R Bootcamp: Getting Started in R for Biologists

Marguerite A. Butler

Department of Zoology, University of Hawaii, Honolulu, HI 96822

mbutler@hawaii.edu

June 5, 2018

Contents

Chapter 1

Preliminaries

1.1 Computer Requirements and Installing R

This chapter is about the software we will be using in class. *If you've installed these software a long time ago, please update to recent versions to avoid compatibility issues.*

Computers I will be using a macintosh running El Capitan (OS 10.11.6), however, R is open source and available on PC and Linux as well. For the most part, the R commands are cross-platform compatible. The only exceptions are those that deal directly with other programs on your computer (the main one being to bring up a new graphics window – quartz() on a mac, and x11() on a PC or Linux).

R version 3.4.0 (Amusingly nicknamed "You Stupid Darkness". The later versions in a series usually have bug fixes). You can install R from the binaries available at the R website http://www.r-project.org. They are available as disk images and very straightforward to use. On the left Menu bar, click on "CRAN" (the Comprehensive R Archive Network). Choose a mirror (the closest geographically), then click on your operating system (MacOS X) and click on R-3.4.0.pkg. Follow the directions from there.

1.1.1 Installing from source

If you would like to be able to install packages from source, you will need these components: C compiler (gcc), a fortran compiler (e.g., gfortran), and X11. If you don't know what this is about, it's OK – just skip it. If you do want to do it, take a look at the instructions on: http://cran.r-project.org/bin/macosx/tools

Xcode Tools This contains the C/C++ compiler. Install from the system disks that

came with your computer. If you don't have the disks, you can also download it from the Apple Developers site after signing up for a free account.

gfortran Install from the link above (tools directory on the CRAN install page).

X11 Comes with OS X, but it may be an optional install.

You can find detailed instructions on how to install these software components and links to the software itself at the R website , under FAQ's > R for Mac OS X FAQ > Building R from sources.

Note: for people who've recently upgraded their systems, please make sure you have Xcode Tools and X11 installed from the discs that came with your computer (they have to be the correct version for your new OS. For example, you can't use your Xcode Tools from Tiger on your mac running Snow Leopard).

1.2 R packages

Many of the packages that you will ever use are available on CRAN. The easiest way to install from CRAN is to do it from within R. From the "Packages & Data" menu option, choose "Package Installer". You may have to choose a mirror if you haven't done so already (choose a geographically close one). The package installer should open up with "CRAN (binaries)" already selected. Click on "Get list", which will refresh the menu with all the available packages and the version numbers that you have installed. Highlight the packages that you want to install, choose "Install Dependencies" then click on "Install Selected". You can also download the packages from the R website, on the left menu bar click on CRAN.

Install the following from CRAN (binaries):

ggplot2

pspline

ks

1.3 General R References

An introduction to R A comprehensive and easy-to-follow tutorial produced by the R Development Core Team.

R for Beginners A tutorial by Emmanuel Paradis.

1.4 Help! and Useful References

1.4.1 general R help

Jonathan Baron's R help page Bookmark this page! It is the best search engine to find R help. It searches the huge archives of the R-help listserv as well as all R documentation pages. For more technical help, you can also include the R-dev (developers) listserv in the search.

1 page R reference card by Jonathan Baron.

4 page R reference card by Tom Short.

1.5 For Folks who get serious about R programming

Programming with Data: A guide to the S Language by John M. Chambers. 2004. Springer. This is written by one of the authors of the S language, which R is based on. It has a lot of details that you will never find in the glossy books.

Chapter 2

R Environment

2.1 A few words about R

R is an open-source (and free) programming environment for statistical computing, modeling, and graphics. With facilities for basic database programming and good string search and replacement facilities, it is excellent for reshaping data. It is scalable, and extendible, so that you can tackle large or complex analyses with confidence. It has quickly become the platform of choice for many biologists with many contributors world-wide writing their code as "packages" that can be shared with the rest of the R community. It is somewhat unusual in being both a powerful programming language as well as an interactive analysis environment.

