R tutorial: Beginner to Expert

Santhose

2025-12-18

Table of contents

About 6

Welcome to R for SAS Programmers 7
- Course Overview . . . 7
- Learning Objectives . . . 7
- Course Structure . . . 7
  - Module 1: R Fundamentals for SAS Users . . . 7
  - Module 2: Data Import and Export . . . 8
  - Module 3: Data Manipulation - dplyr . . . 8
  - Module 4: Creation of ADSL dataset . . . 8
  - Module 5: Statistical Analysis . . . 8
  - Module 6: Package Develpment and testing . . . 9
  - Module 7: Data Visualization . . . 9
  - Module 8: Reporting and Documentation . . . 9
- SAS to R Translation Guide . . . 10
- Prerequisites . . . 10
- Course Format . . . 10
- Getting Started . . . 10
  - Required Software . . . 10

I datatype & structure 12

1 R Fundamentals for SAS Users 13

2 Introduction 14
- 2.1 Prerequisites . . . 14

3 1. R Syntax Compared to SAS Syntax 15
- 3.1 Basic Code Structure and Comments . . . 15
- 3.2 SAS . . . 15
- 3.3 R . . . 15
  - 3.3.1 Key Syntax Differences . . . 16
  - 3.3.2 Demonstrating Case Sensitivity . . . 16
- 3.4 Procedural vs Functional Programming Paradigm . . . 17
  - 3.4.1 SAS: Step-by-Step Procedures . . . 18

3.4.2 R: Multiple Functional Approaches 18
3.4.3 Understanding the Pipe Operator (%>% and |>) 19
3.5 Execution and Evaluation Models 20
3.5.1 Sequential vs Expression-Based Execution 20
3.5.2 Lazy Evaluation in R 21
3.5.3 Vectorized Operations 21
3.5.4 Recycling Rules 22

4 2. Variables and Assignment Operators 24
4.1 Creating and Assigning Variables 24
4.1.1 Three Assignment Operators 24
4.1.2 Chaining Assignments 25
4.1.3 Variable Naming Rules and Conventions 25
4.1.4 Checking and Removing Variables 26
4.2 Variable Scope and Environments 27
4.2.1 Local vs Global Scope 27
4.2.2 Modifying Global Variables from Functions 28
4.2.3 Understanding «- Super Assignment 29
4.2.4 Function Factories and Closures 30
4.2.5 Working with Environments Explicitly 31
4.3 Copy-on-Modify Behavior 32
4.3.1 Default Copy-on-Modify 33
4.3.2 Implications for Function Arguments 33
4.3.3 Using Environments for True Reference Behavior 34

5 3. Data Types and Structures 36
5.1 Atomic Data Types 36
5.1.1 Numeric Types 36
5.1.2 Character (String) Types 37
5.1.3 Logical (Boolean) Types 39
5.1.4 Special Values 41
5.1.5 SAS to R Type Comparison 43
5.2 Vector Data Structures 44
5.2.1 Creating Vectors 44
5.2.2 Vector Properties and Operations 45
5.2.3 Named Vectors 46
5.2.4 Vector Indexing and Subsetting 48
5.2.5 Vector Type Coercion 49
5.3 Lists - Flexible Containers 50
5.3.1 Creating Lists 51
5.3.2 Accessing List Elements 52
5.3.3 Modifying Lists 53
5.3.4 Practical List Applications 55

5.4 Data Frames - R's Version of SAS Datasets . . . 56
5.4.1 Creating Data Frames . . . 56
5.4.2 Data Frame Properties . . . 57
5.4.3 Accessing Data Frame Elements . . . 59
5.4.4 Modifying Data Frames . . . 62
5.4.5 SAS DATA Step vs R Data Frame Operations . . . 63
5.5 SAS . . . 63
5.6 R (Base) . . . 63
5.7 R (tidyverse) . . . 63
5.8 Matrices and Arrays . . . 64
5.8.1 Creating and Using Matrices . . . 64
5.8.2 Matrix Operations . . . 66
5.8.3 Arrays (Multi-dimensional) . . . 68
5.9 Tibbles - Modern Data Frames . . . 69
5.9.1 Creating Tibbles . . . 69
5.9.2 Tibble vs Data Frame Differences . . . 70
5.9.3 Converting Between Data Frames and Tibbles . . . 72
5.10 Data. table - High Performance Option . . . 72
5.10.1 Creating Data.tables . . . 72
5.10.2 Data.table Syntax: DT[i, j, by] . . . 73
5.10.3 Data.table Performance Advantages . . . 76
5.11 Factors - Categorical Variables . . . 77
5.11.1 Creating Factors . . . 77
5.11.2 Working with Factors . . . 78
5.11.3 Practical Uses for Factors . . . 79
5.11.4 Factor Pitfalls and Solutions . . . 80

6 4. Functions and Help System 82
6.1 Using Built-in Functions . . . 82
6.1.1 Statistical Functions . . . 82
6.1.2 Mathematical Functions . . . 84
6.1.3 String Functions . . . 87
6.1.4 Type Checking and Conversion Functions . . . 88
6.2 SAS PROC to R Function Mapping . . . 91
6.2.1 Practical Examples: SAS to R . . . 93
6.3 PROC MEANS . . . 93
6.4 PROC FREQ . . . 94
6.5 PROC SORT . . . 95
6.6 Getting Help in R . . . 96
6.6.1 Basic Help Commands . . . 96
6.6.2 Exploring Packages and Functions . . . 97
6.6.3 Online Resources and Cheat Sheets . . . 97

6.7 Writing Custom Functions 98
6.7.1 Basic Function Definition 98
6.7.2 Function with Default Arguments 98
6.7.3 Function with Variable Number of Arguments 99

7 datatype and structure exercise **100**

8 Exercise 1 **101**

9 Exercise 2 **102**

II data manipulation **103**

10 Introduction **104**

11 Summary **105**

References **106**

About

Welcome to R for SAS Programmers

Course Overview

This comprehensive course is designed specifically for **SAS programmers** who want to learn R programming. We'll leverage your existing knowledge of data manipulation, statistical analysis, and programming concepts to help you become proficient in R.

Learning Objectives

By the end of this course, you will be able to:

- **Understand R fundamentals**: Master R syntax, data types, and programming concepts
- **Data manipulation**: Perform complex data transformations using `dplyr` and `base R`
- **Statistical analysis**: Apply statistical methods and create models in R
- **Data visualization**: Create compelling visualizations using `ggplot2`
- **Reporting**: Generate dynamic reports with R Markdown and Quarto
- **Bridge knowledge**: Map your SAS skills to equivalent R functions and workflows
- **Best practices**: Write clean, efficient, and reproducible R code

Course Structure

Module 1: R Fundamentals for SAS Users

- **Getting Started**
  - R and RStudio installation and setup
  - Understanding the R environment vs SAS environment
  - Package management
- **Basic R Syntax**
  - R syntax compared to SAS syntax
  - Variables and assignment operators
  - Data types and structures
  - Functions and help system

Module 2: Data Import and Export

- CSV, Excel and text files (`readr`, `readxl` packages)
- SAS dataset import (`haven` package) and export xpt files
- other formats (JSON, XML)

Module 3: Data Manipulation - dplyr

- **Core dplyr Functions**
  - `select()` vs `KEEP/DROP` statements
  - `filter()` vs `WHERE` clause
  - `mutate()` vs `assignment` statements
  - `summarise()` vs `PROC MEANS`
  - `group_by()` vs `BY` statement
- **Advanced Data Manipulation**
  - Joins equivalent to `PROC SQL` joins
  - Reshaping data (`tidyr` vs `PROC TRANSPOSE`)
  - String manipulation (`stringr` vs SAS string functions)

Module 4: Creation of ADSL dataset

- **ADSL and ADVS Dataset Creation**
  - Creating analysis variables
  - Handling missing data and derivations
  - creating ADSL xpt dataset

Module 5: Statistical Analysis

- **Descriptive Statistics**
  - Summary statistics (equivalent to `PROC UNIVARIATE`)
  - Frequency tables (equivalent to `PROC FREQ`)
  - Cross-tabulations and chi-square tests
  - creation of tables like DM and AE outputs
- **Statistical Modeling basics**
  - Linear regression (equivalent to `PROC REG`)
  - Logistic regression (equivalent to `PROC LOGISTIC`)

  - ANOVA (equivalent to `PROC ANOVA`)
  - Mixed models and advanced techniques

Module 6: Package Develpment and testing

- **Creating R Packages**
  - Package structure and essential files
  - Documenting functions with `roxygen2`
  - Building and installing packages
- **Testing with testthat**
  - Writing unit tests for R functions
  - Running tests and interpreting results

Module 7: Data Visualization

- **Base R Graphics**
  - Basic plots and customization
  - Comparison with SAS/GRAPH
- **ggplot2 - Grammar of Graphics**
  - Understanding the layered approach
  - Creating publication-ready plots
  - Advanced visualization techniques

Module 8: Reporting and Documentation

- **R Markdown and Quarto**
  - Creating dynamic reports (equivalent to ODS output)
  - Integrating code, results, and narrative
  - Output formats: HTML, PDF, Word
- **Reproducible Research**
  - Project organization
  - Version control with Git
  - Best practices for code documentation

SAS to R Translation Guide

SAS Concept	R Equivalent	Package
DATA step	`dplyr::mutate()`	dplyr
PROC SQL	`dplyr` verbs	dplyr
PROC MEANS	`dplyr::summarise()`	dplyr
PROC FREQ	`table()`, `xtabs()`	base R
PROC REG	`lm()`	base R
PROC LOGISTIC	`glm()`	base R
PROC TRANSPOSE	`tidyr::pivot_*()`	tidyr
ODS OUTPUT	R Markdown/Quarto	rmarkdown/quarto

Prerequisites

- **SAS Experience**: Familiarity with SAS programming, data steps, and procedures
- **Statistical Knowledge**: Basic understanding of statistical concepts
- **Programming Basics**: Understanding of programming logic and data structures

Course Format

- **Interactive Learning**: Hands-on exercises with real datasets
- **Comparative Examples**: Side-by-side SAS and R code comparisons
- **Practical Projects**: Real-world scenarios mimicking typical SAS workflows
- **Reference Materials**: Quick reference guides and cheat sheets

Getting Started

Required Software

1. **R** (version 4.3+): Download from CRAN
2. **RStudio**: Download from Posit
3. **Essential Packages**: We'll install these as needed

```
install.packages(c("tidyverse", "haven", "readxl", "rmarkdown"))
```

Tip: If you don't have sample SDTM/ADaM data yet, the chapters generate **small synthetic data** as a fallback so everything runs end-to-end. ## contact For questions or feedback, reach out to **r2sas2025@gmail.com**

Part I

datatype & structure

1 R Fundamentals for SAS Users

Basic R Syntax - A Comprehensive Guide

2 Introduction

This guide is designed for SAS users transitioning to R. We'll cover fundamental R concepts by comparing them to familiar SAS constructs, providing comprehensive coverage of each topic.

2.1 Prerequisites

---

3 1. R Syntax Compared to SAS Syntax

3.1 Basic Code Structure and Comments

Understanding how R code is structured compared to SAS is fundamental to making the transition.

