Bio Stats II : Lab 5

Acknowledgements: Punt & Branch Labs (U Washington)

Gavin Fay

02/17/2016

Lab schedule

1/27: Introduction to R and R Studio, working with data
2/03: Intro to plotting, manipulating data
2/10: Probability, linear modeling
2/17: Programming practices, conditional statements
2/24: Creating functions, debugging
3/02: Permutation analysis
3/09: Advanced plotting

Recommended reading

An introduction to R (Venables et al.)
– `http://cran.r-project.org/doc/manuals/R-intro.pdf`
- Today's material: Chapter 10

Advance R (Wickham)
- Chapter 6, 9

Some additional data manipulation

Merging data frames

Often we need to merge data frames based on some common information shared between them.
e.g. NEFSC Bottom trawl survey database contains species codes. Species names associated with those codes are contained in a separate table.

`match(vector1,vector2)` returns the index vector of values in vector2 that match the values in vector1.

```
> xvec <- c(1,1,3,2,1,4,3,2,4,2)
> yvec <- c(1,2,3,4)
> match(xvec,yvec)
 [1] 1 1 3 2 1 4 3 2 4 2
```

Some additional data manipulation

The `merge()` function makes use of `match` to merge dataframes based on common values in certain column(s).

```
> lookup <- data.frame(Code=yvec,species=c("wombat",
+                                  "devil","wallaby","quoll"))
> data <- data.frame(code=xvec,abundance=
+                          c(2,4,6,2,1,3,4,7,4,1))
> head(lookup,2)
  Code species
1    1  wombat
2    2   devil
> head(data,2)
  code abundance
1    1         2
2    1         4
> newdata <- merge(data, lookup, by.x = "code", by.y = "Code")
```

Locating data using grep()

grep() is one of the most useful functions for unix users.

It does pattern matching.

```
> pick <- grep("w",newdata$species)
> pick
[1] 1 2 3 7 8
```

returns the elements of newdata$species containing the letter "w".

gsub() is a convenient wrapper to grep() that can be used to perform string substitution.

e.g. capitalize all the 'a's in the species names.

```
> new_names <- gsub("a", newdata$species, replacement="A")
> new_names
 [1] "wombAt"  "wombAt"  "wombAt"  "devil"   "devil"   "devil"
 [8] "wAllAby" "quoll"   "quoll"
```

Functions

R allows the user to create objects of mode *function*.

Most of the functions supplied as part of R are themselves written in R and do not differ from user written functions.

A function is defined by an assignment of the form:

```
> fn_name <- function(arg_1, arg_2, ...) expression
```

The *expression* is usually a grouped expression, that uses the *arguments* to calculate a value.

The value of the expression is the value returned for the function.

A call to the function takes the form:

```
> fn_name(expr_1, expr2, ...)
```

Scripts vs Functions

R scripts containing lots of lines of code are fine for once-off analyses, but tedious if repeating the same analysis on many datasets.

Instead use functions to code commonly performed tasks.

Allows you to avoid global variables that can be accessed and changed anywhere.

To avoid duplicating code: if you have multiple copies of almost identical code, put it into a function.

You can create a function when R does not have a built-in function for your needs.

When running an analysis over and over again, can just call the function (one line) instead of running many lines of code.

"...learning to write useful functions is one of the main ways to make your use of R comfortable and productive."
(*Venables et al. R manual*)

A simple function: cv()

There is no R function for the CV, or coefficient of variation (mean divided by standard deviation)

```
> cv <- function(xvec) {
+     cv_value <- sd(xvec)/mean(xvec)
+     return(cv_value)
+ }
> values <- rnorm(n=1000, mean=5, sd=2)
> cv(values)
[1] 0.4204793
```

Available objects

Only the new function `cv()` and the object values are available in the workspace (they are global).

Object `cv_value` (defined inside the function) is encapsulated within the function `cv()` and not available outside it.

Well-designed functions only receive inputs through their arguments.

Optional arguments

If `xvec` includes NAs the function won't work, instead, add an optional argument.

```
> cv_new <- function(xvec, na.rm=F) {
+     cv_value <- sd(xvec, na.rm=na.rm)/
+                 mean(xvec, na.rm=na.rm)
+     return(cv_value)
+ }
> values <- rnorm(n=1000, mean=5, sd=2)
> values[100] <- NA
> cv_new(values)
[1] NA
> cv_new(values, na.rm=T)
[1] 0.4140504
```

Some built-in `R` functions (e.g. `lm()`) have many optional arguments.

