Reimagining Intro Stats Through Data Science

A Hands-On Approach with ModernDive

USCOTS 2025

Follow-along slides available at https://bit.ly/md-uscots-slides

Learning Objectives

By the end of this workshop, you will be able to help your students and yourself to:

Perform data wrangling techniques in R via the **tidyverse**

Learn powerful data manipulation tools to clean, transform, and prepare data for analysis

Develop skills in data visualization with **ggplot2**

Create effective, publication-quality graphics to communicate your findings

Apply fundamental concepts of statistical inference with **infer**

Understand sampling distributions, confidence intervals, and hypothesis testing

Integrate Theory-Based and Simulation-Based Approaches

Compare traditional statistical methods with modern computational techniques

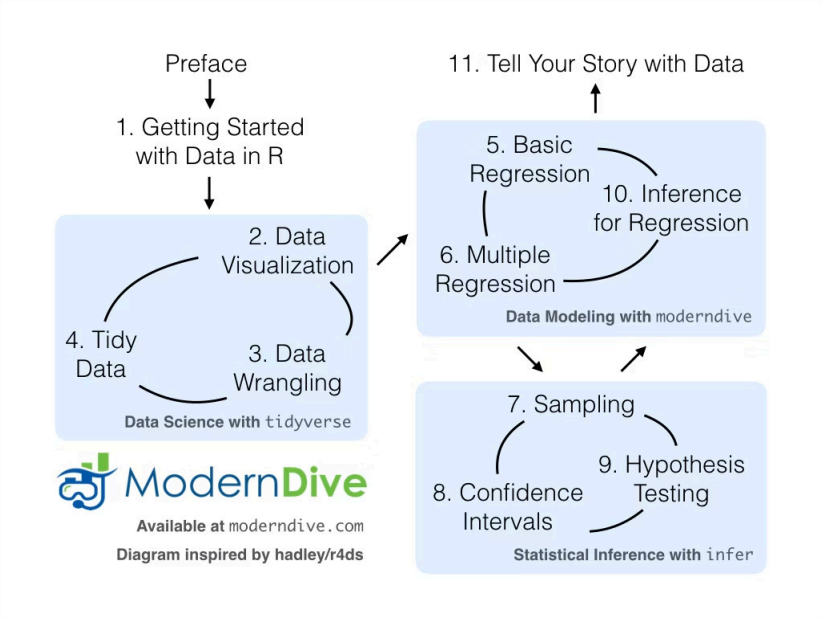
Instructors' Introduction

Chester Ismay

Portland State University

Arturo Valdivia

Indiana University



What to Expect

1. Build a comprehensive, application-driven foundation in both statistics and data science
2. Master essential statistical concepts alongside practical R coding skills
3. Transform your approach to data through reproducible, transparent methodologies
4. Develop powerful programming capabilities that enhance both problem-solving and statistical reasoning

Agenda

Part 0: Introduction to R and RStudio

Getting started with the tools and environment

Part 1: Data Visualization

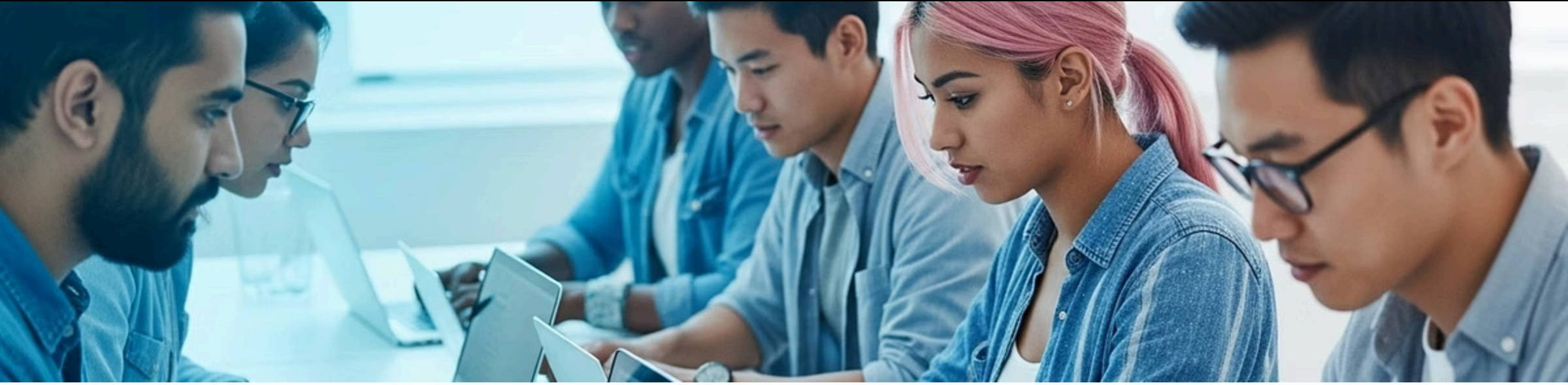
Creating effective graphics using the grammar of graphics

Part 2: Data Wrangling and Tidy Data

Transforming and preparing data for analysis

Part 3: Statistical Inference

Using computation and simulation to understand statistical inference



Part 0: Introduction to R and RStudio

Introduction to R and RStudio

R: Engine	RStudio: Dashboard
	

R:

Programming language designed specifically for statistical computing and data analysis with extensive package ecosystem

RStudio:

Integrated development environment (IDE) that makes working with R easier and more productive

R vs RStudio:

R is the engine; RStudio is the dashboard that helps you drive it efficiently

Installing R and RStudio

Step 1: Download and Install R

Visit: **https://cloud.r-project.org/**

- Select your operating system
- Follow installation instructions for your platform
- R is the core statistical computing engine

Step 2: Download and Install RStudio

Visit: **https://posit.co/download/rstudio-desktop/**

- Download the appropriate version for your OS
- Complete the installation process
- Open RStudio to begin working

> Raise your hand if you need help with installation!

Exploring RStudio

Understanding the RStudio Interface

Console

Where you can type and run your code directly. Results appear immediately after execution, interacting with R in real-time.

Environment

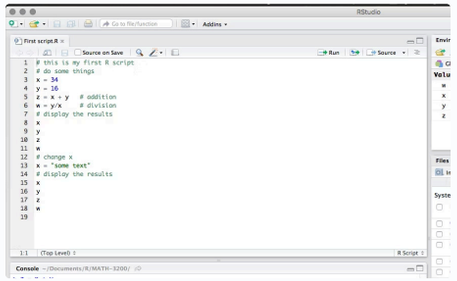
Shows all objects (like datasets, variables, and functions) currently in memory and available for use in your session.

Files

Helps you navigate files and directories in your project. Makes it easy to open and organize your data and code files.

