# 8. Control Flow and Lists

Making decisions in code and working with flexible data structures

Dr. Paul Schmidt

To install and load all packages used in this chapter, run the following code:

```r
for (pkg in c("tidyverse")) {
  if (!require(pkg, character.only = TRUE)) install.packages(pkg)
}

library(tidyverse)
```

## Introduction

Up to this point, the R code in this workshop has been largely linear: one line runs after the next, top to bottom. That works well for straightforward data wrangling, but real analysis tasks often require decisions. Should the code take path A or path B? Should it categorize an observation as “high” or “low”? Should it print a warning when something unexpected happens?

Control flow is the umbrella term for language constructs that let code make decisions and choose between alternatives. The most fundamental of these is the `if` statement. In everyday terms, every analyst already thinks in control flow: “If the p-value is below 0.05, report the result as significant; otherwise, report it as non-significant.” Translating that logic into R code is exactly what this chapter teaches.

This chapter also introduces **lists** — R’s most flexible data structure. While vectors require all elements to share the same type, lists can hold anything: numbers, strings, dataframes, even other lists. Understanding lists is essential because many R functions return their results as lists (for example, `lm()` and `t.test()`), and because lists are the backbone of iteration with `purrr::map()` in Chapter 11. Note that iterative control flow (`for` and `while` loops) is not covered here — it has its own dedicated chapter (Chapter 11).

Together, control flow and lists form the foundation for writing your own functions (Chapter 9) and iterating over data (Chapter 11).

## if and else

### Basic if

The simplest form of control flow is a single `if` statement. It tests a condition and, if the condition is `TRUE`, executes the code inside the curly braces:

```r
temp_mean <- mean(airquality$Temp)
temp_mean
```

```
[1] 77.88235
```

```r
if (temp_mean > 70) {
  message("The average temperature is above 70 degrees Fahrenheit.")
}
```

```
The average temperature is above 70 degrees Fahrenheit.
```

If the condition is `FALSE`, nothing happens — the code inside the braces is simply skipped.

## if / else

More commonly, we want to do one thing when the condition is true and something else when it is false. This is where `else` comes in:

```r
ozone_mean <- mean(airquality$Ozone, na.rm = TRUE)

if (ozone_mean > 50) {
  message(glue::glue("Mean ozone ({round(ozone_mean, 1)} ppb) is elevated."))
} else {
  message(glue::glue("Mean ozone ({round(ozone_mean, 1)} ppb) is within normal
range."))
}
```

```
Mean ozone (42.1 ppb) is within normal range.
```

Note that the `else` keyword must appear on the **same line** as the closing brace `}` of the `if` block. Placing it on the next line causes a syntax error, because R thinks the `if` statement has ended.

## if / else if / else

When there are more than two possible outcomes, additional conditions can be chained with `else if`:

```r
wind_mean <- mean(airquality$Wind)

if (wind_mean > 12) {
  category <- "Windy"
} else if (wind_mean > 8) {
  category <- "Moderate"
} else {
  category <- "Calm"
}

message(glue::glue("Average wind: {round(wind_mean, 1)} mph => {category}"))
```

```
Average wind: 10 mph => Moderate
```

R evaluates the conditions from top to bottom and enters the first branch whose condition is `TRUE`. Once a branch is entered, all remaining branches are skipped.

## Scalar Conditions: && and ||

An important detail is that `if` expects a **single** logical value — `TRUE` or `FALSE`, length 1. Passing a vector of length greater than 1 produces a warning and uses only the first element, which is almost never the intended behavior.

When combining conditions inside `if`, use the **short-circuit** operators `&&` (and) and `||` (or). These evaluate left to right and stop as soon as the result is determined. Their vectorized counterparts `&` and `|` are intended for element-wise operations on vectors and should not be used inside `if`:

```r
x <- 15

# Correct: scalar operators for if
if (x > 10 && x < 20) {
  message("x is between 10 and 20")
}
```

```
x is between 10 and 20
```

```r
# & is vectorized - works element-wise on vectors
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, FALSE)
```

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

The rule of thumb is straightforward: use `&&` / `||` inside `if()` and `&` / `|` inside vectorized operations like `filter()` or `ifelse()`.

