Lecture 10 - `if` Statements and Functions

Packages

```
# install packages for today's lecture
install.packages("dslabs")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

```
# load the necessary packages
library(dslabs)
```

```
# load murders dataset from the dslabs package
data(murders)
```

```
# view the first few lines
head(murders)
```

A data.frame: 6 × 5

	state	abb	region	population	total
	<chr>	<chr>	<fct>	<dbl>	<dbl>
1	Alabama	AL	South	4779736	135
2	Alaska	AK	West	710231	19
3	Arizona	AZ	West	6392017	232
4	Arkansas	AR	South	2915918	93
5	California	CA	West	37253956	1257
6	Colorado	CO	West	5029196	65

`if` Statements and Conditional Expressions

`if` Statements and Conditional Expressions

What is an `if` statement?

- `if` statements are scripts that incorporate conditional logic in your code
- That is, they run lines of code only when a "statement" is `TRUE`
- Conversely, `if` statements can avoid running lines of code when a "statement" is `FALSE`

Why use `if` statements?

- `if` statements can be used to automate data preprocessing and data analysis
  - For example, you can use `if` statements to run a procedure if there are missing values
- `if` statements can also be used to print a message to help with debugging (error handling)
  - For example, you can use an `if` statement to output a message if an `Age` value is less than 0

`if` Statement Syntax

An `if` statement in `R` is created using the following syntax

```
if (Boolean Expression) {
  if statement body
}
```

- `if`
  - An `if` statement is initiated using the `if` keyword
- Boolean Expression

- Following the `if` keyword is a Boolean statement in parenthesis
- This should statement should produce a single logical value `TRUE` or `FALSE`

- `if` Statement Body
  - If the Boolean expression is `TRUE`, then the code in the `if` statement body is executed
  - If the Boolean expression is `FALSE`, then the code in the `if` statement body is NOT executed

`if` Statement Examples

```
# store a logical
my_logical <- TRUE

# prints hello when statement is TRUE
if (my_logical) {
  print("Hello")
}
```

```
[1] "Hello"
```

```
# store a variable
x <- 5

# prints hello when statement is TRUE
if (x < 10) {
  print("Hello")
}
```

```
[1] "Hello"
```

```
# store a character
x <- "five"

# does not print hello since statement is FALSE
if (x == 5) {
  print("Hello")
}
```

```
print("Did not print 'Hello'")
```

```
     [1] "Did not print 'Hello'"
```

```
# 1 is equivalent to TRUE
if (1) {
  print("Hello")
}
```

```
     [1] "Hello"
```

```
# 0 is equivalent to FALSE
if (0) {
  print("Hello")
}
# does not print anything since the statement is false
```

`if, else, else if` Statements

- If the Boolean expression produces a `FALSE`, we can use an `else` statement to execute a different block of code

```
# if else statement
# if the first statement is TRUE then print "Hello"
# But if the first statement is not TRUE then print "Sorry, first statement not true"

x <- 5

if (x > 10) {
  print("Hello")

} else {
  print("Sorry, first statement not true")
}
```

```
     [1] "Sorry, first statement not true"
```

- If the Boolean expression produces a `FALSE`, we can use an `else if` statement to evaluate another Boolean expression and execute a different block of code if the expression is `TRUE`

```
# if else if statement
# if the first statement is TRUE then print "Hello"
# if the first statement is not TRUE and the second statement is TRUE then print "Sorr
# if the first two statements are not true, then default to third statement

x <- 5

if (x > 10) {
  print("Hello")

} else if (x < 7) {
  print("Sorry, first statement not true")

} else {
  print("Default to the third statement")
}
```

```
[1] "Sorry, first statement not true"
```

```
# if else if statement
# if the first statement is TRUE then print "Hello"
# if the first statement is not TRUE and the second statement is TRUE then print "Sorr
# if the first two statements are not true, then default to third statement

x <- 8

if (x > 10) {
  print("Hello")

} else if (x < 7) {
  print("Sorry, first statement not true")

} else {
  print("Default to the third statement")
}
```

```
[1] "Default to the third statement"
```

ifelse()

- A shorthand for of an `if-else` statement can be written using the `ifelse()` function

