Software Carpentry Workshop

Berkeley Institute for Data Science

June 12, 2018

Instructors: Diya Das, Rebecca Barter, Richard Barnes

Helpers: Jenna Baughman, Caroline Cypranowska

Building Programs with R

Workshop website: https://bids.github.io/2018-06-11-bids/

http://swcarpentry.github.io/r-novice-inflammation/
Instructors: Rebecca Barter (rebeccabarter@berkeley.edu), Diya Das (diyadas@berkeley.edu), Richard Barnes (richard.barnes@berkeley.edu)
Helpers: Caroline Cypranowska (casegura@berkeley.edu), Jenna Baughman (jtb@berkeley.edu)

LINK TO INSTRUCTOR GUIDE: http://swcarpentry.github.io/r-novice-inflammation/

Attendees, please sign in:

- Name, email, (and department/university if you like)
- Setsuko Wakao swakao@berkeley.edu, PMB, UCB
- Sean Carim, scarim@berkeley.edu, PMB, UC Berkeley
- Michelle Lu, michelle.lu@berkeley.edu, MCB, UC Berkeley
- Aaron Pomerantz, pomerantz_aaron@berkeley.edu, Integrative Biology
- Kaley Curtis, kaley_curtis@berkeley.edu, Psychology, UC Berkeley
- Nidhi Chandra, nidhivchandra@berkeley.edu, Psych/Social Welfare, UC Berkeley
- Ariane Baudhuin, arianebaudhuin@berkeley.edy, MCB, UC Berkeley
- Huasong Lu, luhuasong@berkeley.edu MCB UC Berkeley
- Mark Yashar, myashar@berkeley.edu, CITRIS, UC Berkeley
- Laura Guzman
- Ariel Li, lipeixian92@gmail.com, CBE, UC Berekeley
- Sophia Friesen, sophia_friesen@berkeley.edu, MCB, UC Berkeley
- Luisa Tacca, latacca@berkeley.edu, QB3, UC Berkeley
- Hsin-Jung Chou, chouhj@berkeley.edu, MCB, UC Berkeley
- Juan Wan, juan.wan@berkeley.edu, PMB, UC Berkeley
- Matthew Parker, mwparker@berkeley.edu, MCB
- Lu Dong, lu.dong@berkeley.edu, Psych, UC Berkeley
- Tom Parkinson, tom.parkinson@berkeley.edu, Arch, UC Berkeley

- Bodil Cass, bncass@ucdavis.edu, Entomology & Nematology, UC Davis
- Maroof Adil, maroof@berkeley.edu, QB3
- Danielle Fleming, myteachermsfleming@berkeley.edu, Grad. School of Ed.
- Fuguo Jiang, jiangfg@berkeley.edu, MCB/QB3, Berkeley
- Sandra Muroy, s.muroy@berkeley.edu, MCB
- Raz Bar-Ziv, barziv@berkeley.edu, MCB, UC Berkeley
- Jonathan Qu, jonathanlqu@gmail.com, IB UC Berkeley
- Xujia Zhou, xujia.zhou@ucsf.edu, PSPG, ucsf
- Meng Li, limengpmb@berkeley PMB, UC Berkeley
- Anchal Mehra, anchalmehra@berkeley.edu, PMB, UC Berkeley
- Carl Schreck, carl.shreck@berkeley.edu, QB3
- Teruyuki Matsunaga
- Cliff Vuong, c.vuong@berkeley.edu, MCB
- Yawei Yu, yyyu@berkeley.edu, MCB
- Yilin Ye, angelynaye@berkeley.edu, Stats & ORMS

•

Monday AM: R

Download R here (choose a Berkeley CRAN mirror): https://cran.r-project.org/mirrors.html
Download RStudio here: https://www.rstudio.com/products/rstudio/download/ (get the desktop version)
Download the data here and put it on your Desktop: http://swcarpentry.github.io/r-novice-inflammation/files/r-novice-inflammation-data.zip

Notes:
R Studio console is like Terminal window. Can type things like **1 + 1** and it will return **2**

R script is where we save the commands we're going to run in the console.

In R the function goes first, then the arguments inside parentheses and quotations, e.g. setwd("~/Desktop/r-novice-inflammation-data")

setwd = set work directory
for example: setwd("~/path/")

comment everything so we know what we're doing

setwd(dir ="~/Desktop/r-novice-inflammation-data") .. if you hit tab after the first bracket it fills in what the argument is

read.csv() reads in a csv file
read_csv() also works [... this doens't work for me?] <- sorry, this is part of the readr package in the tidyverse (I think Rebecca had it loaded)
not read-csv() because - is read as minus
readCsv() also works [... this doens't work for me?]

mouse over the function and it will say what arguments it needs. The one without an = is the only required one.

e.g. **read.csv(file = "data/inflammation-01.csv")**

This file didn't have row names. So R took the first row and used them as names, adding an X in front because row names can't be numbers.
To fix this, we add the argument header:
read.csv(file = "data/inflammation-01.csv", header = FALSE)

TRUE and **FALSE** must be capitalised <- R is special and can also interpret T and F; however True and False do not work

Separate arguments with comma space **(,)**. Space isn't necessary but makes the code easier to read.

