# Plant Breeding Data Analysis with R - Baku

Your Name / Breeding Institute Baku (ICARDA Collaboration)

2025-05-06

# Table of contents

# 1 Welcome to the Plant Breeding Data Analysis Course

**Target Audience:** Breeders in Baku, Azerbaijan with minimal prior data analysis or programming experience. Collaboration with ICARDA.

**Goal:** To provide a practical and understandable introduction to analyzing common breeding data types using the R programming language.

This course covers fundamental concepts in genetics and statistics relevant to breeding programs, alongside hands-on R coding sessions. We aim to build your confidence in handling your own data and interpreting results.

Please use the navigation menu (Table of Contents) to move through the course modules.

Let's begin!

# Part I

# Introduction and Setup

# 2 Welcome and Course Overview

## 2.1 Hello Baku Breeders!

Welcome to this introductory course on data analysis for plant breeding, a collaboration with ICARDA. We are excited to guide you through the essential tools and concepts needed to make sense of your valuable breeding data using R.

## 2.2 Course Objectives

- Learn the fundamentals of the R programming language for data tasks.
- Understand basic concepts of genomic data.
- Perform basic data loading, cleaning, and quality control.
- Grasp key genetic concepts like allele frequency and relatedness (kinship).
- Understand the idea behind marker-trait association studies (GWAS).
- Get introduced to tools like GIGWA and basic AI applications.

## 2.3 Course Structure

This course is divided into several modules, starting with setup and R basics, moving through data handling and genetic concepts, and ending with analysis methods and tools. Each module includes explanations and practical R exercises.

No prior programming experience is required!

# 3 Setting Up Your Environment: R and RStudio

## 3.1 Why R and RStudio?

- **R:** A powerful, free programming language specifically designed for statistical computing and graphics. Widely used in academia and industry for data analysis, including genomics and breeding.
- **RStudio:** An excellent, free Integrated Development Environment (IDE) for R. It makes using R much easier with features like code highlighting, plot viewing, package management, and project organization.

## 3.2 Installation Steps

1. **Install R:** Go to CRAN (the Comprehensive R Archive Network) and download the latest version for your operating system (Windows, macOS, Linux). Follow the installation instructions.
2. **Install RStudio:** Go to the Posit website and download the free RStudio Desktop version for your operating system. Install it after installing R.
3. **Install Quarto:** Go to Quarto's website and download and install Quarto for your system. RStudio often bundles Quarto, but installing the latest version is good practice.
4. **(For PDF Output) Install LaTeX:** Open RStudio, go to the Console panel, and type the following commands one by one, pressing Enter after each: "'r # Run these lines in the R Console # install.packages("tinytex") # Run only once if you dont have it # tinytex::install_tinytex() # Run only once to install LaTeX distribution "' This might take a few minutes. If it fails, consult TinyTeX documentation or ask instructors.

## 3.3 Installing R Packages for the Course

We will use several add-on packages in R. You only need to install packages *once*. Use the R Console in RStudio.

```
# --- Run this code chunk in the R Console --- List of packages we will likely need:

packages_to_install <- c( "tidyverse", # For data manipulation (dplyr, tidyr) and plotting (g
                      "readxl", # For reading Excel files (.xlsx)
                      "writexl", # For writing Excel files (.xlsx)
                      "rrBLUP", # For Kinship calculation and basic GWAS
                      "qqman", # For creating Manhattan and QQ plots for GWAS
                      "vcfR"
                      )
```

## 3.4 Quick RStudio Tour

(We will cover this live, but key windows include: Console, Script Editor/Notebook, Environment/History, Files/Plots/Packages/Help/Viewer). Familiarize yourself with these panes.

# Part II

# R Programming Fundamentals

# 4 Module 1.1: Introduction to R - Your Breeding Data Analysis Tool

### 4.0.1 Introduction to R

R is a powerful language for data manipulation, visualization, and statistical analysis. Think of R as a versatile calculator for data.

- **What is R?** Think of R as a powerful, specialized calculator combined with a programming language. It's designed specifically for handling data, performing statistical analyses, and creating informative graphs.
- **Why R for Breeding?**
  - **Free & Open Source:** Anyone can use it without cost.
  - **Powerful for Data:** Excellent at handling the types of large datasets we generate in breeding (phenotypes, genotypes).
  - **Cutting-Edge Statistics:** Many new statistical methods (like those for genomic selection or GWAS) are first available as R packages.
  - **Great Graphics:** Create publication-quality plots to visualize your results.
  - **Large Community:** Lots of help available online and specialized packages for genetics and breeding (like `rrBLUP` which we might see later).
- **R vs. Excel:** Excel is great for data entry and simple summaries, but R is much better for complex analysis, automation, reproducible research, and handling very large datasets.

Try these examples in the RStudio Console:

```
# Basic arithmetic
2 + 5
10 - 3
4 * 8
100 / 4

# Order of operations (like standard math)
5 + 2 * 3   # Multiplication first
(5 + 2) * 3 # Parentheses first
```

```r
# Built-in mathematical functions
sqrt(16)    # Square root
log(10)     # Natural logarithm
log10(100)  # Base-10 logarithm
```

## 4.1 Variables: Storing Information

Variables are used to store information in R. You can think of them as containers for data. In R, you can create variables using the assignment operator `<-`. You can also use `=` for assignment, but `<-` is more common in R.

Use the `<-` operator to assign and manipulate variables:

```r
# Assign the value 5 to variable x
x <- 5

# Assign the result of 10 + 3 to variable y
y <- 10 + 3

# Print the value of x
x

# Use variables in calculations
z <- x + y
# Print the value of z
z

# Assign the name of a variety to a variable
best_variety <- "ICARDA_Gold" # Text needs quotes ""

print(average_yield)
```

# 5 Module 1.2: R Data Types and Structures - The Building Blocks

## 5.1 Introduction: Types and Structures

Think of data like building blocks:

- **Data Types:** The *kind* of block (e.g., numeric brick, text brick, true/false switch).
- **Data Structures:** How you *organize* those blocks (e.g., a single row of bricks, a flat grid, a complex box holding different things).