```
# traditional approach
if (FALSE) {
  print("Hello")
} else {
  print("Sorry, first statement not true")
}
```

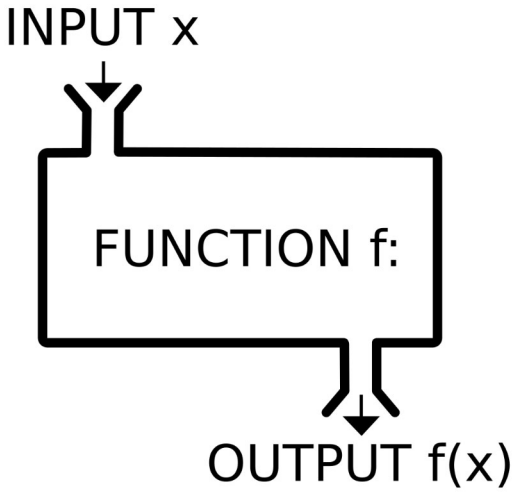
```
[1] "Sorry, first statement not true"
```

```
# using ifelse()
ifelse(FALSE, "Hello", "Sorry, first statement not true")
```

'Sorry, first statement not true'

Functions

What are functions in R?



Functions in R

Functions in R

1. take one or more arguments as input,
2. perform a given task using these inputs, and
3. return one or more objects as output

Example: The minimum function `min()`

1. takes a numeric vector as input,
2. finds the minimum of the vector, and
3. returns the minimum

Why use functions?

- Avoid running several lines over and over again (i.e. more efficient coding)
- Allows us to use functions others have built without having to know what it does internally!
    - This becomes useful when we start using functions to implement complex statistical algorithms

Function Syntax

An R function is created using the keyword `function()` using the syntax below

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
   return(output)
}
```

- Function Name
    - this is the actual name of the function and is stored as an R object of class type 'function'
- Arguments

    - Arguments (`arg_1, arg_2, ...`) are placeholders. When you call a function, you pass a value (e.g., number, vector, dataframe etc.) to the argument
- Function Body
    - This is a collection of R commands performed by the function when it is called
- Return Value
    - Returns the value of the function and is the last expression in the function body

Creating our own mode function `myMode()`

- Three measures of central tendency in data
    - mean
    - median
    - mode

- We are already familiar with the `mean()` and `median()` functions

```
# using the mean function
mean(murders$total)
```

184.372549019608

```
# using the median function
median(murders$total)
```

97

- `R` does not have a built-in function to find the mode
- We can build our own!

- The mode of a vector is the value that occurs most often.

- What is the mode of the following vector (without using R)?

```
c(4, 8, 4, 8, 4, 5)
```

How do we arrive at the answer above?

1. Identified the unique values of the vector
2. Counted the occurrence of the unique values
3. Returned the value that occurred most often

We can replicate these steps in R

- Identify the unique values of a vector and count occurence of unique values using `table()`

```
my_vector <- c(4, 8, 4, 8, 4, 5)

# the table() function finds the frequency/counts of
# all unique values in the vector
vector_freq <- table(my_vector)
vector_freq
```

```
my_vector
4 5 8
3 1 2
```

- Find the index of the maximum

```
# Find the index of the maximum
max_index <- which.max(vector_freq)
max_index
```

4: 1

- Find the number that occurs most often

```
# Find the number corresponding to the max index
mode_out <- names(vector_freq[max_index])
mode_out
```

'4'

```
class(mode_out)
```

'character'

- The variable `mode_out` is a `character` data type
- Therefore, we should convert `mode_out` to a `numeric` data type

```
# need to convert the value to a number
as.numeric(mode_out)
```

4

- Place them all together into a single script
- Not a function yet!!!

```
# vector
my_vector <- c(4, 8, 4, 8, 4, 5)

# count frequency of unique values
vector_freq <- table(my_vector)

# find index of maximum counts
max_index <- which.max(vector_freq)

# extract the number corresponding to max frequency
mode_out <- names(vector_freq[max_index])

# convert the mode to a numeric class
as.numeric(mode_out)
```

4

Creating the myMode function

- We can now convert the script above into a function

```
myMode <- function(my_vector) {

  # find frequencies
  vector_freq <- table(my_vector)

  # find max index
  max_index <- which.max(vector_freq)

  # find value with max index
  mode_out <- names(vector_freq[max_index])

  # our return statement
  return(as.numeric(mode_out))
}
```

```
# running the myMode() function
myMode(c(4, 8, 4, 8, 4, 5))
```

```
4
```

- We aren't quite done yet!
- What happens if we have multiple modes (i.e. more than one value have a tie for occurring most often)?