# ifelse() and case_when()

## Vectorized Decisions with ifelse()

The `if` / `else` construct handles a single condition. But what if we need to classify every row of a dataframe? For that, R provides `ifelse()`, which is vectorized — it tests each element of a vector and returns a value for the `TRUE` case or the `FALSE` case:

```
aq <- airquality %>%
  mutate(temp_class = ifelse(Temp >= 80, "Hot", "Not hot"))

aq %>%
  count(temp_class)
```

```
  temp_class  n
1        Hot 73
2    Not hot 80
```

`ifelse()` works, but it becomes awkward when there are more than two categories, because calls must be nested:

```
aq <- airquality %>%
  mutate(
    temp_class = ifelse(Temp >= 85, "Hot",
                   ifelse(Temp >= 70, "Warm", "Cold"))
  )

aq %>%
  count(temp_class)
```

```
  temp_class  n
1       Cold 32
2        Hot 39
3       Warm 82
```

## Cleaner Multi-Category Logic with case_when()

The `dplyr::case_when()` function provides a much more readable syntax for multiple conditions. Each line contains a condition on the left side of `~` and the corresponding value on the right:

```
aq <- airquality %>%
  mutate(
    temp_class = case_when(
      Temp >= 85 ~ "Hot",
      Temp >= 70 ~ "Warm",
      .default = "Cold"
    )
  )

aq %>%
  count(temp_class)
```

```
  temp_class  n
1       Cold 32
2        Hot 39
3       Warm 82
```

Just like `if` / `else if` chains, `case_when()` evaluates conditions top to bottom and assigns the value from the first matching condition. The `.default = "Cold"` argument serves as the “else” case, catching everything that did not match an earlier condition.

> **Note**
>
> In older code, you will frequently see `TRUE ~ "Cold"` instead of `.default = "Cold"` as the catch-all. Both styles work, but `.default` is the recommended approach since dplyr 1.1.0 because it is more explicit and avoids subtle issues when `NA` conditions are involved.

## When to Use Which

The three approaches serve different purposes:

- **`if`** / **`else`**: Use for control flow decisions that affect program logic (e.g., choosing which analysis to run). Operates on a single scalar condition.
- **`ifelse()`**: Use for simple two-category vectorized operations. Can be used outside the tidyverse.
- **`case_when()`**: Use for multi-category vectorized operations inside `mutate()`. Cleaner and safer than nested `ifelse()`.

Exercise: Categorize Air Quality

Using the `airquality` dataset:

1. Create a column `temp_category` with three levels: “Cold” (below 70), “Warm” (70 to 84), and “Hot” (85 and above) using `case_when()`.
2. Create a column `wind_class` with two levels: “Breezy” (Wind >= 10) and “Calm” (Wind < 10) using `case_when()`.
3. Count the combinations of `temp_category` and `wind_class`.

Solution

```r
aq_classified <- airquality %>%
  mutate(
    temp_category = case_when(
      Temp >= 85 ~ "Hot",
      Temp >= 70 ~ "Warm",
      .default = "Cold"
    ),
    wind_class = case_when(
      Wind >= 10 ~ "Breezy",
      .default = "Calm"
    )
  )

aq_classified %>%
  count(temp_category, wind_class)
```

```
  temp_category wind_class  n
1          Cold     Breezy 22
2          Cold       Calm 10
3           Hot     Breezy  9
4           Hot       Calm 30
5          Warm     Breezy 41
6          Warm       Calm 41
```

## switch()

When a decision depends on matching a single value against several discrete options, `switch()` offers a compact alternative to long `if` / `else if` chains. It takes an expression (typically a character string) and returns the value associated with the matching case:

```r
describe_month <- function(month_num) {
  season <- switch(as.character(month_num),
    "5"  = "Spring",
    "6"  = "Early Summer",
    "7"  = "Summer",
    "8"  = "Late Summer",
    "9"  = "Early Fall",
    "Unknown season"
  )
  glue::glue("Month {month_num}: {season}")
}

describe_month(7)
```

```
Month 7: Summer
```

```r
describe_month(5)
```

```
Month 5: Spring
```

```r
describe_month(12)
```

```
Month 12: Unknown season
```

The last unnamed value (`"Unknown season"`) serves as the default, returned when no case matches. This makes `switch()` particularly useful inside functions where an argument selects between a handful of predefined options:

```r
summarize_stat <- function(x, stat = "mean") {
  switch(stat,
    "mean"   = mean(x, na.rm = TRUE),
    "median" = median(x, na.rm = TRUE),
    "sd"     = sd(x, na.rm = TRUE),
    stop(glue::glue("Unknown statistic: '{stat}'"))
  )
}

summarize_stat(airquality$Temp, "mean")
```

```
[1] 77.88235
```

```r
summarize_stat(airquality$Temp, "median")
```

```
[1] 79
```

The `stop()` call in the default position is a common pattern: it raises an informative error when the user passes an unexpected value, rather than silently returning `NULL`.