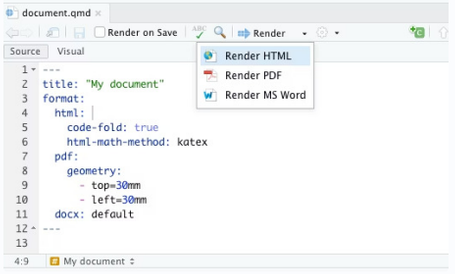
Working in RStudio

In RStudio you have the flexibility to work with different types of documents:



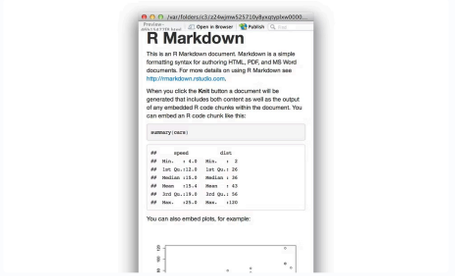
R Scripts (.R files)

Plain text files containing R code. Simple, reusable, and great for developing analysis pipelines.



Quarto documents (.qmd)

Next-generation reproducible documents combining code, results, and narrative text.



R Markdown (.Rmd)

Documents that blend narrative text with embedded R code chunks for reproducible reports.

(The precursor to Quarto)

RStudio also supports project management, Shiny interactive applications, package development, and much more!

Working in RStudio

> ## Download and open in RStudio the following file:
>
> **https://bit.ly/md-uscots-code**

Raise your hand if you need help!

This file contains all the code we'll be working with today. We'll explore it together step-by-step throughout the workshop.

Demo

We'll will now demonstrate:

- Opening and navigating the downloaded file
- Running basic R commands
- Understanding the RStudio interface
- Basic file management in RStudio

Follow along on your own computer and ask questions as needed!

Installing and Loading R Packages

Extend R's capabilities with additional functions and/or datasets

Step 1: Install the package

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

Only needs to be done once per R installation

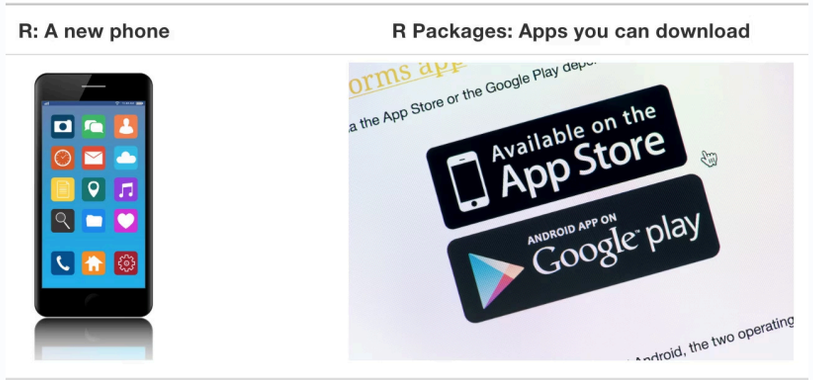
Step 2: Load the package

```
library(tidyverse)
```

Must be done in each new R session where you need the package

Step 3: Use the functions

Once loaded, all functions from the package are available for use



The **tidyverse** is a collection of R packages designed for data science that share a common philosophy and design.

Working with Data Sets

Loading Data

Access datasets from packages or import from external files

```
data(un_member_states_2024, package = "modernive")  # Data in R package
flights <- read_csv("flights.csv")  # External
```

Exploring Structure

Examine data types and organization

```
glimpse(flights)
```

Accessing Columns

Extract specific variables

```
un_member_states_2024$country  # Dollar notation
flights |> select(origin)  # tidyverse
```

Viewing Data

Display data in various formats

```
head(un_member_states_2024)  # First 6 rows
View(flights)  # Open data viewer
```

These basic operations form the foundation of data analysis workflows in R.

Loading and Viewing a Dataset

Access Data from R Packages

```
data(datasets::penguins)
head(penguins)
```

Many R packages include datasets that are immediately available after loading the package.

Import External Data

```
library(readr)
# CSV files
my_data <- read_csv("data.csv")

# Excel files (requires readxl)
library(readxl)
xl_data <- read_excel("data.xlsx")
```

The **tidyverse** provides specialized functions for importing various data formats.

Activities & Exercises

We will demonstrate and you'll practice:

- Installing and loading necessary packages
- Accessing built-in datasets
- Exploring data structure and content
- Basic data manipulation techniques

Exploring Data in R with RStudio

Data Frames Are Like Tables

Data frames organize data into rows and columns, similar to spreadsheets but with powerful programmatic capabilities.

```
dplyr::glimpse(penguins)  # Structure
dim(penguins)  # Dimensions
nrow(penguins)
ncol(penguins)
names(penguins)  # Column names
```

Inspecting Data

- **View()** - Opens data in spreadsheet viewer
- **glimpse()** - Compact summary of structure
- **$** operator - Extracts columns
- **head()** - Shows first few rows

Variable Types

Distinguish between identification variables (e.g., ID, name) and measurement variables (e.g., height, temperature)

An Introduction to Coding in R

Key Programming Concepts

- *Commands* are entered as code in the Console or via scripts
- *Objects* store data, functions, and results
- *Vectors* are sequences of values of the same type
- *Data types* include numeric, character, logical, etc.
- *Conditional statements* control program flow
- *Functions* perform specific tasks with inputs and outputs

> Learning to code takes frequent practice, but it is one of the most rewarding things you can do!

We'll build these skills incrementally throughout the workshop, starting with basic operations.

An Introduction to Coding in R

Basic Operations in R

```
# Arithmetic operations
5 + 3
10 / 2
3 ^ 2

# Variable assignment
x <- 10
y <- 5
result <- x * y
result  # Print the result

# Vectors
numbers <- c(1, 2, 3, 4, 5)
mean(numbers)  # Calculate mean
```

Using Functions in R

```
# Basic functions
sqrt(16)
round(3.14159, digits = 2)

# Statistical functions
data_values <- c(7, 8, 9, 10, 11)
mean(data_values)
median(data_values)
sd(data_values)

# Help documentation
?mean # Get help for a function
```

Activities & Exercises

We will demonstrate and you'll practice:

- Creating and manipulating variables
- Working with vectors
- Applying functions to data
- Accessing help documentation

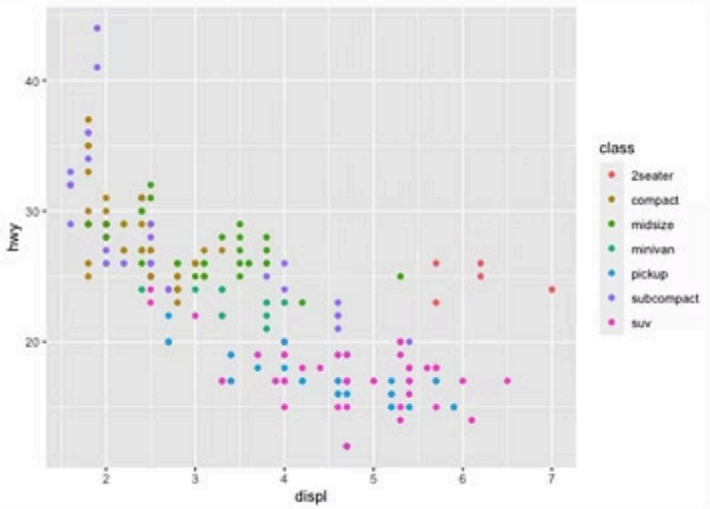
Follow along in your own RStudio session!