- What is the mode of the following vector?

```
c(4, 8, 4, 8, 4, 5, 8)
```

```
# running the myMode() function
myMode(c(4, 8, 4, 8, 4, 5, 8))
```

```
4
```

- The `which.max()` function only returns the index of the first maximum value
- We want to return all values corresponding to the maximum frequency

```
my_vector <- c(4, 8, 4, 8, 4, 5, 8)

# find frequencies
vector_freq <- table(my_vector)
vector_freq
```

```
my_vector
4 5 8
3 1 3
```

- Find the maximum frequency

```
# find maximum frequency
max_freq <- max(vector_freq)
max_freq
```

```
3
```

- We can use a Boolean expression to find which frequencies equal the max frequency

```
# find max index
max_index <- which(vector_freq == max_freq)
max_index
```

```
4: 1 8: 3
```

- Extract all numbers corresponding to the max frequency

```
mode_out <- names(vector_freq)[max_index]
mode_out
```

```
'4' · '8'
```

- Convert the modes to a `numeric` data type

```
as.numeric(mode_out)
```

```
4 · 8
```

- We can now convert the script above into a function

```
myMode <- function(my_vector) {

  # find frequencies
  vector_freq <- table(my_vector)

  # find max index(es)
  max_freq  <- max(vector_freq)
  max_index <- which(vector_freq == max_freq)

  # find value with max index
  mode_out <- names(vector_freq)[max_index]

  # our return statement
  return(as.numeric(mode_out))
}
```

```
myMode(c(4, 8, 4, 8, 4, 5, 8))
```

```
4 · 8
```

More Function Customizations

What if we want a warning if there is more than one mode?

- We can include print and if statements

- Determine the number of modes using the `length()` function

```
modes <- myMode(c(4, 8, 4, 8, 4, 5, 8))
modes
```

```
modes
```

```
length(modes)  # number of modes
```

```
4 · 8
2
```

- `paste()` is another `R` function that enables concatenate character strings and code

```
paste("Number of modes:", length(modes), sep=" ")
```

```
'Number of modes: 2'
```

- Incorporate the print statement in the function

```
myMode <- function(my_vector) {

  # find frequencies
  vector_freq <- table(my_vector)

  # find max index(es)
  max_freq  <- max(vector_freq)
  max_index <- which(vector_freq == max_freq)

  # find value with max index
  mode_out <- names(vector_freq)[max_index]

  # statement for number of modes
  nb_modes <- length(mode_out)
  if (nb_modes > 1) {
    print( paste("Number of modes:", length(mode_out), sep=" ") )
  }

  # our return statement
  return(as.numeric(mode_out))
}
```

```
# single mode
modes <- myMode(c(4, 8, 4, 8, 4, 5))
modes
```

```
4
```

```
# multiple modes
modes <- myMode(c(4, 8, 4, 8, 4, 5, 8))
```

```
     [1] "Number of modes: 2"
```

```
# check model output
modes
```

```
     4 · 8
```

What if we want to be able to turn on/off verbosity (i.e. turn on/off the comments)?

- Can add a function argument
- Function arguments can have default values
- We will make a verbosity argument default to `FALSE`
- When the argument is `TRUE`, the function will print the statement

```
myMode <- function(my_vector, verbose = FALSE) {

  # find frequencies
  vector_freq <- table(my_vector)

  # find max index(es)
  max_freq  <- max(vector_freq)
  max_index <- which(vector_freq == max_freq)

  # find value with max index
  mode_out <- names(vector_freq)[max_index]


  # only print if verbose is TRUE
  if (verbose) {

    # statement for number of modes
    nb_modes <- length(mode_out)
    if (nb_modes > 1) {
      print( paste("Number of modes:", length(modes), sep=" ") )
    }

  }
```

```
  ,

  # our return statement
  return(as.numeric(mode_out))
}
```

```
modes <- myMode(c(4, 8, 4, 8, 4, 5, 8))
modes
```

```
4 · 8
```

```
modes <- myMode(c(4, 8, 4, 8, 4, 5, 8), verbose = TRUE)
modes
```

```
[1] "Number of modes: 2"
4 · 8
```

- What if we want to return multiple outputs:
  - Modes of a vector
  - Number of modes
- Multiple outputs are returned as an ***unstructured list***

```
myMode <- function(my_vector, verbose = FALSE) {

  # find frequencies
  vector_freq <- table(my_vector)

  # find max index(es)
  max_freq  <- max(vector_freq)
  max_index <- which(vector_freq == max_freq)

  # find value with max index
  mode_out <- names(vector_freq)[max_index]


  # only print if verbose is TRUE
  if (verbose) {

    # statement for number of modes
    nb_modes <- length(mode_out)
    if (nb_modes > 1) {
```

```
    if (no_modes > 1) {
      print( paste("Number of modes:", length(modes), sep=" ") )
    }

  }

  # create list of outputs
  function_output <- list(modes        = as.numeric(mode_out),
                          number_modes = length(mode_out))

  # return list
  return(function_output)
}


modes <- myMode(c(4, 8, 4, 8, 4, 5, 8))
modes
```

```
$modes
     4 · 8
$number_modes
     2
```

```
# extract modes from output list
modes$modes
```

```
4 · 8
```

```
# extract number of modes from output list
modes$number_modes
```

```
2
```