:: {.panel-tabset}

3.2 SAS

```
/* Multi-line comments in SAS
   can span multiple lines */
* Single line comment with asterisk;

/* Statements must end with semicolons */
DATA mydata;
    SET olddata;
    new_var = old_var * 2;
RUN;

/* Procedures need RUN statements */
PROC PRINT DATA=mydata;
RUN;

/* Case insensitive - these are all the same */
data test;
DATA test;
Data test;
```

3.3 R

```r
# Single line comments use hash symbol
# There are no multi-line comments in base R
# (though RStudio supports Ctrl+Shift+C for multiple lines)

# Semicolons are optional but allowed
new_var <- old_var * 2  # No semicolon needed
new_var <- old_var * 2; # Semicolon allowed

# No RUN statements required - code executes immediately
print(mydata)

# R is CASE SENSITIVE - these are different variables
data <- 1
Data <- 2
DATA <- 3
```

:::

3.3.1 Key Syntax Differences

Aspect	SAS	R	Notes
Comment	`/* */` or `*;`	`#`	R comments run to end of line only
Statement terminator	`;` required	`;` optional	Most R code omits semicolons
Case sensitivity	Not case-sensitive	**Case-sensitive**	Major difference!
Assignment	`=`	`<-` or `=`	`<-` is preferred in R
Block execution	`RUN;` required	Automatic	Code executes line by line
Line continuation	Automatic	Automatic with open brackets	

3.3.2 Demonstrating Case Sensitivity

```r
# In R, these are THREE DIFFERENT variables
Variable <- 10
variable <- 20
```

```r
VARIABLE <- 30

print(Variable)  # Returns 10
```

```
[1] 10
```

```r
print(variable)  # Returns 20
```

```
[1] 20
```

```r
print(VARIABLE)  # Returns 30
```

```
[1] 30
```

```r
# This would cause confusion in SAS but works in R
mydata <- data.frame(x = 1:5)
MyData <- data.frame(x = 6:10)
MYDATA <- data.frame(x = 11:15)

# Each is a separate object
nrow(mydata)  # 5
```

```
[1] 5
```

```r
nrow(MyData)  # 5
```

```
[1] 5
```

```r
nrow(MYDATA)  # 5
```

```
[1] 5
```

3.4 Procedural vs Functional Programming Paradigm

SAS and R have fundamentally different approaches to data manipulation and analysis.

3.4.1 SAS: Step-by-Step Procedures

```
/* SAS uses distinct procedures with explicit steps */

/* Step 1: Sort data */
PROC SORT DATA=mydata OUT=sorted_data;
    BY age;
RUN;

/* Step 2: Calculate statistics */
PROC MEANS DATA=sorted_data MEAN STD;
    VAR income;
    CLASS gender;
    OUTPUT OUT=summary_stats MEAN=avg_income STD=sd_income;
RUN;

/* Step 3: Print results */
PROC PRINT DATA=summary_stats;
RUN;
```

3.4.2 R: Multiple Functional Approaches

```
# Approach 1: Base R - function chaining
sorted_data <- mydata[order(mydata$age), ]
summary_stats <- aggregate(income ~ gender, data = sorted_data,
                          FUN = function(x) c(mean = mean(x), sd = sd(x)))
print(summary_stats)

# Approach 2: tidyverse - pipe operator (most similar to thinking in steps)
summary_stats <- mydata %>%
    arrange(age) %>%
    group_by(gender) %>%
    summarise(
        avg_income = mean(income, na.rm = TRUE),
        sd_income = sd(income, na.rm = TRUE)
    )

print(summary_stats)

# Approach 3: data.table - high performance
```

```r
library(data.table)
dt <- as.data.table(mydata)
summary_stats <- dt[order(age)][, .(avg_income = mean(income),
                                     sd_income = sd(income)),
                                 by = gender]
print(summary_stats)
```

3.4.3 Understanding the Pipe Operator (%>% and |>)

The pipe operator makes R code read more like SAS procedures:

```r
# Create sample data
employees <- data.frame(
    name = c("John", "Jane", "Bob", "Alice", "Charlie", "Diana"),
    department = c("Sales", "IT", "IT", "Sales", "HR", "Sales"),
    salary = c(50000, 75000, 68000, 52000, 48000, 55000),
    years = c(2, 5, 3, 1, 4, 3)
)

# Without pipes (nested functions - hard to read)
result1 <- head(arrange(filter(employees, department == "Sales"), desc(salary)), 3)

# With pipes (reads left to right, top to bottom)
result2 <- employees %>%
    filter(department == "Sales") %>%
    arrange(desc(salary)) %>%
    head(3)

print(result2)
```

```
   name department salary years
1 Diana      Sales  55000     3
2 Alice      Sales  52000     1
3  John      Sales  50000     2
```

```r
# Native pipe |> (R 4.1+) - similar but slightly different
result3 <- employees |>
    filter(department == "Sales") |>
    arrange(desc(salary)) |>
    head(3)
```

```
print(result3)
```

```
   name department salary years
1 Diana      Sales  55000     3
2 Alice      Sales  52000     1
3  John      Sales  50000     2
```

3.5 Execution and Evaluation Models

Understanding how R evaluates code differently from SAS helps avoid common pitfalls.

3.5.1 Sequential vs Expression-Based Execution

SAS: Executes in clearly defined DATA and PROC steps with explicit boundaries

```
/* SAS processes entire DATA step before moving to next step */
DATA step1;
    SET input_data;
    x = 10;
RUN;  /* Everything above completes before proceeding */

DATA step2;
    SET step1;
    y = x + 5;  /* x is available because step1 completed */
RUN;
```

R: Expression-based evaluation, code executes immediately

```
# R executes line by line
x <- 10            # Executes immediately
y <- x + 5         # Can use x immediately
z <- x + y         # Can use both x and y

print(paste("x:", x, "y:", y, "z:", z))
```

```
[1] "x: 10 y: 15 z: 25"
```

```r
# Functions execute when called
my_calculation <- function() {
    a <- 100
    b <- 200
    return(a + b)
}

# Function is defined but not executed yet
result <- my_calculation()  # Now it executes
print(result)
```

```
[1] 300
```

3.5.2 Lazy Evaluation in R

R uses "lazy evaluation" - function arguments are only evaluated when actually used:

```r
# Demonstrating lazy evaluation
my_function <- function(x, y, z) {
    # If we return early, unused arguments never get evaluated
    if (x > 0) {
        return(x * 2)
    }
    # y and z are never evaluated if x > 0
    return(y + z)
}

# This works even though second and third arguments would error
result <- my_function(5, stop("Error in y!"), stop("Error in z!"))
print(result)  # Returns 10, no errors
```

```
[1] 10
```

```r
# But this would cause an error
# result <- my_function(-1, stop("Error in y!"), 10)  # ERROR!
```

3.5.3 Vectorized Operations

R operates on entire vectors at once (unlike SAS's row-by-row processing):

```r
# Create vectors
vector1 <- c(1, 2, 3, 4, 5)
vector2 <- c(10, 20, 30, 40, 50)

# Vectorized operation - all elements at once
result <- vector1 + vector2
print(result)  # 11 22 33 44 55
```

```
[1] 11 22 33 44 55
```

```r
# Comparison to SAS approach
# In SAS, you'd typically process row by row:
# DATA result;
#     SET input;
#     new_value = value1 + value2;
# RUN;

# In R, you can also do row-wise operations on data frames
df <- data.frame(value1 = vector1, value2 = vector2)
df$result <- df$value1 + df$value2
print(df)
```

```
  value1 value2 result
1      1     10     11
2      2     20     22
3      3     30     33
4      4     40     44
5      5     50     55
```

3.5.4 Recycling Rules

R automatically "recycles" shorter vectors to match longer ones:

```r
# Vector recycling
short_vec <- c(1, 2)
long_vec <- c(10, 20, 30, 40, 50, 60)

# short_vec is recycled:  1, 2, 1, 2, 1, 2
result <- short_vec + long_vec
print(result)  # 11 22 31 42 51 62
```

```
[1] 11 22 31 42 51 62
```

```r
# Warning when lengths don't divide evenly
vec1 <- c(1, 2, 3)
vec2 <- c(10, 20, 30, 40, 50)
# result <- vec1 + vec2  # Would give warning

# Practical use:  add constant to all elements
values <- c(100, 200, 300, 400)
values_plus_10 <- values + 10  # 10 is recycled
print(values_plus_10)
```

```
[1] 110 210 310 410
```

---

4 2. Variables and Assignment Operators

4.1 Creating and Assigning Variables

Understanding variable assignment is crucial for writing effective R code.

4.1.1 Three Assignment Operators

```r
# Left assignment with <- (RECOMMENDED)
x <- 10
patient_age <- 45
department_name <- "Cardiology"

# Left assignment with = (works but not preferred for variables)
y = 20
# Use = for function arguments:  mean(x, na.rm = TRUE)

# Right assignment with -> (rarely used, but valid)
30 -> z

# All three created variables
print(paste("x =", x, "| y =", y, "| z =", z))

# Why <- is preferred:
# 1. Clearly distinguishes assignment from function arguments
# 2. Consistent with R conventions and style guides
# 3. Can be read as "gets" or "assign to"

# Example of clarity with <-
my_data <- read.csv("file.csv", header = TRUE)  # Clear distinction
# vs
# my_data = read.csv("file.csv", header = TRUE)  # Less clear
```

4.1.2 Chaining Assignments

```r
# Multiple assignments in one line
a <- b <- c <- 100
print(paste("a =", a, "| b =", b, "| c =", c))

# Assignment with computation
result <- (x <- 5) + (y <- 10)
print(paste("x =", x, "| y =", y, "| result =", result))

# Practical example: assign and use
data_subset <- subset(employees, salary > (threshold <- 50000))
print(paste("Threshold used:", threshold))
print(data_subset)
```

4.1.3 Variable Naming Rules and Conventions

```r
# VALID variable names
valid_name <- 1
valid.name <- 2          # Dots allowed (unlike most languages)
validName <- 3           # camelCase
valid_name_123 <- 4      # Numbers allowed (but not at start)
.hidden_var <- 5         # Starting with dot (hidden from ls())

# Common naming conventions
snake_case_variable <- "preferred in R"
camelCaseVariable <- "common in some R code"
dot.separated.name <- "traditional R style"
PascalCase <- "typically for functions/classes"

# INVALID variable names (will cause errors)
# 123invalid <- 5         # ERROR:  Cannot start with number
# invalid-name <- 6       # ERROR:  Hyphens not allowed (minus sign)
# _invalid <- 7           # ERROR: Cannot start with underscore
# my variable <- 8        # ERROR:  Spaces not allowed

# Reserved words cannot be used as variable names
reserved_words <- c("if", "else", "repeat", "while", "function",
                    "for", "in", "next", "break", "TRUE", "FALSE",
```

```
                    "NULL", "Inf", "NaN", "NA")
print(reserved_words)
```

```
 [1] "if"       "else"     "repeat"   "while"    "function" "for"
 [7] "in"       "next"     "break"    "TRUE"     "FALSE"    "NULL"
[13] "Inf"      "NaN"      "NA"
```

```
# if <- 10  # ERROR: 'if' is reserved
# BUT you can use them with backticks (not recommended)
`if` <- 10
print(`if`)
```

```
[1] 10
```

4.1.4 Checking and Removing Variables

```
# Create some variables
var1 <- 100
var2 <- 200
var3 <- 300

# List all variables in environment
current_vars <- ls()
print(current_vars)
```

```
 [1] "camelCaseVariable"   "df"                  "dot.separated.name"
 [4] "employees"           "if"                  "long_vec"
 [7] "my_calculation"      "my_function"         "mydata"
[10] "MyData"              "MYDATA"              "PascalCase"
[13] "required_packages"   "reserved_words"      "result"
[16] "result1"             "result2"             "result3"
[19] "short_vec"           "snake_case_variable" "valid_name"
[22] "valid_name_123"      "valid.name"          "validName"
[25] "values"              "values_plus_10"      "var1"
[28] "var2"                "var3"                "variable"
[31] "Variable"            "VARIABLE"            "vec1"
[34] "vec2"                "vector1"             "vector2"
[37] "x"                   "y"                   "z"
```

```r
# Check if variable exists
exists("var1")  # TRUE
```

```
[1] TRUE
```

```r
exists("var999")  # FALSE
```

```
[1] FALSE
```

```r
# Remove specific variables
rm(var3)
exists("var3")  # FALSE
```

```
[1] FALSE
```

```r
# Remove multiple variables
rm(var1, var2)

# Remove all variables (use with caution!)
# rm(list = ls())
```

4.2 Variable Scope and Environments

Understanding scope is crucial for writing functions and avoiding bugs.

