Day 3, Part 2: Wrangling Data Using Conditionals and Iteration

Brennan Terhune-Cotter and Matt Dye

Agenda

1. Vectors
2. %in%
3. Conditionals: case_when() and if_else()
4. Iteration: for loops and map()

https://github.com/brennanglish/2020_IAMO_R

Vectors

Vectors

- Vectors are very simple but can get very complicated. They form the core of a lot of things you can do with R
- Vectors are simply a *list* of *items that are the same type*
- Often, you will want to work with columns in a dataframe as vectors

> **variables in a dataframe == columns == vectors**
>
> **a dataframe is just a collection of (named) vectors with the same length**

Lists

- A type of vector which is more flexible
  - A “collection” of objects or variables
  - Vectors are useful for simplicity; lists are useful for many other things
- Elements in a list can contain any type of R object
  - Elements can be different types and different structures
  - Including other lists! (i.e. *nested list*)
  - Elements can be named

```
1 listylist <- list(3, "42", "hello!", c(3,4,5))
2 listylist[[4]]
```

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

a list of vectors == a data frame

```r
people <- data.frame(
  name = c("Alice", "Spew", "Charlemagne", "Kay", "Mackenzie", "Spirulina", "Jason"),
  age = c(25, 16, 42, 18, 3, 16, 20),
  humor_score = c(7, 14, 16, 9, 1, 20, 11),
  humor_category = c("Bad", "Good", "Good", "Bad", "Terrible", "Too Good", "Good")
)
kable(people)
```

name	age	humor_score	humor_category
Alice	25	7	Bad
Spew	16	14	Good
Charlemagne	42	16	Good
Kay	18	9	Bad
Mackenzie	3	1	Terrible
Spirulina	16	20	Too Good
Jason	20	11	Good

Logical vectors

- In logical vectors, each element can be either `TRUE`, `FALSE`, or `NA`
- Logical vectors are used in many functions which you've seen, like `filter()`
  - `filter()` creates a logical vector based on your logical statement(s) and keeps all rows with `TRUE`
- In the `filter()` example where we chose my besties, this is what we essentially did:

```
1 funny_adults <- people %>%
2   mutate(is_adult = age > 18,
3          is_funny = humor_category == "Good")
4 funny_adults
```

```
         name age humor_score humor_category
is_adult is_funny
1       Alice  25           7            Bad
TRUE    FALSE
2        Spew  16          14           Good
FALSE    TRUE
3 Charlemagne  42          16           Good
TRUE     TRUE
4         Kay  18           9            Bad
FALSE    FALSE
5   Mackenzie   3           1       Terrible
FALSE    FALSE
6   Spirulina  16          20       Too Good
```

```
1 besties <- filter(funny_adults,
2                   is_adult & is_funny)
3 besties
```

```
         name age humor_score humor_category
is_adult is_funny
1 Charlemagne  42          16           Good
TRUE     TRUE
2       Jason  20          11           Good
TRUE     TRUE
```

(what if we wanted to keep everyone who were adults *or* funny?

%in%

- `%in%` keeps all cases that match one of the elements in the vector.
- It's useful when you have several possible matches and don't want to use multiple logical arguments.

```
# Filter cases where name is either Alice, Kay, Jason, or Spirulina
filtered_people <- people %>%
  filter(name %in% c("Alice", "Kay", "Jason", "Spirulina"))

kable(filtered_people)
```

name	age	humor_score	humor_category
Alice	25	7	Bad
Kay	18	9	Bad
Spirulina	16	20	Too Good
Jason	20	11	Good

Conditionals

Mutating values conditionally

```
?dplyr::mutate
?dplyr::case_when
```

- Often, we want to change values in a variable differently depending on the value.
  - i.e., we want to change values *conditionally*
- We use `if_else()` inside `mutate()` to do this:

```
people <- people %>%
  mutate(adult_status = if_else(age >= 18, "Adult", "Minor"))
```

if_else()

- The syntax of `if_else()` is:

```
mutate(output_variable = if_else(conditional statement,
                                 value if true,
                                 value if false)
       )
```

- We use `if_else` to check for only **one** condition.
- What if we have multiple conditions?
  - We would need multiple `if_else` statements

if_else() with multiple conditions

- What if we want to make a new variable reflecting the decade of life people are in, so we can group them by decade?

```
people <- people %>%
  mutate(decade_of_life =
           if_else(age < 10, "0-9",
             if_else(age < 20, "10-19",
               if_else(age < 30, "20-29",
                 if_else(age < 40, "30-39",
                   if_else(age < 50, "40-49",
                           "50+"
                   )
                 )
               )
             )
           )
  )
```

name	age	humor_score	humor_category
Alice	25	7	Bad
Spew	16	14	Good
Charlemagne	42	16	Good
Kay	18	9	Bad
Mackenzie	3	1	Terrible
Spirulina	16	20	Too Good
Jason	20	11	Good

- This is horribly ugly :(
- In comes `case_when()` to save the day!