Understanding these is fundamental to working with data in R.

## 5.2 Basic Data Types

R needs to know what *kind* of information it's dealing with.

1. **Numeric:** Represents numbers. Can be integers (whole numbers) or doubles (with decimals). Used for measurements like yield, height, counts.

```
yield <- 75.5      # Double (decimal)
num_plots <- 120   # Integer (whole number)
class(yield)       # Check the type
class(num_plots)   # Often stored as 'numeric' (double) by default
```

2. **Character:** Represents text (strings). Always enclose text in double (") or single (') quotes. Used for IDs, names, descriptions.

```
variety_name <- "ICARDA_RustResist"
plot_id <- 'Plot_A101'
class(variety_name)
```

3. **Logical:** Represents TRUE or FALSE values. Often the result of comparisons. Crucial for filtering data.

```r
is_resistant <- TRUE
yield > 80 # This comparison results in a logical value
class(is_resistant)
```

4. **Factor:** Special type for categorical data (variables with distinct levels or groups). R stores them efficiently using underlying numbers but displays the text labels. Very important for statistical models and plotting.

```r
# Example: Different locations in a trial
locations <- c("Baku", "Ganja", "Baku", "Sheki", "Ganja")
location_factor <- factor(locations)

print(location_factor) # Shows levels
class(location_factor)
levels(location_factor) # See the unique categories
```

## 5.3 Key Data Structures

How R organizes collections of data:

1. **Vector:** The most basic structure! A sequence (ordered list) containing elements **of the same data type**. Created using `c()` (combine function).

```r
# Vector of plot yields (numeric)
plot_yields <- c(75.5, 81.2, 78.9, 85.0)
# Vector of variety names (character)
plot_varieties <- c("ICARDA_Gold", "Local_Check", "ICARDA_Gold", "ICARDA_RustResist")
# Vector of resistance status (logical)
plot_resistance <- c(TRUE, FALSE, TRUE, TRUE)

plot_yields[1]       # Access the first element (Indexing starts at 1!)
plot_yields[2:4]     # Access elements 2 through 4
length(plot_yields)  # Get the number of elements
```

*Important:* If you mix types in `c()`, R will force them into a single common type (usually

```r
mixed_vector <- c(10, "VarietyA", TRUE)
print(mixed_vector) # All become character strings!
class(mixed_vector)
```

2. **Matrix:** A two-dimensional grid (rows and columns) where all elements **must be of the same data type**. Useful for genotype data (0,1,2 are all numeric).

```
# Example: Small genotype matrix (Individuals x SNPs)
genotype_data <- matrix(c(0, 1, 2, 1, 1, 0), nrow = 2, ncol = 3, byrow = TRUE)
rownames(genotype_data) <- c("Line1", "Line2")
colnames(genotype_data) <- c("SNP1", "SNP2", "SNP3")
print(genotype_data)
class(genotype_data)
dim(genotype_data) # Get dimensions (rows, columns)
genotype_data[1, 2] # Access element row 1, column 2
```

3. **Data Frame: The most important data structure for breeders!** Like a spreadsheet or table in R.

   - It's a collection of vectors (columns) of equal length.
   - **Crucially, columns can be of different data types!** (e.g., character ID, numeric yield, factor location).
   - Rows represent observations (e.g., plots, plants, samples).
   - Columns represent variables (e.g., ID, traits, treatments).

```
# Create a simple breeding trial data frame
trial_data <- data.frame(
 PlotID = c("A101", "A102", "B101", "B102"),
 Variety = factor(c("ICARDA_Gold", "Local_Check", "ICARDA_RustResist", "ICARDA_Gold")),
 Yield_kg_plot = c(5.2, 4.5, 6.1, 5.5),
 Is_Resistant = c(TRUE, FALSE, TRUE, TRUE)
)

print(trial_data)
class(trial_data)
str(trial_data)       # Structure: Shows types of each column - VERY USEFUL!
head(trial_data)      # Show first few rows
summary(trial_data)   # Summary statistics for each column

# Access columns using $
trial_data$Yield_kg_plot
mean(trial_data$Yield_kg_plot) # Calculate mean of a column
```

   *(We will work extensively with data frames).*

4. **List:** A very flexible container that can hold *any* collection of R objects (vectors, matrices, data frames, even other lists), and they don't have to be the same type or length. Often used to return complex results from functions.

```r
analysis_results <- list(
  description = "Yield Trial - Baku 2023",
  raw_data = trial_data, # Include the data frame
  significant_snps = c("SNP101", "SNP504"), # A character vector
  model_parameters = list(threshold = 0.05, method = "MLM") # A nested list
)
print(analysis_results$description)
print(analysis_results$raw_data) # Access the data frame inside the list
```

# 6 Module 1.3: Basic Operations in R

Now that we know about data types and structures, let's see how to manipulate them.

## 6.1 Arithmetic Operations (Review)

Works on numbers and numeric vectors/matrices element-wise.

```
yield1 <- 5.2
yield2 <- 6.1

yield1 + yield2
yield1 * 10 # Scale up (e.g., plot yield to estimated per area)

# On vectors
plot_yields <- c(5.2, 4.5, 6.1, 5.5)
plot_yields + 0.5 # Add 0.5 to every plot's yield
plot_yields * 1000 / 10 # e.g. kg/plot to kg/ha if plot area = 10 sqm

# Other operators: ^ (power), %% (remainder)
2^3 # 2 to the power of 3
10 %% 3 # Remainder of 10 divided by 3 is 1
```

## 6.2 Logical Comparisons and Operators

Used to ask TRUE/FALSE questions about our data. Essential for filtering.