4.2.1 Local vs Global Scope

```r
# Global variable (available everywhere)
global_var <- 100

# Function with local variables
my_function <- function() {
    # Local variable (only exists inside function)
    local_var <- 200

    # Can read global variables
    print(paste("Inside function, global_var:", global_var))
```

```r
    print(paste("Inside function, local_var:", local_var))

    # Return something
    return(local_var * 2)
}

# Call function
result <- my_function()
```

```
[1] "Inside function, global_var: 100"
[1] "Inside function, local_var: 200"
```

```r
print(paste("Function returned:", result))
```

```
[1] "Function returned: 400"
```

```r
# local_var doesn't exist outside function
print(paste("Outside function, global_var:", global_var))
```

```
[1] "Outside function, global_var: 100"
```

```r
# print(local_var)  # ERROR: object 'local_var' not found
```

4.2.2 Modifying Global Variables from Functions

```r
# Global variable
counter <- 0

# Function that modifies local copy (default behavior)
increment_local <- function() {
    counter <- counter + 1  # Creates local 'counter'
    print(paste("Inside increment_local:", counter))
}

increment_local()
```

```
[1] "Inside increment_local: 1"
```

```r
print(paste("Global counter after increment_local:", counter))  # Still 0!
```

```
[1] "Global counter after increment_local: 0"
```

```r
# Function that modifies global variable (use <<- or assign)
increment_global <- function() {
    counter <<- counter + 1  # Modifies global 'counter'
    print(paste("Inside increment_global:", counter))
}

increment_global()
```

```
[1] "Inside increment_global: 1"
```

```r
print(paste("Global counter after increment_global:", counter))  # Now 1!
```

```
[1] "Global counter after increment_global: 1"
```

```r
increment_global()
```

```
[1] "Inside increment_global: 2"
```

```r
print(paste("Global counter after second call:", counter))  # Now 2!
```

```
[1] "Global counter after second call: 2"
```

4.2.3 Understanding «- Super Assignment

super assignment operator `<<-` allows modification of variables in parent environments.

```r
# <<- searches parent environments until it finds the variable
outer_function <- function() {
    x <- 10  # Local to outer_function

    inner_function <- function() {
        x <<- x + 5  # Modifies x in outer_function's environment
        print(paste("Inside inner_function, x:", x))
```

```r
    }

    print(paste("Before inner_function, x:", x))
    inner_function()
    print(paste("After inner_function, x:", x))
}

outer_function()
```

```
[1] "Before inner_function, x: 10"
[1] "Inside inner_function, x: 15"
[1] "After inner_function, x: 15"
```

```r
# Use cases for <<-:
# 1. Counters and state in function factories
# 2. Caching/memoization
# 3. Building interactive applications
# WARNING: Use sparingly, can make code hard to understand
```

4.2.4 Function Factories and Closures

```r
# Creating a function that returns a function
create_counter <- function() {
    count <- 0  # This variable persists across calls

    function() {
        count <<- count + 1
        return(count)
    }
}

# Create two independent counters
counter1 <- create_counter()
counter2 <- create_counter()

# Each maintains its own state
print(counter1())  # 1
```

```
[1] 1
```

```r
print(counter1())  # 2
```

```
[1] 2
```

```r
print(counter1())  # 3
```

```
[1] 3
```

```r
print(counter2())  # 1
```

```
[1] 1
```

```r
print(counter2())  # 2
```

```
[1] 2
```

```r
# Practical example: create functions with specific parameters
create_multiplier <- function(factor) {
    function(x) {
        return(x * factor)
    }
}

times_2 <- create_multiplier(2)
times_10 <- create_multiplier(10)

print(times_2(5))   # 10
```

```
[1] 10
```

```r
print(times_10(5))  # 50
```

```
[1] 50
```

4.2.5 Working with Environments Explicitly

the default environment is the global environment, but you can use the `new.env()` function to create and manipulate environments directly.

```r
# Create new environment
my_env <- new.env()

# Assign variables to environment
my_env$var1 <- 100
my_env$var2 <- 200

# Access environment variables
print(my_env$var1)
```

```
[1] 100
```

```r
# List variables in environment
ls(my_env)
```

```
[1] "var1" "var2"
```

```r
# Environments are passed by reference (not copied)
modify_env <- function(env) {
    env$var1 <- 999
    env$new_var <- 777
}

modify_env(my_env)
print(my_env$var1)     # 999 (modified!)
```

```
[1] 999
```

```r
print(my_env$new_var)  # 777 (added!)
```

```
[1] 777
```

```r
# This is different from regular objects which are copied
```

4.3 Copy-on-Modify Behavior

R's copy-on-modify system is important for understanding performance and memory usage.

4.3.1 Default Copy-on-Modify

```r
# Create original vector
original <- c(1, 2, 3, 4, 5)

# Assign to new variable (no copy yet!)
copy_var <- original

# Modify copy_var (NOW a copy is made)
copy_var[1] <- 999

# Original is unchanged
print(original)   # 1 2 3 4 5
```

```
[1] 1 2 3 4 5
```

```r
print(copy_var)   # 999 2 3 4 5
```

```
[1] 999   2   3   4   5
```

```r
# Same with data frames
original_df <- data.frame(x = 1:5, y = 6:10)
copy_df <- original_df
copy_df$x[1] <- 999

print(original_df$x)  # 1 2 3 4 5 (unchanged)
```

```
[1] 1 2 3 4 5
```

```r
print(copy_df$x)      # 999 2 3 4 5 (modified)
```

```
[1] 999   2   3   4   5
```

4.3.2 Implications for Function Arguments

```r
# Functions receive copies (so modifications don't affect original)
modify_vector <- function(vec) {
    vec[1] <- 999
    print(paste("Inside function:", paste(vec, collapse = " ")))
    return(vec)
}

my_vector <- c(1, 2, 3, 4, 5)
result <- modify_vector(my_vector)
```

```
[1] "Inside function: 999 2 3 4 5"
```

```r
print(paste("Original vector:", paste(my_vector, collapse = " ")))  # Unchanged
```

```
[1] "Original vector: 1 2 3 4 5"
```

```r
print(paste("Returned vector:", paste(result, collapse = " ")))     # Modified
```

```
[1] "Returned vector: 999 2 3 4 5"
```

4.3.3 Using Environments for True Reference Behavior

When you need pass-by-reference behavior (like SAS datasets that get modified):

```r
# Environment approach (pass by reference)
create_data_env <- function() {
    env <- new.env()
    env$data <- data.frame(id = 1:5, value = rnorm(5))
    return(env)
}

modify_data <- function(data_env, new_value) {
    data_env$data$value <- data_env$data$value + new_value
}

# Create data environment
my_data_env <- create_data_env()
print("Original:")
```

```
[1] "Original:"
```

```r
print(my_data_env$data)
```

```
  id      value
1  1  0.7563500
2  2 -1.1486746
3  3  0.5043772
4  4  1.1743032
5  5 -1.2969403
```

```r
# Modify (changes original!)
modify_data(my_data_env, 100)
print("After modification:")
```

```
[1] "After modification:"
```

```r
print(my_data_env$data)  # Modified!
```

```
  id     value
1  1 100.75635
2  2  98.85133
3  3 100.50438
4  4 101.17430
5  5  98.70306
```

```r
# This is similar to how SAS datasets work
```

---

5 3. Data Types and Structures

5.1 Atomic Data Types

R has six atomic (basic) types. Understanding these is fundamental to data manipulation.

5.1.1 Numeric Types

```
# Double (default numeric type in R)
num_double <- 42.5
class(num_double)
```

```
[1] "numeric"
```

```
typeof(num_double)
```

```
[1] "double"
```

```
# Integer (requires L suffix)
num_integer <- 42L
class(num_integer)
```

```
[1] "integer"
```

```
typeof(num_integer)
```

```
[1] "integer"
```

```
# Automatic conversion
x <- 10      # Double
y <- 10L     # Integer
z <- x + y   # Result is double
typeof(z)
```

```
[1] "double"
```

```r
# Checking numeric types
is.numeric(num_double)   # TRUE
```

```
[1] TRUE
```

```r
is.numeric(num_integer)  # TRUE
```

```
[1] TRUE
```

```r
is.integer(num_integer)  # TRUE
```

```
[1] TRUE
```

```r
is.integer(num_double)   # FALSE
```

```
[1] FALSE
```

```r
is.double(num_double)    # TRUE
```

```
[1] TRUE
```

```r
# Coercion
as.integer(42.7)  # 42 (truncates)
```

```
[1] 42
```

```r
as.numeric("123")  # 123
```

```
[1] 123
```

```r
as.numeric("abc")  # NA with warning
```

```
[1] NA
```

5.1.2 Character (String) Types

```r
# Creating character variables
char1 <- "Hello"
char2 <- 'World'  # Single or double quotes work
char3 <- "Can include 'quotes' inside"

# Multi-line strings
multi_line <- "This is line 1
This is line 2
This is line 3"

# String operations
paste("Hello", "World")                    # "Hello World"
```

```
[1] "Hello World"
```

```r
paste0("No", "Space")                      # "NoSpace"
```

```
[1] "NoSpace"
```

```r
paste("A", "B", "C", sep = "-")           # "A-B-C"
```

```
[1] "A-B-C"
```

```r
sprintf("Patient %d has BMI %.2f", 101, 24.5)  # Formatted string
```

```
[1] "Patient 101 has BMI 24.50"
```

```r
# String manipulation
toupper("hello")                           # "HELLO"
```

```
[1] "HELLO"
```

```r
tolower("HELLO")                           # "hello"
```

```
[1] "hello"
```

```r
nchar("Hello")                              # 5 (length)
```

```
[1] 5
```

```r
substr("Hello World", 1, 5)                 # "Hello"
```

```
[1] "Hello"
```

```r
gsub("o", "0", "Hello World")               # "Hell0 W0rld" (replace)
```

```
[1] "Hell0 W0rld"
```

```r
# Check and convert
is.character(char1)  # TRUE
```

```
[1] TRUE
```

```r
as.character(123)    # "123"
```

```
[1] "123"
```

