A Comprehensive Guide to R Programming

Packages, Functions, and Best Practices

Kamarul Imran Musa

2025-08-03

Table of contents

1 R Packages **3**
1.1 Latest Updates on CRAN . . . 3
1.2 The Tidyverse Approach . . . 3
1.2.1 Core Principles . . . 3
1.2.2 Key Tidyverse Packages . . . 3
1.2.3 Modern Data Manipulation with dplyr . . . 4
1.2.4 Advanced dplyr Features . . . 4

2 R TaskView **6**
2.1 What is R TaskView? . . . 6
2.2 Major TaskView Categories . . . 6
2.2.1 Statistical Methods . . . 6
2.2.2 Data Science & Machine Learning . . . 6
2.2.3 Specialized Domains . . . 6
2.2.4 Data Import/Export . . . 7
2.2.5 Visualization . . . 7

3 R Functions **7**
3.1 What is a Function in R? . . . 7
3.2 Function Structure . . . 7
3.3 Arguments and Parameters . . . 8
3.3.1 Types of Arguments . . . 8
3.3.2 Parameter Matching . . . 9
3.4 Getting Help: Finding Packages and Functions . . . 9
3.4.1 Built-in Help System . . . 9
3.4.2 Finding Functions for Specific Tasks . . . 10

4 Writing R Scripts **10**
4.1 Tips for Writing Good Functions 10
4.1.1 1. Follow the Single Responsibility Principle 10
4.1.2 2. Use Descriptive Names 10
4.1.3 3. Include Input Validation 11
4.1.4 4. Document Your Functions 11
4.2 Tips for Writing R Scripts 12
4.2.1 1. Script Organization 12
4.2.2 3. Error Handling 12
4.3 Don't Repeat Yourself (DRY) Principles 13
4.3.1 Example 1: Repetitive Data Cleaning 13
4.3.2 Example 2: Repetitive Plotting 14
4.3.3 Example 3: Repetitive Summary Statistics 17
4.4 Conclusion 18

1 R Packages

1.1 Latest Updates on CRAN

The Comprehensive R Archive Network (CRAN) continues to grow rapidly, with thousands of packages available for various statistical and data science applications. As of 2025, CRAN hosts over 20,000 packages, making R one of the most comprehensive statistical computing environments.

Recent trends in R package development include:

- **Enhanced performance**: Many packages now leverage C++ integration through Rcpp
- **Improved user experience**: Better documentation, vignettes, and error messages
- **Cloud integration**: Packages for working with cloud platforms and big data
- **Machine learning focus**: Continued expansion of ML and AI-related packages

1.2 The Tidyverse Approach

The tidyverse represents a modern approach to data science in R, emphasizing:

1.2.1 Core Principles

1. **Tidy data**: Each variable forms a column, each observation forms a row
2. **Functional programming**: Functions should be predictable and side-effect free
3. **Human-centered design**: APIs designed for humans, not computers
4. **Consistency**: Similar functions work in similar ways

1.2.2 Key Tidyverse Packages

```r
# Install the complete tidyverse
# install.packages("tidyverse")

# Core packages included:
library(dplyr)    # Data manipulation
library(ggplot2)  # Data visualization
library(tidyr)    # Data tidying
library(readr)    # Data import
library(purrr)    # Functional programming
library(tibble)   # Modern data frames
library(stringr)  # String manipulation
```

```r
library(forcats)  # Factor handling
library(nycflights13)
```

1.2.3 Modern Data Manipulation with dplyr

The dplyr package provides a grammar of data manipulation with five key verbs:

```r
# Example using starwars dataset
library(dplyr)

starwars %>%
  filter(species == "Human") %>%
  select(name, height, mass, homeworld) %>%
  mutate(bmi = mass / (height/100)^2) %>%
  arrange(desc(bmi)) %>%
  slice_head(n = 5)
```

```
# A tibble: 5 x 5
  name               height  mass homeworld   bmi
  <chr>               <int> <dbl> <chr>     <dbl>
1 Owen Lars             178   120 Tatooine   37.9
2 Darth Vader           202   136 Tatooine   33.3
3 Beru Whitesun Lars    165    75 Tatooine   27.5
4 Wedge Antilles        170    77 Corellia   26.6
5 Luke Skywalker        172    77 Tatooine   26.0
```

