Intermediate R

Allison Theobold & The Carpentries

The goal of this workshop is to teach you to write modular code and some best practices for writing code in `R`.

First, we will start with the building blocks of conditional statements, the relational operator. Next, we will join sequences of relational operations together with logical operators. We will then use these relational operators inside conditional statements (`ifelse()`).

Finally, we will dive into two methods to avoid copying and pasting your code numerous times to accomplish the same task. The `for` loop will be introduced for procedures done multiple times in *one* location in your code, and functions will be introduced for procedures done *throughout* your code.

Let's get started!

Refresher (10 minutes)

This workshop covers content that will require that we remember how to extract elements from vectors and dataframes. Let's work through the following warm-ups to refresh how we can use these operations.

Extracting Elements from a Vector

To extract an element from a vector we use the bracket (`[]`) notation. Recall, a vector has only one dimension, so inside the brackets goes *one* number. To extract elements of a vector, we can give their corresponding index, starting from the number one.

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)

x[1] ## Extracts the first element of x
```

```
## [1] 5.4
```

```
x[1:3] ## Extracts a slice of x, from the first index to the third index
```

```
## [1] 5.4 6.2 7.1
```

```
x[-2] ## Extracts everything in x BUT the second index
```

```
## [1] 5.4 7.1 4.8 7.5
```

Exercise 1: Why does `R` produce an error for the following code?

```
x[-1:3]
```

Extracting Elements from a Dataframe

A dataframe is a list of vectors, where each vector is permitted to have a different data type. Because dataframes have two dimensions (columns and rows), we are able to extract two types of elements.

To extract a column from a dataframe we can the familiar `$` operator:

```
df <- data.frame(A = c(1, 5, 9, 13),
                 B = c(2, 6, 10, 14),
                 C = c(3, 7, 11, 15),
                 D = c(4, 8, 12, 16)
                 )

df$A ## Extracts the A column from the dataframe
```

```
## [1]  1  5  9 13
```

This is useful if we only wish to extract a *single* vector from our dataframe. If we want multiple vectors, then using matrix notation is more simple.

To extract a column from a dataframe using brackets (`[row, column]`):

```
df[1, 1] ## Extracts the first row, first column entry
```

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

```
df[, 1] ## Extracts EVERY row in the first column
```

```
## [1]  1  5  9 13
```

```
df[1, ] ## Extracts EVERY column in the first row
```

```
##   A B C D
## 1 1 2 3 4
```

Exercise 2: I want to extract the 3rd and 4th columns. What is wrong with my code? How would you change it to extract these columns? What if I wanted to extract the 2nd and 4th columns?

```
df[, 3, 4]
```

Relational Operators (25 minutes)

Relational operators tell how one object relates to another, where the output is a logical value. There are a variety of relational operators that you have used before in mathematics but take on a slightly different feel in data science. We will walk through a few examples of different types of relational operators and the rest will be left as exercises.

```
w <- 10.2
x <- 1.3
y <- 2.8
z <- 17.5
dna1 <- "attattaggaccaca"
dna2 <- "attattaggaacaca"
```

- **Equality & Inequality**: This type of operator tells whether an object is equivalent to another object (`==`), or its negation (`!=`).

```
dna1 == dna2
```

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

```
dna1 != dna2
```

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

- **Greater & Less**: These statements should be familiar, with a bit of a notation twist. To write a strict greater than or less than statement you would use `>` or `<` in your statement. To add an equality constraint, you would add a `=` sign after the inequality statement (`>=` or `<=`).

```
w > 10
```

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

```
x > y
```

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

- **Inclusion**: This type of statement checks if a character, number, or factor are included in a vector. The `%in%` operator tells you whether a value is included in an object. These values can be linked together into a vector, and the output will be a vector of logical values.

```
colors <- c("green", "pink", "red")

"blue" %in% colors
```

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

```
numbers <- c(1, 2, 3, 4, 5)

5 %in% numbers
```

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

```
letters <- c("a", "b", "c", "d", "e")

c("a", "b") %in% letters
```

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

Exercise 3:

```
# write R code to see if
    # 2 * x + 0.3 is equal to y

    # "hello" is greater than or equal to "goodbye"

    # TRUE is greater than FALSE

    # dna1 is longer than 5 bases (use nchar() to figure out how long a string is)
```

Comparing Decimal Valued Numbers:
Why does the output from your first equality check not seem right? What do you think might be going on?

General `R` advice is that the `==` operator should only be used for comparisons of integer and Boolean data types. To store decimal valued data types (doubles) `R`, like other programming systems, uses a format (binary floating-point) that doesn't accurately represent a number like 1.3. When the code is interpreted by `R`, the "1.3" is rounded to the nearest number in floating-point format, which results in a small rounding error even before the calculation happens.

This is why, general `R` advice is to use the `all.equal()` function to check for equality of two doubles. This function "is a utility to compare `R` objects `x` and `y` testing *near equality*." This means that the function allows for a little bit of wiggle room in the rounding errors that are associated with doubles.

Comparing Characters:

How did R know how to compare the words hello and goodbye?

Exercise 4: What is going on in the code below? Why is R giving an error?