5.1.3 Logical (Boolean) Types

```r
# Creating logical values
flag1 <- TRUE
flag2 <- FALSE
flag3 <- T       # Shorthand (but TRUE preferred)
flag4 <- F       # Shorthand (but FALSE preferred)

# Logical operations
TRUE & FALSE   # AND:  FALSE
```

```
[1] FALSE
```

```r
TRUE | FALSE   # OR:  TRUE
```

```
[1] TRUE
```

```r
!TRUE          # NOT:  FALSE
```

```
[1] FALSE
```

```r
TRUE && FALSE  # Short-circuit AND: FALSE (only evaluates what's needed)
```

```
[1] FALSE
```

```r
TRUE || FALSE  # Short-circuit OR: TRUE
```

```
[1] TRUE
```

```r
# Comparisons return logical
5 > 3          # TRUE
```

```
[1] TRUE
```

```r
5 == 5         # TRUE
```

```
[1] TRUE
```

```r
5 != 3         # TRUE
```

```
[1] TRUE
```

```r
"a" %in% c("a", "b", "c")  # TRUE
```

```
[1] TRUE
```

```r
# Logical arithmetic (TRUE = 1, FALSE = 0)
sum(c(TRUE, FALSE, TRUE, TRUE))     # 3
```

```
[1] 3
```

```r
mean(c(TRUE, FALSE, TRUE, TRUE))    # 0.75
```

```
[1] 0.75
```

```r
# Conversion
as.logical(0)        # FALSE
```

```
[1] FALSE
```

```r
as.logical(1)        # TRUE
```

```
[1] TRUE
```

```r
as.logical("TRUE")   # TRUE
```

```
[1] TRUE
```

```r
as.logical("yes")    # NA
```

```
[1] NA
```

5.1.4 Special Values

```r
# NA - Missing value (like .  in SAS)
na_value <- NA
is.na(na_value)   # TRUE
```

```
[1] TRUE
```

```r
# Different types of NA
na_real <- NA_real_        # Numeric NA
na_int <- NA_integer_      # Integer NA
na_char <- NA_character_   # Character NA
na_logical <- NA            # Logical NA

# NULL - Absence of value (different from NA)
null_value <- NULL
is.null(null_value)  # TRUE
```

```
[1] TRUE
```

```r
length(NULL)          # 0
```

```
[1] 0
```

```r
length(NA)            # 1
```

```
[1] 1
```

```r
# Inf - Infinity
inf_value <- Inf
1 / 0                 # Inf
```

```
[1] Inf
```

```r
-1 / 0                # -Inf
```

```
[1] -Inf
```

```r
is.infinite(Inf)      # TRUE
```

```
[1] TRUE
```

```r
is.finite(Inf)        # FALSE
```

```
[1] FALSE
```

```r
# NaN - Not a Number
nan_value <- NaN
0 / 0                 # NaN
```

```
[1] NaN
```

```r
is.nan(nan_value)     # TRUE
```

```
[1] TRUE
```

```r
# Testing for special values
test_values <- c(1, NA, NaN, Inf, -Inf)
is.na(test_values)       # TRUE for NA and NaN
```

```
[1] FALSE  TRUE  TRUE FALSE FALSE
```

```r
is.nan(test_values)      # TRUE only for NaN
```

```
[1] FALSE FALSE  TRUE FALSE FALSE
```

```r
is.infinite(test_values) # TRUE for Inf and -Inf
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE
```

```r
is.finite(test_values)   # TRUE only for 1
```

```
[1]  TRUE FALSE FALSE FALSE FALSE
```

5.1.5 SAS to R Type Comparison

```r
# Create comparison table
comparison <- data.frame(
    SAS_Type = c("Numeric", "Character", "Logical (1/0)", ".  (missing)", "Date (numeric)"),
    R_Type = c("numeric/integer", "character", "logical (TRUE/FALSE)", "NA", "Date class"),
    Example = c("42.5 or 42L", '"text"', "TRUE/FALSE", "NA", 'as.Date("2025-12-18")'),
    stringsAsFactors = FALSE
)

print(comparison)
```

```
        SAS_Type               R_Type               Example
1        Numeric      numeric/integer           42.5 or 42L
2      Character            character                "text"
3  Logical (1/0) logical (TRUE/FALSE)            TRUE/FALSE
4   .  (missing)                   NA                    NA
5 Date (numeric)           Date class as.Date("2025-12-18")
```

```r
# Date example
sas_date <- 19709  # SAS date value for 2025-12-18
r_date <- as.Date("2025-12-18")
print(r_date)
```

```
[1] "2025-12-18"
```

```r
print(class(r_date))
```

```
[1] "Date"
```

```r
# Convert between systems
r_date_from_sas <- as.Date(sas_date, origin = "1960-01-01")
print(r_date_from_sas)
```

```
[1] "2013-12-17"
```

5.2 Vector Data Structures

Vectors are the fundamental data structure in R - even single values are vectors of length 1.

5.2.1 Creating Vectors

```r
# Combine function c()
numeric_vec <- c(1, 2, 3, 4, 5)
char_vec <- c("apple", "banana", "cherry", "date")
logical_vec <- c(TRUE, FALSE, TRUE, TRUE, FALSE)

# Sequences
seq1 <- 1:10                          # Simple sequence
seq2 <- 10:1                          # Descending
seq3 <- seq(0, 100, by = 10)          # With step
seq4 <- seq(0, 1, length.out = 11)  # Specific length
seq5 <- rep(5, times = 3)             # Repeat value
seq6 <- rep(c(1, 2), times = 3)       # Repeat vector
seq7 <- rep(c(1, 2), each = 3)        # Repeat each element

print(seq1)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
print(seq3)
```

```
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

```r
print(seq6)  # 1 2 1 2 1 2
```

```
[1] 1 2 1 2 1 2
```

```r
print(seq7)  # 1 1 1 2 2 2
```

```
[1] 1 1 1 2 2 2
```

5.2.2 Vector Properties and Operations

```r
# Vector properties
vec <- c(10, 20, 30, 40, 50)

length(vec)          # 5
```

```
[1] 5
```

```r
class(vec)           # "numeric"
```

```
[1] "numeric"
```

```r
typeof(vec)          # "double"
```

```
[1] "double"
```

```r
str(vec)             # Structure
```

```
 num [1:5] 10 20 30 40 50
```

```r
# Vectorized operations (applied element-wise)
vec + 10              # 20 30 40 50 60
```

```
[1] 20 30 40 50 60
```

```r
vec * 2               # 20 40 60 80 100
```

```
[1]  20  40  60  80 100
```

```r
vec^2                 # 100 400 900 1600 2500
```

```
[1]  100  400  900 1600 2500
```

```r
sqrt(vec)             # Square root of each element
```

```
[1] 3.162278 4.472136 5.477226 6.324555 7.071068
```

```r
log(vec)              # Natural log of each element
```

```
[1] 2.302585 2.995732 3.401197 3.688879 3.912023
```

```r
# Operations between vectors (element-wise)
vec1 <- c(1, 2, 3, 4, 5)
vec2 <- c(10, 20, 30, 40, 50)
vec1 + vec2           # 11 22 33 44 55
```

```
[1] 11 22 33 44 55
```

```r
vec1 * vec2           # 10 40 90 160 250
```

```
[1]  10  40  90 160 250
```

5.2.3 Named Vectors

```r
# Create named vector
scores <- c(Math = 95, English = 88, Science = 92, History = 85)
print(scores)
```

```
   Math English Science History 
     95      88      92      85 
```

```r
# Access by name
scores["Math"]
```

```
Math 
  95 
```

```r
scores[c("Math", "Science")]
```

```
   Math Science 
     95      92 
```

```r
# Get names
names(scores)
```

```
[1] "Math"    "English" "Science" "History"
```

```r
# Add/modify names
grades <- c(95, 88, 92, 85)
names(grades) <- c("Math", "English", "Science", "History")
print(grades)
```

```
   Math English Science History 
     95      88      92      85 
```

```r
# Named vectors are useful for lookup tables
month_days <- c(Jan = 31, Feb = 28, Mar = 31, Apr = 30, May = 31, Jun = 30,
                Jul = 31, Aug = 31, Sep = 30, Oct = 31, Nov = 30, Dec = 31)
month_days["Feb"]
```

```
Feb 
 28 
```

5.2.4 Vector Indexing and Subsetting

which function is very useful for logical indexing.

```r
# Create vector for examples
vec <- c(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)

# Positive indexing (1-based, not 0-based like Python!)
vec[1]               # First element:  10
```

```
[1] 10
```

```r
vec[5]               # Fifth element: 50
```

```
[1] 50
```

```r
vec[c(1, 3, 5)]      # Multiple elements: 10 30 50
```

```
[1] 10 30 50
```

```r
vec[1:5]             # Range: 10 20 30 40 50
```

```
[1] 10 20 30 40 50
```

```r
# Negative indexing (exclusion)
vec[-1]              # All except first
```

```
[1]  20  30  40  50  60  70  80  90 100
```

```r
vec[-c(1, 2)]        # All except first two
```

```
[1]  30  40  50  60  70  80  90 100
```

```r
vec[-(1:5)]          # All except first five
```

```
[1]  60  70  80  90 100
```

```r
# Logical indexing (very powerful!)
vec[vec > 50]          # Elements greater than 50
```

```
[1]  60  70  80  90 100
```

```r
vec[vec %% 2 == 0]    # Even elements
```

```
 [1]  10  20  30  40  50  60  70  80  90 100
```

```r
vec[vec >= 30 & vec <= 70]  # Between 30 and 70
```

```
[1] 30 40 50 60 70
```

```r
# Which function (returns indices)
which(vec > 50)       # Indices where condition is TRUE
```

```
[1]  6  7  8  9 10
```

```r
vec[which.max(vec)]  # Maximum value
```

```
[1] 100
```

```r
vec[which.min(vec)]  # Minimum value
```

```
[1] 10
```

5.2.5 Vector Type Coercion

```r
# Vectors must be homogeneous (all same type)
# R coerces to most flexible type:  logical < integer < double < character

mixed1 <- c(1, 2, "three")     # All become character
print(mixed1)
```

```
[1] "1"     "2"     "three"
```

```r
class(mixed1)
```

```
[1] "character"
```

```r
mixed2 <- c(TRUE, FALSE, 1, 2)  # All become numeric
print(mixed2)
```

```
[1] 1 0 1 2
```

```r
class(mixed2)
```

```
[1] "numeric"
```

```r
mixed3 <- c(TRUE, FALSE, "yes")  # All become character
print(mixed3)
```

```
[1] "TRUE"  "FALSE" "yes"
```

```r
class(mixed3)
```

```
[1] "character"
```

```r
# Explicit coercion
char_nums <- c("1", "2", "3", "4")
as.numeric(char_nums)            # Convert to numeric
```

```
[1] 1 2 3 4
```

```r
logical_vals <- c(1, 0, 1, 1, 0)
as.logical(logical_vals)         # Convert to logical
```

```
[1]  TRUE FALSE  TRUE  TRUE FALSE
```

5.3 Lists - Flexible Containers

Lists can contain elements of different types and structures, including other lists.

5.3.1 Creating Lists

```r
# Basic list creation
my_list <- list(
    numbers = c(1, 2, 3, 4, 5),
    text = "Hello World",
    flag = TRUE,
    matrix_data = matrix(1:6, nrow = 2),
    nested_list = list(a = 10, b = 20)
)

print(my_list)
```

```
$numbers
[1] 1 2 3 4 5

$text
[1] "Hello World"

$flag
[1] TRUE

$matrix_data
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

$nested_list
$nested_list$a
[1] 10

$nested_list$b
[1] 20
```

```r
str(my_list)  # Structure is very helpful for lists
```

```
List of 5
 $ numbers    : num [1:5] 1 2 3 4 5
 $ text       : chr "Hello World"
 $ flag       : logi TRUE
```

```
 $ matrix_data: int [1:2, 1:3] 1 2 3 4 5 6
 $ nested_list:List of 2
  ..$ a: num 10
  ..$ b: num 20
```

5.3.2 Accessing List Elements