# What are Lists?

## Atomic Vectors: One Type Only

Before introducing lists, it is worth recalling how atomic vectors work. An atomic vector is a sequence of values that all share the same type:

```
num_vec <- c(1.5, 2.3, 4.1)       # double
int_vec <- 1:5                     # integer
chr_vec <- c("a", "b", "c")       # character
log_vec <- c(TRUE, FALSE, TRUE)    # logical
```

If types are mixed in a single vector, R silently coerces everything to the most general type:

```
mixed <- c(1, "two", TRUE)
mixed
```

```
[1] "1"    "two"  "TRUE"
```

```
class(mixed)
```

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

Everything became character, because character is the most general type. This “all elements must match” constraint is what makes vectors efficient, but it also limits what a single vector can hold.

## Lists: Any Type, Any Length

A list removes this constraint. Each element of a list can be a different type and a different length — a number, a character vector, an entire dataframe, or even another list:

```
my_list <- list(
  numbers = c(10, 20, 30),
  greeting = "Hello",
  flag = TRUE,
  data = head(mtcars, 3)
)

my_list
```

```
$numbers
[1] 10 20 30

$greeting
[1] "Hello"

$flag
[1] TRUE

$data
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

A helpful analogy is a **train**: each car (element) can carry completely different cargo. A vector, by contrast, is more like a conveyor belt where every item must be the same shape.

Named elements are strongly recommended because they make code self-documenting. However, unnamed lists work too:

```r
unnamed <- list(1:3, "text", FALSE)
unnamed
```

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

[[2]]
[1] "text"

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

## Accessing List Elements

Getting data out of a list is one of the most common sources of confusion for R beginners. There are three operators, and the distinction between them matters.

### [[ — Extract a Single Element

Double brackets `[[` extract a single element from the list. The result is the element itself, not a list:

```r
my_list[["numbers"]]
```

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

```r
class(my_list[["numbers"]])
```

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

Using the train analogy: `[[` opens a car and takes out the cargo.

### [ — Extract a Sub-List

Single brackets `[` extract a subset of the list. The result is always a **list**, even if it contains only one element:

```r
my_list["numbers"]
```

```
$numbers
[1] 10 20 30
```

```r
class(my_list["numbers"])
```

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

Using the train analogy: `[` detaches one or more cars but keeps them as a (smaller) train.

### $ — Shorthand for Named Elements

The dollar sign `$` is a convenient shorthand for `[[` with named elements:

```r
my_list$greeting
```

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

It is equivalent to `my_list[["greeting"]]` but requires less typing. The `$` operator also supports partial matching (e.g., `my_list$gre` would work), though relying on this is discouraged because it makes code fragile.

### The Critical Difference

The distinction between `[` and `[[` trips up many beginners. A visual comparison helps:

```r
# [[ returns the element (a numeric vector)
str(my_list[["numbers"]])
```

```
 num [1:3] 10 20 30
```

```r
# [ returns a list containing that element
str(my_list["numbers"])
```

```
List of 1
 $ numbers: num [1:3] 10 20 30
```

When you need to compute with the extracted data (e.g., take the mean of the numbers), you almost always want `[[` or `$`. When you need to pass a subset of a list to another function that expects a list, use `[`.

**Exercise: Build and Access a List**

1. Create a list called `my_data` with three named elements:
   - `measurements`: the numeric vector `c(4.2, 5.1, 3.8, 6.0, 4.7)`
   - `cars_sample`: the first 5 rows of `mtcars`
   - `note`: the character string `"Collected on day 1"`
2. Extract `measurements` using `$` and compute its mean.
3. Extract `cars_sample` using `[[` and show only the `mpg` column.
4. Use `[` to create a sub-list containing `measurements` and `note`. Verify with `class()` that the result is a list.

**Solution**