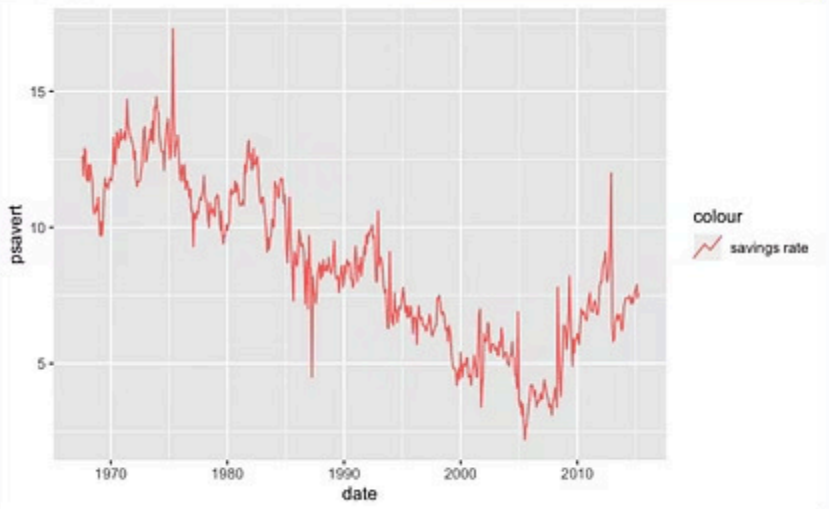
Q & A

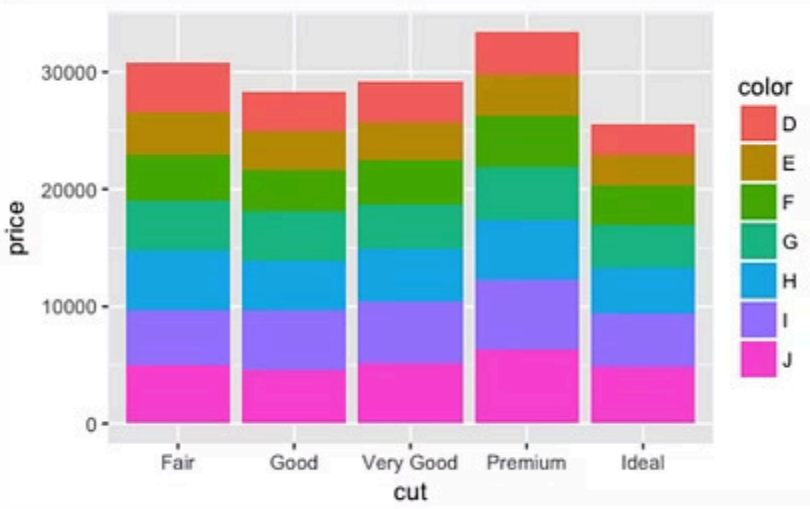
Let's address your questions about:

- R and RStudio setup
- Basic R syntax and commands
- Working with data frames
- Accessing help resources
- Troubleshooting common errors
- Best practices for organizing code
- Package installation issues
- Any other R basics questions

This is a great time to clarify any concepts before we move on to data visualization!







Part 1: Data Visualization

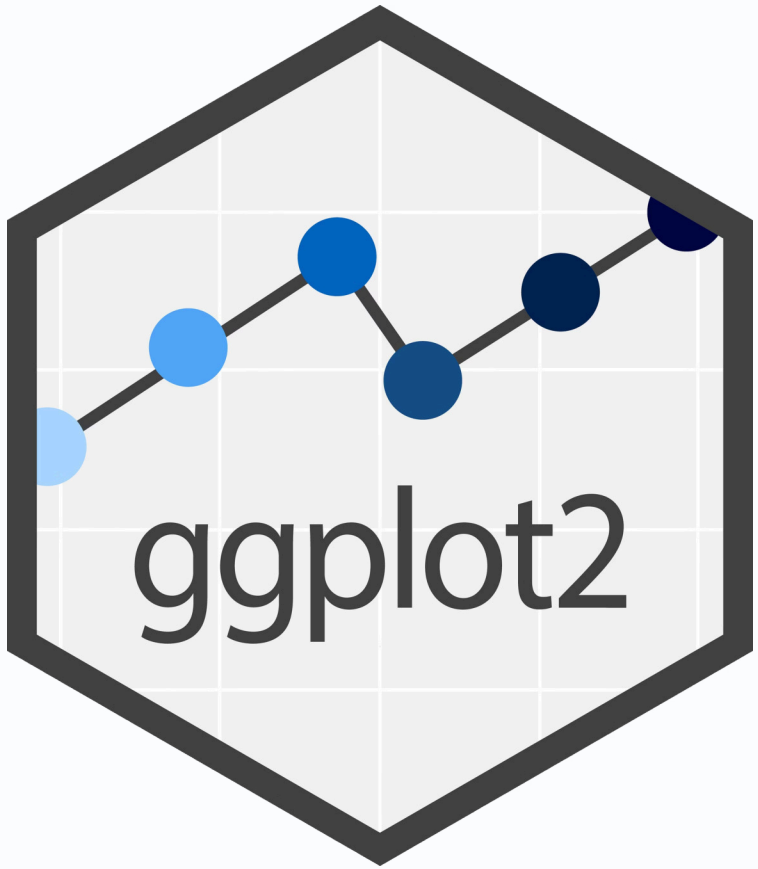
Introduction to Data Visualization

Why Visualize Data?

- Raw data rarely reveals patterns and relationships
- Visualizations help identify outliers, distributions, and correlations
- Graphics communicate findings more effectively than tables
- Visual exploration often leads to new insights and questions

ggplot2 Package

We'll use **ggplot2**, part of the **tidyverse**, which implements the Grammar of Graphics by Leland Wilkinson



Visualizations transform abstract numbers into meaningful patterns that the human brain can quickly interpret.

