R basics and flow control

Michael Alfaro

9/14/2016

Getting started

Welcome to the EEB Quantitative skills bootcamp.

Git repo

create a new document in your bootcamp repo to track your inclass work. Commit to this as desired during the lectures

R basics

Use setwd() to set your directory.

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

And use getwd() to see the working directory.

```
getwd()
```

```
## [1] "/Users/michael_alfaro/Dropbox/git/eeb201/assets"
```

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", "laser"), 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 you session, use the ls() function.

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

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

removing variables

To remove a variable from the workspace, use rm(variable)

```
ls()
```

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

```
rm(xx)
ls()
```

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

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", "Pierce")
numbers <- runif(100)
ls()
```

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

```
rm(list = ls())
ls() #character(0) means the function has returned an empt
```

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

quitting R

You can quit R with q(). Caution, q() will quit your R session!

```
q()
q(save = 'no')
```

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/michael_alfaro/Dropbox/git/eeb201/assets"
```

```
source("/Users/michael_alfaro/Dropbox/bootcamp_examples/sou
#make sure the path to the source file is specified correc
all.I.know.about.life.I.learned.in.grad.school() #a functi
```

```
## Why would I want to go to med school when I can have all
```

Reading in files and manipulating data objects

For this section we are going to work with two kinds of data: a phylogenetic tree, and swimming data for some of the species in this tree. One of the most common tasks you will perform in R will be reading in data and this section should help orient you to ways you can interact with your data objects in the R environment.

Read in the tree

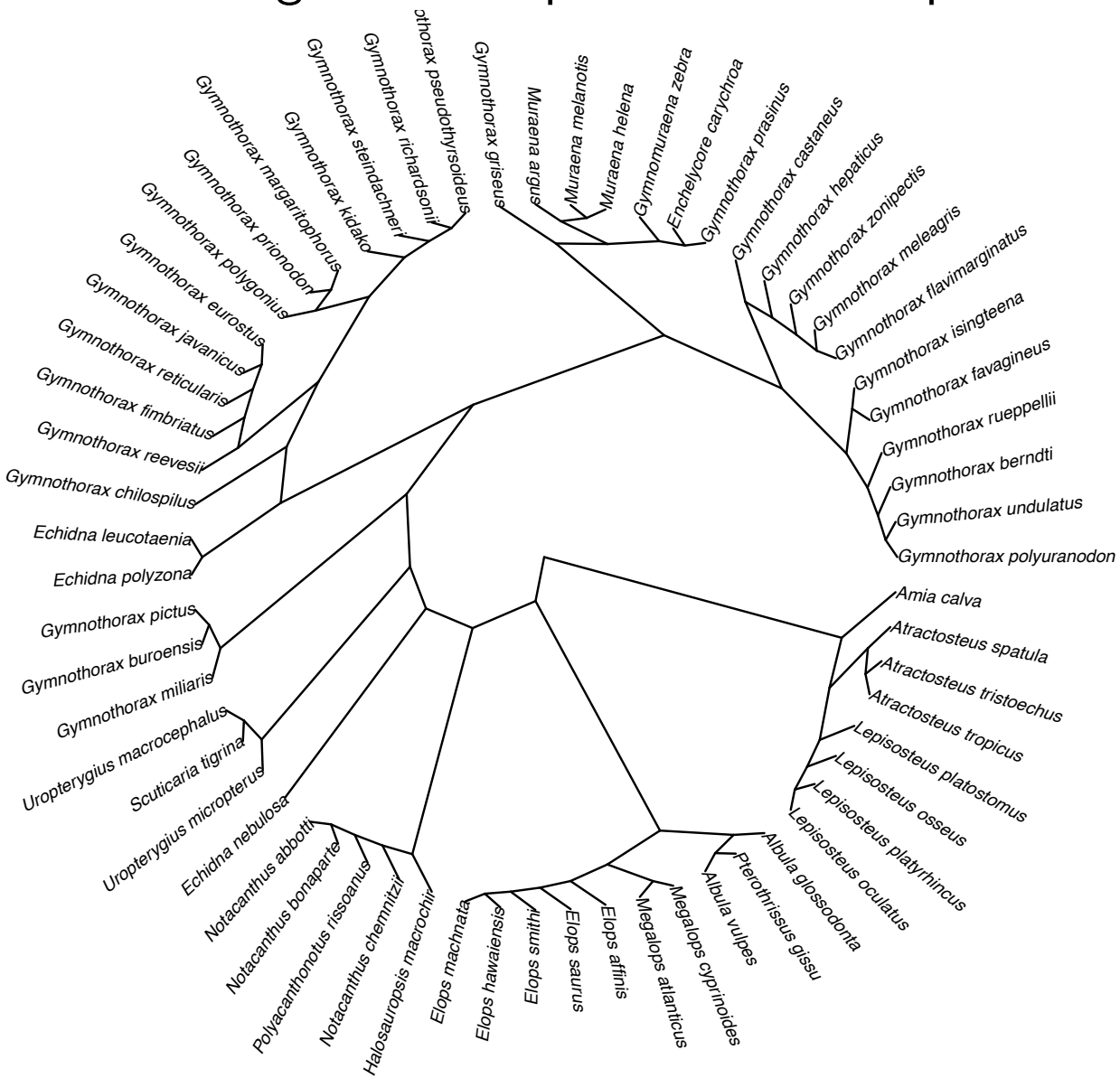
The first thing we will do is use read.tree() to in a phylogenetic tree. readtree() is in the Ape library, so make sure you have that installed. The tree file is a text file that contains informaiton about the tree structure and tip labels in Newick format. Use a text editor to look at this file if you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

```
library(ape)
tt <- read.tree("/Users/michael_alfaro/Dropbox/bootcamp_exa
###see elements of an object
attributes(tt)
```

```
## $names
## [1] "edge"        "Nnode"       "tip.label"   "edge.leng
## [6] "root.edge"
##
## $class
## [1] "phylo"
##
## $order
```