```
letters != c("a", "c")
```

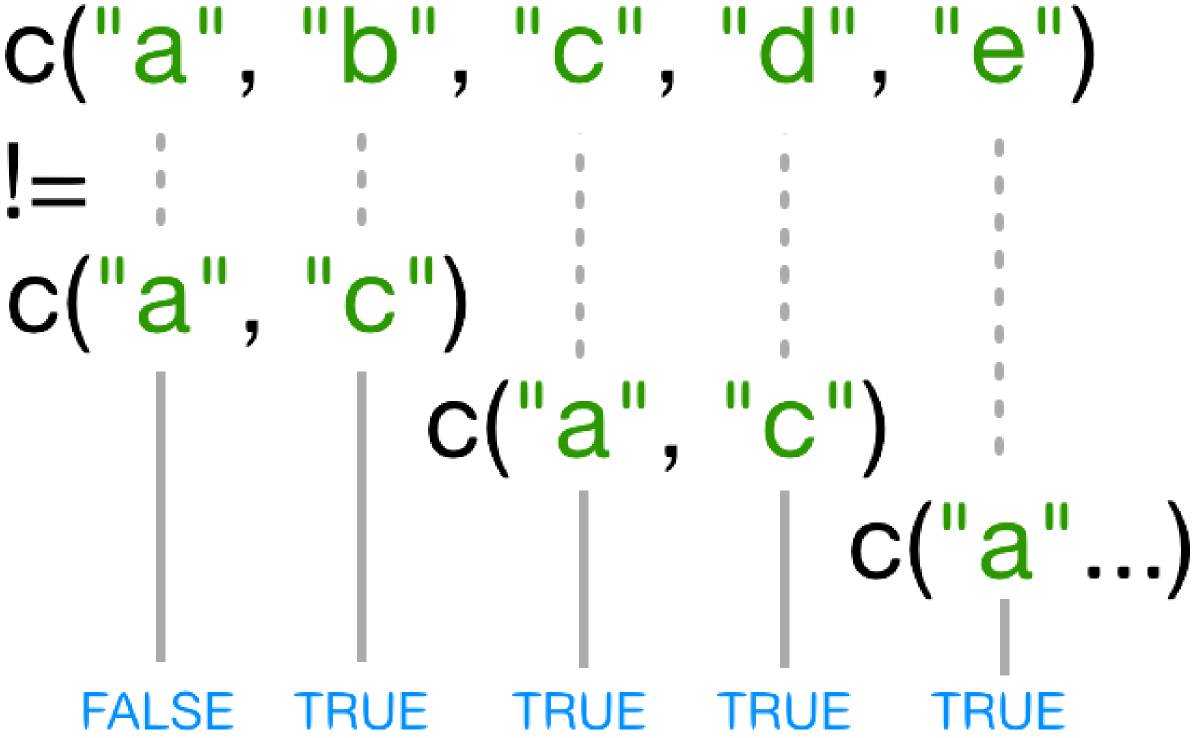
```
## Warning in letters != c("a", "c"): longer object length is not a multiple of
## shorter object length
```

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

Why does R give TRUE as the third element of this vector, when letters[3] != "c" is obviously false?

Recycling: When you use != or ==, R tries to compare each element of the left argument with the corresponding element of its right argument. Then what happens when you compare vectors of different lengths?

When one vector is shorter than the other, it gets *recycled*:



In this case, R **repeats** c("a", "c") as many times as necessary to match the length of the letters vector. So, we get c("a", "c", "a", "c", "a"). Since the recycled "a" doesn't match the third element of letters, the value of != is in fact TRUE.

Note: We got lucky here! `R` output an error because the length of our vectors was not a constant multiple (e.g. 2 times as long). If they were, `R` would have carried out the **same** recycling procedure, but **would not** have output an error!

This is why the inclusion (`%in%`) operator is so great! It does carry out the procedure we wish for `R` to do, without any silly recycling! To exclude values you place a `!` in front of the **entire** statement, not directly in front of the `%in%`.

