

R basics and flow control

Michael Alfaro

9/22/2020

Getting started

Welcome to the EEB Quantitative skills bootcamp.

R basics

Use `setwd()` to set your directory.

```
setwd("~/Dropbox/bootcamp_examples")
```

And use `getwd()` to see the working directory.

```
setwd("~/Dropbox/bootcamp_examples")  
getwd()
```

```
## [1] "/Users/michaelalfaro/Dropbox/bootcamp_examples"
```

comments

The `#` character is used to mark comments. R ignores everything after the `#` to the end of the line

```
2 + 2
```

```
## [1] 4
```

```
#2 + 3
```

R ignores the line `2 + 3` because of the `#`

getting help

R has many options for getting help. You can use the `help()` function on any function:

```
help(lm)
```

You can also use “?” before a function.

```
?lm
```

Two question marks (“??”) tells R to use fuzzy matching on the function name. R will search for functions with names similar to your query in all installed packages. “?” and “??” won’t evaluate in this document but you should try them at your command prompt to see the help files.

```
??lm
```

the `c()` function

The `c()` function combines elements into a vector and is one of the most commonly use functions in R.

```
grad.school.tips <- c( "use a reference manager", "learn a
```


You can use `cat()` to print objects to a screen.

```
cat(grad.school.tips, sep = "\n")
```

```
## use a reference manager  
## learn a programming language  
## write lots of papers
```

install

install() is used to install new packages:

```
install.packages(c("geiger", "picante"), dep = T)
```

variables

As you work in an R session, any variables that you declare will be stored in the session. If you want to see all objects that you have created in your session, use the `ls()` function.

```
xx <- 1000  
ls()
```

```
## [1] "grad.school.tips" "xx"
```

the nuclear option

To remove EVERYTHING, use `rm(list = ls())`. This passes the contents of `ls()` to `rm()` so that everything in the environment is deleted. It is often useful to start your script with this command so that you can be sure that any variables you declare have not been assigned a value already.

```
xx <- 100
names <- c("Paul", "Griffin", "Rivers")
numbers <- runif(100)
ls()
```

```
## [1] "grad.school.tips" "names"           "numbers"
```

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

```
ls() #character(0) means the function has returned an empty
```

```
## character(0)
```

source()

The `source()` function will load functions and variables from another R script into your R session. This means that you can save and reuse functions that you develop for other projects.

```
getwd()
```

```
## [1] "/Users/michaelalfaro/Dropbox/git/bootcamp2019/eeb2019"
```

```
source("~/Dropbox/bootcamp_examples/source.example.R")  
#make sure the path to the source file is specified correctly  
all.I.know.about.life.I.learned.in.grad.school() #a function
```

```
## I want to thank the reviewer for their 15 pages of singl
```

Read in the tree

The first thing we will do is use `read.tree()` to read in a phylogenetic tree. `readtree()` is in the Ape library, so make sure you have that installed. The tree file is a text file that contains information about the tree structure and tip labels in Newick format. Use a text editor to look at this file if you are curious.

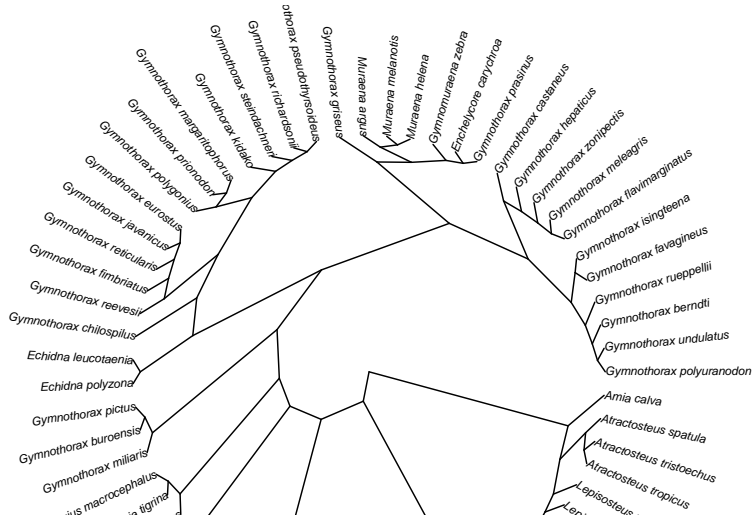
```
library(ape)
tt <- read.tree("~/Dropbox/bootcamp_examples/tree.tre")
###see elements of an object
attributes(tt)
```

```
## $names
## [1] "edge"          "edge.length" "Nnode"        "node.labels"
## [6] "root.edge"
##
## $class
## [1] "phylo"
##
## $order
```