pruning the tree

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



reading in data

```
# d contains length data, family, species, order, etc
inpath = "/Users/michael_alfaro/Dropbox/bootcamp_examples/c
dd <- read.table(inpath, header=T, sep='\t', as.is = T);

###NOTE: R by default reads character columns as FACTORS. T
```

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 objec
#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
```

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_quinquelineatus  BCF
```

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

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 acc
```

```
## [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_qui
```

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

```
## [1] "Pomacentrus_coelestis"     "Pomacentrus_lepidogenys
## [3] "Pomacentrus_nagasakiensis" "Premnas_biaculeatus"
## [5] "Stegastes_apicalis"        "Canthigaster_valentini"
```

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] 0.674472
```

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

```
##                species mode     size
## 1 Naso_brevirostris  BCF 0.674472
```

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_qui
```

```
str(dd)
```

```
## 'data.frame':    92 obs. of  3 variables:
##  $ species: chr  "Naso_brevirostris" "glass_fish" "Zebra
##  $ mode   : chr  "BCF" NA "BCF" "BCF" ...
```

A bit more on subsetting

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

```
##                                                                  sp
## Cheilodipterus_macrodon                                Cheilodipterus_mac
## Cheilodipterus_quinquelineatus Cheilodipterus_quinquelin
## Chaetodon_aureofasciatus                              Chaetodon_aureofasc
## Chaetodon_auriga                                             Chaetodon_a
## Chaetodon_baronessa                                    Chaetodon_barc
## Chaetodon_citrinellus                                Chaetodon_citrin
##                                                 size
## Cheilodipterus_macrodon                      0.6492312
## Cheilodipterus_quinquelineatus               0.3652044
## Chaetodon_aureofasciatus                     0.7771912
## Chaetodon_auriga                             0.8930880
## Chaetodon_baronessa                          0.5301584
## Chaetodon_citrinellus                        0.8191973
```

which()

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

```
#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
## [24] 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
## [47] 80 81 82 83 84 85 86 87 88 89 90 91 92
```

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