The fact that it is a bonafide programming language with the ability to define classes, to write functions and control structures for program loops, in combination with the matrix math facilities make it very useful for customizing inferential models and simulations. It also produces beautiful graphics that can also be customized and reproduced precisely.

How you approach learning R will depend on your background. If you are already proficient in a scientific programming language such as Python, C++, Java, Fortran, Matlab, etc. it will not be very difficult for you to learn R. It is a matter of getting familiar with the particular syntax of R, and adjusting coding conventions to take advantage of some special features in R such as vectorized calculations and its semi-object oriented design (this last point is pretty deep - methods definitions do not reside inside of class definitions bur rather are associated with generic functions and are thus flexibly inherited). A great place to start is the Introduction to R tutorial written by the R core team. This tutorial you?re reading here will also be useful to you, but you can skip to the brief section on the R environment, and read the sections on R objects, writing functions, and indexing.

2.1.1 If you are new to R

If you are very new to R and programming (coding) in general, you may feel overwhelmed because you must come up with your own code which often results in errors. Don?t worry, it will get better! Start by learning how to do a small number of useful tasks and build upon them. *Remember errors and overcoming them is the sign that you are learning.* This tutorial was written for you to master R. Just like learning any foreign language, it helps to understand it?s grammar (syntax) and terminology. You may be here because you have been told that you should learn R, but you may not fully realize its power and capabilities for data exploration, analysis, simulation, and graphics. Mastering these skills will also help you to improve your logic, to think with greater precision and clarity, to more fully explore your ideas, and to be able to write more meaningful papers based in evidence.

Learning to code well involves a lot of practice. Practicing once a week, only repeating the examples in a tutorial or searching Google for every little issue will not help. Just like those who excel at learning foreign languages, the ones who go boldly forth and try out their new words and phrases every chance they get and in every new situation will attain mastery. The same is true of computer languages. Giving your brain the opportunity to struggle for the right commands, to become familiar with putting together lines of code with the proper syntax, and to take guesses at new code to see if it will work are all strategies that will help you to not only remember but to internalize the rules and structure of the language. This is why I personally think it is better to do some of the typing yourself in the beginning rather than cutting and pasting to challenge yourself to come up with the correct code.

2.1.2 Reproducible results

Scientific results must be reproducible. Good coding practices and organization will promote producibility. With the advent of "point and click" statistical software, many powerful tools are available to a wide variety of users. However, users typically lose track of what they did to the dataset, what precise steps they implemented, and in what order. It can be very stressful to try to retrace those steps when your results don‘t match an earlier version!

Writing your analysis in the form of a clean script (a document with all of the code used) will naturally create an archive that will precisely reproduce all of your results and graphics, at any time in the future. A coding language with input and output files, files for code, and organized use of directory structures can ensure reproducibility. With every data analysis project create a directory with:

Data files These are input files, and should be archived and never changed. If there are a lot of files, you may want to include a subfolder or subdirectory for data.

Script files These are the files that contain your code. Any code that manipulates the data should be included and documented here. A primary step is to get your raw data into the format needed for your analysis. Then your code for your analysis, and finally creating the output files and any figures.

Output files These generally include any tables of output data and figures. Often you will have a directory for figures if there are a number of them. Sometimes you may want to save a "clean data file" as a ready-to-go cleaned up version that you may use in other R scripts.

2.1.3 Customized analyses

Parametric statistical procedures are wonderful when your data meet their assumptions, but often we want to go beyond what is available. R encourages users to customize their models to suit their data, not the other way around. In addition, there are many times in which we want to employ simulations to explore the error structures we assume, or the power if our analyses to find significant associations. R provides powerful facilities for randomization tests and simulations.