1.2.4 Advanced dplyr Features

1.2.4.1 Row-wise Operations

```r
# Computing row-wise statistics
df <- tibble(x = 1:3, y = 3:5, z = 5:7)

df %>%
  rowwise() %>%
  mutate(row_mean = mean(c(x, y, z)))
```

```
# A tibble: 3 x 4
# Rowwise:
```

```
      x     y     z row_mean
  <int> <int> <int>    <dbl>
1     1     3     5        3
2     2     4     6        4
3     3     5     7        5
```

1.2.4.2 Column-wise Operations with across()

```
# Apply functions across multiple columns
starwars %>%
  summarise(across(where(is.numeric),
                   list(mean = ~mean(.x, na.rm = TRUE),
                        sd = ~sd(.x, na.rm = TRUE))))
```

```
# A tibble: 1 x 6
  height_mean height_sd mass_mean mass_sd birth_year_mean birth_year_sd
        <dbl>     <dbl>     <dbl>   <dbl>           <dbl>         <dbl>
1        175.      34.8      97.3    169.            87.6          155.
```

1.2.4.3 Two-table Operations

```
# Joining datasets
flights %>%
  left_join(airlines, by = "carrier") %>%
  left_join(weather, by = c("year", "month", "day", "hour", "origin"))
```

```
# A tibble: 336,776 x 30
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 336,766 more rows
```

```
# i 22 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour.x <dttm>, name <chr>, temp <dbl>,
#   dewp <dbl>, humid <dbl>, wind_dir <dbl>, wind_speed <dbl>, wind_gust <dbl>,
#   precip <dbl>, pressure <dbl>, visib <dbl>, time_hour.y <dttm>
```

2 R TaskView

2.1 What is R TaskView?

R TaskView is a curated collection of packages organized by subject area, maintained by domain experts. These views help users navigate the vast ecosystem of R packages by providing structured recommendations for specific analytical domains.

2.2 Major TaskView Categories

2.2.1 Statistical Methods

- **Bayesian**: Stan, brms, MCMCpack, rjags
- **Survival Analysis**: survival, survminer, flexsurv
- **Time Series**: forecast, ts, xts, zoo
- **Multivariate Statistics**: vegan, FactoMineR, cluster

2.2.2 Data Science & Machine Learning

- **Machine Learning**: caret, randomForest, e1071, nnet
- **Deep Learning**: torch, tensorflow, keras
- **Text Mining**: tm, quanteda, tidytext
- **Natural Language Processing**: openNLP, spacyr

2.2.3 Specialized Domains

- **Epidemiology**: epiR, epitools, EpiModel
- **Econometrics**: AER, plm, systemfit
- **Bioinformatics**: Bioconductor ecosystem
- **Spatial Analysis**: sf, sp, raster, leaflet

2.2.4 Data Import/Export

```r
# Common data import packages
library(readr)      # CSV, TSV files
library(readxl)     # Excel files
library(haven)      # SPSS, Stata, SAS
library(DBI)        # Databases
library(jsonlite)   # JSON data
library(xml2)       # XML data
```

2.2.5 Visualization

```r
# Visualization ecosystem
library(ggplot2)     # Grammar of graphics
library(plotly)      # Interactive plots
library(leaflet)     # Interactive maps
library(DT)          # Interactive tables
library(htmlwidgets) # Web-based widgets
```

3 R Functions

3.1 What is a Function in R?

A function in R is a set of statements organized together to perform a specific task. Functions are fundamental building blocks that allow you to:

- Avoid code repetition
- Make code more readable and maintainable
- Create reusable components
- Organize complex analyses

3.2 Function Structure

```r
function_name <- function(argument1, argument2 = default_value) {
  # Function body
  result <- some_computation(argument1, argument2)
```

```r
  return(result)  # Optional - R returns last expression
}
```