The Grammar of Graphics using ggplot2

data

The dataset containing variables to visualize

Aesthetics (**aes**)

Mapping variables to visual properties like position, color, size, and shape

Layering

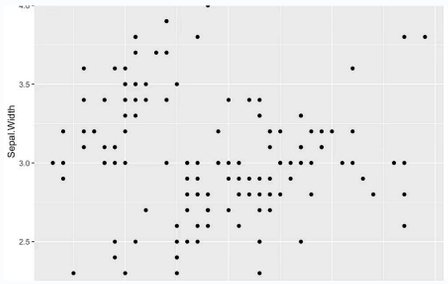
Building plots by adding components one layer at a time

Geometries (**geom**)

Visual elements representing data points (bars, points, lines)

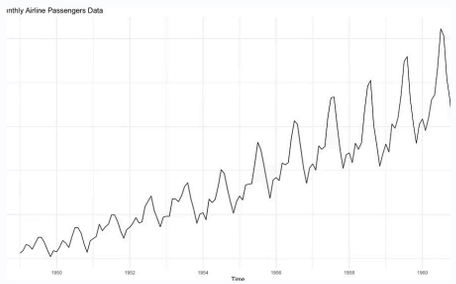
ggplot2's power comes from its consistent grammar and layered approach to building visualizations.

The Five Named Graphs



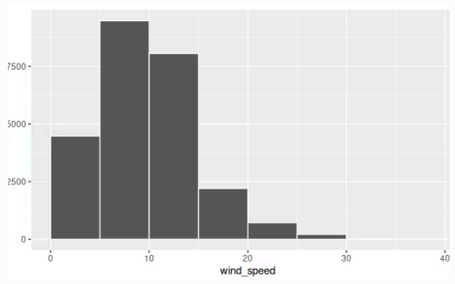
Scatterplots

Show relationships between two continuous variables, revealing patterns, correlations, and outliers



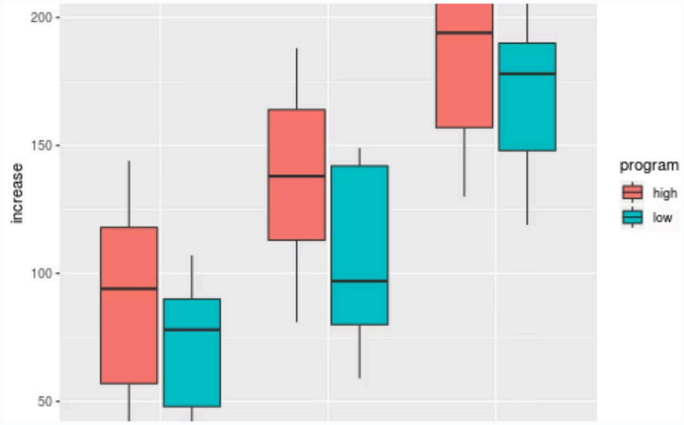
Line Graphs

Display trends over time or ordered categories, emphasizing continuity and change



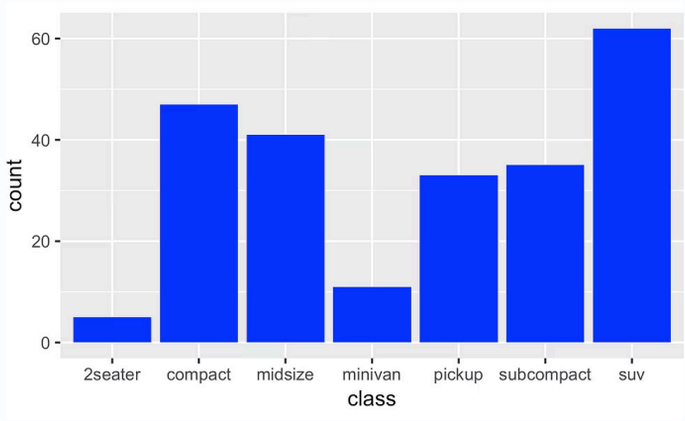
Histograms

Visualize distributions of a single continuous variable, showing frequency and shape



Boxplots

Summarize distributions using five-number summaries, excellent for comparing groups



Bar Plots

Display counts or summary statistics for categorical variables or discrete groups

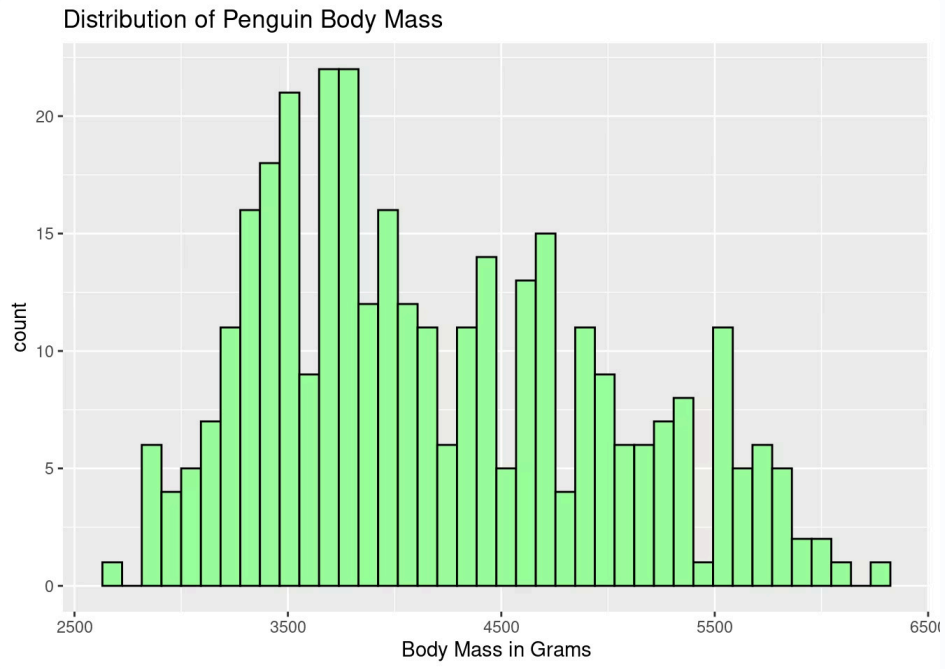
Histograms

Key Features

- Visualize distribution of a single numerical variable
- Show frequency of values falling in specific ranges (bins)
- Reveal shape, center, spread, and unusual features
- Height represents count or density

ggplot2 Implementation

```
ggplot(data = penguins,
       aes(x = body_mass)) +
  geom_histogram(bins = 20,
                 fill = "lightgreen",
                 color = "black") +
  labs(title = "Distribution of Penguin Body Mass",
       x = "Body Mass in Grams",
       y = "count")
```



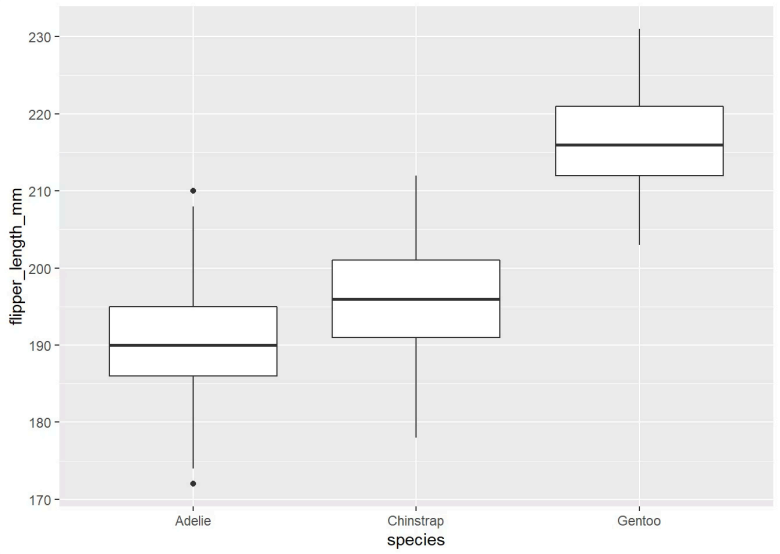
> *Tip*: Adjust bin width or number of bins to better represent your data. Too few bins hide detail; too many create noise.