- **Comparison Operators:**
  - `>` : Greater than
  - `<` : Less than
  - `>=`: Greater than or equal to
  - `<=`: Less than or equal to
  - `==`: **Exactly equal to** (TWO equal signs! Very common mistake to use just one `=`)

  - `!=`: Not equal to

- **Logical Operators (Combine TRUE/FALSE):**
  - `&` : AND (both sides must be TRUE)
  - `|` : OR (at least one side must be TRUE)
  - `!` : NOT (reverses TRUE to FALSE, FALSE to TRUE)

```
yield <- 5.2
min_acceptable_yield <- 5.0
variety <- "ICARDA_Gold"

# Comparisons
yield > min_acceptable_yield # Is yield acceptable? TRUE
variety == "Local_Check"    # Is it the local check? FALSE
variety != "Local_Check"    # Is it NOT the local check? TRUE

# On vectors
plot_yields <- c(5.2, 4.5, 6.1, 5.5)
plot_yields > 5.0 # Which plots yielded above 5.0? [TRUE FALSE TRUE TRUE]

plot_varieties <- c("ICARDA_Gold", "Local_Check", "ICARDA_RustResist", "ICARDA_Gold")
plot_varieties == "ICARDA_Gold" # Which plots are ICARDA_Gold? [TRUE FALSE FALSE TRUE]

# Combining conditions
# Find plots where yield > 5.0 AND variety is ICARDA_Gold
(plot_yields > 5.0) & (plot_varieties == "ICARDA_Gold") # [TRUE FALSE FALSE TRUE]

# Find plots where yield > 6.0 OR variety is Local_Check
(plot_yields > 6.0) | (plot_varieties == "Local_Check") # [FALSE TRUE TRUE FALSE]
```

## 6.3 Vectorization: R's Superpower

Many R operations are **vectorized**, meaning they automatically apply to each element of a vector without needing you to write a loop. This makes R code concise and efficient. We've already seen this with arithmetic (`plot_yields + 0.5`) and comparisons (`plot_yields > 5.0`).

Functions like `mean()`, `sum()`, `min()`, `max()`, `sd()` (standard deviation), `length()` also work naturally on vectors:

```r
plot_yields <- c(5.2, 4.5, 6.1, 5.5)

mean(plot_yields)
sd(plot_yields)
sum(plot_yields > 5.0) # How many plots yielded > 5.0? (TRUE=1, FALSE=0)
length(plot_yields) # How many plots?
```

## 6.4 Working with Data Frames (Indexing and Filtering)

This is crucial for selecting specific data from your tables.

Let's use the `trial_data` data frame from the previous section:

```r
trial_data <- data.frame(
  PlotID = c("A101", "A102", "B101", "B102"),
  Variety = factor(c("ICARDA_Gold", "Local_Check", "ICARDA_RustResist", "ICARDA_Gold")),
  Yield_kg_plot = c(5.2, 4.5, 6.1, 5.5),
  Is_Resistant = c(TRUE, FALSE, TRUE, TRUE)
)
```

1. **Accessing Columns:** Use `$` (most common) or `[[ ]]`. `r trial_data$Variety trial_data[["Yield_kg_plot"]] mean(trial_data$Yield_kg_plot)`
2. **Accessing Rows/Columns/Cells using `[row, column]`:**

   ```r
   # Get the value in Row 2, Column 3
   trial_data[2, 3] # Should be 4.5

   # Get the entire Row 1 (returns a data frame)
   trial_data[1, ]

   # Get the entire Column 2 (Variety column, returns a vector/factor)
   trial_data[, 2]
   # Get Columns 1 and 3 (PlotID and Yield)
   trial_data[, c(1, 3)] # Use c() for multiple column indices
   trial_data[, c("PlotID", "Yield_kg_plot")] # Can also use column names
   ```

3. **Filtering Rows Based on Conditions (VERY IMPORTANT):** Use a logical condition inside the `row` part of the square brackets.

   ```r
   # Select rows where Yield_kg_plot is greater than 5.0
   high_yield_plots <- trial_data[trial_data$Yield_kg_plot > 5.0, ]
   print(high_yield_plots)
   ```

```r
# Select rows where Variety is "ICARDA_Gold"
icarda_gold_plots <- trial_data[trial_data$Variety == "ICARDA_Gold", ]
print(icarda_gold_plots)

# Select rows where Variety is "ICARDA_Gold" AND yield > 5.0
# (We generated the logical vector for this earlier)
condition <- (trial_data$Variety == "ICARDA_Gold") & (trial_data$Yield_kg_plot > 5.0)
print(condition) # Shows [TRUE FALSE FALSE TRUE]
selected_plots <- trial_data[condition, ]
print(selected_plots)

# Select rows where the variety is resistant
resistant_plots <- trial_data[trial_data$Is_Resistant == TRUE, ] # Or just trial_data[tr
print(resistant_plots)
```

**Exercise:** Select the data for the 'Local_Check' variety from the `trial_data` data frame. Calculate its yield.

# 7 Module 1.4: Reading and Writing Data

So far, we've created data inside R. But usually, your breeding data exists in external files, like Excel spreadsheets or CSV files. We need to get this data *into* R and save our results *out* of R.

## 7.1 Common Data File Formats

- **CSV (Comma Separated Values - `.csv`):** Plain text file where columns are separated by commas. Very common, easily readable by many programs (including R and Excel). **Often the best format for sharing data.**
- **TSV (Tab Separated Values - `.tsv`):** Similar to CSV, but uses tabs to separate columns.
- **Excel Files (`.xls`, `.xlsx`):** Native Microsoft Excel format. Can contain multiple sheets, formatting, formulas. Requires specific R packages to read/write.

## 7.2 Paths, Working Directory, and RStudio Projects (Best Practice!)

R needs to know *where* to find your files.