```
! c("a", "b") %in% letters
```

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

- **Logical Values**: You can also use logical values (`TRUE`, `FALSE`) to extract elements of a vector.

```
letters[c(TRUE, TRUE, FALSE, FALSE, FALSE)]
```

```
## [1] "a" "b"
```

```
## Extracts the first two elements of the letters vector
```

This should give you an intuition as to how we might be able to use the results of relational statements to extract the elements of the vector that satisfy the relation.

- **Which**: The `which` statement returns the *indices* of a vector, where a relational statement is satisfied (evaluates to `TRUE`).

```
x <- c(3, 5, 7, 9, 11, 13, 15)

which(x > 8)
```

```
## [1] 4 5 6 7
```

```
which(x == 7)
```

```
## [1] 3
```

```
## Challenge: How would you use the indicies from these which statements to
##            extract the elements of x that meet the criteria?
```

- **Matrices**: The above relational and which statements can be applied to a matrix, or to a subset of the matrix (e.g. a vector). The relational and `which` statements are applied element wise (a step-by-step progression through the matrix/vector entries).

```
# The following are dating data, of one person's messages received per day for one week

okcupid <- c(16, 9, 13, 5, 2, 17, 14)
match <- c(17, 7, 5, 16, 8, 13, 14)
messages <- data.frame(Okcupid = okcupid, Match = match)
# This makes the data from OkCupid the first column and the data from Match the second column
row.names(messages) <- c("Monday", "Tuesday", "Wednesday", "Thursday",
                         "Friday", "Saturday", "Sunday")
```

Exercise 5: Use the messages matrix to return a matrix of *logical* values which answer the following question:

```
# Create a logical matrix to answer the question:
# For what days were the number of messages at either site greater than 12?
```

Exercise 6: Use the messages matrix to return the *rows* of `messages` which answer the following questions:

```
# Use rows of the matrix to answer:

# when were the messages at OkCupid equal to 13?


# when were the messages at OkCupid greater than Match?
```

- **Subset**: The `subset` command takes in an object (vector, matrix, data frame) and returns the subset of that object where the entries meet the relational conditions specified (evaluates to `TRUE`).

```
x <- c(3, 5, 7, 9, 11, 13, 15)

subset(x, x > 6)
```

```
## [1]  7  9 11 13 15
```

```
## Challenge: Change the which() statement code to a subset() statement,
##            extracting the days that the number of messages at Okcupid
##            greater than the messages at Match
```

Logicals (10 minutes)

These statements allow for us to change or combine the results of the relational statements we discussed before, using **and**, **or**, or **not**.

- **and** statements evaluate to `TRUE` only if every relational statement evaluates to `TRUE`.
  - `(3 < 5) & (9 > 7)` would evaluate to `TRUE` because both relational statements are `TRUE`
  - `(3 > 5) & (9 > 7)`, this would evaluate to `FALSE` as only one of the relational statements is `FALSE` (the first one)

- **or** statements evaluate to `TRUE` if at least one relational statement evaluates to `TRUE`.
  - `(3 > 5) | (9 > 7)` would evaluate to `TRUE` because one of the relational statements is `TRUE` (the second one)
  - `(3 > 5) | (9 < 7)` would evaluate to `FALSE` as both relational statements evaluate to `FALSE`

- **not** statements converts (negates) the statement it proceeds, changing `TRUE` to `FALSE` and `FALSE` to `TRUE`.
  - `is.numeric(5)` would evaluate to `TRUE` because 5 is a number
  - `!is.numeric(5)` would evaluate to `FALSE` as it negates the statement it proceeds (`!TRUE = FALSE`)

- **Remark:** The `&&` and `||` logical statements do not evaluate the same as their single counterparts. Instead, these logical operators evaluate to `TRUE` or `FALSE` based only on the first element of the statement, vector, or matrix.

```
c(TRUE, TRUE, FALSE) & c(TRUE, FALSE, FALSE)
```

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

```
c(TRUE, TRUE, FALSE) | c(TRUE, FALSE, FALSE)
```

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

```
c(TRUE, TRUE, FALSE) && c(TRUE, FALSE, FALSE)
```

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

Exercise 7: Using the okcupid vector from above, answer the following questions:

```
# Is the last day of the week under 5 messages or above 10 messages?
# (hint: last <- tail(okcupid, 1) could be helpful)


# Is the last day of the week between 15 and 20 messages, excluding 15 but including 20?
```

The `subset` command can from before, can also accept more than one relational conditions, joined by logicals.