pruning the tree

This tree is giant! Lets prune down and plot the pruned tree.

```
pruned.tree <- drop.tip(tt, tt$tip.label[1:7900])  
plot(ladderize(pruned.tree), cex = 0.5, type = "radial")
```



reading in data

```
# d contains length data, family, species, order, etc  
inpath = "~/Dropbox/bootcamp_examples/data.txt"  
dd <- read.table(inpath, header=T, sep='\t', as.is = T);  
  
###NOTE: R by default reads character columns as FACTORS.
```


a quick note about data frames

You have just read your data in as a data frame object. check this with the `str()` function

```
str(dd)
```

```
## 'data.frame':    92 obs. of  2 variables:  
## $ species: chr  "Naso_brevirostris" "glass_fish" "Zebra  
## $ mode    : chr  "BCF" NA "BCF" "BCF" ...
```

```
#a data frame is a collection of columns where every object  
#get the dimensions of a data frame
```

```
dim(dd)
```

```
## [1] 92  2
```

```
length.dd <- dim(dd)[1] #what does this line do?  
#dimensions are rows, columns  
attributes(dd)
```

```
## $names
```

```
## [1] "species" "mode"
```

adding data to a data frame

Lets create some size data and add it to the data frame

```
#get 92 random variables  
size <- runif(length.dd)
```

```
#you can add columns to an existing data frame with cbind  
head(dd) #before
```

```
##                species mode  
## 1      Naso_brevirostris  BCF  
## 2           glass_fish <NA>  
## 3      Zebrasoma_scopas  BCF  
## 4  Apogon_nigrofasciatus  BCF  
## 5  Cheilodipterus_macrodon  BCF  
## 6 Cheilodipterus_quinque-lineatus  BCF
```

```
dd<- cbind(dd, size)  
head(dd) #after
```

```
##                species mode      size
```

accessing data frame elements

You can use the "\$" operator to access rows and head() and tail() check a data frame

```
names(dd) #these are the names of the columns we could access
```

```
## [1] "species" "mode"      "size"
```

```
#dd$species #all the species
```

```
head(dd$species)
```

```
## [1] "Naso_brevirostris" "glass_fish"
```

```
## [3] "Zebrasoma_scopas" "Apogon_nigrofascia"
```

```
## [5] "Cheilodipterus_macrodon" "Cheilodipterus_qu"
```

```
tail(dd$species) # use these functions to check that data is
```

```
## [1] "Pomacentrus_coelestis" "Pomacentrus_lepidogenys"
```

```
## [3] "Pomacentrus_nagasakiensis" "Premnas_biaculeatus"
```

```
## [5] "Stegastes_apicalis" "Canthigaster_valentini"
```

```
#you can pull out individual columns
```

subsetting

use the `[]` after a data frame to access specific cells, rows, and columns

```
dd[1,1] # entry in row 1, column 1
```

```
## [1] "Naso_brevirostris"
```

```
dd[1,2] # entry in row 1, column 2
```

```
## [1] "BCF"
```

```
dd[1,3] # entry in row 1, column 3
```

```
## [1] 0.5503558
```

```
dd[1,] # row 1, all columns
```

```
##           species mode      size  
## 1 Naso_brevirostris  BCF 0.5503558
```

```
dd[,2] # all rows, column 2
```

