

```r
boxplot(mpg ~ cyl,
        data = mtcars,
        main = "MPG Distribution by Number of Cylinders",
        xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon",
        col = "skyblue")
```



Exercises

The following exercises will test your ability to apply the data wrangling skills you've learned in this chapter. Each problem uses datasets that are built into the Tidyverse, so you will need to have the `tidyverse` library loaded. The goal is not just to get the right answer, but to write a clean, readable pipeline of commands for each task. **Don't forget to use text blocks and/or comments to document your thinking and explain what each line of your code is doing.**

```r
library(tidyverse)
```

Problem 1: Analyzing US Arrests

Dataset: `USArrests` (built into R)

Scenario: A law student at William & Mary is researching historical crime data from 1973 and wants to identify states with high urban populations and their corresponding violent crime rates.

Task: Write a single pipeline of `dplyr` commands that: 1. Starts with the `USArrests` dataset. Note that the state names are currently row names. You can turn them into a proper column by running `USArrests %>% rownames_to_column(var = "state")` first. 2. Filters the data to include only states with an `UrbanPop` rate of 80% or higher. 3. Selects only the `state`, `Murder`, and `Rape` columns. 4. Arranges the final result by the `Murder` rate in descending order. 5. Write a sentence or two to describe the result.

```r
# Your code here
```

Problem 2: Mammal Sleep Habits

Dataset: `msleep` (from the `ggplot2` package)

Scenario: A researcher at the Virginia Living Museum wants to know if there's a relationship between an animal's diet and its sleep patterns, specifically the proportion of time spent in REM sleep.

Task: Write a `dplyr` pipeline that: 1. Starts with the `msleep` dataset. 2. Removes rows where the `vore` (diet) column is `NA`, as we can't use them for this analysis. (Hint: use `filter(!is.na(vore))`). 3. Creates a new column called `rem_proportion` which is the ratio of `sleep_rem` to `sleep_total`. 4. Groups the data by `vore`. 5. Calculates the average `rem_proportion` for each diet group. 6. Arranges the final tibble to show the diets with the highest average REM proportion first. 7. Create a bar chart or boxplot to visualize the REM proportion for each diet type. 8. Write a sentence or two to describe the results.

```
# Your code here
```

Problem 3: Tidying Global Health Data

Dataset: `who` (from the `tidyr` package)

Scenario: An intern at the Virginia Department of Health has been given a dataset from the World Health Organization about tuberculosis cases. The data is in an extremely "wide" and messy format and needs to be tidied before any analysis can be done.

Task: This is a classic reshaping challenge. The columns from `new_sp_m014` through `newrel_f65` contain values that should be in their own columns. 1. Start with the `who` dataset. 2. Use `pivot_longer()` to gather all the columns from `new_sp_m014` to `newrel_f65`. 3. The names of these columns should go into a new column called `diagnosis_code`. 4. The values should go into a new column called `count`. 5. After pivoting, many rows will have a `count` of `NA`. Filter these rows out. 6. Display the first few rows of your final tidy tibble.

(Hint: This is a complex pivot! You can select the columns with `cols = new_sp_m014:newrel_f65`)

```
# Your code here
```

Problem 4: Finding the Windiest Year for Storms

Dataset: `storms` (from the `dplyr` package)

Scenario: Meteorologists in Hampton Roads are analyzing the historical Atlantic storm dataset to identify the year with the highest average maximum wind speeds, which could indicate a particularly active storm season.

Task: Write a `dplyr` pipeline to determine which year had the highest average wind speeds for its storms. Your final output should be a tibble with only one row (the winning year). 1. Start with the `storms` dataset. 2. Group the data by `year`. 3. For each year, calculate the average maximum wind speed (`wind`). 4. Arrange the results to find the year with the highest average. 5. Create a scatter plot of year as a function of wind speed. 6. Return only the top row. (Hint: `head(1)` or `slice_max()` are good ways to do this). 7. Write a sentence or two to describe your results.

```
# Your code here
```

Problem 5: From Tidy back to Wide

Dataset: Your tidied data from Problem 3.

Scenario: After you tidied the `who` dataset, your manager asks for a specific summary table: a report showing the total number of cases for Afghanistan, Brazil, and China for the years 1999 through 2001. They want the years as columns for a quick comparison.

Task: This requires you to filter, summarize, and then reshape your tidy data back into a wide format. 1. Start with the tidied data you created in Problem 3. 2. `filter()` for only the countries of interest (`Afghanistan`, `Brazil`, `China`) and the years of interest (`1999`, `2000`, `2001`). 3. Calculate the total number of cases (`count`) for each country and year combination. (Hint: `group_by()` then `summarise()`). 4. Use `pivot_wider()` to reshape this summary table. The new column headers should come from the `year` column, and the values should come from your new `total_cases` column. 5. Write a sentence or two to describe your results.

```
# First, re-create your tidy 'who' data from problem 3 here
# tidy_who <- who %>% ...

# Then, write the pipeline for this problem
# tidy_who %>% ...
```

Problem 6: Ranking Species by Average BMI (Star Wars)

Dataset: starwars (from dplyr)

Scenario: A sci-fi physiology class wants to compare species by average Body Mass Index (BMI), computed from height and mass.

Task:

Write a pipeline that:

1. Starts from starwars, and keeps name, species, height, mass.
2. Creates height_m = height / 100 and bmi = mass / (height_m^2).
3. Groups by species, computes avg_bmi and n (characters per species).
4. Ungroup and return the top 5 species by avg_bmi, breaking ties by n (larger first).
5. Create a visualization (plot) of your results.
6. Write a sentence or two to describe your results.

(Tip: This reinforces mutate(), summarise(), arrange(), and the importance of ungroup().)

```
# Your code here
```

Problem 7: Advanced Reshaping with `.value` (Water Quality)

Dataset: Recreate the `water_quality_wide` tibble from the chapter (or copy it in your chunk).

Scenario: You need a tidy table that separates pollutant and statistic, then a compact report.

Task:
1. Start from `water_quality_wide` (columns like `nitrogen_mean`, `nitrogen_sd`, etc.).

2. Use `pivot_longer()` with `names_to = c("pollutant", ".value")` and `names_sep = "_"` to get columns `site, collection_date, pollutant, mean, sd`.
3. Compute **overall means by pollutant** (average of the `mean` column across sites/dates).
4. Return a **one-row per pollutant** report and then `pivot_wider()` so pollutants are columns and the overall means are values.
5. Write a sentence or two describing your results.

(Tip: This exercises the advanced `.value` pattern and a round-trip pivot.)

```
# Your code here
```

Problem 8: Create Your Own Pipeline

Dataset: Import your own dataset, or choose from one of the built-in datasets

Scenario: Examine the variables and formulate a research question about the data

Task: Create a data wrangling pipeline appropriate to your research question, then summarize and plot the results.

```
# Your code here
```