2.1.4 Improving your analyses

It's a funny thing, no matter how well you understand your work before you write about it, it seems that going through and actually writing that manuscript really helps to reveal what is truly significant about your work. Similarly, when you begin to code, you will find associations or structure in your data that you didn't appreciate before. Forcing yourself to be more explicit and rigorous can help you to see what is really and truly robust and important in your study system as explanatory factors. It also helps you to see what is not important. It helps you to grow intellectually.

2.2 The R Environment

2.2.1 Text Editors and GUI interfaces

At its heart, R works via a command line interface. Data objects are accessed by commands, functions are groups of commands put together to perform a specific task, and scripts are collections of commands. How you interact with R strongly influence your experience.

The R GUI (graphical user interface) for Mac is much more feature rich than that for the PC. The R GUI for Mac has syntax highlighting which means that R commands, comments, and strings are colored differently than other text, and will automatically

indent when writing functions and loops. It also allows code to be run from the script to the console via a hot key (command-R), and working from the console, once you begin to type a function it shows you the form of the function call with the arguments below the command line. Another helpful feature is that starting R by double clicking on a script file (a file named with a .R ending) will automatically start the R console will set the working directory to the one where the script resides, and will autosave your history as well as other nice features.

For PC, a plain text editor that has syntax highlighting and automatic indenting when writing functions and loops is TextPad which can be obtained at https://www.textpad.com.

Rstudio provides a graphical front-end can be a nice help, in that it will auto-complete functions and object. However, for beginners it can sometimes be too helpful and take away some of the learning opportunities. Just as relying too heavily on spellcheck hurts the learning of spelling, many new users using Rstudio are unaware of where their files are and find that too much autocompletion (of functions, objects, etc) can take away opportunities to solidify their recall. If this is you, it might be better to hold off on R studio until you?ve given yourself the opportunity to gain some proficiency. Struggle is a good thing – that?s when you are learning. In any case, if you are on a Mac, the R GUI is perfectly fine. Unfortunately the PC interface is really very bare, and so you will probably want Rstudio right away. Any way you approach it, learning R will take some effort, but it will be worth it.

2.2.2 R Works in RAM

Your entire R session - the workspace and all computations are held in RAM. On the one hand this makes R very fast, however if your computer crashes you lose all of your work. It is important that you save your working lines of code in a script. Make a habit of saving each line of code as you go in a script file that you name meaningfully and save with a ".R" extension. Do make sure that if you are on a Mac or a PC that your file viewing preferences are set to show all extensions.

Older 32-bit machines limited R workspaces to 2GB or less. However, most computers nowadays are 64-bit machines, which will only be limited by the size of the RAM chips you purchased. Therefore, most of us will not notice any size limitation in our R sessions, unless you are doing a very large analysis (such as spatial or phylogenetic analyses which have huge data objects).

2.2.3 R workspace

When you start up R, running the program will occupy a portion of your RAM called the workspace. (Note that the RAM will also be shared with any other applications

you have running - close them down if they are memory hogs.) The workspace contains whatever you load into it: any data, any functions or packages, any code you enter or load, any objects or variables that you create, and contains your processed computations. The R workspace is the virtual "space" occupied in RAM by running the R program, and everything in it will no longer exist at the end of your R session when you shut down R. The R workspace refers to the virtual space occupied by your activities in R (the objects), whereas the R session refers to timeframe during your activities in R, or one bout of using the R environment.

2.2.4 R session and R working directory

Your R session begins when you start up R and ends when you shut it down. Upon start, R will treat your current hard drive location as your R working directory, the place where R is "parked". R will look for any "`.Rdata`" file in this working directory and load the data from that file into your workspace. The working directory will be the default location where R will look for external data. So if you ask R to load data from an external file, it will look in the working directory unless you give it the full path to your file.

By default, the working directory on a Mac and Unix/Linux systems is your home user directory " ". On a PC the default working directory is where your R program is installed "`C:\Program Files\R\R` *`version`*`\bin`". This is not where you want all of your work to be stored, so please establish a working directory on your hard drive where you will save all of your personal work: your scripts, your data, and your output. Make a directory for this class. I would recommend naming it "`Rclass`" and putting it at some accessible location in your user directory (like at the top level of your User directory on a Mac, or at `C:\` on a windows machine).