```
bad_days <- subset(messages, Okcupid < 6 | Match < 6)

bad_days
```

```
##           Okcupid Match
## Wednesday      13     5
## Thursday        5    16
## Friday          2     8
```

```
good_days <- subset(messages, Okcupid > 10 & Match > 10)

good_days
```

```
##          Okcupid Match
## Monday        16    17
## Saturday      17    13
## Sunday        14    14
```

Conditional Statements (30 minutes)

Conditional statements utilize relational and logical statements to change the results of your `R` code. You may have encountered an `ifelse` statement before (or not), but let's breakdown exactly what `R` is doing when it evaluates them.

If Statements

First, let's start with an `if` statement, the often overlooked building block of the `ifelse` statement. The `if` statement is structured as follows:

```
if(condition){
              statement
}
```

- the condition inside the parenthesis (a relational statement) is what the computer executes to check its logical value,
    - if the condition evaluates to `TRUE` then the statement inside the curly braces (`{}`) is output, and
    - if the condition is `FALSE` nothing is output.

Remark: In `R` the `if` statement, as described above, will only accept a **single** value (not a vector or matrix).

```
y <- -3

if(y < 0){
  "y is a negative number"
}
```

```
## [1] "y is a negative number"
```

```
y <- 5
```

Exercise 8: Using the `last` number from above, write an if statement that prints "You're popular!" if the number of messages exceeds 10.

```
# hint: use the last day of the week for okcupid that you made above
```

If Else Statements

Since whenever an `if` statement evaluates to `FALSE` nothing is output, you might see why an `else` statement could be beneficial! An `else` statement allows for a different statement to be output whenever the `if` condition evaluates to `FALSE`. The `ifelse` statement is structured as follows:

```
if(condition){
              statement1
}
else{
              statement2
}
```

- again, the `if` condition is executed first,
  - if it evaluates to `TRUE` then the first statement (`statement1`) is output,
  - if the condition is `FALSE` the computer moves on to the `else` statement, and the second statement (`statement2`) is output.

Remark: In `R` the `ifelse` statement, as described above, will only accept a **single** value (not a vector or matrix).

```r
y <- -3

if(y < 0){
  "y is a negative number"
} else{
  "y is either positive or zero"
}
```

```
## [1] "y is a negative number"
```

```r
y <- 5
```

Remark: `R` accepts both `ifelse` statements structured as outlined above, but also `ifelse` statements using the built-in `ifelse()` function. This function accepts **both singular and vector** inputs and is structured as follows:

`ifelse(condition, statement1, statement2),`

where the first argument is the conditional (relational) statement, the second argument is the statement that is evaluated when the condition is `TRUE` (`statement1`), and the third statement is the statement that is evaluated when the condition is `FALSE` (`statement2`).

```r
ifelse(y < 0, "y is a negative number", "y is either positive or zero")
```

```
## [1] "y is either positive or zero"
```

Exercise 9: Using the if statement from Exercise 8 add the following else statement: When the if-condition on messages is not met, R prints out “Send more messages!”.

```r
# Challenge: Rewrite the if, else function from above using R's built in ifelse() function.
```

Else If Statements

On occasion, you may want to add a third (or fourth, or fifth, ...) condition to your `ifelse` statement, which is where the `elseif` statement comes in. The `elseif` statement is added to the `ifelse` as follows:

```
if(condition1){
              statement1
}
else if(condition2){
              statement2
}
else{
              statement3
}
```

- The `if` condition (`condition1`) is executed first,
  - if it evaluates to `TRUE` then the first statement (`statement1`) is output,
  - if the condition is `FALSE` the computer moves on to the `elseif` condition,
- Now the second condition (`condition2`) is executed,
  - if it evaluates to `TRUE` then the second statement (`statement2`) is output,
  - if the condition is `FALSE` the computer moves on to the `else` statement, and
- the third statement (`statement3`) is output.

```
y <- -3

if(y < 0){
  "y is a negative number"
} else if(y == 0){
    "y is zero"
} else{
    "y is positive"}
```

