Coding Lab: If statements and conditionals

Ari Anisfeld

Summer 2020

Conditional statements (control flow 1)

We often want to our code to do something depending on the context. We start with "if" statements.

```
if (condition is true) {
  do this
} else {
  do this other thing
}
```

In this lesson, we'll

- review logical operators and comparing vectors
- introduce `if` and `else` statements
- introduce vectorized if with `ifelse` in tibbles

Review: Logical Operators

The logical operators are AND (&), OR (|), and NOT (!). What happens when we use them on booleans?

Let's start with NOT (!).

```
!TRUE
```

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

```
!FALSE
```

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

Review: Logical Operators

Replace the conditional statements

```
!(2 > 1)
```

Review: Logical Operators

Replace the conditional statements

```
!(2 > 1)
```

```
!TRUE
```

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

What does this produce?

```
# NOT (0 does not equal 0)
!(0 != 0)
```

What does this produce?

```
# NOT (0 does not equal 0)
!(0 != 0)
```

```
!FALSE
```

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

Review: Logical OR

OR returns `TRUE` if at least one term is `TRUE`.

```
TRUE | FALSE
```

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

```
FALSE | FALSE
```

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

Notice that Logical OR has a different meaning than "or" the conjunction has in common English.

Review: Logical OR

```
(5 > 7) | (10 == 10)
```

Review: Logical OR

Recall == is the logical comparison for if two things are equal.

```
# 5 is greater than 7 OR 10 equals 10"
(5 > 7) | (10 == 10)
```

```
FALSE | TRUE
```

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

Finally, AND (&)

Returns `TRUE` when both operands are `TRUE`

```
TRUE & FALSE
```

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

```
TRUE & TRUE
```

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

This one is harder...

```
!(2 > 6) & (4 > 9 | 3 == 3)
```

This one is harder...

```
!(2 > 6) & (4 > 9 | 3 == 3)
```

Break it down:

```
# Start with the left term
# first
2 > 6
# then
! 2 > 6
```

This one is harder...

```
!(2 > 6) & (4 > 9 | 3 == 3)
```

Break it down:

```
# Start with the left term
# first
2 > 6
```

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

```
# then
! 2 > 6
```

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

This one is harder...

```
!(2 > 6) & (4 > 9 | 3 == 3)
```

Break it down:

```
# Now try the right term
# first
4 > 9
# then
3 == 3
# so
(4 > 9 | 3 == 3)
```

This one is harder...

```
!(2 > 6) & (4 > 9 | 3 == 3)
```

Break it down:

```
# Now try the right term
# first
4 > 9
```

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

```
# then
3 == 3
```

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

```
# so
(4 > 9 | 3 == 3)
```

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

This one is harder...

```
!(2 > 6) & (4 > 9 | 3 == 3)
```

```
!(FALSE) & (FALSE | TRUE)
```

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

If statements

The general syntax of an if statement is as follows:

```
if (condition is TRUE) {
  do this
}
```

For example:

```
x <- 100

if (x > 0) {
  print("x is positive")
}
```

```
## [1] "x is positive"
```

If/else statements

Slightly more interesting, the syntax of an if else statement is as follows:

```
if (condition is TRUE) {
  do this
} else {
  do this other thing
}
```

If/else statements example:

When working on a project with others, it's sometimes helpful to set

```
if (Sys.info()[["user"]] == "arianisfeld") {
  base_path <- "~/Documents/coding_lab_examples/"
} else {
  base_path <- "~/gdrive/coding_lab_examples/"
}

data <- read_csv(paste0(base_path, "our_data.csv"))
```

1

[1] Try running `Sys.info()` in your console to understand the code a bit more deeply.

multiple tests with `if`, `else if` and `else`

```
if (condition is TRUE) {
  do this
} else if (second condition is TRUE) {
  do this other thing
} else if (third condition is TRUE) {
  do this third thing
} else {
  do a default behavior
}
```

NB: a default behavior with `else` is not necessary.

multiple tests with `if`, `else if` and `else`

Here's a cheap version of black jack.

```
my_cards <- sample(2:11, 1) +  sample(2:11, 1)
computers_cards <- sample(2:11, 1) +  sample(2:11, 1)
```

```
if (my_cards > computers_cards) {
  score <- score + 1
  print("You win")
} else if (my_cards < computers_cards) {
  score <- score - 1
  print("Better luck next time.")
} else {
  print("It's a tie")
}
```

```
## [1] "Better luck next time."
```

`if` can take a compound condition

```
if ((my_cards > computers_cards & my_cards <= 21) |
    computers_cards > 21) {
  score <- score + 1
  print("You win")
} # etc
```

As the statement gets more complex, we're more likely to make errors.