```r
# 1. Create the list
my_data <- list(
  measurements = c(4.2, 5.1, 3.8, 6.0, 4.7),
  cars_sample = head(mtcars, 5),
  note = "Collected on day 1"
)

# 2. Extract and compute
mean(my_data$measurements)
```

```
[1] 4.76
```

```r
# 3. Extract dataframe column
my_data[["cars_sample"]]$mpg
```

```
[1] 21.0 21.0 22.8 21.4 18.7
```

```r
# 4. Sub-list
sub <- my_data[c("measurements", "note")]
class(sub)
```

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

```r
str(sub)
```

```
List of 2
 $ measurements: num [1:5] 4.2 5.1 3.8 6 4.7
 $ note        : chr "Collected on day 1"
```

## Nested Lists

Lists can contain other lists, creating hierarchical structures of arbitrary depth:

```r
experiment <- list(
  metadata = list(
    researcher = "Dr. Smith",
    date = "2025-03-15"
  ),
  results = list(
    treatment_A = c(4.2, 3.8, 5.1),
    treatment_B = c(6.7, 7.2, 6.9)
  )
)
```

To access nested elements, chain the extraction operators:

```
# The researcher name
experiment[["metadata"]][["researcher"]]
```

```
[1] "Dr. Smith"
```

```
# Equivalent with $
experiment$metadata$researcher
```

```
[1] "Dr. Smith"
```

```
# The first value of treatment B
experiment$results$treatment_B[1]
```

```
[1] 6.7
```

The `str()` function is invaluable for understanding the structure of nested lists:

```
str(experiment)
```

```
List of 2
 $ metadata:List of 2
  ..$ researcher: chr "Dr. Smith"
  ..$ date      : chr "2025-03-15"
 $ results :List of 2
  ..$ treatment_A: num [1:3] 4.2 3.8 5.1
  ..$ treatment_B: num [1:3] 6.7 7.2 6.9
```

`str()` provides a compact summary showing the type and first few values of each element, indented to reflect the nesting hierarchy. For large or deeply nested lists, the `max.level` argument limits how deep `str()` descends:

```
str(experiment, max.level = 1)
```

```
List of 2
 $ metadata:List of 2
 $ results :List of 2
```

# Lists in Practice

Lists are not just a teaching concept — they appear constantly in day-to-day R work. Three common situations illustrate this.

## Model Objects are Lists

When you fit a linear model with `lm()`, the result is a list. It contains the coefficients, residuals, fitted values, and much more:

```
fit <- lm(mpg ~ wt, data = mtcars)
typeof(fit)
```

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

```
names(fit)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"         
 [5] "fitted.values" "assign"        "qr"            "df.residual"  
 [9] "xlevels"       "call"          "terms"         "model"        
```

Individual components can be extracted with `$`:

```
# Coefficients
fit$coefficients
```

```
(Intercept)          wt 
  37.285126   -5.344472 
```

```
# First 10 residuals
fit$residuals[1:10]
```

```
        Mazda RX4     Mazda RX4 Wag        Datsun 710    Hornet 4 Drive 
       -2.2826106        -0.9197704        -2.0859521         1.2973499 
Hornet Sportabout           Valiant        Duster 360         Merc 240D 
       -0.2001440        -0.6932545        -3.9053627         4.1637381 
         Merc 230          Merc 280 
        2.3499593         0.2998560 
```

The `summary()` function returns yet another list with additional computed statistics like R-squared:

```
fit_summary <- summary(fit)
fit_summary$r.squared
```

```
[1] 0.7528328
```

```
fit_summary$coefficients
```

```
             Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 37.285126   1.877627 19.857575 8.241799e-19
wt          -5.344472   0.559101 -9.559044 1.293959e-10
```

## split() Returns a Named List

The `split()` function divides a dataframe into a named list of sub-dataframes, one per group:

```
by_cyl <- split(mtcars, mtcars$cyl)
names(by_cyl)
```

```
[1] "4" "6" "8"
```

```
# The 6-cylinder cars
head(by_cyl[["6"]], 3)
```

```
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

This pattern becomes powerful in combination with `lapply()` or `purrr::map()`, which are covered in Chapter 11.

## JSON Data and APIs

When working with data from web APIs, the response is typically JSON, which R parses into a nested list structure. Understanding how to navigate lists is therefore essential for anyone working with external data sources. While this chapter does not cover JSON in detail, the skills for accessing nested list elements apply directly.