```
## [1] "y is a negative number"
```

```
y <- 5
```

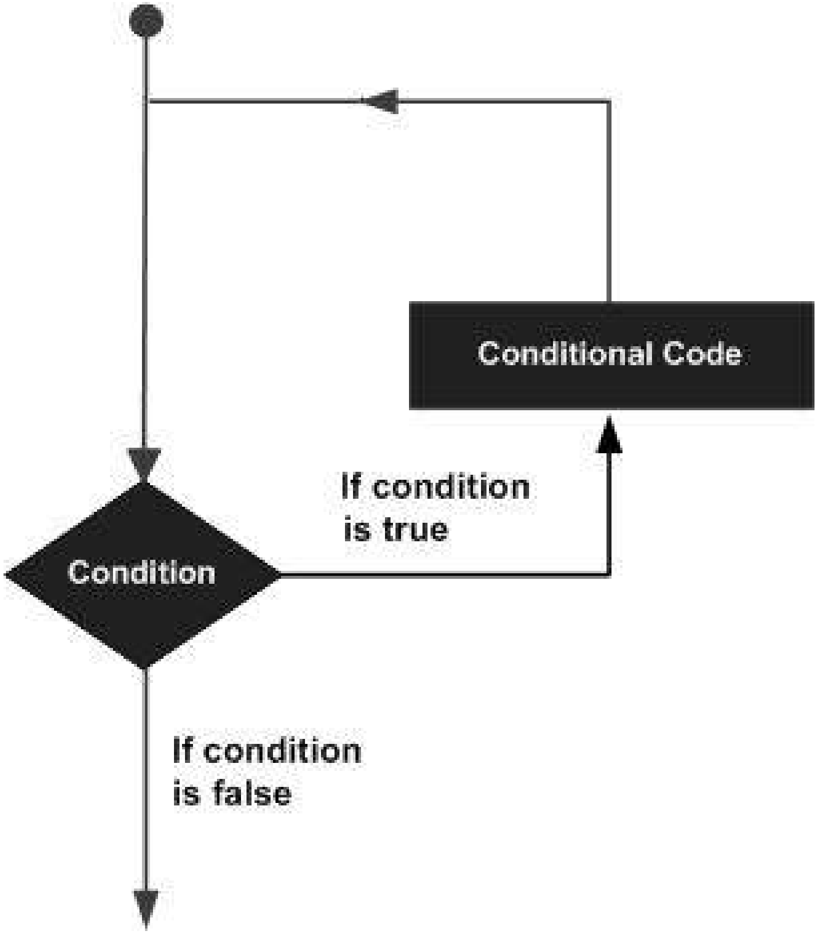
- **Remark:** Conditional statements should be written so that the components are mutually exclusive (an observation can only belong to one piece).

```
x <- 6

if(x %% 2 == 0){
  "x is divisible by 2"
}else if(x %% 3 == 0){
  "x is divisible by 3"
}else{
  "x is not divisible by 2 or 3"
}
```

```
## [1] "x is divisible by 2"
```

Loops (30 minutes)



Loops are a popular way to iterate or replicate the same set of instructions many times. It is no more than creating an automated process by organizing a sequence of steps in your code that need to be repeated. In general, the advice of many `R` users would be to learn about loops, but once you have a clear understanding of them to instead use vectorization, when your current iteration doesn't depend on the previous iteration.

Loops will give you a detailed view of the process that is happening and how it relates to the data you are manipulating. Once you have this understanding, you should put your effort into learning about vectorized alternatives as they pay off in efficiency. These loop alternatives (the `apply` and `purrr` families) will be covered in a later workshop.

Typically in data science, we separate loops into two types. The loops that execute a process a specified number of times, where the "index" or "counter" is incremented after each cycle are part of the `for` loop family. Other loops that only repeat themselves until a conditional statement is evaluated to be `FALSE` are part of the `while` loop family.

For Loops

In the for loop figure above:

- The starting point (black dot) represents the initialization of the object(s) being used in the `for` loop (i.e. the variables). `R` requires for you to initialize the objects you will use in your loop **before** you use them, it does not automatically create the objects for you!
- The diamond shapes represent the repeat condition the computer is required to evaluate. The computer evaluates the conditional statement (`i in seq`) as either `TRUE` or `FALSE`. In other terms, you are testing if in the current value of `i` (the index/counter) is within the specified range of values `seq`, where this range is either defined in the initialization or directly within the condition statement (like `1:100`).
- The rectangle represents the set of instructions to execute for every iteration. This could be a simple statement, a block of instructions, or another loop (nested loops).

The computer marches through this process until the repeat decision (condition) evaluates to `FALSE` (`i` is not in `seq`).

Notation

The `for` loop repeat statement is placed between parentheses after the `for` statement. Directly after the repeat statement, in curly braces, are the the instructions that are to be repeated. You will find the formatting that you like best for things such as loops and functions, but here is my preferred syntax:

```
seq <- 1:100

for(i in seq){
              statement
}
```

OR

```
num <- 100
for(i in 1:num){
              statement
}
```

Example:

```
for(i in 1:10) {
print(i)
  }
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```
i ## should be 100 (the same as sim), unless the loop had issues!
```

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

Now, let's take add the corresponding entries of two vectors together!