Boxplots

Key Features

- Summarize data using quartiles (25th, 50th, 75th percentiles)
- Box represents interquartile range (IQR) with median line
- Whiskers extend to ±1.5*IQR or to min/max values
- Points beyond whiskers indicate potential outliers
- Perfect for comparing distributions across groups

```
ggplot(data = penguins,
       aes(x = species,
           y = flipper_len)) +
  geom_boxplot() +
  labs(x = "species",
       y = "flipper_length_mm")
```



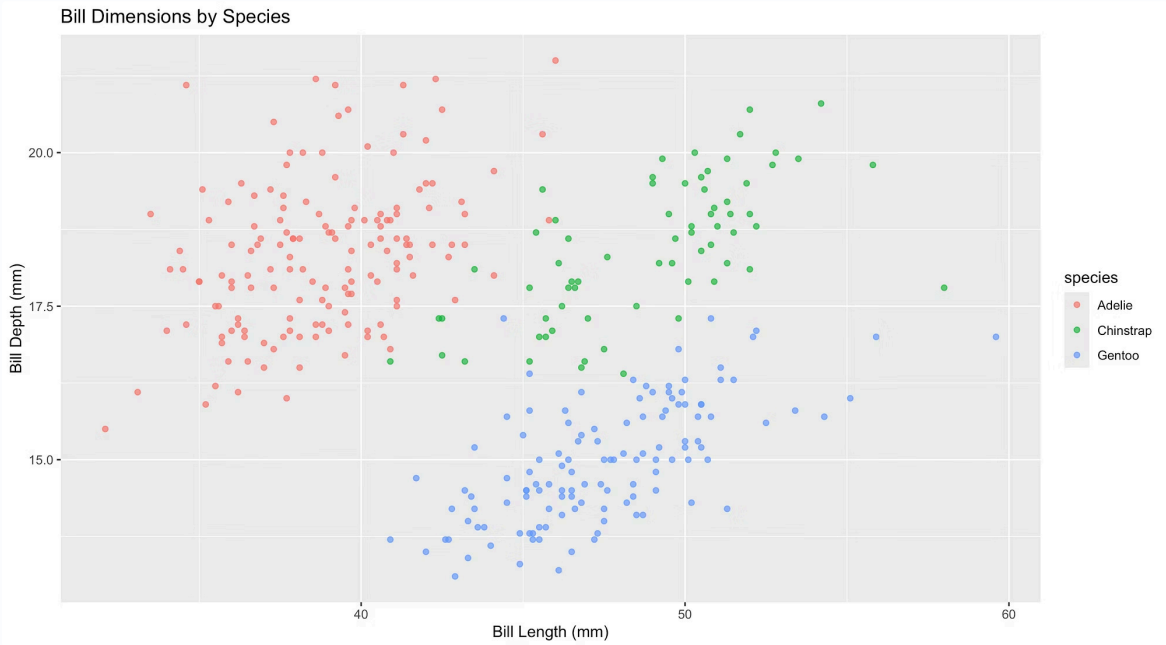
> *Tip*: Use boxplots when you need to compare distributions across multiple groups or categories. They efficiently show center, spread, and outliers.

Scatterplots

Key Features

- Display relationship between two numerical variables
- Each point represents one observation
- Reveal patterns, correlations, clusters, and outliers
- Can encode additional variables using color, shape, or size

```
ggplot(data = penguins,
       aes(x = bill_length_mm,
           y = bill_depth_mm,
           color = species)) +
  geom_point(alpha = 0.7) +
  labs(title = "Bill Dimensions by Species",
       x = "Bill Length (mm)",
       y = "Bill Depth (mm)")
```



Handling Overplotting

- `alpha` **transparency:** Reveal density by making points semi-transparent
- **jittering:** Add small random noise to prevent exact overlaps
- `geom_jitter()`: A specialized `geom` for scatterplots with built-in jittering

Activities & Exercises

We will demonstrate and you'll practice:

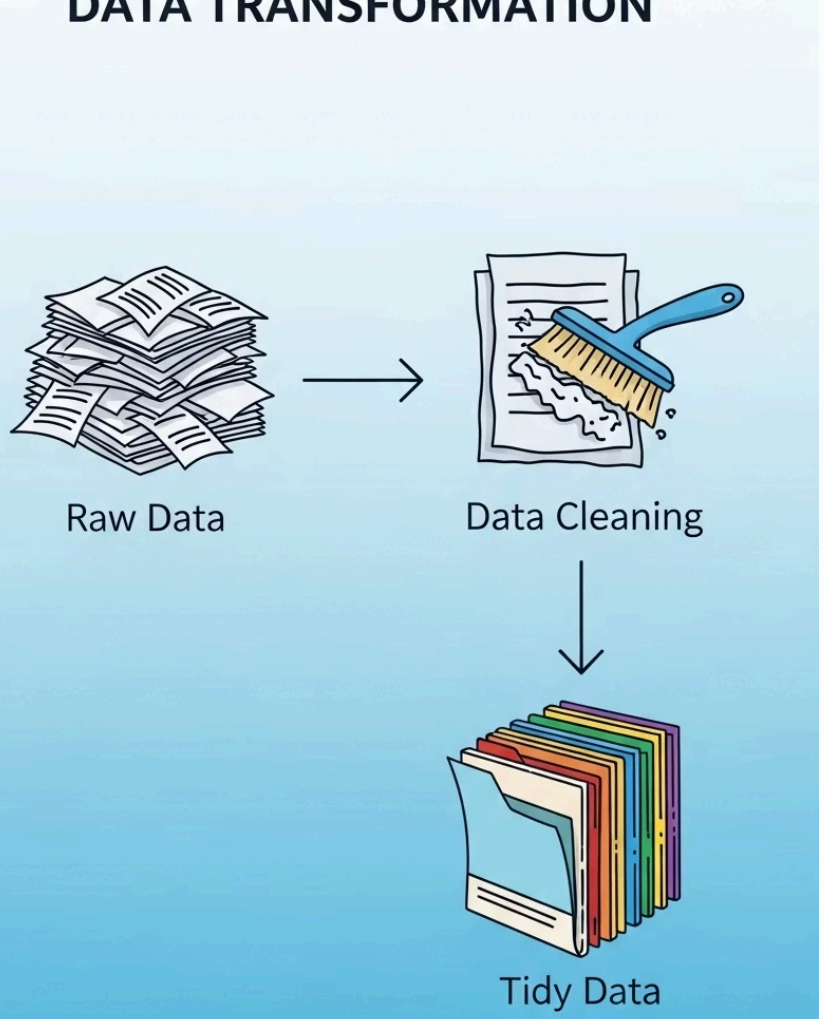
- Building visualizations layer by layer with ggplot2
- Creating histograms, boxplots, and scatterplots
- Customizing plot appearance with colors, labels, and themes
- Handling common visualization challenges