3.3 Arguments and Parameters

3.3.1 Types of Arguments

1. **Required arguments**: Must be provided by the user
2. **Optional arguments**: Have default values
3. **... (dots)**: Accept variable number of arguments

```r
# Example function with different argument types
calculate_stats <- function(x, na.rm = FALSE, ...) {
  if (na.rm) {
    x <- x[!is.na(x)]
  }

  list(
    mean = mean(x, ...),
    median = median(x, ...),
    sd = sd(x, ...)
  )
}

# Usage
data <- c(1, 2, 3, NA, 5)
calculate_stats(data, na.rm = TRUE)
```

```
$mean
[1] 2.75

$median
[1] 2.5

$sd
[1] 1.707825
```

3.3.2 Parameter Matching

R matches arguments in three ways:

1. **Exact matching**: Argument names match exactly
2. **Partial matching**: Argument names are partially matched
3. **Positional matching**: Arguments matched by position

```
# All equivalent calls
mean(x = c(1, 2, 3), na.rm = TRUE)
```

```
[1] 2
```

```
mean(c(1, 2, 3), na.rm = TRUE)
```

```
[1] 2
```

```
mean(c(1, 2, 3), na = TRUE)  # Partial matching
```

```
[1] 2
```

3.4 Getting Help: Finding Packages and Functions

3.4.1 Built-in Help System

```
# Get help for a function
?mean
help(mean)

# Search for functions
??regression  # Fuzzy search
help.search("regression")

# View package documentation
help(package = "dplyr")

# View vignettes
vignette("dplyr")
browseVignettes("dplyr")
```

3.4.2 Finding Functions for Specific Tasks

1. **CRAN Task Views**: Organized by domain
2. **RSeek.org**: Specialized R search engine
3. **Stack Overflow**: Community-driven solutions
4. **R Documentation sites**: rdocumentation.org, rdrr.io
5. **Package websites**: Often hosted on GitHub or pkgdown sites

4 Writing R Scripts

4.1 Tips for Writing Good Functions

4.1.1 1. Follow the Single Responsibility Principle

```r
# Good: Function does one thing well
calculate_bmi <- function(weight_kg, height_m) {
  bmi <- weight_kg / (height_m^2)
  return(bmi)
}

# Bad: Function tries to do too many things
calculate_everything <- function(weight_kg, height_m) {
  bmi <- weight_kg / (height_m^2)
  category <- if (bmi < 18.5) "Underweight" else "Normal"
  plot(weight_kg, height_m)  # Side effect!
  return(list(bmi = bmi, category = category))
}
```

4.1.2 2. Use Descriptive Names

```r
# Good
calculate_confidence_interval <- function(data, confidence_level = 0.95) {
  # Function implementation
}

# Bad
ci_calc <- function(d, cl = 0.95) {
  # Function implementation
}
```

4.1.3 3. Include Input Validation

```r
safe_divide <- function(x, y) {
  # Input validation
  if (!is.numeric(x) || !is.numeric(y)) {
    stop("Both x and y must be numeric")
  }

  if (y == 0) {
    warning("Division by zero, returning Inf")
    return(Inf)
  }

  return(x / y)
}
```

4.1.4 4. Document Your Functions

```r
#' Calculate Body Mass Index
#'
#' This function calculates BMI from weight and height measurements
#'
#' @param weight_kg Numeric vector of weights in kilograms
#' @param height_m Numeric vector of heights in meters
#' @return Numeric vector of BMI values
#' @examples
#' calculate_bmi(70, 1.75)
#' calculate_bmi(c(70, 80), c(1.75, 1.80))
#' @export
calculate_bmi <- function(weight_kg, height_m) {
  if (length(weight_kg) != length(height_m)) {
    stop("weight_kg and height_m must have the same length")
  }

  bmi <- weight_kg / (height_m^2)
  return(bmi)
}
```

4.2 Tips for Writing R Scripts

4.2.1 1. Script Organization

```r
# Header with script information
# Title: Data Analysis Pipeline
# Author: Your Name
# Date: 2025-01-01
# Purpose: Analyze survey data and generate report

# Load required packages
library(tidyverse)
library(here)
library(scales)

# Set global options
options(stringsAsFactors = FALSE)

# Define constants
SIGNIFICANCE_LEVEL <- 0.05
OUTPUT_DIR <- here("output")

# Source custom functions
# source(here("R", "helper_functions.R"))

# Main analysis code...
```