```
x <- seq(1, 10, by = 1)
y <- seq(10, 28, by = 2)

## create an empty vector to store added quantities
z <- rep(NA, length(x))


for(i in 1:10) {
# write a loop to store the entries of x added to the entries of y in z

}

i ## should be 100 (the same as sim), unless the loop had issues!
```

Exercise 10: How could the above calculation be done outside of a loop?

```
# code to execute squaring of the u vector NOT in a loop
```

Remark: Like I used in the above `for` loop, it is often useful, to add a `print()` statement inside the loop. This allows for you to verify that the process is executing correctly and can save you some major headaches!

Exercise 11: Recursive `for-loop`!

Read in the BlackfootFish dataset and modify the code to write a `for-loop` to find the indices needed to sample every 7^{th} row from the dataset, starting with the 1^{st} row, until you've sampled 1900 rows.

```
BlackfootFish <- read.csv("data/BlackfootFish.csv", header = TRUE)

size <- 1900

samps <- rep(NA, 1900)
## Initializing the samps vector, for storing indicies

samps[1] <- 1
## Setting first sample index to 1 (first row)

# Code snippet:
# Create the code for the process will be executed at each step
# e.g. How will you get the next sample after 1?

for(i in 2:#ending index here){
  samps[i] <- # process you execute at every index
}

testing <- BlackfootFish[samps, ]
training <- BlackfootFish[-samps, ]
```

Nesting For Loops

Now that you know that for loops can also be nested, you're probably wondering how this would look.

In nested `for` loops, you have two indices keeping track of where you are in the loop. The outer loop will have an index, say `i`, and the inner loop will have an index, say `j`. Let's look at the nested `for` loop below and see if we can figure out how the loop proceeds through the indices.

```
for(i in 1:5){
  for(j in c("a", "b", "c", "d", "e")){
    print(paste(i, j))
  }
}
```

```
## [1] "1 a"
## [1] "1 b"
## [1] "1 c"
## [1] "1 d"
## [1] "1 e"
## [1] "2 a"
## [1] "2 b"
## [1] "2 c"
## [1] "2 d"
## [1] "2 e"
## [1] "3 a"
## [1] "3 b"
## [1] "3 c"
## [1] "3 d"
## [1] "3 e"
## [1] "4 a"
## [1] "4 b"
## [1] "4 c"
## [1] "4 d"
## [1] "4 e"
## [1] "5 a"
## [1] "5 b"
## [1] "5 c"
## [1] "5 d"
## [1] "5 e"
```

When would you possibly use this in your code?

Well, suppose you wish to manipulate a matrix by setting its elements to specific values, based on their row and column position. You will need to use nested `for loops` in order to assign each of the matrix's entries a value.

- The index `i` runs over the rows of the matrix
- The index `j` runs over the columns of the matrix

But when should you use a loop? Couldn't you just repeat the set of instructions the number of times you want to do it?

A rule of thumb is that if you need to perform the same action in one place of your code three or more times, then you are better served by using a loop. A loop makes your code more compact, readable, maintainable, and saves you some typing! If you have the same process multiple places in your code, that's where functions come in handy!

Functions (30 minutes)

What is a function? In rough terms, a function is a set of instructions that you would like repeated (similar to a loop), but are more efficient to be self-contained in a sub-program and called upon when needed. A function is a piece of code that carries out a specific task, accepting arguments (inputs), and returning an output.

Functions gather a sequence of operations into a whole, preserving it for ongoing use. Functions provide:

- a name we can remember and invoke it by
- relief from the need to remember individual operations
- a defined set of inputs and expected outputs
- rich connections to the larger programming environment

As the basic building block of most programming languages, user-defined functions constitute "programming" as much as any single abstraction can. If you have written a function, you are a computer programmer!

At this point, you may be familiar with a smattering of `R` functions in a few different packages (e.g. `lm`, `glm`, `str`, `tapply`, etc.). Some of these functions take multiple arguments and return a single output, but the best way to learn about the inner workings of functions is to write your own!

User Defined Functions

A function allows for us to repeat several operations with a single command. Take for instance a function which converts heights from feet and inches to centimeters, named `inch_to_cm()`.