You can start R within your desired working directory by putting an R script in there and starting R by double clicking the script file there. Or you can navigate to it from within R by using the `setwd(`*`path to working directory`*`)` function, for example "`setwd("~\Rclass")`".

2.2.5 Two special files

At the end of your R session, R will ask you if you want to save your workspace. If you say yes to the question "Save workspace image", it will save all of the objects in your workspace into the hidden file "`.Rdata`". If you are like me and do a lot of trial and error before settling on the right way to do an analysis, this is generally not a good idea. I usually say no. If I do want to save some data objects, I save the specific ones I want manually and giving them names I can remember using the `save()` function.

Automatically R will save a "`.history`" file in your working directory, which contains a history of every command that you typed during your R session, and possibly your

previous sessions. Looking at this after a session can be useful for starting a script file. You can also create a script file on your own.

Note that both the `.Rdata` and the `.history` files are hidden files meaning that you can't see them ordinarily in your folders because they start with a "." However, they are still there and R will look at them on startup.

2.2.6 R program directory

When you download and install R software from CRAN, it will be saved on your hard drive. You should never touch these files, but only add to the packages using the R `install.packages()` function or the Package Installer menu item from within the R application. In case you're curious, on a Mac the R software is stored in a directory outside of all user directories, from the computer's root directory: "`/Library/Frameworks/R.framework/Versions/C` The R application is here under "`R`", and the packages that you've installed are stored in the "`Resources/library`" subdirectory.

On a PC, R is stored at "`C:\Program Files\R\R` *`version`*`\bin`". For reasons that I'll never understand when you start R on a PC, the default save location for all user files is right in the middle of this R program file directory. Never use this location to save your personal work! It is a recipe for corrupting your R installation. Make a separate working directory for yourself somewhere else.

Chapter 3

Objects

Everything in R is an object. Data is stored in objects, functions are objects. Gaining a solid understanding of R objects is the most important step in mastering R. Doing so will expand your possibilities in coding and prepare you to be able to understand other people‘s code, even when you have never seen that strategy before. Understanding “object anatomy” will help you internalize the rules governing how objects are manipulated, reshaped or changed from one form to another, understand and fix errors, and make indexing much more intuitive (or selecting the parts of the dataset that you are interested in).

3.1 Objects and Classes

R is a semi-object oriented language. It differs from truly object-oriented languages such as C++ or Java, where objects contain the methods that operate on them. Methods are common tasks that we would want to operate on objects, such as plotting (this would be called the “`plot()` method for such-and-such object”. Therefore the methods (functions) are local to the class of objects, and don?t generalize to new classes that are created by users. In R, methods are associated with generic functions. The consequence is that R is a much better environment for interactive use - users can define new classes that will inherit the methods of classes that are already existing. This is what makes R a very powerful yet flexible and easily extendible programming language. Users can build upon the work of others (functions and packages) without having to create all of the lower-level infrastructure themselves.

Here‘s why classes are so important. When I first started using R, I was amazed and bewildered by the fact that the function `plot()` worked on a vector, a data frame, a linear model object, or a tree object, or whatever class of object you tried. How did R know? Then I learned about generic functions and methods. To display all of the available methods for the generic `plot()` function that are currently loaded in your workspace:

```
> methods(plot)

 [1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
 [4] plot.default        plot.dendrogram*    plot.density*
 [7] plot.ecdf           plot.factor*        plot.formula*
[10] plot.function       plot.hclust*        plot.histogram*
[13] plot.HoltWinters*   plot.isoreg*        plot.lm*
[16] plot.medpolish*     plot.mlm*           plot.ppr*
[19] plot.prcomp*        plot.princomp*      plot.profile.nls*
[22] plot.raster*        plot.spec*          plot.stepfun
[25] plot.stl*           plot.table*         plot.ts
[28] plot.tskernel*      plot.TukeyHSD*
see '?methods' for accessing help and source code
```