`if` is not vectorized and doesn't handle `NA`s

```
if (c(TRUE, FALSE)) { print("if true") }
#> [1] "if true"
#> Warning in if (c(TRUE, FALSE)) {:
#  the condition has length > 1 and only the
#> first element will be used

if (NA) { print("if true") }
#> Error in if (NA) {: missing value where TRUE/FALSE need
```

Vectorized if `ifelse` statements

At first blush, `ifelse()` statements look like a quicker way to write an if else statement

```
today <- Sys.Date()
ifelse(today == "2020-11-03",
       "VOTE TODAY!!",
       "Don't forget to vote on Nov 3rd.")
```

```
## [1] "Don't forget to vote on Nov 3rd."
```

```
ifelse(condition, returns this if TRUE, returns this if FAI
```

What will the following statements return?

```
ifelse(TRUE, 1, 2)
ifelse(FALSE, 1, 2)
```

What will the following statements return?

```
ifelse(TRUE, 1, 2)
```

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

```
ifelse(FALSE, 1, 2)
```

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

What will the following statements return?

```r
ifelse(c(TRUE, FALSE, TRUE), 1, 2)
```

What will the following statements return?

Unlike `if`, `ifelse` is vectorized! It evaluates item by item.

```
ifelse(c(TRUE, FALSE, TRUE), 1, 2)
```

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

Detour: NAs and missing data

What's going on in this `ifelse()` statement?

```
ifelse(NA, 1, 2)
```

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

Unlike `if`, `ifelse` can handle `NA`s and as usual `NA`s are contagious.

Ifelse statements in dataframes

Ifelse statements work well in dataframes with the `mutate()` function. Let's add a column to the `texas_housing_data` based on a conditional.

```
texas_housing_data %>%
  mutate(in_january = ifelse(month == 1, TRUE, FALSE)) %>%
  select(city, year, month, sales, in_january)
```

```
## # A tibble: 8,602 x 5
##    city     year month sales in_january
##    <chr>   <int> <int> <dbl> <lgl>
##  1 Abilene  2000     1    72 TRUE
##  2 Abilene  2000     2    98 FALSE
##  3 Abilene  2000     3   130 FALSE
##  4 Abilene  2000     4    98 FALSE
##  5 Abilene  2000     5   141 FALSE
##  6 Abilene  2000     6   156 FALSE
##  7 Abilene  2000     7   152 FALSE
##  8 Abilene  2000     8   131 FALSE
```

case_when statements, supercharged for multiple cases

If you have a lot of categories, ditch the ifelse statement and use dplyr's `case_when()` function, which allows for multiple conditions, like the `else ifs` we saw earlier.

```
texas_housing_data %>%
  mutate(housing_market =
           case_when(
             median < 100000 ~ "first quartile",
             100000 <= median & median < 123800 ~ "second q
             123800 <= median & median < 150000 ~ "third qu
             150000 <= median & median < 350000 ~ "fourth q
           )) %>%
  select(city, median, housing_market)
```

```
## # A tibble: 8,602 x 3
##    city    median housing_market
##    <chr>    <dbl> <chr>
##  1 Abilene  71400 first quartile
##  2 Abilene  58700 first quartile
```

case_when statements are a bit "surly"

case_when will not do type coercion.

```
texas_housing_data %>%
  mutate(housing_market =
   case_when(
     median < 100000 ~ 1,
     100000 <= median & median < 123800 ~ "second quartile"
     123800 <= median & median < 150000 ~ "third quartile",
     150000 <= median & median < 350000 ~ "fourth quartile"
    )) %>%
  select(city, median, housing_market)

Error: must be a double vector, not a character vector
Run `rlang::last_error()` to see where the error occurred.
```

Here we try to but doubles and characters in the housing_market column, but atomic vectors only have one type!

- Rather than coerce and provide a warning, the developers

Recap: `if` and `ifelse`

Today we learned how to:

- better understand logical operators and conditional statements
- use control flow with `if` and `if/else` statements
- use `ifelse()` and `case_when()` statements in conjunction with `mutate` to create columns based on conditional statements.