Exercise: Extract Model Output

1. Fit a linear model: `lm(mpg ~ wt + hp, data = mtcars)`.
2. Extract the R-squared value from the model summary.
3. Extract the coefficient estimates (intercept, wt, hp) as a named numeric vector.
4. Extract the first 6 residuals.
5. Combine R-squared, the three coefficient estimates, and the number of observations into a single tibble with one row.

*Hint:* The number of observations can be determined from the length of `fit$residuals`.

Solution

```r
# 1. Fit the model
fit <- lm(mpg ~ wt + hp, data = mtcars)

# 2. R-squared
fit_summary <- summary(fit)
r_sq <- fit_summary$r.squared
r_sq
```

```
[1] 0.8267855
```

```r
# 3. Coefficients
coefs <- fit$coefficients
coefs
```

```
(Intercept)          wt          hp 
37.22727012 -3.87783074 -0.03177295 
```

```r
# 4. First 6 residuals
fit$residuals[1:6]
```

```
        Mazda RX4     Mazda RX4 Wag        Datsun 710    Hornet 4 Drive 
       -2.5723294        -1.5834826        -2.4758187         0.1349799 
Hornet Sportabout           Valiant 
        0.3727334        -2.3738163 
```

```r
# 5. Summary tibble
tibble(
  r_squared = r_sq,
  intercept = coefs[["(Intercept)"]],
  coef_wt   = coefs[["wt"]],
  coef_hp   = coefs[["hp"]],
  n_obs     = length(fit$residuals)
)
```

```
# A tibble: 1 × 5
  r_squared intercept coef_wt coef_hp n_obs
      <dbl>     <dbl>   <dbl>   <dbl> <int>
1     0.827      37.2   -3.88 -0.0318    32
```

## From Lists to Dataframes

In many workflows, the final step after working with lists is converting the results into a tidy dataframe. Several approaches exist depending on the structure of the list.

For simple lists where each element is a scalar or a vector of equal length, direct conversion works:

```
results <- list(
  group = c("A", "B", "C"),
  mean  = c(4.2, 5.8, 3.1),
  sd    = c(0.9, 1.2, 0.7)
)

as_tibble(results)
```

```
# A tibble: 3 × 3
  group  mean    sd
  <chr> <dbl> <dbl>
1 A       4.2   0.9
2 B       5.8   1.2
3 C       3.1   0.7
```

For lists of dataframes (like the output of `split()`), `bind_rows()` from dplyr combines them back into a single dataframe. The `.id` argument creates a column that records which list element each row came from:

```
by_cyl <- split(mtcars, mtcars$cyl)

bind_rows(by_cyl, .id = "cyl") %>%
  head(6)
```

```
                mpg cyl  disp hp drat    wt  qsec vs  am gear carb
Datsun 710     22.8   4 108.0 93 3.85 2.320 18.61  1   1    4    1
Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1   0    4    2
Merc 230       22.8   4 140.8 95 3.92 3.150 22.90  1   0    4    2
Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1   1    4    1
Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1   1    4    2
Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1   1    4    1
```

These conversions are a recurring pattern in the workflow introduced in Chapter 9 (writing functions that return lists) and Chapter 11 (mapping functions over many inputs and collecting the results).

## Summary

This chapter introduced two foundational R concepts: control flow for making decisions in code, and lists for storing heterogeneous data.

ℹ Key Takeaways

1. **Conditional Logic:** `if` / `else` for scalar decisions, `ifelse()` for simple vectorized decisions, `case_when()` for multi-category vectorized decisions, and `switch()` for matching against discrete values.
2. **Lists:** R's most flexible data structure — each element can hold a different type and length. Created with `list()`, accessed with `$`, `[[`, or `[`.
3. **Lists in Practice:** Many R functions return lists (e.g., `lm()`, `t.test()`). Understanding list access with `$` and `[[` is essential for extracting model results.
4. **Lists to Dataframes:** `as_tibble()` converts simple lists, `bind_rows(.id = )` combines lists of dataframes.
5. **Next Steps:**
   - Chapter 9: Writing Functions — packaging control flow and list construction into reusable units.
   - Chapter 11: Iteration — applying functions to lists of inputs with `for` loops and `purrr::map()`.

## Bibliography