accessing by row name

Naming rows allows you to access a row by name (Note that rownames are a part of a data frame but not a separate column of the data frame)

```
head(rownames(dd))
```

```
## [1] "1" "2" "3" "4" "5" "6"
```

```
rownames(dd) <- dd$species
```

```
head(rownames(dd))
```

```
## [1] "Naso_brevirostris"      "glass_fish"
## [3] "Zebrasoma_scopas"      "Apogon_nigrofascia
## [5] "Cheilodipterus_macrodon" "Cheilodipterus_qu
```

```
str(dd)
```

```
## 'data.frame':      92 obs. of  3 variables:
## $ species: chr  "Naso_brevirostris" "glass_fish" "Zebrasoma_scopas"
## $ mode : chr  "BCF" NA "BCF" "BCF" ...
## $ size : num  0.5504 0.468 0.9805 0.1684 0.0481 ...
```

A bit more on subsetting

```
# a bit on subsetting  
dd[5:10,] # rows 5-10, all columns
```

```
##  
## Cheilodipterus_macrodon Cheilodipterus_macrodon  
## Cheilodipterus_quinquelineatus Cheilodipterus_quinquelineatus  
## Chaetodon_aureofasciatus Chaetodon_aureofasciatus  
## Chaetodon_auriga Chaetodon_auriga  
## Chaetodon_baronessa Chaetodon_baronessa  
## Chaetodon_citrinellus Chaetodon_citrinellus
```

```
dd[5:10,3] # rows 5-10, column 3
```

```
## [1] 0.04812316 0.19206977 0.93339070 0.28761622 0.837791
```

which()

if we store the results of this `which()` function we can subset the dataframe to include only MPF swimmers

```
#if you want only the MPF swimmers, you can use the which()  
which(dd$mode == 'MPF')
```

```
## [1] 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  
## [26] 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75  
## [51] 84 85 86 87 88 89 90 91 92
```

```
mpfs <- which(dd$mode == 'MPF') #stores rows of mpf swimmers  
mpf_swimmers <- dd[mpfs,] #stored this as a separate df  
head(mpf_swimmers)
```

```
##                                     species mode  
## Acanthurus_blochii          Acanthurus_blochii  MPF 0.927  
## Acanthurus_dussumieri       Acanthurus_dussumieri  MPF 0.892  
## Acanthurus_lineatus         Acanthurus_lineatus  MPF 0.488  
## Acanthurus_nigrofuscus      Acanthurus_nigrofuscus  MPF 0.308  
## Acanthurus_olivaceus        Acanthurus_olivaceus  MPF 0.614
```

How would you make dataframe with all of the big (size > 0.9) species only?

```
head(dd)
```

```
##                                     sp
## Naso_brevirostris                 Naso_breviro
## glass_fish                       glass
## Zebrasoma_scopas                 Zebrasoma_s
## Apogon_nigrofasciatus           Apogon_nigrofasc
## Cheilodipterus_macrodon         Cheilodipterus_mac
## Cheilodipterus_quinquelineatus Cheilodipterus_quinquelin
```

```
which(dd$size > 0.9) #shows us rows with large fish in the
```

```
## [1] 3 7 14 34 56 57 76 78 83 86 88
```


R challenge

make a new data frame with large species only

hints

- ▶ use the **which()** function to select only rows of some size or greater
- ▶ the **\$** operator lets you specify columns from a data frame
- ▶ you can subset a data frame by specifying a list of row names within the square brackets **[]**

Once you have it (or have something) try committing it to your github repository

one solution

```
big.fish <- dd[which(dd$size > 0.9),] #remember the , after  
head(big.fish)
```

```
##                                                                 species  
## Zebrasoma_scopas                                             Zebrasoma_scopas  
## Chaetodon_aureofasciatus                                     Chaetodon_aureofasciatus  
## Chaetodon_plebius                                           Chaetodon_plebius  
## Acanthurus_blochii                                           Acanthurus_blochii  
## Labroides_dimidiatus                                         Labroides_dimidiatus  
## Macropharyngodon_negrosensis Macropharyngodon_negrosensis
```

