Introduction to R: **Control Flow**

Day 4, Part A

IHME | UNIVERSITY *of* WASHINGTON

Institute for Health Metrics and Evaluation

In this lecture

1. `if`/`else` statements
2. Assertions
3. `for` loops
4. `break` and `next`
5. `while` loops

if statements (single line)

`if` statements can be used to specify that a line of code should only be executed if a given condition is met:

```
> x <- -5
> if (x < 0) x <- -1 * x
> x
[1] 5
```

```
> x <- 5
> if (x < 0) x <- -1 * x
> x
[1] 5
```

else statements (single line)

else statements can be used to specify alternate code that should be executed if a given condition is NOT met:

```
> x <- -5
> if (x < 0) y <- -1 * x else y <- 100 * x
> y
[1] 5
```

```
> x <- 5
> if (x < 0) y <- -1 * x else y <- 100 * x
> y
[1] 500
```

if and else blocks

if and else can be used alongside curly brackets to specify blocks of code that should be executed if a condition is or is not met:

```
> x <- rnorm(1)
>
> if (x < 0) {
+     y <- -1 * x
+ } else {
+     y <- 100 * x
+ }
>
> x
[1] 1.025571
> y
[1] 102.5571
```

Note: else must follow on the same line as the close brace for the if block.

else if

Adding in `else if` allows you to specify additional alternatives:

```
> x <- rnorm(1)
>
> if (x < 0) {
+     y <- -1 * x
+ } else if (x < 100) {
+     y <- x * 100
+ } else {
+     y <- x * 5
+ }
>
> x
[1] -0.284773
> y
[1] 0.284773
```

Note: you can have as many `else if`'s as you want, and you do not necessarily need to follow with an `else`

Nesting if, else, and else if types of statements

If needed, you can also nest `if`, `else`, and `else if` types of statements:

```
> x <- rnorm(1)
>
> if (x < 0) {
+     y <- x * -10
+ } else {
+     if (x < 10) {
+         y <- x * 100
+     } else {
+         y <- x * 10
+     }
+ }
>
> x
[1] -1.220718
> y
[1] 12.20718
```

Logicals in if, else, and else if statements

Any kind of logical statement (i.e., something that returns TRUE or FALSE) can be used in an if, else, or else if, so long as it returns a *single* value.

```
> x <- rnorm(4)
> x
[1]  0.181303480 -0.138891362  0.005764186  0.385280401
```

This is fine:

```
> if (sum(x) > 0) print("the sum of x is greater than 0")
[1] "the sum of x is greater than 0"
> if (x[1] > 0) print("the first value of x is greater than 0")
[1] "the first value of x is greater than 0"
```

This is (usually) wrong:

```
> if (x > 0) print("x is greater than 0?")
Warning in if (x > 0) print("x is greater than 0?"): the
condition has length > 1 and only the first element will be
used
[1] "x is greater than 0?"
```

Logicals in `if`, `else`, and `else if` statements

In addition to relational operators (!=, ==, <, <=, >, >=) and logical operators (&, |), **functions** that return TRUE or FALSE can be used in `if`, `else`, and `else if` statements:

```
> today <- as.character(Sys.time())
> today
[1] "2019-06-17 21:21:00"
> if (grepl("2017", today)) print("The year is 2017")
```

```
> x <- log(rnorm(4))
Warning in log(rnorm(4)): NaNs produced
> x
[1]       NaN -0.439472       NaN -1.103277
> if (sum(is.na(x)) > 0) print("Vector x contains missing values")
[1] "Vector x contains missing values"
```

Assertions

One particularly useful application of `if` statements are assertions: testing to ensure that some condition that should be met is actually met, and stopping the code from executing otherwise.

`if` can be combined with `stop()` to accomplish this:

```
> if (sum(is.na(x)) > 0) stop("There is missingness!")
Error in eval(expr, envir, enclos): There is missingness!
```

Or you can use the `stopifnot()` function:

```
> stopifnot(sum(is.na(x)) == 0)
Error in eval(expr, envir, enclos): sum(is.na(x)) == 0 is not TRUE
```

Messages and warnings

In some cases, it makes sense to provide a message or warning instead of completely halting execution.

```
> if (sum(is.na(x)) > 0) message(paste("x has", sum(is.na(x)),
+     "missing values"))
x has 2 missing values
```

```
> if (sum(is.na(x)) > 0) warning(paste("x has", sum(is.na(x)),
+     "missing values"))
Warning: x has 2 missing values
```

`message()` is also frequently useful apart from `if/else` type statements, just for reporting information the user of some code may want to know.

for loops

for loops allow you to repeat a block of code while iterating through a vector of values that (usually) impact that block of code in some way.

```
> for (x in c("red", "orange", "blue", "green", "yellow")) {
+     print(paste("I like the color", x))
+ }
[1] "I like the color red"
[1] "I like the color orange"
[1] "I like the color blue"
[1] "I like the color green"
[1] "I like the color yellow"
```

`for` loops

`for` loops are generally used to repeat a set of operations for a series of closely related values.