Note that `qnorm(0.25,0,1)`, `qnorm(x = 0.25)`, and `qnorm(0.25,sd=1)` all return the same result.

Lab exercise 1/5

Create a function `wtlen()` with three arguments: a vector of lengths, and values `a` and `b`.
The function should return the weights of the fishes using the weight-length equation: $W = aL^b$
Use the function to calculate the weight (in g) of fish of length 100, 200, 300 cm for:

Species	a	b
Mola mola	0.0454	3.05
Regalecus glesne	0.0039	2.90

Properties of functions

Functions:
- Have a name
- Are provided with input ("passed" some input)
- Do something
- Return something for later use

Functions can call other functions.
A program is a collection of functions.
Functions can be re-used.
A good function is general, carefully written, and well documented.

Help with functions

Type `ls()` at the prompt in the console to see all of your current objects, including functions.

Type the name of a function to see how it was created, e.g. `lm`

To copy and then change the source code of a built-in function in `R`, use the edit function, e.g. `edit(lm)`

`R` contains numerous functions, the trick is finding the right one and being able to understand the documentation.

Outputs of functions

The command `return(my_object)` makes the object `my_object` available to whatever called the function.

If the function is called from the command line, `return(my_object)` prints out the object.

Replacing `return(my_object)` with `invisible(my_object)` returns the value without printing out the object.

What is the most common function that returns invisibly? (*Hint*: you use it on nearly every line of code)

Functions do not have to return anything.

You can pass and return all sorts of things: a single number, a vector of numbers, a character, a boolean value, and even another function (!)

Outputting many values from a function

Achieved by returning a list object.
(e.g. value returned by `lm()`)

```
> compute_stats <- function(xvec) {
+   ave <- mean(xvec)
+   var <- var(xvec)
+   cv <- cv_new(xvec)  # call to our own function
+   cat(ave, var, cv, "\n")
+   out <- list(ave = ave, var = var,
+               sd = sqrt(var), cv = cv)
+   return(out)
+ }
> values <- rnorm(n=1000, mean=5, sd=2)
> values_summary <- compute_stats(values)
5.007874 3.874996 0.3930811
```

Lab exercise 2/5

Write a function `cor_vec()` which takes two vectors `X` and `Y`.
Normalize both (subtract their respective means) so they average 0;
plot `X` versus `Y`; return the correlation between `X` and `Y`.
Apply your function to the following data set:

```
> xx <- seq(from=1, to=100, by=1)
> yy <- 0.2 + xx*0.5 + rnorm(n=length(xx),mean=0,sd=5)
```

Add `, ...` to your parameters and to the end of the `plot()` command.
The `...` stands for any parameter.
Call the function with parameters `cex=3` or `col="blue"`.

Scope: where is an object visible?

```
> rm(list=ls())
> x1 <- 10
> func1 <- function(x,y) {
+     print(x)
+     print(y)
+     print(x1)
+ }
> func1(x=5, y=6)
[1] 5
[1] 6
[1] 10
> print(x)
Error in print(x): object 'x' not found
> print(y)
Error in print(y): object 'y' not found
```

First rule of good programming

AVOID GLOBAL VARIABLES [whenever possible]

Each function should be self-contained.

All inputs passed as parameters.

All outputs returned using `return()` or `invisible()`.

Nothing to see here

```
> rm(list=ls()) #clear everything
> x1 <- 10
> func1 <- function(x) {
+     x1 <<- x
+     return(x1)
+ }
> func1(x=5)
[1] 5
> print(x1)
[1] 5
```

The <<- means change the value of the global variable.
NEVER use this.

Scoping rule: if there is a local variable and a global variable with the same name, R will use the local variable.

Lab exercise 3/5

Write a function which takes a file name and two numbers (the defaults for the two numbers should be 1 and 2).
Reads in a table of data (assume that the file is comma delimited)
Plots the columns represented by the two numbers against each other.