```
##                                        species mode
## Acanthurus_blochii           Acanthurus_blochii  MPF 0.468
## Acanthurus_dussumieri     Acanthurus_dussumieri  MPF 0.983
## Acanthurus_lineatus         Acanthurus_lineatus  MPF 0.812
## Acanthurus_nigrofuscus   Acanthurus_nigrofuscus  MPF 0.557
```

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 backets **[]**

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 , afte
head(big.fish)
```

```
##                                          species mode
## Chaetodon_lunulatus       Chaetodon_lunulatus  BCF 0.947
## Chelmon_rostratus           Chelmon_rostratus  BCF 0.916
## Amblygobius_decussatus Amblygobius_decussatus  BCF 0.930
## Stegastes_nigricans       Stegastes_nigricans  BCF 0.914
## Cephalopholis_boenak     Cephalopholis_boenak  BCF 0.943
## Acanthurus_dussumieri   Acanthurus_dussumieri  MPF 0.983
```

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_brevir
## glass_fish                                                    glass
## Zebrasoma_scopas                                       Zebrasoma_s
## Apogon_nigrofasciatus                                Apogon_nigrofas
## Cheilodipterus_macrodon                            Cheilodipterus_mac
## Cheilodipterus_quinquelineatus Cheilodipterus_quinqueli
##                                          size
## Naso_brevirostris                   0.6744720
## glass_fish                          0.4146935
## Zebrasoma_scopas                    0.2670809
## Apogon_nigrofasciatus               0.1691091
```

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

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_qu
##  [3] "Chaetodon_lunulatus"            "Chaetodon_plebius
##  [5] "Chaetodon_rainfordii"           "Heniochus_singula
##  [7] "Amblygobius_decussatus"         "Scolopsis_bilinea
##  [9] "Acanthurus_lineatus"            "Sufflamen_chryso
## [11] "Cheilinus_chlorurus"            "Cirrhilabrus_punc
## [13] "Oxycheilinus_digrammus"         "Pseudocheilinus_h
## [15] "Thalassoma_janseni"             "Chrysiptera_brown
## [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')]<-'Chae
```

matching rest of data to tree

```
del_from_data <- setdiff(dd$species, tt$tip.label)
match(del_from_data, rownames(dd)) #row numbers of dd that
```

```
##  [1]  3  5 12 14 19 20 23 35 41 44 47 57 58 60 71 78
```

```
dd.pruned <- dd[-match(del_from_data, rownames(dd)),]
###ok now lets check for overlap
setdiff(dd.pruned$species, tt$tip.label) # this should pro
```

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

```
#this would also work
#pruned_data <- dd[!(dd$species %in% del_from_data),]
```

matching tree to data

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

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

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

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

```
## [1] "Polyodon_spathula"              "Psephurus_gladius"
## [3] "Scaphirhynchus_albus"           "Scaphirhynchus_plator
## [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.

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**
- code sample as mainland

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

while (SOME CONDITION is TRUE){ do something }
while loops keeps running until the conditional part of the expression fails. At this point, the loop is terminated.

```
while(thesis_idea_sucks){
    get_New_Thesis_Idea();
}
```

When a good idea is returned, the program breaks out of the loop.

while continued

You can use the break statement to set conditions for breaking out of the while loop too

```
xx <- 1
xx <- 1
while(xx < 5) {
  xx <- xx+1;
  if (xx == 3) {
    break; }
  }
print(xx)
```

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

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;
```

if and else

if only two conditions are possible and are mutually exclusive, you could use **if** and **else** to control your program.

```
for(ii in 1:6){
  if (ii %% 2) {
    cat(ii, " is odd\n")
    }
  else{
    cat(ii, " is even\n")
    }
  }
```

```
## 1  is odd
## 2  is even
## 3  is odd
## 4  is even
## 5  is odd
## 6  is even
```

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 partiular 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
```

```
# 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){
  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)
}
```

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?

Repo Update

PLEASE VISIT **`http://tinyurl.com/bootcamp-repos`** and post your repo
Thanks!