4.2.2 3. Error Handling

```r
# Robust data reading
read_data_safely <- function(file_path) {
  tryCatch({
    data <- read_csv(file_path)
    message(paste("Successfully loaded", nrow(data), "rows"))
    return(data)
  }, error = function(e) {
    stop(paste("Failed to read file:", file_path, "\nError:", e$message))
  })
}
```

4.3 Don't Repeat Yourself (DRY) Principles

```
data <- read_csv('data.csv')
```

4.3.1 Example 1: Repetitive Data Cleaning

4.3.1.1 Before (Violates DRY):

```
# Bad: Repetitive code
data$age[data$age < 0] <- NA
data$age[data$age > 120] <- NA

data$income[data$income < 0] <- NA
data$income[data$income > 1000000] <- NA

data$height[data$height < 50] <- NA
data$height[data$height > 250] <- NA
```

4.3.1.2 After (Follows DRY):

```
# Good: Reusable function
clean_outliers <- function(x, min_val, max_val) {
  x[x < min_val | x > max_val] <- NA
  return(x)
}

# Usage
data <- data %>%
  mutate(
    age = clean_outliers(age, 0, 120),
    income = clean_outliers(income, 0, 1000000),
    height = clean_outliers(height, 50, 250)
  )

# Even better: Use across() for multiple columns
outlier_rules <- list(
  age = c(0, 120),
  income = c(0, 1000000),
  height = c(50, 250)
)
```

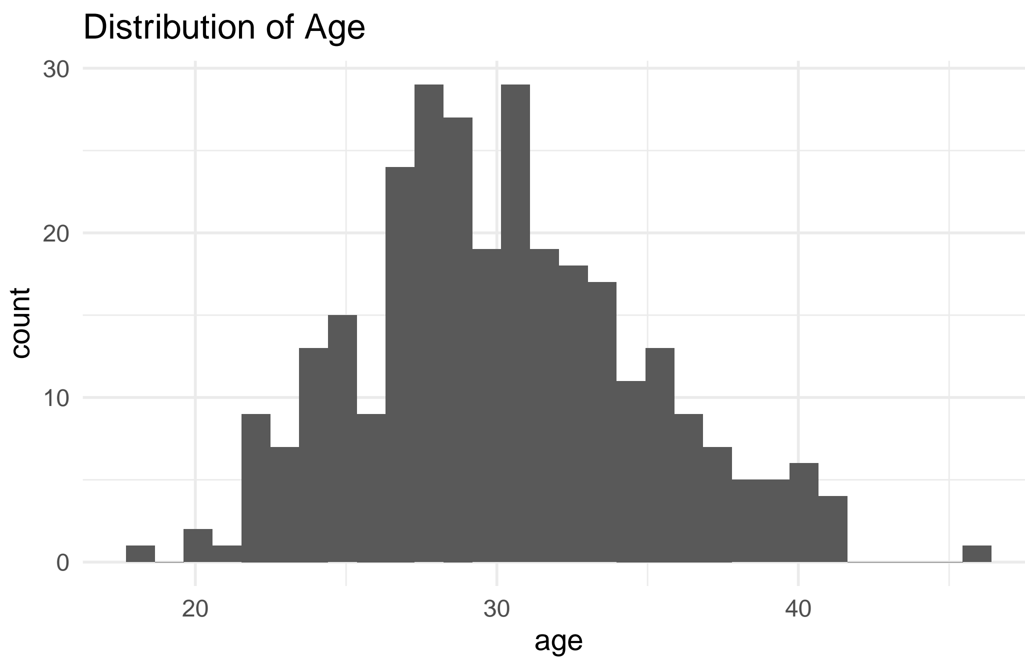
```
clean_multiple_outliers <- function(data, rules) {
  for (col in names(rules)) {
    if (col %in% names(data)) {
      data[[col]] <- clean_outliers(data[[col]], rules[[col]][1], rules[[col]][2])
    }
  }
  return(data)
}

data_clean <- clean_multiple_outliers(data, outlier_rules)
```