- **Working Directory:** The default folder location R looks in. You can see it with `getwd()` and set it with `setwd("path/to/folder")`, but **setting it manually is usually bad practice** because it makes your code non-portable.
- **Absolute Path:** The full path from the root of your computer (e.g., `"C:/Users/YourName/Documents/Bre`
  **Avoid this!** It breaks if you move folders or share your code.
- **Relative Path & RStudio Projects (RECOMMENDED):**
  1. Organize your work using an **RStudio Project**. Create one via `File -> New Project -> Existing Directory...` and select your main course folder (`course_project_baku`).
  2. When you open the `.Rproj` file, RStudio automatically sets the working directory to that project folder.
  3. Keep your data files *inside* the project folder, ideally in subdirectories like `data/raw` (original data) or `data/example` (cleaned data for examples).

4. Refer to files using **relative paths** starting from the project root, like `"data/example/phenotypes.csv"`. This makes your analysis reproducible and easy to share!

## 7.3 Reading Data into R

We'll use functions from the `readr` (for CSV/TSV) and `readxl` (for Excel) packages. Make sure they are installed (see Module 1.1).

```
# Load the necessary libraries
library(readr)
library(readxl)
library(dplyr) # for glimpse

# --- Reading a CSV file ---
# Assumes you have a file 'sample_phenotypes.csv' in the 'data/example' folder
# relative to your project root.
pheno_file_path <- "data/example/sample_phenotypes.csv"

# Check if file exists before trying to read (good habit)
if (file.exists(pheno_file_path)) {
  # Use read_csv from the readr package (generally preferred)
  phenotype_data <- read_csv(pheno_file_path)

  print("CSV data loaded successfully:")
  head(phenotype_data)   # Look at the first 6 rows
  glimpse(phenotype_data) # See column names and data types

} else {
  print(paste("Error: Phenotype file not found at", pheno_file_path))
  phenotype_data <- NULL # Set to NULL if file not found
}

# Note: Base R has read.csv() - it works but readr::read_csv() is often faster
# and handles data types more consistently (e.g., doesn't default strings to factors).

# --- Reading an Excel file ---
# Assumes you have 'sample_trial.xlsx' in 'data/example'
excel_file_path <- "data/example/sample_trial.xlsx" # You'll need to create this file

if (file.exists(excel_file_path)) {
```

```
  # See what sheets are in the workbook
  excel_sheets(excel_file_path)

  # Read data from a specific sheet (e.g., "YieldData")
  # yield_data_excel <- read_excel(excel_file_path, sheet = "YieldData")

  # Or read by sheet number (first sheet is 1)
  # yield_data_excel <- read_excel(excel_file_path, sheet = 1)

  # print("Excel data loaded:")
  # glimpse(yield_data_excel)

} else {
  print(paste("Warning: Example Excel file not found at", excel_file_path))
}
```

- **Always inspect your data after loading!** Use `head()`, `str()`, `glimpse()`, `summary()`. Did R read the column names correctly? Are the data types what you expected (numeric, character, etc.)?

## 7.4 Writing Data out of R

After cleaning data or performing analysis, you'll want to save results.

```
# Load libraries if not already loaded
library(readr)
library(writexl)

# Let's assume we filtered our phenotype data (from previous module example)
# Make sure phenotype_data exists first
if (!is.null(phenotype_data)) {
  # Example: Create a subset of high yielders (Yield > 11, assuming Yield column exists)
  # Check if Yield column exists before filtering
  if ("Yield" %in% names(phenotype_data)) {
    high_yielders <- phenotype_data[phenotype_data$Yield > 11, ] # Adjust threshold as neede

    # --- Writing to a CSV file ---
    # Use write_csv from readr. It doesn't write row numbers by default.
    output_csv_file <- "output/high_yielders_output.csv"
    write_csv(high_yielders, output_csv_file)
    print(paste("High yield data saved to:", output_csv_file))
```

```
    # --- Writing to an Excel file ---
    # Use write_xlsx from writexl. Can write multiple data frames to different sheets.
    output_excel_file <- "output/analysis_summary.xlsx"
    sheets_to_write <- list(
      HighYielders = high_yielders,
      OriginalDataSummary = summary(phenotype_data) # Example: write a summary too
      # Add other results data frames here
    )
    # write_xlsx(sheets_to_write, path = output_excel_file)
    # print(paste("Analysis results saved to:", output_excel_file))

  } else {
    print("Column 'Yield' not found in phenotype_data. Cannot filter or write.")
  }
} else {
  print("phenotype_data object does not exist. Cannot write data.")
}
```

**Exercise:** If you have a simple Excel file with some breeding data (e.g., Plot ID, Variety, Yield), try reading it into R using `read_excel()`. Inspect the loaded data frame using `glimpse()`.

# 8 Why Visualize Your Data?

"A picture is worth a thousand words" - this is especially true for data! Plots help us to:

- **Explore:** Understand the distribution of your traits (e.g., `Yield`, `Height`). See relationships between variables. Identify patterns.
- **Diagnose:** Spot potential problems like outliers (strange values) or unexpected groupings. Check assumptions of statistical models.
- **Communicate:** Clearly present your findings to colleagues, managers, or in publications.

## 8.1 Introducing `ggplot2`: The Grammar of Graphics

R has basic plotting functions, but we will focus on the **`ggplot2`** package, which is part of the `tidyverse`. It's extremely powerful and flexible for creating beautiful, publication-quality graphics.