```
inch_to_cm <- function(feet, inches) {
  inches <- feet*12 + inches
  cm <- inches * 2.54

  return(cm)
}
```

Exercise 12: Use the `inch_to_cm` function to convert your neighbor's height!

We define `inch_to_cm()` by assigning it to the output of function. The list of argument names are contained within parentheses. Next, the body of the function (the statements that are executed when it runs) is contained within the curly braces (`{}`). The statements in the body are indented by two spaces. This makes the code easier to read but does not affect how the code operates.

It is useful to think of creating functions like writing a cookbook.

1. Define the "ingredients" that your function needs. In this case, we only need one ingredient to use our function: "temp".
2. State what we to do with the ingredients. In this case, we are taking our ingredient and applying a set of mathematical operators to it.
3. Declare what the final product (output) of your function will be. This typically lives in a `return()` statement, but we will see other methods to use.

When we call the function, the values we pass to it as arguments are assigned to those variables so that we can use them inside the function. Inside the function, we use a return statement to send a result back to whoever asked for it.

Function Scoping

When writing and using functions, you should be careful to consider where each of the variables you are using are defined. Variables that are defined *within* a function belong to a local environment, and are not accessible from outside of the function. Take this example:

```r
x <- 10
## globally defined variable

f <- function(){
  ## locally defined x and y
  x <- 1
  y <- 2
  c(x, y)
}

f()
```

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

```r
x
```

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

We notice that the globally defined value of `x` is not called inside the function, as the function defines `x` within itself, so it does not search for a value of `x`! Now look at another example, where `x` is not defined within the function.

```r
g <- function(){
  ## locally defined y
  y <- 2
  c(x, y)
}

x <- 15

g()
```

```
## [1] 15  2
```

```r
x <- 20

g()
```

```
## [1] 20  2
```

Summary:

1. When you call a function, a new environment is created for the function to do its work in.
2. The new environment is populated with the argument values passed in to the function.
3. Objects are looked for *first* in the function environment.
4. If objects are not found in the function environment, the are then looked for in the global environment.

Why Write a Function?

Similar to loops, you will often find that you have performed the same task multiple times. A good rule of thumb is that if you've copied and pasted the same code twice (so you have three copies), you should write a function instead!

Example: If I wanted to rescale every quantitative variable in my dataset so that the variables have values between 0 and 1. I could use the following formula:

$$y_{scaled} = \frac{y_i - min\{y_1, y_2, ..., y_n\}}{max\{y_1, y_2, ..., y_n\} - min\{y_1, y_2, ..., y_n\}}$$

The following `R` code would carry out this rescaling procedure for the `length` and `weight` columns of my data:

```r
BlackfootFish$length <- (BlackfootFish$length - min(BlackfootFish$length, na.rm = TRUE)) /
  (max(BlackfootFish$length, na.rm = TRUE) - min(BlackfootFish$length, na.rm = TRUE))

BlackfootFish$weight <- (BlackfootFish$weight - min(BlackfootFish$weight, na.rm = TRUE)) /
  (max(BlackfootFish$weight, na.rm = TRUE) - min(BlackfootFish$length, na.rm = TRUE))
```

I could continue to copy and paste for other columns of my data, but you can probably get the idea. This process of duplicating an action multiple times makes it difficult to understand the intent of the process. Additionally, it makes it very difficult to spot the mistakes. *Did you spot the mistake in the weight conversion?*

Often you will find yourself in the position of needing to find a function that performs a specific task, but you do not know of a function or a library that would help you. You could spend time Googling for a solution, but in the amount of time it takes you to find something you could have already written your own function!

Example: Let's transform the repeated process above into a rescaling function.

The following snippet of code rescales a column to be between 0 and 1:

```r
(BlackfootFish$length - min(BlackfootFish$length, na.rm = TRUE)) /
  (max(BlackfootFish$length, na.rm = TRUE) - min(BlackfootFish$length, na.rm = TRUE))
```

Our goal is to turn this snippet of code into a general rescale function that we can apply to any numeric vector.

- The first step is to examine the process to determine how many inputs there are.
- The second step is to change the code snippet to instead refer to these inputs using temporary variables.

For this process there is one input, the numeric vector to be rescaled (currently `BlackfootFish$length`). We instead want to refer to this input using a temporary variable. It is common (in `R`) to refer to a vector of data as `x`.

Exercise 13: Modify the code snippet above to instead refer to a temporary variable `x`.