```r
# Three ways to access list elements

# 1. Using $ (by name)
my_list$numbers
```

```
[1] 1 2 3 4 5
```

```r
my_list$text
```

```
[1] "Hello World"
```

```r
# 2. Using [[ ]] (by name or index) - extracts element
my_list[["numbers"]]
```

```
[1] 1 2 3 4 5
```

```r
my_list[[1]]
```

```
[1] 1 2 3 4 5
```

```r
my_list[["nested_list"]]$a
```

```
[1] 10
```

```r
# 3. Using [ ] - returns a list
my_list["numbers"]   # Returns list with one element
```

```
$numbers
[1] 1 2 3 4 5
```

```r
my_list[1]           # Same
```

```
$numbers
[1] 1 2 3 4 5
```

```r
class(my_list[[1]])  # "numeric"
```

```
[1] "numeric"
```

```r
class(my_list[1])    # "list"
```

```
[1] "list"
```

```r
# Accessing nested elements
my_list$nested_list$a
```

```
[1] 10
```

```r
my_list[["nested_list"]][["a"]]
```

```
[1] 10
```

```r
my_list[[5]][[1]]
```

```
[1] 10
```

5.3.3 Modifying Lists

```r
# Add new elements
my_list$new_element <- c(100, 200, 300)
my_list[["another_element"]] <- "New value"

# Modify existing elements
my_list$text <- "Modified text"
```

```r
# Remove elements
my_list$new_element <- NULL

# Append lists
list1 <- list(a = 1, b = 2)
list2 <- list(c = 3, d = 4)
combined <- c(list1, list2)
print(combined)
```

```
$a
[1] 1

$b
[1] 2

$c
[1] 3

$d
[1] 4
```

```r
# Get list names
names(my_list)
```

```
[1] "numbers"        "text"           "flag"           "matrix_data"
[5] "nested_list"    "another_element"
```

```r
# Unnamed lists
unnamed_list <- list(10, "text", TRUE)
print(unnamed_list)
```

```
[[1]]
[1] 10

[[2]]
[1] "text"

[[3]]
[1] TRUE
```

```r
unnamed_list[[2]]  # Access by index only
```

```
[1] "text"
```

5.3.4 Practical List Applications

```r
# Storing analysis results
perform_analysis <- function(data) {
    results <- list(
        summary_stats = summary(data),
        mean_value = mean(data, na.rm = TRUE),
        sd_value = sd(data, na.rm = TRUE),
        n_missing = sum(is.na(data)),
        n_total = length(data)
    )
    return(results)
}

sample_data <- c(12, 15, 18, NA, 22, 25, 28, 31)
analysis_results <- perform_analysis(sample_data)
print(analysis_results)
```

```
$summary_stats
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
  12.00   16.50   22.00   21.57   26.50   31.00       1 

$mean_value
[1] 21.57143

$sd_value
[1] 6.948792

$n_missing
[1] 1

$n_total
[1] 8
```

```
# Accessing results
print(paste("Mean:", analysis_results$mean_value))
```

```
[1] "Mean: 21.5714285714286"
```

```
print(paste("Missing values:", analysis_results$n_missing))
```

```
[1] "Missing values: 1"
```

5.4 Data Frames - R's Version of SAS Datasets

Data frames are the primary structure for tabular data in R.

5.4.1 Creating Data Frames

```
# Method 1: From vectors
employees_df <- data.frame(
    employee_id = 1:6,
    name = c("John", "Jane", "Bob", "Alice", "Charlie", "Diana"),
    department = c("Sales", "IT", "IT", "Sales", "HR", "Sales"),
    salary = c(50000, 75000, 68000, 52000, 48000, 55000),
    years_employed = c(2, 5, 3, 1, 4, 3),
    full_time = c(TRUE, TRUE, FALSE, TRUE, TRUE, FALSE),
    stringsAsFactors = FALSE  # Keep strings as character (important!)
)

print(employees_df)
```

```
  employee_id    name department salary years_employed full_time
1           1    John      Sales  50000              2      TRUE
2           2    Jane         IT  75000              5      TRUE
3           3     Bob         IT  68000              3     FALSE
4           4   Alice      Sales  52000              1      TRUE
5           5 Charlie         HR  48000              4      TRUE
6           6   Diana      Sales  55000              3     FALSE
```

```
# Method 2: From lists
emp_list <- list(
    employee_id = 1:3,
    name = c("John", "Jane", "Bob"),
    salary = c(50000, 75000, 68000)
)
df_from_list <- as.data.frame(emp_list)
print(df_from_list)
```

```
  employee_id name salary
1           1 John  50000
2           2 Jane  75000
3           3  Bob  68000
```

```
# Method 3: Read from file (most common)
# employees_df <- read.csv("employees. csv")
# employees_df <- read.table("employees.txt", header = TRUE)
```

5.4.2 Data Frame Properties

```
# Dimensions
nrow(employees_df)      # Number of rows
```

```
[1] 6
```

```
ncol(employees_df)      # Number of columns
```

```
[1] 6
```

```
dim(employees_df)       # Both:  rows, columns
```

```
[1] 6 6
```

```
# Names
names(employees_df)     # Column names
```

```
[1] "employee_id"    "name"           "department"     "salary"        
[5] "years_employed" "full_time"     
```

```r
colnames(employees_df)  # Same
```

```
[1] "employee_id"    "name"           "department"     "salary"        
[5] "years_employed" "full_time"     
```

```r
rownames(employees_df)  # Row names (usually just numbers)
```

```
[1] "1" "2" "3" "4" "5" "6"
```

```r
# Structure and summary
str(employees_df)       # Structure (like PROC CONTENTS)
```

```
'data.frame':	6 obs. of  6 variables:
 $ employee_id   : int  1 2 3 4 5 6
 $ name          : chr  "John" "Jane" "Bob" "Alice" ...
 $ department    : chr  "Sales" "IT" "IT" "Sales" ...
 $ salary        : num  50000 75000 68000 52000 48000 55000
 $ years_employed: num  2 5 3 1 4 3
 $ full_time     : logi  TRUE TRUE FALSE TRUE TRUE FALSE
```

```r
summary(employees_df)   # Summary statistics
```

```
  employee_id        name            department            salary     
 Min.   :1.00   Length:6           Length:6           Min.   :48000  
 1st Qu.:2.25   Class :character   Class :character   1st Qu.:50500  
 Median :3.50   Mode  :character   Mode  :character   Median :53500  
 Mean   :3.50                                         Mean   :58000  
 3rd Qu.:4.75                                         3rd Qu.:64750  
 Max.   :6.00                                         Max.   :75000  
 years_employed full_time      
 Min.   :1.00   Mode :logical  
 1st Qu.:2.25   FALSE:2        
 Median :3.00   TRUE :4        
 Mean   :3.00                  
 3rd Qu.:3.75                  
 Max.   :5.00                  
```

```r
head(employees_df, 3)   # First 3 rows
```

```
  employee_id name department salary years_employed full_time
1           1 John      Sales  50000              2      TRUE
2           2 Jane         IT  75000              5      TRUE
3           3  Bob         IT  68000              3     FALSE
```

```r
tail(employees_df, 2)   # Last 2 rows
```

```
  employee_id    name department salary years_employed full_time
5           5 Charlie         HR  48000              4      TRUE
6           6   Diana      Sales  55000              3     FALSE
```

```r
glimpse(employees_df)   # tidyverse version (if loaded)
```

```
Rows: 6
Columns: 6
$ employee_id    <int> 1, 2, 3, 4, 5, 6
$ name           <chr> "John", "Jane", "Bob", "Alice", "Charlie", "Diana"
$ department     <chr> "Sales", "IT", "IT", "Sales", "HR", "Sales"
$ salary         <dbl> 50000, 75000, 68000, 52000, 48000, 55000
$ years_employed <dbl> 2, 5, 3, 1, 4, 3
$ full_time      <lgl> TRUE, TRUE, FALSE, TRUE, TRUE, FALSE
```

5.4.3 Accessing Data Frame Elements

```r
# Access columns
employees_df$name                    # By name with $
```

```
[1] "John"    "Jane"    "Bob"     "Alice"   "Charlie" "Diana"
```

```r
employees_df[["name"]]               # By name with [[]]
```

```
[1] "John"    "Jane"    "Bob"     "Alice"   "Charlie" "Diana"
```

```r
employees_df[, "name"]               # By name with [,]
```

```
[1] "John"    "Jane"    "Bob"     "Alice"   "Charlie" "Diana"
```

```r
employees_df[, 2]                    # By position
```

```
[1] "John"    "Jane"    "Bob"     "Alice"   "Charlie" "Diana"
```

```r
# Access rows
employees_df[1, ]                    # First row
```

```
  employee_id name department salary years_employed full_time
1           1 John      Sales  50000              2      TRUE
```

```r
employees_df[1:3, ]                  # First three rows
```

```
  employee_id name department salary years_employed full_time
1           1 John      Sales  50000              2      TRUE
2           2 Jane         IT  75000              5      TRUE
3           3  Bob         IT  68000              3     FALSE
```

```r
# Access specific elements
employees_df[1, 2]                   # Row 1, column 2
```

```
[1] "John"
```

```r
employees_df[1, "name"]              # Row 1, column "name"
```

```
[1] "John"
```

```r
employees_df$name[1]                 # Element 1 of column "name"
```

```
[1] "John"
```

```r
# Multiple columns
employees_df[, c("name", "salary")]
```

```
     name salary
1    John  50000
2    Jane  75000
3     Bob  68000
4   Alice  52000
5 Charlie  48000
6   Diana  55000
```

```r
employees_df[, c(2, 4)]
```

```
     name salary
1    John  50000
2    Jane  75000
3     Bob  68000
4   Alice  52000
5 Charlie  48000
6   Diana  55000
```

```r
# Subset with conditions
employees_df[employees_df$salary > 50000, ]
```

```
  employee_id  name department salary years_employed full_time
2           2  Jane         IT  75000              5      TRUE
3           3   Bob         IT  68000              3     FALSE
4           4 Alice      Sales  52000              1      TRUE
6           6 Diana      Sales  55000              3     FALSE
```

```r
employees_df[employees_df$department == "Sales", ]
```

```
  employee_id  name department salary years_employed full_time
1           1  John      Sales  50000              2      TRUE
4           4 Alice      Sales  52000              1      TRUE
6           6 Diana      Sales  55000              3     FALSE
```

```r
employees_df[employees_df$salary > 50000 & employees_df$full_time, ]
```

```
  employee_id  name department salary years_employed full_time
2           2  Jane         IT  75000              5      TRUE
4           4 Alice      Sales  52000              1      TRUE
```

5.4.4 Modifying Data Frames

```r
# Add new column
employees_df$bonus <- employees_df$salary * 0.10

# Modify existing column
employees_df$salary <- employees_df$salary * 1.05  # 5% raise

# Add calculated column
employees_df$total_comp <- employees_df$salary + employees_df$bonus

# Delete column
employees_df$bonus <- NULL

# Rename columns
names(employees_df)[names(employees_df) == "name"] <- "employee_name"

# Add rows
new_employee <- data.frame(
    employee_id = 7,
    employee_name = "Eve",
    department = "IT",
    salary = 70000,
    years_employed = 2,
    full_time = TRUE,
    total_comp = 77000
)
employees_df <- rbind(employees_df, new_employee)