Control+ENTER: shortcut for RUN (put the cursor on that line) (command + enter also works on macs)

to save something as a variable (this is similar to $variable we used yesterday in the terminal), we use a backwards arrow to **assign** it
define a variable caleld "weight_kg" with a value of 55
weight_kg <- 55

define variables with **<-**
functions use **=**

Can now use the saved variable for computations, e.g.
Multiply weight_kg by 2.2
2.2 * weight_kg
[1] 121

Can define the result as a new variable
weight_lb <- 2.2 * weight_kg

When you define a new variable it doens't print anthing in the console, you need to type the name to get it to print out the value of the variable.

If you then change weight_kg, it doens't change or 'update' the value of weight_lb, unless you run the script again.

Like in the terminal, when you hit up in the console it shows the previous commands used

head() shows the first 6 rows
tail() shows the last 6 rows
It prints the row numbers on the left side

comma # adjusts how many rows are printed e.g. **head(data, 1)** prints only the first row

class() shows the type of data storage. e.g. data.frame

in R, 2.0 is coded the same as 2.. i.e. it doensn't care about the 0 deicmal places

dim() prints the immensions of the data

to access individual cells:
data[#1,#2] where #1 = column number, #2 = row number
"indexing"

look at the first two entries of the third column
inflammation[c(1,2), 3]
the **c** groups the "1,2" together "combine"

print sequential values use colon
e.g. **c(1, 2, 3, 4, 5)** is the same as **1:5**

extracting whole columns (i.e. "all the rows") - leave the entry blank
e.g. get the whole fifth column
inflammation[, 5]
e.g. extract the whole first row
inflammation[1,]

max() calculates maximum
e.g. calculate maximum of whole first row
max(inflammation[1,])

mean() calculates the average
median() calcuates the median

In our dataframe, the class of the rows is 'dataframe', not 'integer' like the columns, because it keeps the header.
We can force it to be numeric with a function inside our function. Then the median calculation will work, which needs a numeric variable.
as.numeric() forces the data to be numeric, stripping the column headers away, making it a 'vector' format
class(inflammation[4,]) returns data.frame, which is why running median(inflammation[4,]) doesn't work

?function_name shows the help info {base} -- base is the name of the package the function is storred in

apply() calculates something ("applies a function") across the whole dataset

if you put the arguments in the order R lists them you don't have to define them with the =

usage: apply(dataset, row(1) or column(2), function)
e.g. apply(inflammation, 1, mean)

seq() gives a sequence of numbers from #, to #, skipping by #
e.g. seq(2, 40, 2)
gives every other integer from two to forty (2, 4, 6, 8, ... 40)

can use **####### text ######** to create structure within the script then navigate in the "Visualization" tab

plot() makes a basic graph

ggplot2 package makes better graphs

install.package("package_name") to install a package
only have to do this once ever, but need to load in the package each session
with **library()**
can put these listed at the top of the script <- it's a good idea so that when someone else runs your script, they can see what packages they need first! (otherwise your code will run for awhile, then there will be an error because R can't find the package :( )

to check if a package is installed (browse in RStudio) or type installed.packages()
- that probably gives you a lot of information you don't particularly need, but it can be handy for finding the **package version**

if something is weird and R is complaining about things being corrupted, you can remove the package and try installing again with:
remove.packages("name of package to remove in quotes")
- - - - -

Functions

syntax: function(argument1, argument2)
e.g. read.csv, mean, apply are all functions

function() defines a function

name_of_function <- function(name_of_argument) { body of the function that does the calculation and can use additional arguments and tells us what to **return()** as the result}

e.g. define a function that converts temperature from F to K

```
#define function
fahrenheit_to_kelvin  <- function(temp_F) {
  temp_K <- ((temp_F - 32) * (5 / 9)) + 273.15
return(temp_K)
}
```

# sample
fahrenheit_to_kelvin(32)
[1] 273.15

Can use functions within functions
e.g. go from fahrenheit_to_celsius using two functions we already defined
```
fahrenheit_to_celsius <- function(temp_F) {
 temp_K <- fahrenheit_to_kelvin(temp_F)
 temp_C <- kelvin_to_celsius(temp_K)
 return(temp_C)
}
```

we could also string them together like this, with a function inside a function:
```
  kelvin_to_celcius(fahrenheit_to_kelvin())
```

  standard documentation goes within a function as comments in the second line
  e.g.
```
  my_sum <- function(num1, num2) {
    # explain what the function does
      # Args:
        #   num_1: a number
        #   num_2: a number
      # Returns:
        #   a number corresponding to the sum of num_1 and num_2
      # Example:
        #   my_sum(1, 3)
    num_sum <- num1 + num2
    return(num_sum)
  }
```

can provide defaults in the arguments that will be used if we don't define anything
e.g. my_sum <- function(num_1 = 4, num_2 = 5) {...}