`ggplot2` is based on the **Grammar of Graphics**. The idea is to build plots layer by layer:

1. **`ggplot()` function:** Start the plot. You provide:
    - `data`: The data frame containing your variables.
    - `mapping = aes(...)`: **Aesthetic mappings**. This tells `ggplot` *how variables in your data map to visual properties* of the plot (e.g., map `Yield` to the y-axis, `Height` to the x-axis, `Variety` to color).
2. **`geom_` functions:** Add geometric layers to actually *display* the data. Examples:
    - `geom_point()`: Creates a scatter plot.
    - `geom_histogram()`: Creates a histogram.
    - `geom_boxplot()`: Creates box-and-whisker plots.
    - `geom_line()`: Creates lines.
    - `geom_bar()`: Creates bar charts.
3. **Other functions:** Add labels (`labs()`), change themes (`theme_bw()`, `theme_minimal()`), split plots into facets (`facet_wrap()`), customize scales, etc.

## 8.2 Let's Make Some Plots!

First, load the necessary libraries:

```
# Load necessary libraries
library(ggplot2)
library(dplyr) # Often used with ggplot2 for data prep
```

Now, let's create a sample breeding data frame for plotting.

```
set.seed(123) # for reproducible random numbers
breeding_plot_data <-
  tibble(
    PlotID = paste0("P", 101:120),
    Variety = factor(rep(c("ICARDA_A", "ICARDA_B", "Check_1", "Check_2"), each = 5)),
    Location = factor(rep(c("Baku", "Ganja"), each = 10)),
    Yield = rnorm(20, mean = rep(c(6, 7, 5, 5.5), each = 5), sd = 0.8),
    Height = rnorm(20, mean = rep(c(90, 110, 85, 88), each = 5), sd = 5)
  )

# Take a quick look at the data structure
glimpse(breeding_plot_data)
```

```
Rows: 20
Columns: 5
$ PlotID   <chr> "P101", "P102", "P103", "P104", "P105", "P106", "P107", "P108~
$ Variety  <fct> ICARDA_A, ICARDA_A, ICARDA_A, ICARDA_A, ICARDA_A, ICARDA_B, I~
$ Location <fct> Baku, Baku, Baku, Baku, Baku, Baku, Baku, Baku, Baku, Baku, G~
$ Yield    <dbl> 5.551619, 5.815858, 7.246967, 6.056407, 6.103430, 8.372052, 7~
$ Height   <dbl> 84.66088, 88.91013, 84.86998, 86.35554, 86.87480, 101.56653, ~
```

### 8.2.1 1. Scatter Plot: Relationship between Yield and Height

See if taller plants tend to have higher yield in this dataset.

```
# 1. ggplot(): data is breeding_plot_data, map Height to x, Yield to y
# 2. geom_point(): Add points layer
# 3. labs() and theme_bw(): Add labels and theme
plot1 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Height, y = Yield)) +
```

```
  geom_point() +
  labs(
    title = "Relationship between Plant Height and Yield",
    x = "Plant Height (cm)",
    y = "Yield (kg/plot)",
    caption = "Sample Data"
  ) +
  theme_bw() # Use a clean black and white theme

# Display the plot
plot1
```

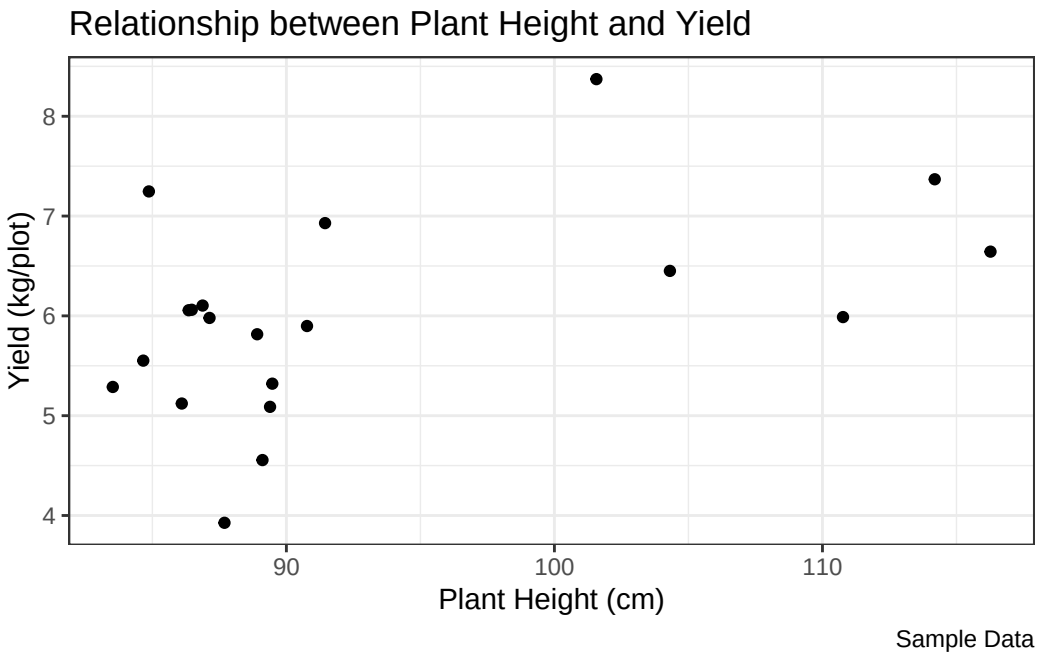


Figure 8.1: Relationship between Plant Height and Yield.

Let's color the points by Variety:

```
# Map 'color' aesthetic to the Variety column
# Adjust point size and transparency for better visibility
plot2 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Height, y = Yield, color = Variety)) +
  geom_point(size = 2.5, alpha = 0.8) + # Make points slightly bigger, semi-transparent
  labs(
```

```
    title = "Height vs. Yield by Variety",
    x = "Plant Height (cm)",
    y = "Yield (kg/plot)"
  ) +
  theme_minimal() # Use a different theme

# Display the plot
plot2
```


Figure 8.2: Height vs. Yield by Variety, colored by Variety.