So you can see that there is a `plot.default` which will be used for two vectors passed to plot, but also a `plot.factor`, `plot.data.frame`, `plot.lm`, `plot.function`, among many others. R developers, when they contribute new object classes, have to write generic functions for those classes so that users don't have to worry about it. If you want to plot, you just use the generic `plot()` function on the object you want to plot, and the appropriate plot function will be deployed according to the class of object. *This is why the class of the object is so important.* When you start writing your own functions, a common source of error is using the wrong class of object. For example, if a user writes a function to expect a vector, but you try to give it a dataframe. (Generally speaking, most generic functions have been developed to handle many types of objects.)

Another beauty of the structure of R is *inheritance*. For example, if you were to write a class of objects for phylogenetic trees that it itself made up of matrices, it will inherit all of the methods for matrices as well. This property makes it very easy to write extensions to existing code.

3.2 The Classes

R stores information of three fundamental types which form the building blocks for everything else: **numeric** (e.g., 1, 2, 65.321), **character** (e.g., "tom", "mary", "apple", "x11", "v_26"), and **logical** (e.g., TRUE, FALSE, FALSE, TRUE), also **complex**, but we won't worry about that one). Information may also be stored as a **factor**, which is used to indicate categorial data such as (e.g., "large", "medium", "small", "small", "small"), which looks superficially like a character but is stored internally as a numerical integer (more on this later).

Information is stored and kept organized in objects. The most important built-in object types or classes are given in table 3.1. All objects have several attributes: class, mode, and length.

Class	Description
vector	A one-dimensional array of arbitrary length, each value is the same type (i.e., numeric, character, logical, factor)
matrix	A two-dimensional array with an arbitrary number of rows and columns, all of same type.
data frame	Resembles a matrix, however columns can contain different types of data. Columns typically correspond to variables in a statistical study, while rows correspond to observations of these variables.
list	An arbitrary collection of other R objects (which may include other lists).
function	a set of commands that are packaged into a unit with optional input and output.

Table 3.1: A description of the most used built-in R object classes.

3.2.1 Attributes

The common attributes of objects (the most helpful to know are **class, mode, length, dimension,** and **names**:

class Describes the type of object, it may be a built-in class or written by a package author or user. Classes are used for methods dispatch.

mode Also describes the type of object. Sometimes called "storage mode". Can be the same as the class, if it is a plain built-in object.

length The number of fundamental elements of the object.

dimension Rectangular objects such as matrices and data frames have dimension, indicating the numbers of rows and columns. 3D and higher dimensional arrays have 3 or more dimensions.

names Names are optional but often useful to apply labels to internal elements of objects. If no names are given, the internal values can be referenced by their position (index).

3.2.2 Vectors

The simplest R object is a vector. It is the building block of all other objects. It is a chain of values that are stored together. It has length, which describes how many values are contained in the vector. All must be of the same type, so that R treats the entire vector assuming that each value is numeric, character, factor, etc. If you add names to a vector it will apply to each element (value) of the vector.

Various ways to create vectors:

```
> x <- c( 1, 5, 7)  # numeric
> x

[1] 1 5 7

> rep( x, times=2)

[1] 1 5 7 1 5 7

> y <- rnorm(8)
> y

[1]  1.21264406  0.84095957  0.08781878  2.40779791  1.16261456  0.22951318
[7]  0.14967345 -1.80939979

> goodanswer <- c(TRUE, TRUE, FALSE)  # logical
> goodanswer

[1]  TRUE  TRUE FALSE

> species <- letters[1:4]    # special stored data object: lower case letters a - d
> species

[1] "a" "b" "c" "d"

> LETTERS[1:3]   # A B C  # character

[1] "A" "B" "C"

> treatment <- c("low", "med", "high")
> treat <- factor(treatment)   # create a factor
> treat

[1] low  med  high
Levels: high low med
```