checking for NAs

Sometimes your data frame will include missing values. Often you will want to exclude these rows from the analysis. There are several ways to do this.

```
#ways to check for NAs
```

```
head(dd) # there are NAs in the data
```

```
##                                     sp
## Naso_brevirostris                  Naso_breviro
## glass_fish                         glass
## Zebrasoma_scopas                   Zebrasoma_s
## Apogon_nigrofasciatus              Apogon_nigrofasc
## Cheilodipterus_macrodon            Cheilodipterus_mac
## Cheilodipterus_quinquelineatus Cheilodipterus_quinquelin
```

```
head(is.na(dd))
```

```
##                                     species mode size
## Naso_brevirostris                  FALSE FALSE FALSE
## glass_fish                         FALSE  TRUE FALSE
```

removing NAs

We can remove these missing cases in a variety of ways.

```
#one way to get only complete cases  
cleaned_1 <- dd[complete.cases(dd),]  
#another  
cleaned_2 <- na.omit(dd)  
  
dd <- cleaned_1
```

Note that we have reassigned the cleaned data set to **dd** so that **dd** only includes the complete cases.

Renaming data frame entries and matching data objects

The following example demonstrates how to manipulate data frames from different sources to find elements in common. We will not go through this example in class this year due to constraints on time. However I am including it as a optional exercise to work through for those of you who might find it useful.

You will often need to find common elements between two data sets before you can do an analysis of the data. In our example we have a phylogeny that is taken from one study and a data set on swimming mode that is taken from another. Problems:

- ▶ tree huge
- ▶ data may not match species in tree

setdiff()

setdiff() is a useful tool. setdiff() compares two lists and returns the items in the first list that are not present in the second list. Also see intersect(), union(), and setdiff().

```
setdiff(dd$species, tt$tip.label)
```

```
## [1] "Apogon_nigrofasciatus"      "Cheilodipterus_quadrangulatus"
## [3] "Chaetodon_lunulatus"        "Chaetodon_plebius"
## [5] "Chaetodon_rainfordii"      "Heniochus_singularis"
## [7] "Amblygobius_decussatus"    "Scolopsis_bilineata"
## [9] "Acanthurus_lineatus"       "Sufflamen_chrysops"
## [11] "Cheilinus_chlorurus"       "Cirrhilabrus_punctatus"
## [13] "Oxycheilinus_digrammus"    "Pseudocheilinus_melanocephalus"
## [15] "Thalassoma_janseni"        "Chrysiptera_browni"
## [17] "Neoglyphidodon_melas?"
```

Changing one field in a record

OK, it looks like there are 18 species in the swimming data set that don't match the tree. Some of these mismatches are due to spelling errors or taxonomic inconsistency between the two data sets. Here is one way we could correct a name.

```
dd$species[which(dd$species == 'Chaetodon_plebius')] <- 'Chaetodon_plebius'
```

matching rest of data to tree

```
del_from_data <- setdiff(dd$species, tt$tip.label)
# tips with data not in tree
del_from_data
```

```
## [1] "Apogon_nigrofasciatus"      "Cheilodipterus_quadrangus"
## [3] "Chaetodon_lunulatus"       "Chaetodon_rainfordi"
## [5] "Heniochus_singularis"     "Amblygobius_decussatus"
## [7] "Scolopsis_bilineatus"     "Acanthurus_lineatus"
## [9] "Sufflamen_chrysopterus"    "Cheilinus_chlorourus"
## [11] "Cirrhilabrus_punctatus"   "Oxycheilinus_dignatus"
## [13] "Pseudocheilinus_hexataenia" "Thalassoma_jansseni"
## [15] "Chrysiptera_brownriggi"    "Neoglyphidodon_nigricans"
```

#keep all species in data file except those that match the

```
pruned_data <- dd[!(dd$species %in% del_from_data),]
```

```
setdiff(pruned_data$species, tt$tip.label) # this should print
```

```
## character(0)
```


matching tree to data

Now we've pruned the data set. How can figure out what tips of the tree to prune? `setdiff()` again, but this time switching the order of the arguments

```
not.in.dd <-setdiff(tt$tip.label, pruned_data$species )  
length(not.in.dd) #this will be a large number because the
```