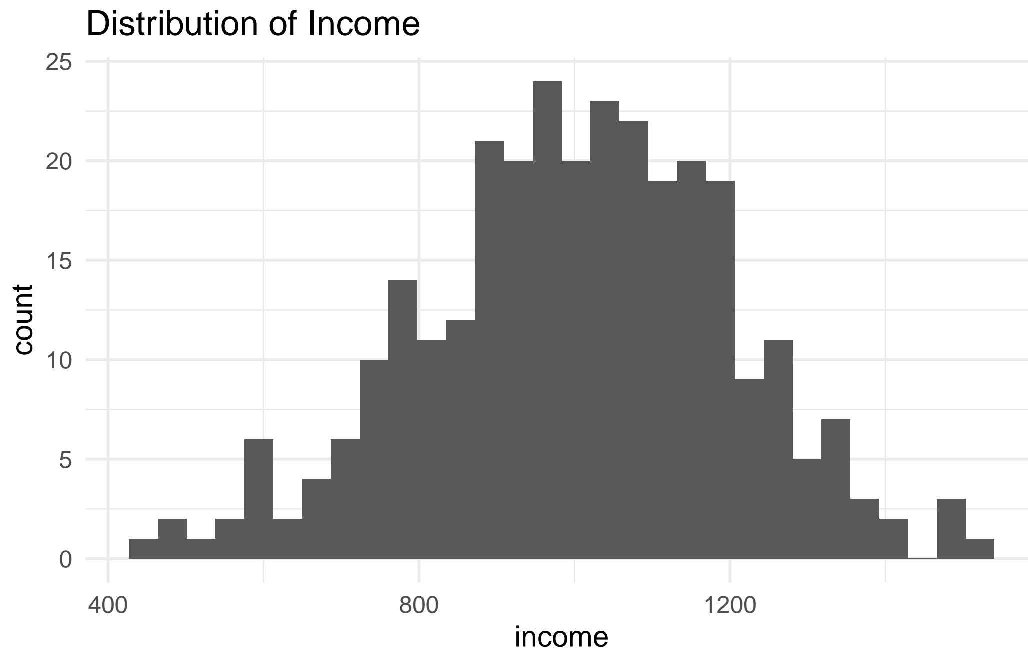
4.3.2 Example 2: Repetitive Plotting

4.3.2.1 Before (Violates DRY):

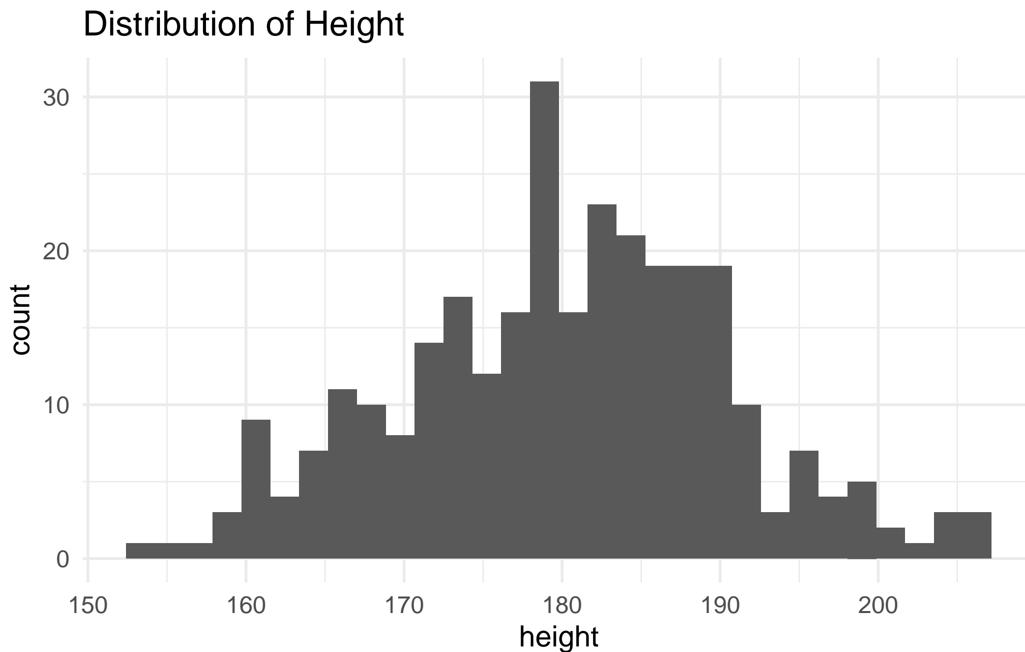
```
# Bad: Repetitive plotting code
ggplot(data, aes(x = age)) +
  geom_histogram(bins = 30) +
  theme_minimal() +
  labs(title = "Distribution of Age")
```



```r
ggplot(data, aes(x = income)) +
  geom_histogram(bins = 30) +
  theme_minimal() +
  labs(title = "Distribution of Income")
```




```r
ggplot(data, aes(x = height)) +
  geom_histogram(bins = 30) +
  theme_minimal() +
  labs(title = "Distribution of Height")
```