Q & A

Let's address your questions about data visualization:

- **ggplot2** syntax and structure
- Choosing appropriate plot types
- Customizing plot aesthetics
- Combining multiple plots
- Handling specific data challenges
- Saving and exporting plots
- Advanced visualization techniques
- Common errors and troubleshooting

Now is the perfect time to clarify any visualization concepts before moving on to data wrangling!



Part 2: Data Wrangling

Data Wrangling

Overview of the tidyverse

The **tidyverse** is an integrated collection of R packages designed for data science with a consistent design philosophy.

Importance of Data Wrangling

- Real data is rarely analysis-ready
- Cleaning and transformation take up to 80% of analysis time
- Proper wrangling ensures valid statistical conclusions
- Reproducible wrangling documents data processing decisions

Key Package: **dplyr**

We'll focus on **dplyr**, which provides intuitive functions for data manipulation with a consistent grammar.

Filter Rows

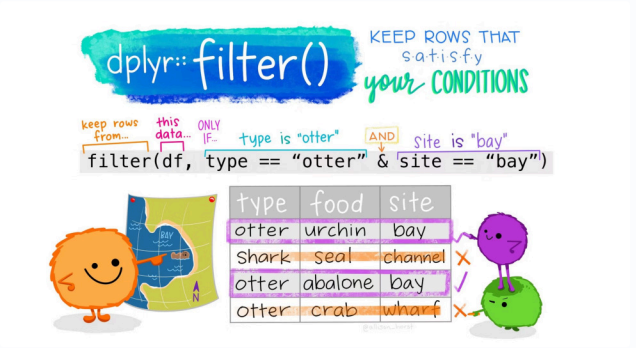
Selecting Observations Based on Conditions

```
library(dplyr)

# Keep only Adelie penguins
adelie_penguins <- penguins |>
  filter(species == "Adelie")

# Multiple conditions
large_adelie <- penguins |>
  filter(species == "Adelie" &
         body_mass > 4000)

# Either/or conditions
not_gentoo <- penguins |>
  filter(species != "Gentoo")
```



Comparison Operators:

- `==` equal to
- `!=` not equal to
- `>`, `>=` greater than, greater than or equal
- `<`, `<=` less than, less than or equal
- `&` AND (both conditions must be true)
- `|` OR (either condition can be true)

Tip: Use `!=` to filter out specific values

Mutate Columns

Creating or Transforming Variables

```
library(dplyr)

# Create new column
penguins_with_ratio <- penguins |>
  mutate(bill_ratio = bill_len / bill_dep)

# Create multiple columns
penguins_enhanced <- penguins |>
  mutate(
    bill_ratio = bill_len / bill_dep,
    body_mass_kg = body_mass / 1000,
    size_category = if_else(body_mass > 4000,
                            "large", "small"))
```

Common Transformations:

- Arithmetic operations (`+`, `-`, `*`, `/`, `^`)
- Mathematical functions (**log()**, **sqrt()**, **abs()**)
- String operations (**str_length()**, **str_to_upper()**)
- Conditional logic (**if_else()**, **case_when()**)

Tip: **mutate()** can also modify existing columns by using the same column name

Summarize Data

Computing Summary Statistics

```
library(dplyr)

# Calculate overall summary statistics
penguin_summary <- penguins |>
  summarize(
    mean_body_mass = mean(body_mass, na.rm = TRUE),
    median_body_mass = median(body_mass, na.rm = TRUE),
    sd_body_mass = sd(body_mass, na.rm = TRUE),
    count = n(),
    n_species = n_distinct(species)
  )
```

Common Summary Functions:

- `mean()`, `median()`: Measures of center
- `min()`, `max()`, `range()`: Extremes
- `sd()`, `var()`, `IQR()`: Measures of spread
- `n()`: Count observations
- `n_distinct()`: Count unique values

Tip: Handle missing data with **na.rm = TRUE**

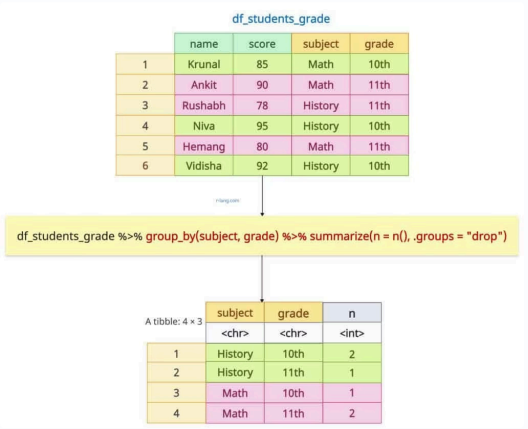
Group By and Summarize

Computing Statistics by Group

```
library(dplyr)

# Summary statistics by species
species_summary <- penguins |>
  group_by(species) |>
  summarize(
    count = n(),
    mean_body_mass = mean(body_mass, na.rm = TRUE),
    sd_body_mass = sd(body_mass, na.rm = TRUE))

# Multiple grouping variables
island_species_summary <- penguins |>
  group_by(island, species) |>
  summarize(count = n(), .groups = "drop")
```



Key Points:

- `group_by()` doesn't change the data; it changes how subsequent functions operate
- Use multiple variables in `group_by()` for nested grouping
- `ungroup()` removes grouping structure
- The `.groups` argument controls the resulting grouping structure

Tip: `ungroup()` data after grouping if further processing is needed

Arrange Data

Sorting Rows Based on Values

```
library(dplyr)

# Sort by body mass (ascending)
penguins_by_mass <- penguins |>
  arrange(body_mass)

# Sort by body mass (descending)
penguins_heaviest_first <- penguins |>
  arrange(desc(body_mass))

# Sort by multiple columns
penguins_sorted <- penguins |>
  arrange(species, desc(body_mass))
```

Key Points:

- `arrange()` sorts in ascending order by default
- Use `desc()` to sort in descending order
- With multiple columns, arranges by first column, then second, etc.
- Missing values (NA) are placed at the end

