

Week 11: Making Choices

Ellen Bledsoe

2025-04-15

Making Choices

Conditionals

When we want to make choices in R, we are most often using some type of conditional statement.

We talked about conditional statements early in the class when we talked about conditional sub-setting, and we've continued to use them frequently throughout the course in our `filter` functions!

Here are some examples below.

```
weight > 50  
species == "DM"
```

When we run these lines of code, these statements produce an output that is a generate a "logical" data type.

- TRUE if the condition is satisfied
- FALSE if the condition is not satisfied

As a reminder, these are not the same as strings "TRUE" and "FALSE." These are a special type of value called "boolean."

In some ways, they are similar to NA values. True NA values do not go in quotation marks because they are a special data type; the same is true for TRUE (or T) and FALSE (F).

Conditional Operators

As we've seen in previous classes, conditional statements can be created with a number of different operators.

We have the "traditional" operators:

- == for equals
- != for not equals
- < and > for less than or greater than
- <=, >= for less than or equal to or greater than or equal to

For example:

```
10 >= 5
```

```
## [1] TRUE
```

There are other operators that we have not yet used, such as `%in%`. This is a very handy little operator! It checks to see if a value is present in a vector of possible values.

```
"DM" %in% c("DM", "DO", "DS")
```

```
## [1] TRUE
```

```
"PP" %in% c("DM", "DO", "DS")
```

```
## [1] FALSE
```

There are also a number of functions that return these boolean data types. For example, the `is.na()` function returns `TRUE` or `FALSE` depending on whether a value is null or not.

```
is.na(NA)
```

```
## [1] TRUE
```

```
is.na(5)
```

```
## [1] FALSE
```

Reversing Conditions

When we want the opposite of a certain condition, we can use the `!` operator to negate the condition. In doing so, it reverses the boolean values that are returned (`TRUE` becomes `FALSE` and `FALSE` becomes `TRUE`).

```
!(10 >= 5)
```

```
## [1] FALSE
```

```
!is.na(c(4, NA, 76))
```

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

Work on Question 1: Choice Operators

Testing One Condition with an if Statement

Conditional statements generate logical values. One type of statement that we can write that uses conditional statements are `if` statements. We can use these to control the flow of our program.

Note that the conditional statement to be tested goes inside a parenthetical. Then, whatever action is to be taken if that condition is met goes inside curly braces (`{}`), similar to how we write a function.

```
# if (conditional statement is TRUE) {
#   do something
# }
```

For example, let's say we have a value, `x`. If `x` is 5 or less, it can stay the way it is. However, if `x` is more than 5, we want to double it.:

```
# create object
x <- 6

# if statement
if (x > 5){
  x <- x * 2
}

# check output
x
```

```
## [1] 12
```

What happens when the conditional statement is `FALSE`?

```
x <- 4
if (x > 5){
  x <- x * 2
}
x
```

```
## [1] 4
```

As a reminder, this `if` statement is *not* a function, so everything that happens in the `if` statement influences the global environment.

Let's look at another example, this time with some character values.

Perhaps different vegetation types require different biomass calculations.

```
# set objects
veg_type <- "shrub"
volume <- 16.08

if (veg_type == "shrub") {
  mass <- 2.65 * volume^0.9
}

mass
```

```
## [1] 32.27775
```

Work on Question 2a

Testing Multiple Conditions with if-else Statements

Often, we want to choose one of several options rather than just one.

To do this, we can add more conditions and associated actions with `else if`

Let's take a look at an example.

```
# set objects
veg_type <- "grass"
volume <- 16.08

if (veg_type == "shrub") {
  mass <- 2.65 * volume^0.9
} else if (veg_type == "grass") {
  mass <- 0.65 * volume^1.2
}

mass
```

```
## [1] 18.21615
```

The code above checks the first condition. If `TRUE`, it will run that condition's code and skip the rest. If `FALSE`, it will check the next one and keep going until it runs out of conditions to check.

We can also specify what should happen if none of the conditions are `TRUE` using `else` on its own.

```
veg_type <- "tree"
volume <- 16.08

if (veg_type == "shrub") {
  mass <- 2.65 * volume^0.9
} else if (veg_type == "grass") {
  mass <- 0.65 * volume^1.2
} else {
  mass <- NA
}

mass
```

```
## [1] NA
```

Work on Question 2b-2c

Conditionals Inside Functions

So far, we've used a conditional to estimate biomass for different types of vegetation.