### 8.2.2 2. Histogram: Distribution of Yield

See the frequency of different yield values.

```
# 1. ggplot(): data, map Yield to x-axis
# 2. geom_histogram(): Add histogram layer. Adjust 'binwidth' or 'bins'.
# 3. labs() and theme_classic(): Add labels and theme
plot3 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Yield)) +
  geom_histogram(binwidth = 0.5, fill = "lightblue", color = "black") + # Specify binwidth,
```

```
  labs(
    title = "Distribution of Plot Yields",
    x = "Yield (kg/plot)",
    y = "Frequency (Number of Plots)"
  ) +
  theme_classic()

# Display the plot
plot3
```

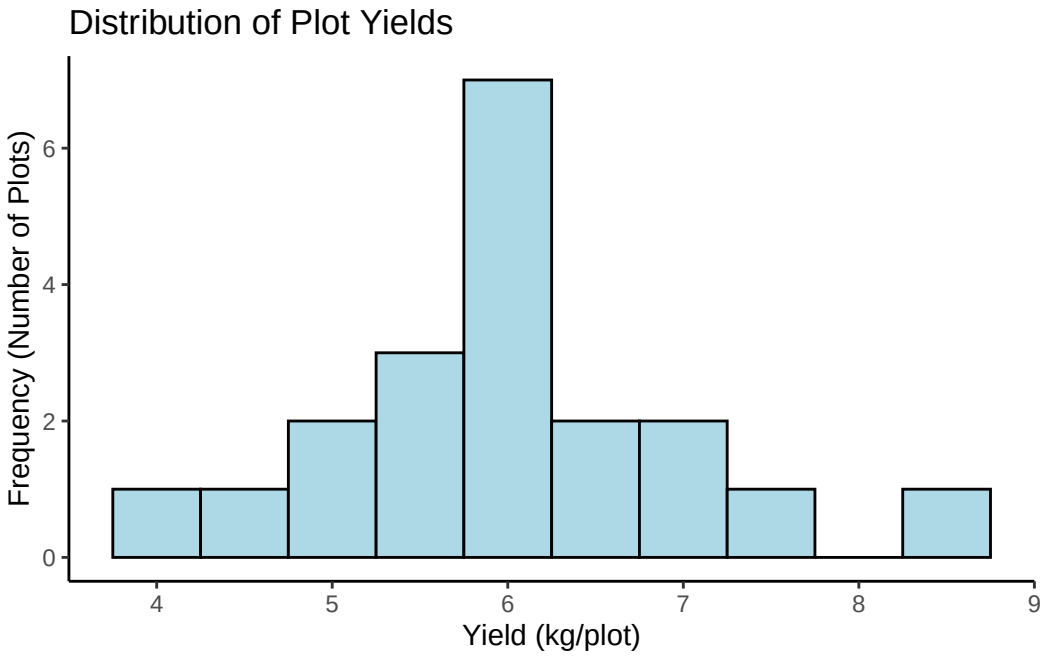


Figure 8.3: Distribution of Plot Yields.

### 8.2.3 3. Box Plot: Compare Yield across Locations

Are yields different in Baku vs. Ganja? Box plots are great for comparing distributions across groups.

```
# 1. ggplot(): data, map Location (categorical) to x, Yield (numeric) to y
# 2. geom_boxplot(): Add boxplot layer. Map 'fill' to Location for color.
# 3. labs() and theme_light(): Add labels and theme
# 4. theme(): Customize theme elements (e.g., remove legend)
```

```r
plot4 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Location, y = Yield, fill = Location))
  geom_boxplot() +
  labs(
    title = "Yield Comparison by Location",
    x = "Location",
    y = "Yield (kg/plot)"
  ) +
  theme_light() +
  theme(legend.position = "none") # Hide legend if coloring is obvious from x-axis

# Display the plot
plot4
```

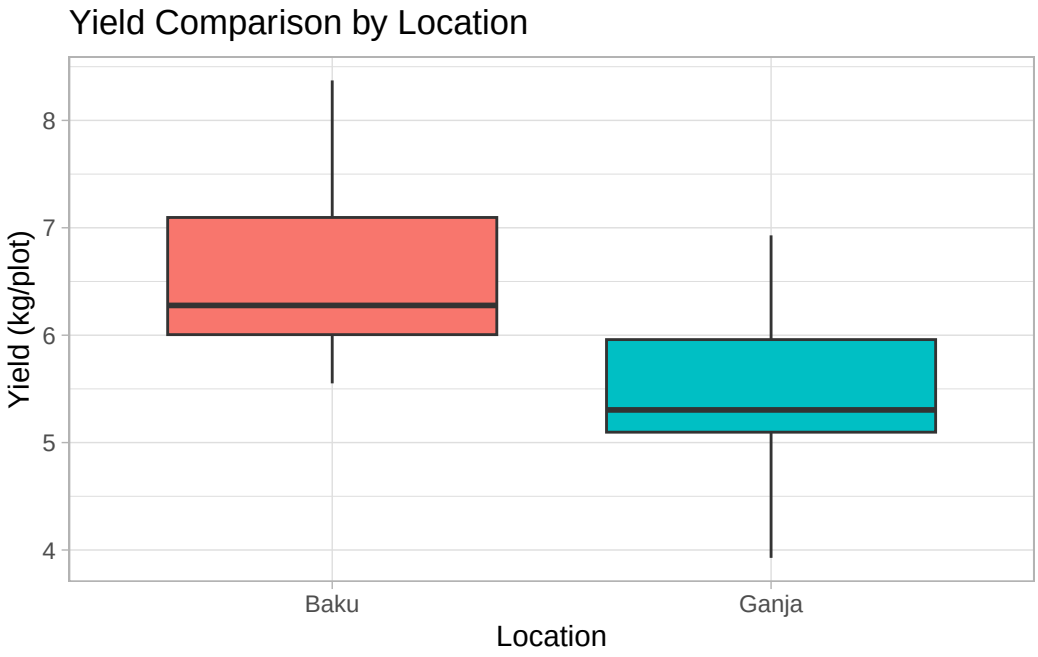


Figure 8.4: Yield Comparison by Location.