Tip: Sort in ascending order by default; use `desc()` for descending

Select Columns

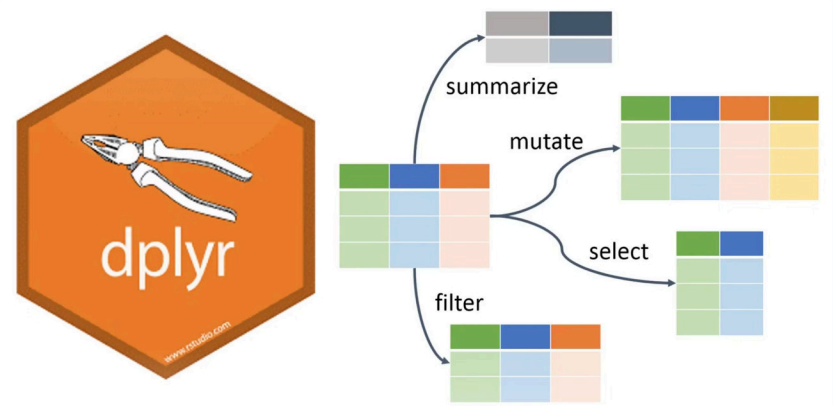
Choosing Specific Variables

```
library(dplyr)

# Select specific columns
penguin_dims <- penguins |>
  select(species, bill_len, bill_dep, body_mass)

# Remove column
penguins_no_year <- penguins |>
  select(-year)

# Select using helper functions
measurement_cols <- penguins |>
  select(species, ends_with("_len"))
```



Selection Helper Functions:

- `starts_with("str")`: Columns that start with a string
- `ends_with("str")`: Columns that end with a string
- `contains("str")`: Columns containing a string
- `matches("regex")`: Columns matching a regular expression
- `everything()`: All remaining columns

Tip: Use helpers like `starts_with()` to select columns by pattern

Pipe Operator (|>)

Chaining Operations Together

```
# Without pipe operator
filtered_data <- filter(penguins,
                        species == "Adelie")
selected_data <- select(filtered_data,
                        bill_len, body_mass)
result <- arrange(selected_data,
                  desc(body_mass))

# With pipe operator
result <- penguins |>
  filter(species == "Adelie") |>
  select(bill_len, body_mass) |>
  arrange(desc(body_mass))
```

Benefits of the Pipe:

- More readable code that flows from left to right
- Avoids nested function calls and temporary variables
- Follows the natural order of operations
- Easier to add, remove, or reorder steps

Tip: Think of `|>` as "then" to improve readability

Activities & Exercises

We will demonstrate and you'll practice:

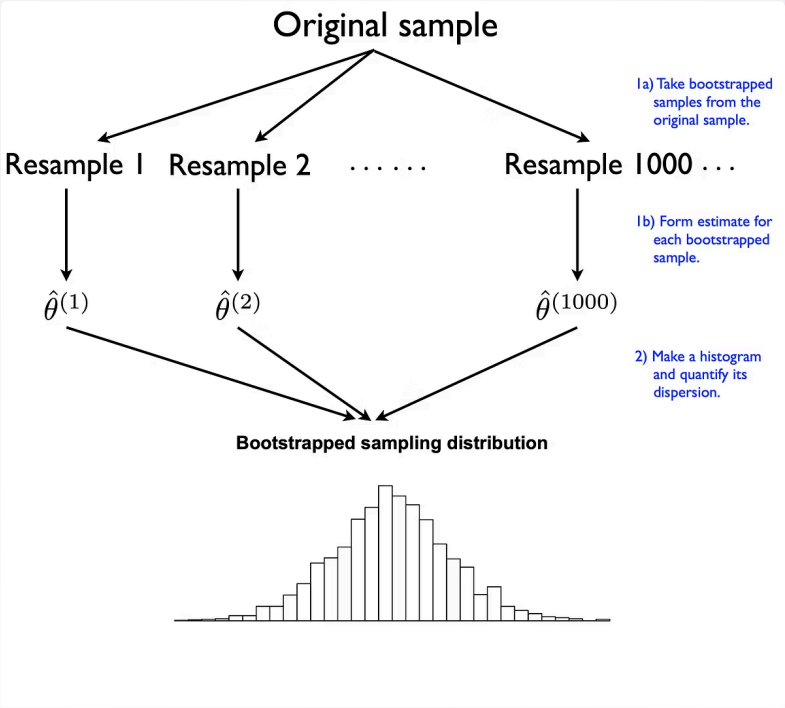
- Filtering data to subset observations
- Creating new variables with `mutate()`
- Grouping and summarizing data
- Chaining operations with the pipe operator
- Combining multiple `dplyr` functions in a workflow

Q & A

Let's address your questions about data wrangling:

- dplyr function syntax and usage
- Working with missing data
- Combining multiple operations
- Common data transformation patterns
- Grouping and summarizing strategies
- Joining and combining datasets
- Optimizing data wrangling workflows
- Troubleshooting data manipulation issues

Now is the perfect time to clarify any data wrangling concepts before moving to statistical inference!



Part 3: Statistical Inference

In this final section, we'll explore how the tools we've learned can be applied to statistical inference.

Population Data

Understanding Population Parameters

```
library(tidyverse)
set.seed(2025)  # For reproducibility

# Create a population
population <- tibble(value = rnorm(10000, mean = 50, sd = 10))

# Calculate population parameters
population_stats <- population |>
  summarize(mean = mean(value), median = median(value),
            sd = sd(value))

# Visualize the population
ggplot(population, aes(x = value)) +
  geom_histogram(bins = 30, fill = "white", color = "black") +
  geom_vline(xintercept = population_stats$mean,
             color = "red", linetype = "dashed", linewidth = 2)
```

Key Population Parameters:

- **Population Mean (μ):** The average of all values
- **Population Standard Deviation (σ):** Measure of spread
- **Population Distribution:** The complete distribution of all values

In real research, we rarely have access to the entire population and must work with samples instead. This is presented to help you see how simulation-based techniques can approximate sampling distribution characteristics.

Sampling

Taking Samples from a Population

```
library(tidyverse); library(moderndive)

# Take samples
samples <- population |>
  rep_slice_sample(n = 100, reps = 1000)

# Calculate sample means
sample_means <- samples |>
  summarize(sample_mean = mean(value))

# Calculate stats about sample means
sample_means_stats <- sample_means |>
 summarize(mean_of_means = mean(sample_mean),
           sd_of_means = sd(sample_mean))

# Calculate theoretical standard error
theoretical_se <- population_stats$sd / sqrt(100)
```

Central Limit Theorem:

- The sampling distribution of the mean approaches a normal distribution as sample size increases
- The mean of sample means equals the population mean
- The standard deviation of sample means (standard error) equals $\sigma/\sqrt{n}$

This is the foundation for inferential statistics and allows us to make inferences about populations from samples.