4.3.2.2 After (Follows DRY):

```
# Good: Reusable plotting function
create_histogram <- function(data, variable, bins = 30) {
  ggplot(data, aes(x = .data[[variable]])) +
    geom_histogram(bins = bins, fill = "steelblue", alpha = 0.7) +
    theme_minimal() +
    labs(
      title = paste("Distribution of", str_to_title(variable)),
      x = str_to_title(variable),
      y = "Count"
    )
}

# Usage
variables <- c("age", "income", "height")
plots <- map(variables, ~create_histogram(data, .x))
names(plots) <- variables

# Even better: Create all plots at once
create_histograms_for_numeric <- function(data) {
  numeric_vars <- select(data, where(is.numeric)) %>% names()
```

```
  map(numeric_vars, ~create_histogram(data, .x)) %>%
    set_names(numeric_vars)
}

all_plots <- create_histograms_for_numeric(data)
```

4.3.3 Example 3: Repetitive Summary Statistics

4.3.3.1 Before (Violates DRY):

```
# Bad: Repetitive summary calculations
age_summary <- data %>%
  summarise(
    mean = mean(age, na.rm = TRUE),
    median = median(age, na.rm = TRUE),
    sd = sd(age, na.rm = TRUE),
    min = min(age, na.rm = TRUE),
    max = max(age, na.rm = TRUE)
  )

income_summary <- data %>%
  summarise(
    mean = mean(income, na.rm = TRUE),
    median = median(income, na.rm = TRUE),
    sd = sd(income, na.rm = TRUE),
    min = min(income, na.rm = TRUE),
    max = max(income, na.rm = TRUE)
  )
```

4.3.3.2 After (Follows DRY):

```
# Good: Using across() to avoid repetition
summary_stats <- data %>%
  summarise(across(
    where(is.numeric),
    list(
      mean = ~mean(.x, na.rm = TRUE),
      median = ~median(.x, na.rm = TRUE),
      sd = ~sd(.x, na.rm = TRUE),
      min = ~min(.x, na.rm = TRUE),
```

```r
      max = ~max(.x, na.rm = TRUE)
    ),
    .names = "{.col}_{.fn}"
  ))

# Or create a reusable function
get_summary_stats <- function(data, variables = NULL) {
  if (is.null(variables)) {
    variables <- select(data, where(is.numeric)) %>% names()
  }

  data %>%
    select(all_of(variables)) %>%
    summarise(across(
      everything(),
      list(
        n = ~sum(!is.na(.x)),
        mean = ~mean(.x, na.rm = TRUE),
        median = ~median(.x, na.rm = TRUE),
        sd = ~sd(.x, na.rm = TRUE),
        min = ~min(.x, na.rm = TRUE),
        max = ~max(.x, na.rm = TRUE)
      ),
      .names = "{.col}_{.fn}"
    )) %>%
    pivot_longer(everything(), names_to = "statistic", values_to = "value") %>%
    separate(statistic, into = c("variable", "stat"), sep = "_(?=[^_]*$)")
}

# Usage
summary_table <- get_summary_stats(data, c("age", "income", "height"))
```

4.4 Conclusion

Following these principles will help you write more maintainable, readable, and efficient R code. Remember:

1. **Functions should do one thing well**
2. **Use descriptive names**
3. **Document your code**
4. **Avoid repetition through functions and vectorization**

5. **Use the tidyverse approach for consistent and readable data manipulation**
6. **Leverage R's functional programming capabilities**

The combination of good function design, proper script organization, and DRY principles will make your R code more professional and easier to maintain over time.