Box plot anatomy: The box shows the interquartile range (IQR, middle 50% of data), the line inside is the median, whiskers extend typically 1.5*IQR, points beyond are potential outliers.

## 8.3 Saving Your Plots

Use the `ggsave()` function after you've created a ggplot object (like plot1, plot2, etc.).

```
# Make sure the 'output/figures' directory exists
# The 'recursive = TRUE' creates parent directories if needed
output_dir <- "output/figures"
if (!dir.exists(output_dir)) {
  dir.create(output_dir, recursive = TRUE)
}

# Save the height vs yield scatter plot (plot2)
ggsave(
  filename = file.path(output_dir, "height_yield_scatter.png"), # Use file.path for robust pa
  plot = plot2,                                  # The plot object to save
  width = 7,                                     # Width in inches
  height = 5,                                    # Height in inches
  dpi = 300                                      # Resolution (dots per inch)
)

# You can save in other formats too, like PDF:
# ggsave(
#   filename = file.path(output_dir, "yield_distribution.pdf"),
#   plot = plot3,
#   width = 6,
#   height = 4
# )

cat("Plot saved to", file.path(output_dir, "height_yield_scatter.png"), "\n")
```

```
Plot saved to output/figures/height_yield_scatter.png
```

## 8.4 Exercise

Create a box plot comparing Plant Height (`Height`) across the different Varieties (`Variety`) in the `breeding_plot_data`. Save the plot as a PNG file named `height_variety_boxplot.png` in the `output/figures` directory.

```r
# Exercise: Box plot comparing Plant Height across Varieties
plot5 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Variety, y = Height, fill = Variety)) +
  geom_boxplot() +
  labs(
    title = "Plant Height Comparison by Variety",
    x = "Variety",
    y = "Plant Height (cm)"
   ) +
  theme_light() +
  theme(legend.position = "none")

# Display the new plot
plot5
```

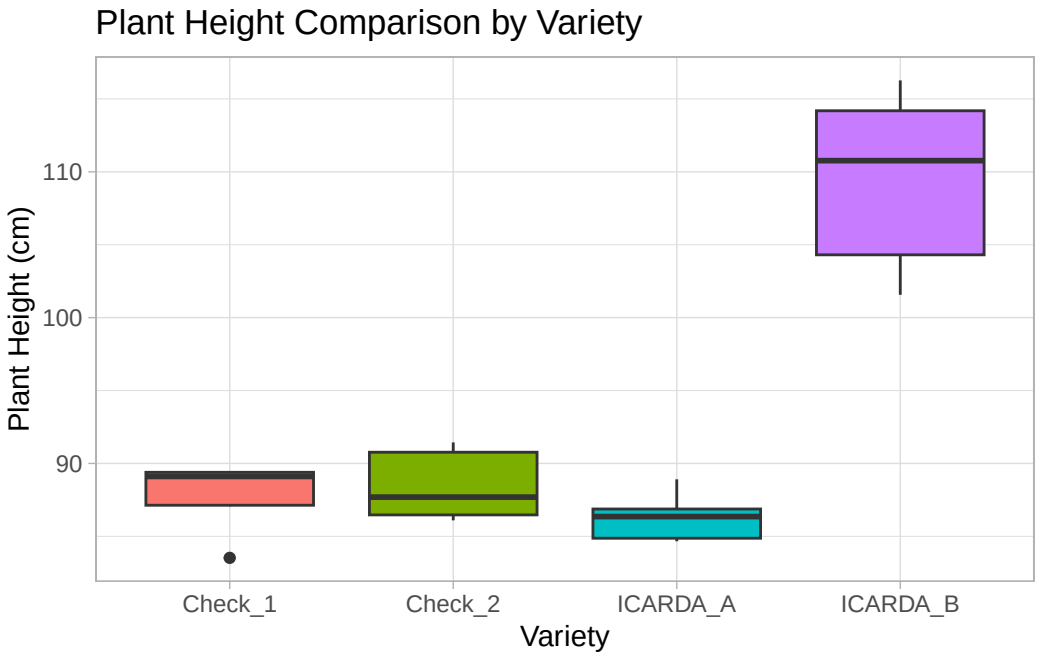


Figure 8.5: Plant Height Comparison by Variety.

```r
# Ensure output directory exists
output_dir <- "output/figures"
if (!dir.exists(output_dir)) {
  dir.create(output_dir, recursive = TRUE)
}
```

```r
# Save the box plot as a PNG file
ggsave(
  filename = file.path(output_dir, "height_variety_boxplot.png"),
  plot = plot5,
  width = 7,
  height = 5,
  dpi = 300
)

cat("Exercise plot saved to", file.path(output_dir, "height_variety_boxplot.png"), "\n")
```

```
Exercise plot saved to output/figures/height_variety_boxplot.png
```

# Part III

# Handling Breeding Data

# 9 Module 2.1: Loading Breeding Data - ICARDA Barley Example

## 9.1 Introduction to the Dataset

In this module, we'll learn how to load typical phenotypic data into R. We'll use a real-world example: data from a study on **275 barley accessions conducted at ICARDA in 2019**. This dataset contains various measurements related to agronomic traits, grain quality, and morphological characteristics.

**Why this dataset?** * It's representative of the kind of multi-trait data breeders work with. * It allows us to practice loading, inspecting, and performing basic summaries on realistic data. * This data comes from ICARDA's valuable work in crop improvement for dry areas.