For example:

1. Loading a series of closely related files (e.g., one for each calendar year, country, or age group).
2. Making a series of similar graphs (e.g., one for each location or indicator).
3. Carrying out transformations to a series of data frame columns (e.g., calculating z-scores for all variables).

for loops: loading data

If you want to store information generated in each iteration of a for loop in R's memory, you'll need to set up a data structure to hold that information.

```
> all_data <- NULL
>
> for (year in c(2010, 2012, 2013)) {
+     data <- read.csv(paste0(main_dir, "data/wa_data_", year,
+         ".csv"))
+     all_data <- rbind(all_data, data)
+ }
>
> table(all_data$year)

2010 2012 2013
   6    6    6
```

for loops: graphing

We often use for loops for making a series of graphs: e.g., for multiple indicators or for multiple years or areas.

```
> mmr <- read.csv(paste0(main_dir, "data/mmr_data_time_series.csv"))
> all_locations <- unique(mmr$location_name)
>
> pdf(paste0(main_dir, "output/mmr_time_trends_by_country.pdf"))
> for (loc in all_locations) {
+   gg <- ggplot(mmr[mmr$location_name == loc,],
+                aes(x=year_id, y=mmr)) +
+     geom_line()
+   print(gg)
+ }
> dev.off()
```

NOTE: for ggplot graphics you must *explicitly print* the graph object (e.g., print(gg)) when making graphs inside a loop.

for loops: transforming columns

for loops are also useful for carrying out the same operation on a series of columns in a data frame, for example, applying the same transformation to a series of columns:

```
> head(mmr[, c("mmr", "lower", "upper")], 2)
       mmr    lower    upper
1 31.36912 27.91811 35.16374
2 33.76302 29.61305 38.24765
>
> for (var in c("mmr", "lower", "upper")) {
+   mmr[, var] <- mmr[, var] / 100000
+ }
>
> head(mmr[, c("mmr", "lower", "upper")], 2)
           mmr        lower        upper
1 0.0003136911 0.0002791810 0.0003516374
2 0.0003376302 0.0002961305 0.0003824765
```

nesting `for` loops

You can nest loops! This is useful if you need to iterate over multiple dimensions:

```
> pdf(paste0(main_dir, "output/mmr_barchart_by_super_region.pdf"))
> for (sr in unique(mmr$super_region_name)) {
+   for (yr in unique(mmr$year_id)) {
+     gg <- ggplot(mmr[mmr$super_region_name == sr & mmr$year_id == yr, ],
+              aes(x = location_name, y = mmr)) +
+       geom_bar(stat = 'identity') +
+       labs(title = paste0(sr, ", ", yr))
+     print(gg)
+   }
+ }
> dev.off()
pdf
  2
```

messages in loops

Sometimes it is useful to print statements or put messages in a loop. This can help with debugging (it makes it obvious where you were when something went awry) and may also be useful if you just want to provide a status update to the user.

```
> pdf(paste0(main_dir, "output/mmr_barchart_by_super_region.pdf"))
> for (sr in unique(mmr$super_region_name)) {
+   message(sr)
+
+   for (yr in unique(mmr$year_id)) {
+     message(paste0('..', yr))
+     gg <- ggplot(mmr[mmr$super_region_name == sr & mmr$year_id == yr, ],
+            aes(x = location_name, y = mmr)) +
+       geom_bar(stat = 'identity') +
+       labs(title = paste0(sr, ", ", yr))
+     print(gg)
+   }
+ }
High-income
..1990
..1995
..2000
..2005
..2010
..2015
Latin America and Caribbean
..1990
..1995
..2000
..2005
..2010
..2015
North Africa and Middle East
..1990
..1995
..2000
```

next

Sometimes you may want to end a single iteration of a `for` loop early. This is accomplished with the **`next`** function:

```
> all_data <- NULL
>
> for (year in 2010:2013) {
+   file <- paste0(main_dir, "data/wa_data_", year, ".csv")
+   if (!file.exists(file)) next
+
+   data <- read.csv(file)
+   all_data <- rbind(all_data, data)
+ }
>
> table(all_data$year)

2010 2012 2013
   6    6    6
```

break

Similarly, you may want to stop the entire loop (not just a single iteration) early. **break** can accomplish this.

```
> all_data <- NULL
>
> for (year in 2010:2013) {
+   file <- paste0(main_dir, "data/wa_data_", year, ".csv")
+   if (!file.exists(file)) break
+
+   data <- read.csv(file)
+   all_data <- rbind(all_data, data)
+ }
>
> table(all_data$year)

2010
   6
```

while loops

R also has **while** loops, though these are used less commonly than for loops. A while loop continues to execute until some pre-specified condition is met.

```
> x <- 0
>
> while (x < 5) {
+   x <- x + abs(rnorm(1))
+   print(x)
+ }
[1] 1.096839
[1] 1.532021
[1] 1.857952
[1] 3.00676
[1] 4.000264
[1] 4.548661
[1] 4.787392
[1] 5.415298
```