# Reset column name back
names(employees_df)[names(employees_df) == "employee_name"] <- "name"
```

5.4.5 SAS DATA Step vs R Data Frame Operations

: :: {.panel-tabset}

5.5 SAS

```
/* SAS DATA step */
DATA analysis;
    SET employees;
    WHERE salary > 50000;

    /* Create variables */
    bonus = salary * 0.10;
    total_comp = salary + bonus;

    /* Conditional logic */
    IF department = 'Sales' THEN sales_flag = 1;
    ELSE sales_flag = 0;

    /* Keep specific variables */
    KEEP employee_id name salary bonus total_comp;
RUN;
```

5.6 R (Base)

```
# Base R approach
analysis <- employees[employees$salary > 50000, ]
analysis$bonus <- analysis$salary * 0.10
analysis$total_comp <- analysis$salary + analysis$bonus
analysis$sales_flag <- ifelse(analysis$department == "Sales", 1, 0)
analysis <- analysis[, c("employee_id", "name", "salary", "bonus", "total_comp")]
```

5.7 R (tidyverse)

```r
# tidyverse approach (more readable)
analysis <- employees_df %>%
    filter(salary > 50000) %>%
    mutate(
        bonus = salary * 0.10,
        total_comp = salary + bonus,
        sales_flag = if_else(department == "Sales", 1, 0)
    ) %>%
    select(employee_id, name, salary, bonus, total_comp)

print(analysis)
```

:::

5.8 Matrices and Arrays

Matrices (2D) and arrays (N-dimensional) are homogeneous data structures.

5.8.1 Creating and Using Matrices

```r
# Create matrix
mat <- matrix(1:12, nrow = 3, ncol = 4)
print(mat)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```r
# Matrix with specific fill pattern
mat_byrow <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
print(mat_byrow)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```r
# Named dimensions
mat_named <- matrix(1:12, nrow = 3, ncol = 4,
                    dimnames = list(c("Row1", "Row2", "Row3"),
                                   c("Col1", "Col2", "Col3", "Col4")))
print(mat_named)
```

```
     Col1 Col2 Col3 Col4
Row1    1    4    7   10
Row2    2    5    8   11
Row3    3    6    9   12
```

```r
# Matrix properties
dim(mat)         # Dimensions
```

```
[1] 3 4
```

```r
nrow(mat)        # Number of rows
```

```
[1] 3
```

```r
ncol(mat)        # Number of columns
```

```
[1] 4
```

```r
length(mat)      # Total elements
```

```
[1] 12
```

```r
# Matrix indexing
mat[1, 2]        # Element at row 1, column 2
```

```
[1] 4
```

```r
mat[1, ]         # First row
```

```
[1]  1  4  7 10
```

```r
mat[, 2]         # Second column
```

```
[1] 4 5 6
```

```r
mat[1: 2, 2: 3]    # Submatrix
```

```
     [,1] [,2]
[1,]    4    7
[2,]    5    8
```

5.8.2 Matrix Operations

```r
# Create matrices for operations
A <- matrix(c(1, 2, 3, 4), nrow = 2)
B <- matrix(c(5, 6, 7, 8), nrow = 2)

# Element-wise operations
A + B            # Element-wise addition
```

```
     [,1] [,2]
[1,]    6   10
[2,]    8   12
```

```r
A * B            # Element-wise multiplication
```

```
     [,1] [,2]
[1,]    5   21
[2,]   12   32
```

```r
A / B            # Element-wise division
```

```
          [,1]      [,2]
[1,] 0.2000000 0.4285714
[2,] 0.3333333 0.5000000
```

```r
# Matrix multiplication
A %*% B           # Matrix product
```

```
     [,1] [,2]
[1,]   23   31
[2,]   34   46
```

```r
# Transpose
t(A)
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

```r
# Inverse (if square and invertible)
solve(A)
```

```
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

```r
# Determinant
det(A)
```

```
[1] -2
```

```r
# Eigenvalues and eigenvectors
eigen(A)
```

```
eigen() decomposition
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

5.8.3 Arrays (Multi-dimensional)

```r
# Create 3D array
arr <- array(1:24, dim = c(3, 4, 2))
print(arr)
```

```
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

```r
# Access elements
arr[1, 2, 1]     # Single element
```

```
[1] 4
```

```r
arr[, , 1]       # First "slice"
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```r
arr[1, , ]       # First row across all slices
```

```
     [,1] [,2]
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

```r
# Array properties
dim(arr)
```

```
[1] 3 4 2
```

```r
length(arr)
```

```
[1] 24
```

5.9 Tibbles - Modern Data Frames

Tibbles are enhanced data frames from the tidyverse with better printing and behavior.

5.9.1 Creating Tibbles

```r
library(tibble)

# Create tibble
employees_tbl <- tibble(
    employee_id = 1:6,
    name = c("John", "Jane", "Bob", "Alice", "Charlie", "Diana"),
    department = c("Sales", "IT", "IT", "Sales", "HR", "Sales"),
    salary = c(50000, 75000, 68000, 52000, 48000, 55000),
    start_date = as.Date(c("2023-01-15", "2020-06-01", "2022-03-10",
                           "2024-02-20", "2021-09-05", "2022-11-12"))
)

print(employees_tbl)  # Better printing than data.frame
```

```
# A tibble: 6 x 5
  employee_id name    department salary start_date
        <int> <chr>   <chr>       <dbl> <date>
1           1 John    Sales       50000 2023-01-15
2           2 Jane    IT          75000 2020-06-01
3           3 Bob     IT          68000 2022-03-10
4           4 Alice   Sales       52000 2024-02-20
5           5 Charlie HR          48000 2021-09-05
6           6 Diana   Sales       55000 2022-11-12
```

```r
# Tribble:  row-wise tibble creation
employees_tribble <- tribble(
    ~employee_id, ~name,      ~department, ~salary,
    1,            "John",     "Sales",     50000,
    2,            "Jane",     "IT",        75000,
    3,            "Bob",      "IT",        68000
)
print(employees_tribble)
```

```
# A tibble: 3 x 4
  employee_id name  department salary
        <dbl> <chr> <chr>       <dbl>
1           1 John  Sales       50000
2           2 Jane  IT          75000
3           3 Bob   IT          68000
```

5.9.2 Tibble vs Data Frame Differences

```r
# Difference 1: Printing
# Data frames print everything, tibbles show first 10 rows
df_large <- data.frame(x = 1:100, y = 101:200)
tbl_large <- tibble(x = 1:100, y = 101:200)
# print(df_large)   # Would print all 100 rows
print(tbl_large)    # Prints first 10 rows nicely
```

```
# A tibble: 100 x 2
       x     y
   <int> <int>
 1     1   101
 2     2   102
 3     3   103
 4     4   104
 5     5   105
 6     6   106
 7     7   107
 8     8   108
 9     9   109
10    10   110
# i 90 more rows
```

```r
# Difference 2: Subsetting
df <- data.frame(x = 1:3, y = 4:6)
tbl <- tibble(x = 1:3, y = 4:6)

df[, "x"]           # Returns vector
```

```
[1] 1 2 3
```

```r
tbl[, "x"]          # Returns tibble
```

```
# A tibble: 3 x 1
      x
  <int>
1     1
2     2
3     3
```

```r
# Difference 3: Character vectors
df_char <- data.frame(name = c("John", "Jane"))
tbl_char <- tibble(name = c("John", "Jane"))

str(df_char)        # Might convert to factor (depends on R version)
```

```
'data.frame':   2 obs. of  1 variable:
 $ name: chr  "John" "Jane"
```

```r
str(tbl_char)       # Always character
```

```
tibble [2 x 1] (S3: tbl_df/tbl/data.frame)
 $ name: chr [1:2] "John" "Jane"
```

```r
# Difference 4: Column names
# Tibbles allow non-syntactic names
tbl_special <- tibble(
    `Column with spaces` = 1:3,
    `2024` = 4:6
)
print(tbl_special)
```

```
# A tibble: 3 x 2
  `Column with spaces` `2024`
                 <int>  <int>
1                    1      4
2                    2      5
3                    3      6
```

5.9.3 Converting Between Data Frames and Tibbles

```r
# Convert data frame to tibble
df <- data.frame(x = 1:3, y = 4:6)
tbl <- as_tibble(df)

# Convert tibble to data frame
tbl <- tibble(x = 1:3, y = 4:6)
df <- as.data.frame(tbl)

# Check type
is.data.frame(tbl)   # TRUE (tibbles are also data frames)
```

```
[1] TRUE
```

```r
is_tibble(df)        # FALSE
```

```
[1] FALSE
```

```r
is_tibble(tbl)       # TRUE
```

```
[1] TRUE
```

5.10 Data. table - High Performance Option

Data.table is optimized for speed and memory efficiency with large datasets.

5.10.1 Creating Data.tables

```r
library(data.table)

# Create data. table
employees_dt <- data.table(
    employee_id = 1:1000,
    department = sample(c("Sales", "IT", "HR", "Finance"), 1000, replace = TRUE),
    salary = round(rnorm(1000, mean = 60000, sd = 15000), 0),
    years = sample(1:20, 1000, replace = TRUE)
)

print(head(employees_dt))
```

```
   employee_id department salary years
         <int>     <char>  <num> <int>
1:           1         HR  51842    17
2:           2         HR  66407    13
3:           3         IT  71294    11
4:           4         HR  81347    20
5:           5    Finance  45480     6
6:           6      Sales  74822    20
```

```r
# Convert from data frame
df <- data.frame(x = 1:3, y = 4:6)
dt <- as.data.table(df)
```

5.10.2 Data.table Syntax: DT[i, j, by]

The data.table syntax follows the pattern: `DT[where, select, group by]`

```r
# i: Filter rows (WHERE clause)
employees_dt[salary > 70000]
```

```
     employee_id department salary years
           <int>     <char>  <num> <int>
  1:           3         IT  71294    11
  2:           4         HR  81347    20
  3:           6      Sales  74822    20
  4:          10    Finance  70873    13
  5:          12         HR  80826     1
```

```
 ---
234:         977          IT  80616     8
235:         978          HR  70166    14
236:         989     Finance 105046    14
237:         996          IT  93737     2
238:         997       Sales  75090     1
```

```r
employees_dt[department == "IT"]
```

```
     employee_id department salary years
           <int>     <char>  <num> <int>
  1:           3         IT  71294    11
  2:           9         IT  51208    16
  3:          16         IT  56581    15
  4:          19         IT  35140     7
  5:          20         IT  64713     9
 ---
241:         977         IT  80616     8
242:         994         IT  54681     9
243:         996         IT  93737     2
244:         999         IT  49916    16
245:        1000         IT  68992     9
```

```r
# j: Select/compute columns (SELECT clause)
employees_dt[, .(employee_id, salary)]
```

```
      employee_id salary
            <int>  <num>
   1:           1  51842
   2:           2  66407
   3:           3  71294
   4:           4  81347
   5:           5  45480
  ---
 996:         996  93737
 997:         997  75090
 998:         998  56055
 999:         999  49916
1000:        1000  68992
```

```r
employees_dt[, mean_salary := mean(salary)]

# by: Group by (GROUP BY clause)
employees_dt[, .(avg_salary = mean(salary)), by = department]
```

```
   department avg_salary
       <char>      <num>
1:         HR   59172.37
2:         IT   60116.66
3:    Finance   56665.81
4:      Sales   58846.69
```

```r
employees_dt[, .(count = .N), by = department]  # . N is row count
```

```
   department count
       <char> <int>
1:         HR   251
2:         IT   245
3:    Finance   249
4:      Sales   255
```

```r
# Combining i, j, by
employees_dt[salary > 50000,
             .(avg_salary = mean(salary), count = .N),
             by = department]
```

```
   department avg_salary count
       <char>      <num> <int>
1:         HR   66055.12   184
2:         IT   66951.79   184
3:      Sales   65162.32   186
4:    Finance   64984.46   164
```

```r
# Multiple group by
employees_dt[, .(avg_salary = mean(salary)),
             by = .(department, years_bucket = cut(years, breaks = c(0, 5, 10, 20)))]
```

```
    department years_bucket avg_salary
        <char>       <fctr>      <num>
```

```
 1:          HR      (10,20]   58712.06
 2:          IT      (10,20]   60364.82
 3:     Finance       (5,10]   56291.61
 4:       Sales      (10,20]   58037.80
 5:     Finance      (10,20]   56846.53
 6:          HR        (0,5]   59166.40
 7:          IT       (5,10]   59374.37
 8:       Sales        (0,5]   59515.41
 9:     Finance        (0,5]   56726.71
10:       Sales       (5,10]   59447.41
11:          IT        (0,5]   60374.79
12:          HR       (5,10]   60264.12
```

