Conditionals, Loops, and Apply

http://datascience.tntlab.org

Module 3

Today’s Agenda

- Are you still doing this daily?
- Cheatsheet reminder
- Highlights from *Intermediate to R* (i.e., grab bag of functions)
  - Comparators and Boolean logic
  - Conditional statements
  - Loops
  - Functions
  - Apply-family
  - Regular expressions
  - Dates and times

Cheatsheet Reminder

- Why cheatsheets?
  - Convenient, all-in-one references
  - They’re like flashcards

- How to use a cheatsheet
  - If you’ve never heard of something but think you might need it, don’t try to learn it from a cheatsheet
  - If you think you need something you’ve never heard of in this class, you’re probably not thinking about the problem the way we discussed it in class (or on DataCamp)

Comparators and Booleans

- **Comparators from last week, plus new**
  - < and <=
  - > and >=
  - ==
  - !=

- **Boolean logic**
  - & or | , and parentheses

- Try this on paper, step by step
  - 9 < (5 %% 3) + 7 | (5 > 9) | FALSE

Comparators and Booleans

- **Boolean madness**
  - Don’t forget to re-state each complete comparison:
    - DO THIS: x > y & x > z
    - NOT THIS: x > y & z
  - TRUE & FALSE == ?
  - TRUE & TRUE == ?
  - FALSE & FALSE == ?
  - TRUE | FALSE == ?
  - TRUE | TRUE == ?
  - FALSE | FALSE == ?
  - Remember that R treats TRUE as 1 and FALSE as 0, which is convenient in a lot of specific situations.

Conditional Statements

- **if** # not a function
- **else** # often nested into "else if"

- Curly braces can be confusing
  - They indicate an entire grouping of code follows the last conditional; you can omit them if there is only one line.
  - These are identical
    - if (x > 5) { x <- x - 1 }
    - if (x > 5) x <- x – 1
  - R always looks for a distinct line of code to run after an if or else; curly braces count as a line of code even if they are empty
  - When in doubt, use curly braces!! They don't hurt anything.

Nested Conditional Statements

- Consider this statement:

-
```
if (x < 0) {
   print("Negative")
} else if (x > 0) {
   print("Positive")
} else
   print("Zero")
```

- How does this work, step by step, for:
  - x <- -5
  - x <- 0
  - x <- 8
  - x <- “”
  - x <- “1”
  - x <- “0”   # beware *type coercion*

Loops

- while () { }   # these are not functions
- for () { }   # in other languages, this is a foreach
  - for (x in 1:10) { }

- You learned about **break** and **next**, but try not to use them.

- Curly braces are also optional in all loop specifications, but often help readability
  - After setting x to 0, try: while(x<10) print(x<-x+1)

Nesting Functions

- We've already talked about functions (and their parameters)

- The output of a function is called its *return*
- A return can be *passed* to another function or used in any other way you want; think of it like its own (temporary) variable

- When designing nested functions, start by testing the innermost function and gradually build your way out:
  - "I need to store the mean of two variables in a list"
  - mean(x)
  - mean(y)
  - list(mean(x), mean(y))
  - means_list <- list(mean(x), mean(y)) # this becomes the only line in your .R

Writing Functions

- funcname <- function(param1, param2withdefault = 5) { return(something) }

- What does this do, and why?
  - add_one <- function(x, y = 1) return(x + y)

  - What do these return?
    - add_one
    - add_one()
    - add_one(1)
    - add_one(2)
    - add_one(4,1)
    - add_one(4,5,2)

- Worry about **scoping**!!
  - Variables inside a function *only exist inside that function.*
  - Parameters in and the return out *as values only;* their names are lost.

Anonymous Functions

- A function with no name

- Useful if you only need to use a function for a single purpose, such as in an apply-family statement

- Just don’t assign it to a variable

Packages

- Two commands to remember:
  - install.packages(“stringpackagename”)
  - library(stringpackagename)

- We will use a *lot* of packages in this course.
  - Each provides different functionality and capabilities.
  - Each is written by someone different, so do not expect consistency in variable names or parameters.
  - Groups of people have banded together to address this problem within what are called *frameworks*. We will explore a few of these.

Apply Family

- **lapply():** For each item in a vector or list, run a function on it, and return a vector or list
  - You usually don't need it for vectors, because you can just run most functions on vectors directly
    - lowercase_v <- tolower(string_v)
    - lowercase_v <- lapply(string_v, tolower)
  - You can add parameters needed for the function you want to run as additional parameters to lapply()
  - lowercase_vector <-
- **sapply():** Same as lapply, but simplifies data type *if possible*
  - Example
    - If you lapply over a dataframe with 5 variables to calculate maximums using max(), it'll run the function on each variable, one at a time, and return a 5-item list of maximums
    - If you sapply over a dataframe, it'll do the same but return a 5-item vector
- **vapply()**: Same as sapply, but pre-specifies return format.

Regular Expressions (regex)

- **grepl():** Searches for a regex “pattern” within a vector of character strings and returns TRUE if found
- **grep():** Searhces for a regex “pattern” within a vector of character strings and returns a vector of indices from that vector where that pattern is found
  - agreements <- c(“yes”,”no”,”yes”)
  - Results for “yes” pattern:
    - grepl: TRUE FALSE TRUE (vector)
    - grep: 1 3 (vector)
- Guesses?: ^(\([0-9]{3}\) | [0-9]{3}-) [0-9]{3}-[0-9]{4}$
- We will return to this concept later in MUCH more detail.

Dates and Times

- Sys.Date() and Sys.time() return the date and the time
  - You probably won't need Sys.Date() much
  - Sys.time() returns the number of seconds since the **Unix epoch** (Jan 1 1970 UTC)
  - A time in this format is also called a **Unix timestamp** or **POSIX time**. It is a universal time format in computing.
  - Having a time format that is an integer makes a lot of processes much easier

- You can convert a POSIX-ish formatted character vector (e.g. a string that reads "2018-02-03 05:12:03") into a POSIX class using as.POSIXct(), then convert it into a timestamp using unclass() or as.numeric()

Useful Mathematical Functions

mean()	# mean!
sd()	# standard deviation
abs()	# absolute value
round()	# round a value
min()	# minimum value
max()	# maximum value

Other Useful Data Functions

seq()	# create a vector of numbers in order
sort()	# sorts!
length()	# length of a vector, list, df, matrix
nchar()	# number of characters in a character vector
identical()	# check if two objects are *exactly* identical
str()	# display object structure
typeof()	# display atomic class (be careful)
is. functions	# check if something is the class you think
as. functions	# recast a variable as another class
unlist()	# **tries** to convert a list into a vector
na.rm	# this is not rm.na