**Column Descriptions (Partial List - full list would be in a data dictionary):** * `Taxa`: The identifier for each barley accession (genotype). * `Area`: Grain area (e.g., $mm^2$). * `B_glucan`: Beta-glucan content (%), a quality trait. * `DTH`: Days to Heading (days), an agronomic trait. * `Fe`: Iron content in grain (ppm), a nutritional trait. * `FLA`: Flag Leaf Area ($cm^2$). * `GY`: Grain Yield (e.g., t/ha or kg/plot - units should always be known!). * `PH`: Plant Height (cm). * `Protein`: Grain protein content (%). * `TKW`: Thousand Kernel Weight (grams). * `Zn`: Zinc content in grain (ppm). * *(And many others related to grain morphology and plant characteristics…)*

Our goal is to load this data (which is typically stored in a file like a CSV or Excel sheet) into an R data frame so we can start analyzing it.

## 9.2 Setting Up: Libraries and File Path

First, we need to load the R packages that help us read data. The `readr` package (part of `tidyverse`) is excellent for reading text files like CSVs.

Remember our RStudio Project setup! We will assume the data file is saved in the `data/example/` subfolder of our project.

```r
# Load the necessary libraries
# 'tidyverse' includes 'readr' (for read_csv) and 'dplyr' (for glimpse, etc.)
library(tidyverse)
```

## 9.3 Reading the CSV File

Let's say our barley data is stored in a CSV file named `icarda_barley_2019_pheno.csv`.

```r
# Define the path to our data file (relative to the project root)
barley_data_file_path <- "data/example/icarda_barley_2019_pheno.csv"

# Check if the file exists (good practice!)
if (file.exists(barley_data_file_path)) {
  # Use read_csv() from the readr package to load the data
  barley_pheno_data <- read_csv(barley_data_file_path)

  print("ICARDA Barley Phenotype data loaded successfully!")
} else {
  print(paste("ERROR: File not found at:", barley_data_file_path))
  print("Please make sure 'icarda_barley_2019_pheno.csv' is in the 'data/example' folder.")
  # If the file isn't found, we'll create an empty placeholder to avoid later errors in the c
  barley_pheno_data <- tibble() # Creates an empty tibble (tidyverse data frame)
}
```

## 9.4 First Look: Inspecting the Loaded Data

It's **CRUCIAL** to always inspect your data immediately after loading it to make sure it looks correct.

1. **`head()`**: Shows the first few rows (default is 6). `r # Only run this if barley_pheno_data was loaded successfully if (nrow(barley_pheno_data) > 0) { head(barley_pheno_data) }`
2. **`dim()`**: Shows the dimensions (number of rows, number of columns). `r if (nrow(barley_pheno_data) > 0) { dim(barley_pheno_data) # We expect around 275 rows (accessions) and several columns (traits) }`
3. **`glimpse()`** (from `dplyr`): A great way to see column names, their data types, and the first few values. Better than `str()` for tibbles. `r if (nrow(barley_pheno_data) > 0) { glimpse(barley_pheno_data) # Pay attention to the`

```
data types:       # - <chr> for character (like Taxa)       # - <dbl>
for double (numeric with decimals, like most traits)       # - <int>
for integer (whole numbers)       # read_csv usually does a good job
guessing, but sometimes you might need to specify.     }
```

4. **summary()**: Provides basic summary statistics for each column (Min, Max, Mean, Median, Quartiles for numeric; counts for character/factor). `r if (nrow(barley_pheno_data) > 0) { summary(barley_pheno_data) # This helps spot: # - Obvious errors (e.g., negative yield if not possible) # - Range of values for each trait # - Number of NAs (missing values) if any }`

## 9.5 Understanding Data Types in Our Barley Data

When `glimpse()` runs, you'll see types like: * `Taxa`: Should be `<chr>` (character) as it's an identifier. * `Area`, `B_glucan`, `DTH`, `GY`, `PH`, etc.: Should mostly be `<dbl>` (double-precision numeric) as they are measurements.

If `read_csv` misinterprets a numeric column as character (e.g., if there's a text entry like "missing" in a numeric column), you'll need to clean that data or specify column types during import using the `col_types` argument in `read_csv()`. (We'll cover data cleaning later).

## 9.6 Quick Summary of a Specific Trait

Let's say we are interested in Grain Yield (`GY`).

```
# Make sure the data and the 'GY' column exist
if (nrow(barley_pheno_data) > 0 && "GY" %in% names(barley_pheno_data)) {
  # Access the GY column
  yield_values <- barley_pheno_data$GY

  # Calculate some basic statistics
  mean_yield <- mean(yield_values, na.rm = TRUE) # na.rm=TRUE ignores missing values in calcu
  min_yield <- min(yield_values, na.rm = TRUE)
  max_yield <- max(yield_values, na.rm = TRUE)
  sd_yield <- sd(yield_values, na.rm = TRUE)

  print(paste("Average Grain Yield (GY):", round(mean_yield, 2)))
  print(paste("Minimum Grain Yield (GY):", round(min_yield, 2)))
  print(paste("Maximum Grain Yield (GY):", round(max_yield, 2)))
  print(paste("Standard Deviation of GY:", round(sd_yield, 2)))
```

```
  # How many accessions do we have yield data for (non-missing)?
  num_yield_obs <- sum(!is.na(yield_values))
  print(paste("Number of accessions with GY data:", num_yield_obs))
} else if (nrow(barley_pheno_data) > 0) {
  print("Column 'GY' not found in the loaded data.")
}
```

**Exercise:** 1. Load the `icarda_barley_2019_pheno.csv` file into R. 2. Use `glimpse()` to check the column names and data types. 3. Calculate and print the average Plant Height (`PH`) from the dataset. Remember to handle potential missing values (`na.rm = TRUE`).

This module has shown you the first critical step: getting your valuable field data into R. In the next modules, we'll learn how to clean, manipulate, and visualize this data.

# 10

# Part IV

# Core Genomic Concepts

# 11

# 12

# Part V

# Population Structure and Relatedness

# 13

# 14

# 15

# Part VI

# Marker-Trait Association (GWAS)

# 16

# 17

# 18

# Part VII

# Tools and Advanced Concepts

# 19

# 20