5.10.3 Data.table Performance Advantages

```r
# Create large datasets for comparison
n <- 1000000

# Data frame approach
df_large <- data.frame(
    id = 1:n,
    group = sample(letters[1:100], n, replace = TRUE),
    value = rnorm(n)
)

# Data.table approach
dt_large <- data.table(
    id = 1:n,
    group = sample(letters[1:100], n, replace = TRUE),
    value = rnorm(n)
)

# Speed comparison:  group aggregation
system.time({
    result_df <- aggregate(value ~ group, data = df_large, FUN = mean)
})
```

```
   user  system elapsed
  0.092   0.014   0.106
```

```r
system.time({
    result_dt <- dt_large[, .(mean_value = mean(value)), by = group]
})
```

```
   user  system elapsed 
  0.081   0.003   0.044 
```

```r
# data.table is typically much faster for large data operations
```

5.11 Factors - Categorical Variables

Factors represent categorical data, similar to SAS formats/value labels.

5.11.1 Creating Factors

```r
# Create factor from character vector
gender_char <- c("M", "F", "F", "M", "M", "F")
gender_factor <- factor(gender_char)
print(gender_factor)
```

```
[1] M F F M M F
Levels: F M
```

```r
# Levels are shown
levels(gender_factor)
```

```
[1] "F" "M"
```

```r
# With explicit levels and labels
gender_factor2 <- factor(
    c("M", "F", "F", "M", "M", "F"),
    levels = c("M", "F"),
    labels = c("Male", "Female")
)
print(gender_factor2)
```

```
[1] Male   Female Female Male   Male   Female
Levels: Male Female
```

```r
# Ordered factors
education <- factor(
    c("HS", "BS", "MS", "PhD", "BS", "HS"),
    levels = c("HS", "BS", "MS", "PhD"),
    ordered = TRUE
)
print(education)
```

```
[1] HS  BS  MS  PhD BS  HS
Levels: HS < BS < MS < PhD
```

```r
print(education[2] < education[3])  # TRUE:  BS < MS
```

```
[1] TRUE
```

5.11.2 Working with Factors

```r
# Factor properties
levels(gender_factor)
```

```
[1] "F" "M"
```

```r
nlevels(gender_factor)
```

```
[1] 2
```

```r
is.factor(gender_factor)
```

```
[1] TRUE
```

```r
is.ordered(education)
```

```
[1] TRUE
```

```r
# Convert factor to character/numeric
as.character(gender_factor)
```

```
[1] "M" "F" "F" "M" "M" "F"
```

```r
as.numeric(gender_factor)  # Returns level codes (1, 2, ...)
```

```
[1] 2 1 1 2 2 1
```

```r
# Reorder levels
gender_reordered <- factor(gender_factor, levels = c("F", "M"))
levels(gender_reordered)
```

```
[1] "F" "M"
```

```r
# Add levels
gender_expanded <- factor(gender_factor, levels = c("M", "F", "O"))
levels(gender_expanded)
```

```
[1] "M" "F" "O"
```

```r
# Relabel levels
levels(gender_factor) <- c("Male", "Female")
print(gender_factor)
```

```
[1] Female Male   Male   Female Female Male
Levels: Male Female
```

5.11.3 Practical Uses for Factors

```r
# Create survey data
survey <- data.frame(
    id = 1:10,
    satisfaction = factor(
        c("High", "Low", "Medium", "High", "Low", "High", "Medium", "High", "Low", "Medium"),
        levels = c("Low", "Medium", "High"),
```

```r
        ordered = TRUE
    ),
    region = factor(c("North", "South", "North", "West", "East",
                     "North", "South", "East", "West", "North"))
)

# Frequency tables (like PROC FREQ)
table(survey$satisfaction)
```

```

   Low Medium   High 
     3      3      4 
```

```r
table(survey$region)
```

```

 East North South  West 
    2     4     2     2 
```

```r
table(survey$satisfaction, survey$region)  # Cross-tabulation
```

```

         East North South West
  Low       1     0     1    1
  Medium    0     2     1    0
  High      1     2     0    1
```

```r
# Proportions
prop.table(table(survey$satisfaction))
```

```

   Low Medium   High 
   0.3    0.3    0.4 
```

```r
# Factors preserve level order in plots and analyses
# (Very useful for ordered categories like satisfaction, education, etc.)
```

5.11.4 Factor Pitfalls and Solutions

```r
# Pitfall 1: Factors look like characters but aren't
factor_var <- factor(c("10", "20", "30"))
# as.numeric(factor_var)  # Returns 1, 2, 3 (NOT 10, 20, 30!)
as.numeric(as.character(factor_var))  # Returns 10, 20, 30 (correct)
```

```
[1] 10 20 30
```

```r
# Pitfall 2: Can't add values not in levels
colors <- factor(c("red", "blue", "green"), levels = c("red", "blue", "green"))
# colors[4] <- "yellow"  # Would produce NA (not an error!)

# Solution: Add level first
levels(colors) <- c(levels(colors), "yellow")
colors[4] <- "yellow"
print(colors)
```

```
[1] red    blue   green  yellow
Levels: red blue green yellow
```

```r
# Pitfall 3: stringsAsFactors in data.frame (older R versions)
# In R < 4.0.0, characters were automatically converted to factors
# Always use stringsAsFactors = FALSE or upgrade to R >= 4.0.0
df_safe <- data.frame(
    name = c("John", "Jane"),
    stringsAsFactors = FALSE
)
str(df_safe)  # character, not factor
```

```
'data.frame':	2 obs. of  1 variable:
 $ name: chr  "John" "Jane"
```

6 4. Functions and Help System

6.1 Using Built-in Functions

R has thousands of built-in functions for data analysis.

6.1.1 Statistical Functions

```
# Create sample data
numbers <- c(12, 18, 23, 28, 15, NA, 34, 29, 19, 25)

# Measures of central tendency
mean(numbers, na.rm = TRUE)         # Mean
```

```
[1] 22.55556
```

```
median(numbers, na.rm = TRUE)       # Median
```

```
[1] 23
```

```
# mode (no built-in mode function)

# Measures of dispersion
sd(numbers, na.rm = TRUE)           # Standard deviation
```

```
[1] 7.16085
```

```
var(numbers, na.rm = TRUE)          # Variance
```

```
[1] 51.27778
```

```r
IQR(numbers, na.rm = TRUE)          # Interquartile range
```

```
[1] 10
```

```r
range(numbers, na.rm = TRUE)        # Min and max
```

```
[1] 12 34
```

```r
mad(numbers, na.rm = TRUE)          # Median absolute deviation
```

```
[1] 7.413
```

```r
# Summary statistics
min(numbers, na.rm = TRUE)
```

```
[1] 12
```

```r
max(numbers, na.rm = TRUE)
```

```
[1] 34
```

```r
sum(numbers, na.rm = TRUE)
```

```
[1] 203
```

```r
prod(numbers, na.rm = TRUE)         # Product
```

```
[1] 977240376000
```

```r
length(numbers)                      # Count all
```

```
[1] 10
```

```r
sum(! is.na(numbers))                 # Count non-missing
```

```
[1] 9
```

```r
# Quantiles
quantile(numbers, na.rm = TRUE)     # Default:  0%, 25%, 50%, 75%, 100%
```

```
  0%  25%  50%  75% 100% 
  12   18   23   28   34 
```

```r
quantile(numbers, probs = c(0.1, 0.9), na.rm = TRUE)  # 10th and 90th percentiles
```

```
 10%  90% 
14.4 30.0 
```

```r
# Summary function (multiple stats at once)
summary(numbers)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
  12.00   18.00   23.00   22.56   28.00   34.00       1 
```

6.1.2 Mathematical Functions

```r
# Arithmetic
abs(-5)                     # Absolute value
```

```
[1] 5
```

```r
sqrt(16)                    # Square root
```

```
[1] 4
```

```r
exp(2)                      # Exponential
```

```
[1] 7.389056
```

```r
log(100)                  # Natural logarithm
```

```
[1] 4.60517
```

```r
log10(100)                # Base-10 logarithm
```

```
[1] 2
```

```r
log(8, base = 2)          # Custom base logarithm
```

```
[1] 3
```

```r
# Rounding
round(3.14159, digits = 2) # 3.14
```

```
[1] 3.14
```

```r
floor(3.9)                # 3 (round down)
```

```
[1] 3
```

```r
ceiling(3.1)              # 4 (round up)
```

```
[1] 4
```

```r
trunc(3.9)                # 3 (truncate decimal)
```

```
[1] 3
```

```r
signif(123456, digits = 3) # 123000 (significant figures)
```

```
[1] 123000
```

```r
# Trigonometry
sin(pi/2)                   # 1
```

```
[1] 1
```

```r
cos(0)                      # 1
```

```
[1] 1
```

```r
tan(pi/4)                   # 1
```

```
[1] 1
```

```r
asin(1)                     # pi/2 (arcsin)
```

```
[1] 1.570796
```

```r
# Powers and roots
2^10                        # 1024
```

```
[1] 1024
```

```r
10^3                        # 1000
```

```
[1] 1000
```

```r
27^(1/3)                    # 3 (cube root)
```

```
[1] 3
```

6.1.3 String Functions

```r
# Basic string operations
text <- "  Hello World  "

nchar(text)                          # Length (with spaces)
```

```
[1] 15
```

```r
toupper(text)                        # "  HELLO WORLD  "
```

```
[1] "  HELLO WORLD  "
```

```r
tolower(text)                        # "  hello world  "
```

```
[1] "  hello world  "
```

```r
trimws(text)                         # Remove leading/trailing whitespace
```

```
[1] "Hello World"
```

```r
# Substring operations
substr("Hello World", 1, 5)          # "Hello"
```

```
[1] "Hello"
```

```r
substring("Hello World", 7)          # "World"
```

```
[1] "World"
```

```r
# Search and replace
grepl("World", text)                 # TRUE (pattern found)
```

```
[1] TRUE
```

```r
grep("World", c("Hello", "World", "Goodbye"))  # 2 (position)
```

```
[1] 2
```

```r
sub("World", "Universe", text)       # Replace first occurrence
```

```
[1] "  Hello Universe  "
```

```r
gsub("o", "0", text)                 # Replace all occurrences
```

```
[1] "  Hell0 W0rld  "
```

```r
# Split strings
strsplit("apple,banana,cherry", ",") # Split by delimiter
```

```
[[1]]
[1] "apple"  "banana" "cherry"
```

```r
# Paste strings
paste("Hello", "World")              # "Hello World" (with space)
```

```
[1] "Hello World"
```

```r
paste0("Hello", "World")             # "HelloWorld" (no space)
```

```
[1] "HelloWorld"
```

```r
paste(c("A", "B", "C"), collapse = "-")  # "A-B-C"
```

```
[1] "A-B-C"
```

```r
# Formatted strings (like sprintf in C)
sprintf("Patient %d:  BMI = %.2f", 12345, 24.567)
```

```
[1] "Patient 12345:  BMI = 24.57"
```