```
# code modification with temporary variables
```

But now take a closer look at our process. Is there any duplication? An obvious duplicated statement is `min(x, na.rm = TRUE)`. It would make more sense to calculate the minimum of the data once, store the result, and then refer to it when needed. We also notice that we use the maximum value of `x`, so we could instead calculate the range of `x` and refer to the first (minimum) and second (maximum) elements when they are needed.

The `range` function in `R` takes a single numeric input and returns the minimum and maximum:

```
range(seq(1, 5))
```

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

What should we call this variable containing the range of `x`? Some important advice on naming variables in `R`:

- Make sure that the name you choose for your variable is not a reserved word in `R` (`range`, `mean`, `sd`, `hist`, etc.).
    - A way to avoid this is to use the help directory to see if that name already exists (`?functionName`).
- It is possible to overwrite the name of an existing variable, but it is not recommended!
- Variable names cannot begin with a number.
- Keep in mind that capitalization matters in `R`!

I suggest we call this intermediate variable `rng` (for range).

Exercise 14: Create another intermediate variable called `rng` that contains the range of the intermediate variable `x`, using the `range()` function. Make sure that you specify the `na.rm()` option to ignore any NAs in the input vector.

```
# Create rng = the range of x


# Rewrite the code snippet from Excercise 13 to now
# refer to rng instead of min(x) and max(x)
```

How does this process end up with us writing our own function? What do you need in order to write a function?

- The task the function will solve (what you've been copying and pasting) **and**
- The inputs of the function (the intermediate variables)

We now have all these pieces ready to put together. It's time to write the function!

In `R` the function you define will have the following construction:

```
myFunction <- function(argument1, argument2,...){
                body
}
```

Exercise 15: Using the template above, write a function that rescales a vector to be between 0 and 1. The function should take a single argument, `x`.

```
# Use the function template to create a function named rescale01 that perfoms the process outlined abov

myfunction <- function(arg1,...){
  # body
}
```

Once you have written your function, the next step is to test it out. The simplest place to start is to set value(s) for the function's argument(s) (e.g. `x <- c(1, 2, 3, 4, 5)`). You can then run the code *inside* the function, line by line, to make sure that each line successfully executes for this variable. This is what we call unit testing in data science. This is a similar process to testing out a `for` loop, by setting the value of the index (`i <- 1`) and running the inside of the loop.

Exercise 16: Test your function on a simple vector, with the same name as your function's input. What do you expect the values of the test to be after you input it into your function?

```
# Test your function out on a simple vector named x
```

If your unit test was successful, now you can move on to testing your function with your data.

Exercise 17: Test your function on the `length` column of the BlackfootFish dataset. Inspect the rescaled values. Do they look correct?

```
# Test your function out on the BlackfootFish data
```

Process of Creating a Function

You should **never** start writing a function with the function template. Instead, you start with a problem that you need to solve and work through the following steps:

1. Define the problem you need to solve.
2. Get a working snippet of code that solves the simple problem. (*Where you start when you're copying and pasting*)
3. Rewrite the code snippet to use temporary variables.
4. Rewrite the code for clarity (remove redundancy or multiple calculations of the same value).
5. Turn everything into a function! The code snippet forms the body of the function, the temporary variables are the arguments, and you choose the name of your function.
6. Test your function! Start with a simple example and then go big!

Exercise 18: Putting it All Together!

```
matrix <- data.frame(l = BlackfootFish$length, w = BlackfootFish$weight)
```

Now, given the matrix of fish lengths and weights above, use the tools from the workshop (`for loop`, function, conditional and relational statements) to:

- compute the condition index of each fish (condition $= \frac{\text{weight}^{1/3}}{\text{length}} * 50$)
- remove the fish from the dataset whose condition index is NA or less than 2

References

`R` for Data Science (Grolemund & Wickham, 2017)

- Functions: https://r4ds.had.co.nz/functions.html
- Iteration: https://r4ds.had.co.nz/iteration.html

Advanced `R` (Wickham, 2015)

- Subsetting: https://adv-r.hadley.nz/subsetting.html
- Control Flow: https://adv-r.hadley.nz/control-flow.html
- Functions: https://adv-r.hadley.nz/functions.html
- Conditionals: https://adv-r.hadley.nz/conditions.html
- Environments: https://adv-r.hadley.nz/environments.html

Workshop Materials & Recordings Available:

- Through the MSU Library YouTube channel: https://www.youtube.com/watch?v=xV7JNWTfsww
- Through the MSU Data Science Workshop hub: http://www.montana.edu/datascience/training/#workshop-recordings

How to Learn More About `R`

This material is intended to provide you with an introduction to using `R` for scientific analyses of data. The best way for you to continue to learn more about `R` is to use it in your research! This may sound daunting, but writing `R` scripts is the best way to become familiar with the syntax. This will help you progress through more advanced operations, such as cleaning your data, using statistical methods, or creating graphics.

The best place to start is playing around with the code from today's workshop. Change parts of the code and see what happens! Better yet, use the code from the workshop to investigate your own data!

Workshop build on: 2020-11-19 17:07:42