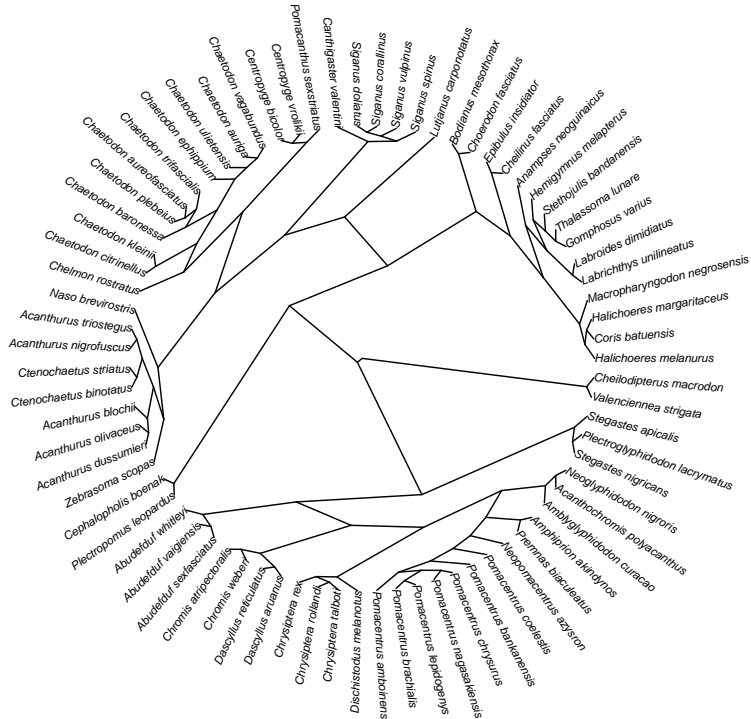
```
## [1] 7888
```

```
head(not.in.dd)
```

```
## [1] "Polyodon_spathula"           "Psephurus_gladus"  
## [3] "Scaphirhynchus_albus"        "Scaphirhynchus_platorum"  
## [5] "Scaphirhynchus_suttkusi"     "Huso_huso"
```

Now we will use the `drop.tip()` function from `ape` to any tip that is in `not.in.dd`. `drop.tip()` needs a tree and a list of taxa to be dropped as arguments and returns a pruned tree. Use the help function to verify this.

```
pruned.tree <- drop.tip(tt, not.in.dd)
setdiff(pruned.tree$tip.label, pruned_data$species) #should be empty
## character(0)
plot(pruned.tree, type = "radial", cex = 0.5)
```



Introduction to control statements

Control statements order operations

- ▶ **for** each line in a text file
 - ▶ capitalize the first word
- ▶ **while** the number of simulations is less than 100:
 - ▶ perform new simulation
- ▶ **if** the sample is from Cuba, Hispaniola, or Jamaica:
 - ▶ assign sample to “island”
- ▶ **else**
 - ▶

assign sample as mainland

More help with data manipulation in R

We have only scratched the surface in exploring the power of R to wrangle data. For more practice and guided tutorials on R data fundamentals check out the Software Carpentry site [here](#). In addition there is a popular and powerful set of tools for working with data wrapped up in a set of packages called the tidyverse. This is my preferred way for working with data now. If you want to learn the tidyverse tools you can start at this [UCLA tidyverse course site](#) and [here](#) and [here](#). Please also let me know if you find sites that are especailly helpful and I will share them with the class.