Theory-Based Confidence Interval

Estimating Population Parameters

```
library(tidyverse)

# Take one sample of size 100
one_sample <- slice_sample(population,
                           n = 100)

# Calculate sample statistics
sample_stats <- one_sample |>
  summarize(sample_mean = mean(value),
    sample_sd = sd(value),
    sample_size = n(),
    se = sample_sd / sqrt(100),
    margin_of_error = qt(0.975, df = 100 - 1) * se,
    ci_lower = sample_mean - margin_of_error,
    ci_upper = sample_mean + margin_of_error)
```

```
# Interpretation
sprintf("We are 95%% confident that the population ",
        "mean is between %.2f and %.2f.",
        sample_stats$ci_lower, sample_stats$ci_upper)
```

Components of a Confidence Interval:

- **Point Estimate:** The sample mean ($\bar{x}$)
- **Standard Error:** $s/\sqrt{n}$
- **Critical Value:** t* or z* (depends on sample size)
- **Margin of Error:** Critical value × Standard error
- **Confidence Interval:** Point estimate ± Margin of error

Interpretation: If we took many samples and calculated the 95% CI for each, about 95% of those intervals would contain the true population mean.

Simulation-Based Confidence Interval

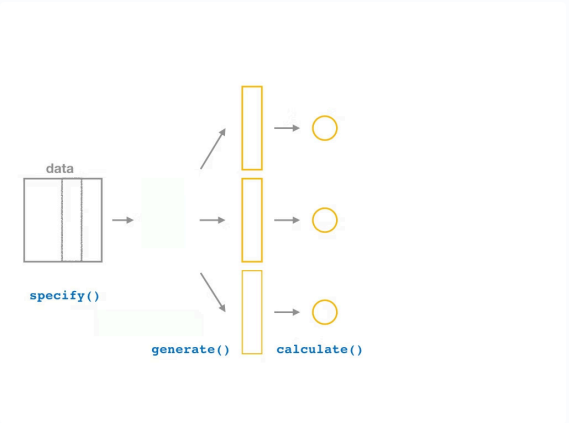
The infer Framework

```
library(infer)

# Step 1: Specify the variable of interest
bootstrap_dist <- one_sample |>
  specify(response = value)

# Step 2: Generate bootstrap samples
bootstrap_dist <- bootstrap_dist |>
  generate(reps = 1000, type = "bootstrap")

# Step 3: Calculate the mean of each bootstrap sample
bootstrap_dist <- bootstrap_dist |>
  calculate(stat = "mean")
```



Bootstrapping Process:

- **Start with one sample** from the population
- **Resample with replacement** to create many bootstrap samples of the same size
- **Calculate the statistic** (e.g., mean) for each bootstrap sample
- **Use the distribution** of bootstrap statistics to estimate uncertainty

Bootstrapping simulates the sampling process without requiring theoretical assumptions about distributions.

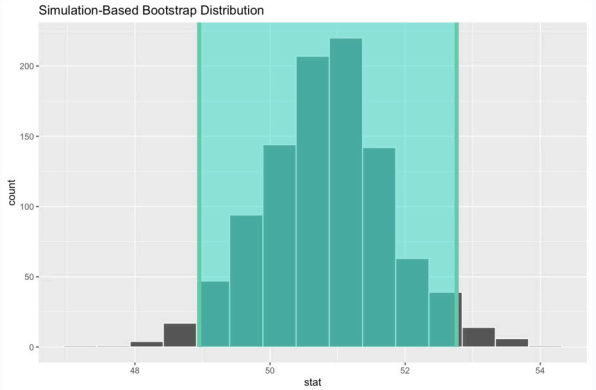
Simulation-Based Confidence Interval

Visualizing and Calculating the Bootstrap CI

```
# Calculate percentile confidence interval
percentile_ci <- bootstrap_dist |>
  get_confidence_interval(
    level = 0.95,
    type = "percentile")

# Calculate standard error confidence interval
se_ci <- bootstrap_dist |>
  get_confidence_interval(
    point_estimate = mean(one_sample$value),
    level = 0.95,
    type = "se")

# Visualize the bootstrap distribution with
# percentile confidence interval
visualize(bootstrap_dist) +
    shade_ci(endpoints = percentile_ci)
```



Types of Bootstrap CIs:

- **Percentile Method:** Uses quantiles of the bootstrap distribution (e.g., 2.5th and 97.5th percentiles for 95% CI)
- **Standard Error Method:** Uses the standard deviation of bootstrap statistics as an estimate of the standard error
- **Bias-Corrected Method:** Adjusts for potential bias in the bootstrap distribution

Interpretation: Similar to theory-based CIs, but based on simulated data rather than theoretical distributions.

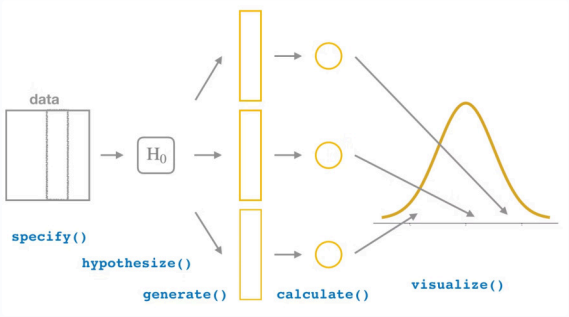
Simulation-Based Hypothesis Testing

Conducting a Bootstrap-Based Hypothesis Test

```
library(infer)

null_dist <- one_sample |>
  # Step 1: Specify the variable of interest
  specify(response = value)
  # Step 2: State the hypotheses
  hypothesize(null = "point", mu = 75)
  # Step 3: Generate bootstrap samples shifted
  # assuming the null is true
  generate(reps = 1000, type = "bootstrap")
  # Step 4: Calculate the mean of each
  # bootstrap sample
  calculate(stat = "mean")

null_dist |>
  get_p_value(obs_stat = mean(one_sample$value),
              direction = "two-sided")
```



Bootstrapping Process:

- **Start with one sample** from the population
- **Resample with replacement** to create many bootstrap samples of the same size, shifted to account for the null hypothesis
- **Calculate the statistic** (e.g., mean) for each bootstrap sample
- **Use the distribution** of bootstrap statistics to estimate uncertainty

Bootstrapping simulates the sampling process without requiring theoretical assumptions about distributions.

Activities & Exercises

We will demonstrate and you'll practice:

- Creating a population and calculating parameters
- Taking samples and exploring the sampling distribution
- Calculating theory-based confidence intervals
- Using the `infer` package for simulation-based inference (bootstrapping)
- Comparing different confidence interval methods
- Conducting hypothesis tests using simulation-based inferenceC

Q & A

Final Questions and Discussion

Thank you for participating in our workshop!

- For additional resources, visit **ModernDive**.
- Check out the solutions to the exercises you worked on in this workshop:

https://bit.ly/md-uscots-answers

Feedback form: **burl.live/workshop**