Debugging

- What if we want to find the mode of a character vector? What do we change?

```
modes <- myMode(c("four", "eight", "four", "eight",
                  "four", "five", "eight"))
modes
```

```
Warning message in myMode(c("four", "eight", "four", "eight", "four", "five", "eig
“NAs introduced by coercion”
```

```
$modes
     <NA> · <NA>
$number_modes
     2
```

- We are attempting to convert a character vector to numeric
- We can avoid this by checking for this in our function

```
myMode <- function(my_vector, verbose = FALSE) {

  # find frequencies
  vector_freq <- table(my_vector)

  # find max index(es)
  max_freq  <- max(vector_freq)
  max_index <- which(vector_freq == max_freq)

  # find value with max index
  mode_out <- names(vector_freq)[max_index]


  # only print if verbose is TRUE
  if (verbose) {

    # statement for number of modes
    nb_modes <- length(mode_out)
    if (nb_modes > 1) {
      print( paste("Number of modes:", length(modes), sep=" ") )
    }

  }

  # our return statement
  if (class(my_vector) == 'numeric') {

    function_output <- list(modes        = as.numeric(mode_out),
                            number_modes = length(mode_out))
    return(function_output)

  } else {

    function_output <- list(modes        = mode_out,
                            number_modes = length(mode_out))
    return(function_output)

  }

}
```

```
modes <- myMode(c("four", "eight", "four", "eight",
                  "four", "five", "eight"))
modes
```

```
$modes
	'eight' · 'four'
$number_modes
	2
```

Built-in Functions

The `unique()` function finds all unique values in a vector or table

```
my_vector <- c(3, 2, 4, 4, 2, 4, 4, 4, 2)
unique(my_vector)
```

```
3 · 2 · 4
```

`sort()` sorts a vector

```
my_vector <- c(3, 2, 4, 4, 2, 4, 4, 4, 2)
sort(my_vector)
```

```
2 · 2 · 2 · 3 · 4 · 4 · 4 · 4 · 4
```

We are already familiar with the `mean()` function, but the `mean()` function, like many others, have additional arguments you can use

```
my_vector <- c(3, 2, 4, 4, 2, 4, 4, 4, 2, NA)
mean(my_vector)
```

```
<NA>
```

- The `mean()` function outputs `NA` if there are missing values in the numeric vector

- The `mean()` function has an `na.omit` argument with a default value of `FALSE`
- The `na.omit` argument can be changed to `TRUE` to remove `NA` values before calculating the mean

```
mean(my_vector, na.rm = TRUE)
```

3.22222222222222

- Identify which variables you have already stored in your workspace

```
ls()
```

'max_freq' · 'max_index' · 'mode_out' · 'modes' · 'murders' · 'my_logical' · 'my_vector' · 'myMode' · 'vector_freq' · 'x'

- Remove objects from your workspace

```
rm(list = ls())
```

```
ls()
```

Scope of functions

- Scope is essentially knowing where variables can and cannot be accessed
- This concept is best understood through example

```
# function adds 5 to x
add_5 <- function(x) {
  y <- x + 5
  return(y)
}
```

```
add_5(2)
```

```
7
```

- We stored the variable `y` in this function
- Let's see if we can access it

```
y
```

```
Error: object 'y' not found
Traceback:
```

Étapes suivantes : Expliquer l'erreur

- The variable `y` only exists with the function `add_5` (i.e. the scope of `y`)
- In this case, `y` is a **local variable** since it is define locally within a function

- What happens in the following?

```
# function adds x and y
add_y <- function(x) {
  return(x + y)
}
```

```
y <- 3
add_y(2)
```

```
5
```

- Careful when using variables that are not defined in a function!
- R will pull the variable globally if possible
- This can lead to annoying bugs down the road

```
# remove all variables
rm(list = ls())

# function adds 5 to x
add_5 <- function(x) {
  y <- x + 5
  return(y)
}


add_5(2)
y
```

```
7
Error: object 'y' not found
Traceback:
```

Étapes suivantes : Expliquer l'erreur