Hints:
Use the read.csv() or read.table() function.
Use print() to check the values of intermediate results (to see if your function is working).
Use the Laengelmavesi data from lab 2 to check your program.

Dynamic lookup

R looks for values when the function is run, not when it's created.
The output of a function can be different depending on objects outside its environment:

```
> f <- function() x
> x <- 15
> f()
[1] 15
> x <- 20
> f()
[1] 20
```

Generally want to avoid this behavior - function is not self-contained.
Detect using `findGlobals()` function from package `codetools`.

```
> f <- function() x + 1
> codetools::findGlobals(f)
[1] "+" "x"
```

All of the standard operators in R are in fact functions.
You can therefore override them with your own alternatives.
- may or may not be a good April Fool's joke if your colleague leaves their computer unlocked.

Arguments

Arguments can be specified by position, by complete name, or by partial name.

Avoid using positional matching for less commonly used arguments.

Calling a function given a list of arguments using `do.call()`

```
> args <- list(1:10, na.rm = TRUE)
> do.call(mean, list(1:10, na.rm = TRUE))
[1] 5.5
```

Determine if an argument was supplied or not using `missing()`.

```
> i <- function(a, b) {
+   c(missing(a), missing(b))
+ }
> i()
[1] TRUE TRUE
> i(a = 1)
[1] FALSE  TRUE
> i(b = 2)
[1]  TRUE FALSE
```

Discover undocumented features by reading the source code for `plot()`

infix functions

Most functions in R are *prefix* functions, the function name comes before the arguments.

Can also have *infix* arguments where function comes between the arguments, like + or −.

All user created infix functions must start and end with %

e.g. new function to paste together two strings

```
> `%+%` <- function(a, b) paste(a, b, sep = "")
> "new" %+% "string"
[1] "newstring"
```

NB: when creating, function name has to be put in backticks.

Debugging and defensive programming

What happens when something goes wrong with your `R` code?

What do you do? What tools do you have to address the problem?

Debugging is the art and science of fixing unexpected problems in your code.

Useful functions include `traceback()` and `browser()`. Also interactive tools in RStudio.

Not all problems are unexpected.
- when writing functions, you can often anticipate potential problems.

"Finding your bug is a process of confirming the many things that you believe are true - until you find one which is not true."
(Norm Matloff)

Add statements to your function that stop the calculations and return messages when function assumptions are not met.

You can debug with just `print()`, there are times when additional help is welcome.

Debugging procedure

Realise that you have a bug.
- (the first step is admitting you have a problem. . . .)

Make it repeatable.
- once found, need to be able to reproduce it on command.
- slowly whittle down to the smallest snippet that still causes the error.
- binary search is useful for this. (repeatedly remove half the code).
- if generating the bug takes time, figure out how to get it faster (create minimal example).
- during this you'll discover similar inputs that don't trigger the bug.

WRITE THESE DOWN.

Figure out where the bug is.
- Use helpful tools to try and identify the offending line of code.
- Generate hypotheses, design experiments to test them, record results.
- A systematic approach will often end up saving you time.

Fix it and test it.
- Figure out how to fix it and to check that the fix worked.
- Carefully record the correct output, and check against the inputs that previously failed.

Lab exercise 4/5, Tasmanian Devils

If there are 5,000 Tasmanian devils in year 2015 and they are declining at 9% per year, in what year will they fall below 500 individuals?

```
> devil.loop <- function(N, year=2015) {
+   while (N>500) {
+     N <- N*0.91
+     year <- year+1
+   }
+   return(year)
+ }
> devil.loop(N=5000)
[1] 2040
```

Add code to this function to allow for demographic stochasticity in the survival rate. (assume $N_{t+1} \sim \text{Binomial}(n = N_t, p = 0.91)$)

Record and plot the distribution of the year the population drops below 500 individuals obtained from 1000 simulations (function calls).

Lab exercise 5/5

Write a function (or set of functions) that takes two vectors of data, produces a scatterplot of one vs the other, adds a title that is a species name, performs a regression if enough data, and returns the coefficients of the regression.

The functions should also add the estimated w-l curve to the plot.

Add an argument that determines whether the regression is based on the logs of the input vectors.

Hint Test your functions with some dummy data. Reuse code from earlier today and from other labs.

Apply your function(s) to the Laengelmavesi data.