Common control statements

for statements perform an action over a range ****for** (some range)
{do something}

```
for (ii in 1:5){  
  cat("\nthe number is ", ii)  
}
```

```
##
```

```
## the number is 1
```

```
## the number is 2
```

```
## the number is 3
```

```
## the number is 4
```

```
## the number is 5
```

more about **for**

You can also loop over all items in a vector

```
notfish <- c("bat", "dolphin", "toad", "soldier")
```

```
for(animal in notfish){  
  cat(animal, "fish\n", sep="")  
}
```

```
## batfish
```

```
## dolphinfish
```

```
## toadfish
```

```
## soldierfish
```

while loops keeps running until the conditional part of the expression fails. At this point, the loop is terminated.

```
thesis_idea_sucks <- True #initialize ideas to suck
```

```
while(thesis_idea_sucks){  
  current_idea <- get_New_Thesis_Idea();  
}
```

if

if statements allow your code to diverge depending on conditions

IF (condition is true) {do something}

```
if (xx == 'a') doSomething1;  
if (xx == 'b') doSomething2;  
if (xx == 'c') doSomething3;
```


else if

Use **else if** with **if** and **else** when you have multiple conditions

```
if (x == 'a'){
    doSomething1;
}
else if (x == 'b'){
    doSomething2;
}...
else if (x == 'z'){
    doSomething26;
}
else{
    cat('x != a letter\n')
}
```

Pseudocode

Pseudocode is an informal way to plan out the structure and flow of your program.

- ▶ don't worry about syntax of a particular language
- ▶ **do** think about variables and control structure
- ▶ Pseudocode can be translated across many languages easily

Pseudocode example

Write a script that prints a number and its square over a given range on integers and then sums them.

```
# set lower and upper range values  
# set squaresum to 0  
  
# loop over the range and for each value print  
    # currentvalue and the currentvalue^2  
    # add currentvalue^2 to squaresum  
# print "here is the sum of it all"m squaresum
```

Try this now!

one solution

```
lower = 1; upper = 5; squaresum = 0
```

```
for (ii in lower:upper){  
  cat(ii, ii^2, "\n")  
  squaresum <- squaresum + ii^2  
}
```

```
## 1 1
```

```
## 2 4
```

```
## 3 9
```

```
## 4 16
```

```
## 5 25
```

```
cat("the sum of it all is ", squaresum)
```

```
## the sum of it all is 55
```

functions

A function is a self-contained bit of code that performs a task. It might sum a set of numbers, run a simulation, or print your name backwards 500 times.

Functions are useful because

- ▶ they make code modular
- ▶ they make code reuseable
- ▶ they isolate code from unintended consequences (**scope**)

how functions work

Usually functions. . .

- ▶ take one or more arguments
- ▶ perform some operations
- ▶ return something

```
doubler <- function(num){  
  ## this function takes a number and doubles it  
  doubled = 2 * num  
  cat("witness the awesome power of the doubler\n")  
  cat("I changed ", num, " to ", doubled, "\n")  
  cat("you're welcome!\n")  
  return(doubled)  
}  
  
for (ii in 1:100){  
  doubled <- doubler(ii)  
  cat(doubled, "\n")  
}
```

```
## witness the awesome power of the doubler  
## I changed 1 to 2  
## you're welcome!  
## 2  
## witness the awesome power of the doubler  
## I changed 2 to 4
```

functions don't need to return anything

```
takeNoArguments <- function() {  
  cat('this function takes no arguments\n'); cat('it also\n')  
  cat('returns nothing\n');  
  cat('you never get something for nothing.\n')  
}  
takeNoArguments()
```

```
## this function takes no arguments  
## it also  
## returns nothing  
## you never get something for nothing.
```


creating functions

To define a function, you use the function keyword like this:

```
myFunction <- function(arg1, arg2)
```

This says that you want you create a function named 'myFunction' which takes two arguments, arg1 and arg2. Below this line, you enclose the statements belonging to the function in curly braces:

```
{  
  cat('this is my function');  
  cat('dont mess with it');  
}
```

using functions

Once you have defined your function, it is part of your workspace. Until you remove it, you can use it. Enter the following function:

```
greeter <- function(name) {  
  cat('Hello, ', name, '\n');  
}
```

greeter() takes the variable **name** as an argument and performs the greeting.

- ▶ what happens if you fail to supply argument name?
- ▶ what happens if you just type the name of the function without any parentheses?

Further function help

Of course there are a large number of web resources to help you learn various aspects of R programming. I like the Software Carpentry site here. Please send me other sites you find useful and I can share them with the class.