This seems like code we would want to reuse, so it would be a good idea to create a function to do this.

We can place our entire `if-else` statement inside a function, making sure that the function takes all required variables as arguments.

```

est_biomass <- function(volume, veg_type){
  if (veg_type == "shrub") {
    mass <- 2.65 * volume^0.9
  } else if (veg_type == "grass") {
    mass <- 0.65 * volume^1.2
  } else {
    mass <- NA
  }
  return(mass)
}

```

Now, we can run this function with different vegetation types and get different estimates for mass.

```
est_biomass(1.6, "shrub")
```

```
## [1] 4.045329
```

```
est_biomass(1.6, "grass")
```

```
## [1] 1.142503
```

```
est_biomass(24, "tree")
```

```
## [1] NA
```

Work on Question 2d.

Compare How Multiple Conditions Work

Let's compare multiple if statements to if-else if statements.

Multiple if statements are going to check each conditional separately. The code will execute for all conditions which are true.

```

x <- 5

if (x > 2){
  x * 2
}

```

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

```

if (x > 4){
  x * 4
}

```

```
## [1] 20
```

```
x
```

```
## [1] 5
```

On the other hand, an `else if` statement is going to check each condition sequentially. It will only execute code for the first condition which is `TRUE`.

```
x <- 5

if (x > 2){
  x * 2
} else if (x > 4){
  x * 4
}
```

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

Nested Conditionals

Sometimes decisions are complicated and need more than one series of conditional statements.

For example, we might have different equations for some vegetation types based on the age of the plant.

In order to do this, we can “nest” conditionals inside of one another.

```
est_mass <- function(volume, veg_type, age){
  if (veg_type == "shrub") {
    if (age < 5) {
      mass <- 1.6 * volume^0.8
    } else {
      mass <- 2.65 * volume^0.9
    }
  } else if (veg_type == "grass" | veg_type == "sedge") {
    mass <- 0.65 * volume^1.2
  } else {
    mass <- NA
  }
  return(mass)
}

est_mass(1.6, "shrub", age = 2)
```

```
## [1] 2.330322
```

```
est_mass(1.6, "shrub", age = 6)
```

```
## [1] 4.045329
```

```
est_mass(24, "grass")
```

```
## [1] 29.45553
```

This function first checks if the vegetation type is “shrub”.

If it is, it then checks to see if it is < 5 years old. If so, it will do the specified calculation. If not, it move on to the next calculation.

Spend some time working through Questions 3 and 4.

Making Choices in the tidyverse

All of the statements we have been working with have been outside of the **tidyverse**.

The **tidyverse** (**dplyr** in particular) has functions that we can use to accomplish the same tasks.

```
library(dplyr)
```

```
##  
## Attaching package: 'dplyr'  
  
## The following objects are masked from 'package:stats':  
##  
##   filter, lag  
  
## The following objects are masked from 'package:base':  
##  
##   intersect, setdiff, setequal, union
```

When the choice is binary (choose one of two outcomes), we can use the **if_else** function.

The arguments for an **if_else** function are:

- the conditional statement
- the outcome if the statement is TRUE
- the outcome if the statement is FALSE

```
x <- 7  
  
# if x is > 5, multiply x by 4. If not, multiply by 10  
if_else(x > 5, x * 4, x * 10)
```

```
## [1] 28
```

We can also nest the **if_else** functions.

```
veg_type <- "grass"  
  
if_else(veg_type == "grass", "small",  
        if_else(veg_type == "shrub", "medium", "large"))  
  
## [1] "small"
```

When we have situations where we would need to nest `if_else` statements, these nested statements can get pretty hard to interpret.

In order to avoid these nested `if_else` statements, we can use the `case_when` function.

The `case_when` function uses the `~` to set expressions.

```
veg_type <- "tree"

case_when(veg_type == "grass" ~ "small",
          veg_type == "shrub" ~ "medium",
          veg_type == "tree" ~ "large")
```

```
## [1] "large"
```

In a circumstance where you want something similar to the `else` statement (meaning, if none of these conditions are met, do something else), we have to do something rather unique in the `case_when` function.