Powerful packages:
https://www.tidyverse.org/

People say that for loops in R are slow, so tend to use purrr instead from these tidyverse packages

variables defined inside the function are not accessible outside the function

dev.off() escapes if we get stuck in the plot function (?)

paste0() takes arguments and concatenates them
e.g. paste0("banana", "apple")
[1] "bananaapple"

for loop

```
for (variable in collection of variables) {
   do things
   }
```

- - - - -

Global environment shows all the variables we have storred

ls() lists all the variables in the workspace

list.files() lists all the files in a directory or folder
. = current working directory
full.names = TRUE option returns the entire path, not just from the current working directory

file is a function in R so don't use that as a variable name because we don't want to overwrite it

ctrl+L in the console clears the space

conditionals

```
if (this is true) {
   do this }
   else {
      do this instead
      }
```

== test for equality (because single = assigns a value)

R evaluates character strings to be positive
e.g. "hi" > 0
[1] TRUE

Option in Code menu to "Reindent lines" makes the code pretty again

>= this means greater than equal
<= this means less than equal
!= this means not equal to

& means "AND"
| means "OR"

Challenge exercise: make a function that boxplots a vector (supplied as input by the user) if it contains more values than some threshold (also supplied as input by the user)

```
plot_dist <- function(myvector, threshold){
# your input variable names are myvector and threshold
  if (length(myvector) > threshold){
 boxplot(vector)
}
}
```

can combine multiple statements with **else if**

Expansion on this using conditions: let's make a histogram if the length of the vector is <= threshold :

```
plot_dist <- function(myvector, threshold){
# your input variable names are myvector and threshold
  if (length(myvector) > threshold){
 boxplot(vector)
} else {
    hist(vector)
    }
}
```

To run it on some vectors and thresholds:

```
plot_dist(inflammation[,10],threshold=10) # our input data is the 10th column of inflammation
```

or

```
myvector <- c(1,3,5,9)
plot_dist(myvector, threshold=10)
```

can combine multiple statements with **else if**

the **length** of a **matrix** is the total number of elements in the matrix

the **length of** a **dataframe** is the number of columns

graph functions:
boxplot()
hist()
stripchart()

which.max() tells us which element is the max
e.g. which.max(c(1,1,3,5))
[1] 4

sessionInfo() prints version info about R and attached or loaded packages

running an R script from the shell (might want to do this to get the supercomputer to run your file)

save the R script
in the Terminal, run the script with
Rscript name_of_script
where **Rscript** is the command and **name_of_script** is the name of the R script file

https://www.rstudio.com/resources/cheatsheets/

R studio IDE Cheat Sheet

(what is IDE? Google: "An integrated development environment (***IDE***) is a software application that provides comprehensive facilities to computer programmers for software development. An***IDE*** normally consists of a source code editor, build automation tools, and a debugger. Most modern ***IDEs*** have intelligent code completion.")

R studio has a **find and replace** bar at the top (little magnifying glass)

you can navigate through files using the tab in the pannel on the right called "Files" other tabs here "Plots", "Packages" (can browse and see the help contents, search)

in the console, use **??word** to search for a word within the packages

there's an R mailing list that's archived online you can find it by google

history tab in top right (next to "Environment" where the variables showed up) keeps a list of the things you ran in the console. You can highlight them and use the little green arrow to send them 'to source' that copies it into the R script.

str() structure of the object i.e. all the information about that

factors are used to represent categorical data

is.na() checks if something is NA

table() makes a table counting things? <- exactly

all() check the whole data
any() checks if any of the data

Inf is a special value = infinity

R Markdown makes file output in a specific format that contains text, code and plots (create a new script that's an R Markdown instead of an R Script) - useful for sharing code, output, & explanation with others

use library() not require() to load a package because require() doesn't return an error if it doesn't work

gridExtra // layout command in R is useful

package **dplyr** allows piping, similar to "|" in shell <- there's a cheatsheet

Fun with plotting

base R (without ggplot):

```
colpal <- c("red", "green", "blue")
plot(iris$Sepal.Length, iris$Sepal.Width, col = colpal[iris$Species])
legend("topright", levels(iris$Species), fill = colpal)
```

ggplot (and also dplyr)

```
head(iris)
library(ggplot2)

ggplot(iris, aes(x = Sepal.Length,
                 y = Petal.Length)) +
  geom_point(aes(colour = Sepal.Width),
             size = 6,
             alpha = 0.5) +
  geom_smooth() +
  theme_classic()

ggplot(iris) +
  geom_violin(aes(x = 1, y = Sepal.Width)) +
  geom_jitter(aes(x = 1, y = Sepal.Width, col = Petal.Length),
              width = 0.1) +
  facet_wrap(~Species) +
  theme_bw() +
  ylab("Sepal Width") +
  xlab("")

install.packages("dplyr")
library(dplyr)
iris %>% head()

iris %>%
  filter(Species == "setosa") %>%
  summarize(mean_sepal_length = mean(Sepal.Length))

iris %>%
  group_by(Species) %>%
  summarise(mean_sepal_length = mean(Sepal.Length)) %>%
  ggplot() +
  geom_point(aes(x = 1, y = mean_sepal_length))
```

-
-