6.1.4 Type Checking and Conversion Functions

```r
# Type checking (is.* functions)
x <- 42
is.numeric(x)
```

```
[1] TRUE
```

```r
is.integer(x)
```

```
[1] FALSE
```

```r
is.character(x)
```

```
[1] FALSE
```

```r
is.logical(x)
```

```
[1] FALSE
```

```r
is.factor(x)
```

```
[1] FALSE
```

```r
is.data.frame(x)
```

```
[1] FALSE
```

```r
is.list(x)
```

```
[1] FALSE
```

```r
is.matrix(x)
```

```
[1] FALSE
```

```r
is.na(x)
```

```
[1] FALSE
```

```r
is.null(x)
```

```
[1] FALSE
```

```r
# Type conversion (as.* functions)
as.character(42)              # "42"
```

```
[1] "42"
```

```r
as.numeric("42")              # 42
```

```
[1] 42
```

```r
as.integer(42.7)              # 42 (truncates)
```

```
[1] 42
```

```r
as.logical(1)                 # TRUE
```

```
[1] TRUE
```

```r
as.factor(c("A", "B", "C"))
```

```
[1] A B C
Levels: A B C
```

```r
as.data.frame(matrix(1:6, nrow = 2))
```

```
  V1 V2 V3
1  1  3  5
2  2  4  6
```

```r
as.list(c(1, 2, 3))
```

```
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
```

```r
as.matrix(data.frame(x = 1:3, y = 4:6))
```

```
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

6.2 SAS PROC to R Function Mapping

Here's a comprehensive mapping of common SAS procedures to R equivalents:

```r
# Create mapping table
sas_to_r <- data.frame(
    SAS_Proc = c(
        "PROC MEANS",
        "PROC FREQ",
        "PROC PRINT",
        "PROC SORT",
        "PROC CONTENTS",
        "PROC SQL",
        "PROC UNIVARIATE",
        "PROC CORR",
        "PROC REG",
        "PROC LOGISTIC",
        "PROC TABULATE",
        "PROC TRANSPOSE",
        "PROC IMPORT",
        "PROC EXPORT"
```

```r
    ),
    R_Function = c(
        "summary(), mean(), sd()",
        "table(), prop.table()",
        "print(), head(), View()",
        "sort(), order(), arrange()",
        "str(), glimpse(), attributes()",
        "sqldf(), dplyr verbs",
        "summary(), psych::describe()",
        "cor(), cor.test()",
        "lm(), summary()",
        "glm(family='binomial')",
        "ftable(), aggregate()",
        "t(), pivot_longer()/pivot_wider()",
        "read.csv(), read_excel()",
        "write.csv(), write_xlsx()"
    ),
    Package = c(
        "base",
        "base",
        "base",
        "base/dplyr",
        "base/dplyr",
        "sqldf/dplyr",
        "base/psych",
        "base",
        "stats",
        "stats",
        "base/tidyr",
        "base/tidyr",
        "base/readxl",
        "base/writexl"
    ),
    stringsAsFactors = FALSE
)

print(sas_to_r)
```

```
           SAS_Proc                        R_Function      Package
1        PROC MEANS           summary(), mean(), sd()         base
2         PROC FREQ             table(), prop.table()         base
3        PROC PRINT           print(), head(), View()         base
```

```
4         PROC SORT        sort(), order(), arrange()   base/dplyr
5     PROC CONTENTS    str(), glimpse(), attributes()   base/dplyr
6          PROC SQL              sqldf(), dplyr verbs  sqldf/dplyr
7   PROC UNIVARIATE      summary(), psych::describe()   base/psych
8         PROC CORR                  cor(), cor.test()         base
9          PROC REG                    lm(), summary()        stats
10    PROC LOGISTIC            glm(family='binomial')        stats
11    PROC TABULATE             ftable(), aggregate()   base/tidyr
12   PROC TRANSPOSE t(), pivot_longer()/pivot_wider()   base/tidyr
13      PROC IMPORT            read.csv(), read_excel()  base/readxl
14      PROC EXPORT          write.csv(), write_xlsx() base/writexl
```

6.2.1 Practical Examples: SAS to R

: :: {.panel-tabset}

6.3 PROC MEANS

```r
# Create data
data <- data.frame(
    group = rep(c("A", "B", "C"), each = 10),
    value = c(rnorm(10, 100, 15), rnorm(10, 110, 15), rnorm(10, 95, 15))
)

# SAS:  PROC MEANS DATA=data MEAN STD MIN MAX; VAR value; CLASS group; RUN;

# R (base)
aggregate(value ~ group, data = data,
          FUN = function(x) c(mean = mean(x), sd = sd(x),
                              min = min(x), max = max(x)))
```

```
  group value.mean  value.sd value.min value.max
1     A   94.89372  15.21235  62.79850 113.28803
2     B  110.89518  14.17912  86.46798 127.66605
3     C  100.08641  15.35787  81.35258 123.05135
```

```r
# R (dplyr)
data %>%
    group_by(group) %>%
```

```
    summarise(
        n = n(),
        mean = mean(value),
        sd = sd(value),
        min = min(value),
        max = max(value)
    )
```

```
# A tibble: 3 x 6
  group     n  mean    sd   min   max
  <chr> <int> <dbl> <dbl> <dbl> <dbl>
1 A        10  94.9  15.2  62.8  113.
2 B        10 111.   14.2  86.5  128.
3 C        10 100.   15.4  81.4  123.
```

6.4 PROC FREQ

```
# Create data
survey_data <- data.frame(
    gender = sample(c("M", "F"), 100, replace = TRUE),
    satisfaction = sample(c("Low", "Medium", "High"), 100, replace = TRUE)
)

# SAS: PROC FREQ DATA=survey_data; TABLES gender satisfaction gender*satisfaction; RUN;

# R:  One-way frequency
table(survey_data$gender)
```

```

 F  M 
50 50 
```

```
prop.table(table(survey_data$gender))
```

```

  F   M 
0.5 0.5 
```

```r
# Two-way cross-tabulation
table(survey_data$gender, survey_data$satisfaction)
```

```

    High Low Medium
  F   16  17     17
  M   13  18     19
```

```r
prop.table(table(survey_data$gender, survey_data$satisfaction))
```

```

    High  Low Medium
  F 0.16 0.17   0.17
  M 0.13 0.18   0.19
```

```r
# With dplyr
survey_data %>%
    count(gender, satisfaction) %>%
    mutate(proportion = n / sum(n))
```

```
  gender satisfaction  n proportion
1      F         High 16       0.16
2      F          Low 17       0.17
3      F       Medium 17       0.17
4      M         High 13       0.13
5      M          Low 18       0.18
6      M       Medium 19       0.19
```

6.5 PROC SORT

```r
# Create data
unsorted <- data.frame(
    id = c(3, 1, 4, 2, 5),
    name = c("Charlie", "Alice", "David", "Bob", "Eve"),
    score = c(85, 92, 78, 88, 95)
)
```

```
# SAS: PROC SORT DATA=unsorted OUT=sorted; BY score; RUN;

# R (base)
sorted_base <- unsorted[order(unsorted$score), ]
sorted_desc <- unsorted[order(-unsorted$score), ]  # Descending

# R (dplyr)
sorted_dplyr <- unsorted %>% arrange(score)
sorted_desc_dplyr <- unsorted %>% arrange(desc(score))

print(sorted_dplyr)
```

```
  id    name score
1  4   David    78
2  3 Charlie    85
3  2     Bob    88
4  1   Alice    92
5  5     Eve    95
```

:::

6.6 Getting Help in R

R has comprehensive built-in documentation and help systems.

6.6.1 Basic Help Commands

```
# Help on specific function
? mean
help(mean)

# Search help for keyword
?? regression
help.search("regression")

# Find functions with pattern in name
apropos("mean")
```

```r
# Examples from help page
example(mean)
example(plot)

# See function arguments
args(mean)
args(lm)

# See function source code
mean.default
lm  # For most functions, just type the name
```

6.6.2 Exploring Packages and Functions

```r
# List all installed packages
installed.packages()

# List functions in a package
ls("package:dplyr")

# Help on a package
help(package = "dplyr")

# Vignettes (package tutorials)
vignette()                    # List all vignettes
vignette("dplyr")             # Specific vignette
browseVignettes("dplyr")      # Browse all package vignettes
```

6.6.3 Online Resources and Cheat Sheets

```r
# R documentation website
# https://www.rdocumentation.org/

# RStudio cheat sheets
# https://posit.co/resources/cheatsheets/

# Stack Overflow for R questions
# https://stackoverflow.com/questions/tagged/r
```

```
# R-bloggers for tutorials
# https://www.r-bloggers.com/
```

6.7 Writing Custom Functions

Creating reusable functions is essential for efficient R programming.

6.7.1 Basic Function Definition

```
# Simple function
calculate_bmi <- function(weight_kg, height_m) {
    bmi <- weight_kg / (height_m^2)
    return(bmi)
}

# Test function
my_bmi <- calculate_bmi(70, 1.75)
print(paste("BMI:", round(my_bmi, 2)))
```

```
[1] "BMI: 22.86"
```

```
# Calculate for multiple people
weights <- c(70, 85, 60)
heights <- c(1.75, 1.80, 1.65)
bmis <- mapply(calculate_bmi, weights, heights)
print(round(bmis, 2))
```

```
[1] 22.86 26.23 22.04
```

6.7.2 Function with Default Arguments

```
# Function with default parameters
greet_user <- function(name, greeting = "Hello") {
    message <- paste(greeting, name)
    return(message)
```

```
}
# Test function
print(greet_user("Alice"))                    # Uses default greeting
```

```
[1] "Hello Alice"
```

```
print(greet_user("Bob", greeting = "Hi")) # Custom greeting
```

```
[1] "Hi Bob"
```

6.7.3 Function with Variable Number of Arguments

```
# Function that accepts variable number of arguments
sum_numbers <- function(...) {
    numbers <- c(...)
    total <- sum(numbers, na.rm = TRUE)
    return(total)
}
# Test function
print(sum_numbers(1, 2, 3, 4, 5))
```

```
[1] 15
```

```
print(sum_numbers(10, 20, NA, 30))
```

```
[1] 60
```

7 datatype and structure exercise

8 Exercise 1

Install and load the following packages

{tidyverse} {admiral} {dplyr} {tidyr} {admiral.test}

```
#installing the packages
install.packages(c("tidyverse", "admiral", "dplyr", "tidyr"))

library(tidyverse)
library(admiral)
library(admiral.test)
library(dplyr)
library(tidyr)
```

9 Exercise 2

Import `adsl.sas7bdat` as adsl

```
library(haven)
adsl <-  read_sas("data/adsl.sas7bdat")
```

Part II

data manipulation

10 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

```
1 + 1
```

```
[1] 2
```

11 Summary

In summary, this book has no content whatsoever.

```r
1 + 1
```

```
[1] 2
```

References

Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.