Essentially, we have to force a `TRUE` outcome, which we do by literally using `TRUE` as the final option.

```
veg_type <- "forb"

case_when(veg_type == "grass" ~ "small",
          veg_type == "shrub" ~ "medium",
          veg_type == "tree" ~ "large",
          TRUE ~ NA)
```

```
## [1] NA
```

Using `if_else` and `case_when` with `mutate`

One of the nicest aspects of the `if_else` and `case_when` functions is that we can use them within `mutate` functions to make new columns (or edit existing columns) in our dataframes.

Let's look at an example using a dataframe from a package called `lterdatasampler`. This package has a number of cleaned datasets from Long Term Ecological Research (LTER) sites around the US.

```
#install.packages("lterdatasampler")
library(lterdatasampler)
```

We are going to use the `and_vertebrales` dataset, which contains data about trout and salamanders from Andrews Forest LTER.

```
and_vertebrales
```

```
## # A tibble: 32,209 x 16
##   year sitecode section reach pass unitnum unittype vert_index pitnumber
##   <dbl> <chr>    <chr>  <chr> <dbl> <dbl> <chr>      <dbl>    <dbl>
## 1 1987 MACKCC-L CC      L      1      1 R          1      NA
## 2 1987 MACKCC-L CC      L      1      1 R          2      NA
## 3 1987 MACKCC-L CC      L      1      1 R          3      NA
## 4 1987 MACKCC-L CC      L      1      1 R          4      NA
## 5 1987 MACKCC-L CC      L      1      1 R          5      NA
```



```
## 6 1987 MACKCC-L CC      L      1      1 R      6      NA
## 7 1987 MACKCC-L CC      L      1      1 R      7      NA
## 8 1987 MACKCC-L CC      L      1      1 R      8      NA
## 9 1987 MACKCC-L CC      L      1      1 R      9      NA
## 10 1987 MACKCC-L CC      L      1      1 R     10      NA
## # i 32,199 more rows
## # i 7 more variables: species <chr>, length_1_mm <dbl>, length_2_mm <dbl>,
## #   weight_g <dbl>, clip <chr>, sampleddate <date>, notes <chr>
```

The `section` column denotes whether the data were collected in clear cut forest (“CC”) or old growth forest (“OG”).

Let’s make a new column in the data frame that has “Clear Cut” when the value is “CC”, “Old Growth” when the value is “OG”, and NA if there is something else.

```
and_vertebrates %>%
  mutate(ForestType = case_when(section == "CC" ~ "Clear Cut",
                                section == "OG" ~ "Old Growth",
                                TRUE ~ NA))
```

```
## # A tibble: 32,209 x 17
##   year sitecode section reach  pass unitnum unittype vert_index pitnumber
##   <dbl> <chr>    <chr>  <chr> <dbl>  <dbl> <chr>      <dbl>    <dbl>
## 1 1987 MACKCC-L CC      L      1      1 R      1      NA
## 2 1987 MACKCC-L CC      L      1      1 R      2      NA
## 3 1987 MACKCC-L CC      L      1      1 R      3      NA
## 4 1987 MACKCC-L CC      L      1      1 R      4      NA
## 5 1987 MACKCC-L CC      L      1      1 R      5      NA
## 6 1987 MACKCC-L CC      L      1      1 R      6      NA
## 7 1987 MACKCC-L CC      L      1      1 R      7      NA
## 8 1987 MACKCC-L CC      L      1      1 R      8      NA
## 9 1987 MACKCC-L CC      L      1      1 R      9      NA
## 10 1987 MACKCC-L CC      L      1      1 R     10      NA
## # i 32,199 more rows
## # i 8 more variables: species <chr>, length_1_mm <dbl>, length_2_mm <dbl>,
## #   weight_g <dbl>, clip <chr>, sampleddate <date>, notes <chr>,
## #   ForestType <chr>
```

Why if-else if?

If `dplyr` has such nice functions for making choices, why did I teach you about the seemingly more complicated `if-else if` structures?

There are a few reasons.

- 1) In setting up `if_else` or `case_when` functions, it is helpful to know what is happening in the background.
- 2) When you see other people’s code, you are very likely to come across the `if-else if` structure, and you should know what it means and how to use it.
- 3) Finally, while `if_else` and `case_when` are really handy for somewhat straight-forward decisions, sometimes you need to perform a lot of tasks if a condition is true or false. It is much easier to add in multiple tasks to do in the `if-else if` structure than in the `tidyverse` structure.