case_when()

```
?dplyr::mutate
?dplyr::case_when
```

- `case_when()` can be used within `mutate()` to *selectively* mutate values within a variable.
- It is **extremely** useful to use with `mutate()` when you have multiple conditions and you want to assign different values or perform different operations based on those conditions.

```
# Create a new variable 'decade_of_life'
# for which decade of life people are in
people <- people %>%
  mutate(decade_of_life =
           case_when(
           age < 10 ~ "0-9",
           age < 20 ~ "10-19",
           age < 30 ~ "20-29",
           age < 40 ~ "30-39",
           age < 50 ~ "40-49",
           TRUE ~ "50+"
           )
  )
```

name	age	humor_score	humor_category
Alice	25	7	Bad
Spew	16	14	Good
Charlemagne	42	16	Good
Kay	18	9	Bad
Mackenzie	3	1	Terrible
Spirulina	16	20	Too Good
Jason	20	11	Good

case_when()

- In technical terms, `case_when()` is a **vectorized if-else statement**, in which an if-else statement is a way to perform conditional operations.
  - Each argument in case_when takes the format of `conditional statement ~ output value`.
  - The conditions are evaluated in order, and the first condition that evaluates to TRUE will have its corresponding value returned.

> In case_when statements, if none of the cases match, the output is NA *unless* you add a `TRUE ~ "misc_value"` case at the end. Thinking about how this works (why do you need only TRUE as a conditional statement) will help you understand `case_when` and conditional statements in general.

General if-else statements

- `if_else()` is an R function which reflects general if-else statements, which are very common in programming
- For analyzing data using Tidyverse functions, you will usually use `if_else` or `case_when`
- However, if you start coding more complex things, you will want to use **general if-else statements**
  - You can put **anything** in the conditions!
  - They are used for **control flow** in a function or script

```
classify_adult <- function(age) {
  if (age >= 18) {
    return("Adult")
  } else {
    return("Non-Adult")
  }
}

ben <- 29
classify_adult(ben)
```

```
[1] "Adult"
```

Iteration in R

- Iteration means to do the same thing over and over again.
- **For loops** are a very common way to do this in programming
- R has the **apply** and **map** families of functions, which are vectorized for loops

For Loops

- **For loops** allow you to iterate over a sequence (vector) and perform actions for each item in the sequence
- Let's print the name and age of each person in the `people` dataframe:

manual iteration

```
print(paste("Name:", people$
print(paste("Name:", people$
print(paste("Name:", people$
print(paste("Name:", people$
print(paste("Name:", people$
print(paste("Name:", people$
print(paste("Name:", people$
print(paste("Name:", people$
```

OUTPUT

```
[1] "Name: Alice , Age: 25"
[1] "Name: Spew , Age: 16"
[1] "Name: Charlemagne , Age:
42"
[1] "Name: Kay , Age: 18"
[1] "Name: Mackenzie , Age: 3"
[1] "Name: Spirulina , Age: 16"
[1] "Name: Jason , Age: 20"
```

iteration using a for loop

```
for(i in 1:nrow(people)) {
  print(
    paste(
      "Name:",
      people$name[i],
      ", Age:",
      people$age[i]))
}
```

For Loops

Here is the syntax of a **for loop:**

```
for(i in begin:end) {
  do something with i
}
```

- *i* means the index; it is a variable that represents the index of each item in the sequence
- However, I almost never use *for* loops in R!
- Instead, I use vectorized functions like *map* or *apply* (*map* functions are in the Tidyverse and better than *apply*)
  - They are more concise and readable
  - They are more efficient
    - `For` loops process each item/row sequentially
    - Vectorized functions process multiple items/rows simultaneously

The *map* family

```
1 ?purrr::map
```

Let's say we test all of the people in our dataset a bunch of times, and the number of times varies per person:

name	age	humor_score	humor_category	adult_status	decade_
Alice	25	7	Bad	Adult	20-29
Spew	16	14	Good	Minor	10-19
Charlemagne	42	16	Good	Adult	40-49

Kay	18	9	Bad	Adult	10-19
Mackenzie	3	1	Terrible	Minor	0-9
Spirulina	16	20	Too Good	Minor	10-19
Jason	20	11	Good	Adult	20-29

We want the mean of scores for each person. Let's try this:

```
1  mean(people$scores)
```

```
[1] NA
```

This doesn't work because `people$scores` is a list of vectors. We have to apply the `mean` function to *each vector* in the list, individually.

The *map* family

We could do this with a for loop:

```r
# Create an empty vector to hold the results
average_score <- c()

# Loop over the rows of the data frame
for (i in 1:nrow(people)) {
  # Calculate the mean of the scores for the current person and save it in the results vector
  average_score[i] <- mean(people$scores[[i]])
}

# Add the results vector as a new column in the data frame
people$average_score <- average_score
```

Or with `map()`:

```r
library(purrr)
people$average_score <- map(people$scores, mean)
```

map(), explained

- `map()` applies a function to each element in a list or vector.
- `for` loops are more flexible, but `map()` is simpler and works for most cases in R
- It takes some practice to fully understand how to use *map()* for your data, so if you don't get it yet, don't worry!
- Just keep this in mind and you'll come across it later when you're ready :)

Last thing! Other *map()* functions

- What's the class of the new average_score vector we just created?

```
class(people$average_score)
```

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

- `map()` always outputs a list for each value, even if it's a list of 1.
- If you want the output to be a specific type, you have to use the right function

Function	Type
map_lgl()	logical
map_int()	integer
map_dbl()	numeric
map_df()	dataframe

```
people$average_score <- map_dbl(people$scores
class(people$average_score)
```

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