Notice that your work is only saved if you STORE the result in an obect

It is simple to do arithmetic on vectors:

```
> x

[1] 1 5 7

> x+2

[1] 3 7 9

> x^2

[1]  1 25 49
```

Typically the class of a vector is returned as the type of information contained, so that return values of ”numeric”, ”character”, ”factor”, or ”logical” indicate that the obect is a vector:

```
> class(x)

[1] "numeric"

> mode(x)

[1] "numeric"
```

3.2.3 An aside about Factors

Factors are categorical data, for example, “large” and “small”, or “blue”, “red”’, and “yellow”. Factors may be ordered, which means that the order of the categories has meaning (like size categories). By default, factors are unordered. Levels are the values (i.e., names of the categories) that the factor can take.

You can easily coerce one type to another:

```
> as.numeric(treat)   # coerce to numeric

[1] 2 3 1

> as.character(y)  # coerce to character
```

```
[1] "1.21264406261767"   "0.840959566260821"  "0.0878187841570245"
[4] "2.40779790637393"   "1.16261455684436"   "0.229513184474261"
[7] "0.149673447334957"  "-1.80939978863398"
```

Note that when values are quoted, they are of type character.

However, notice that factors return different values for class and mode:

```
> class(treat)
```

```
[1] "factor"
```

```
> mode(treat)
```

```
[1] "numeric"
```

And sometimes, when doing ordinary manipulations such as coercing to numeric or combining vectors together to create a matrix, we get:

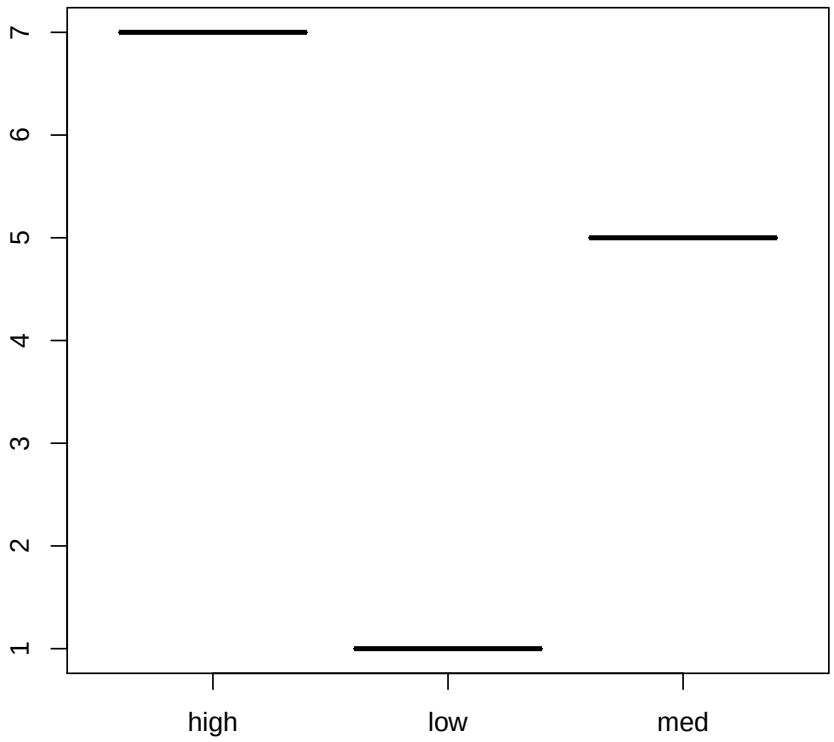
```
> as.numeric(treat)
```

```
[1] 2 3 1
```

```
> cbind(x, treat)
```

```
     x treat
[1,] 1     2
[2,] 5     3
[3,] 7     1
```

```
> plot(treat, x)
```

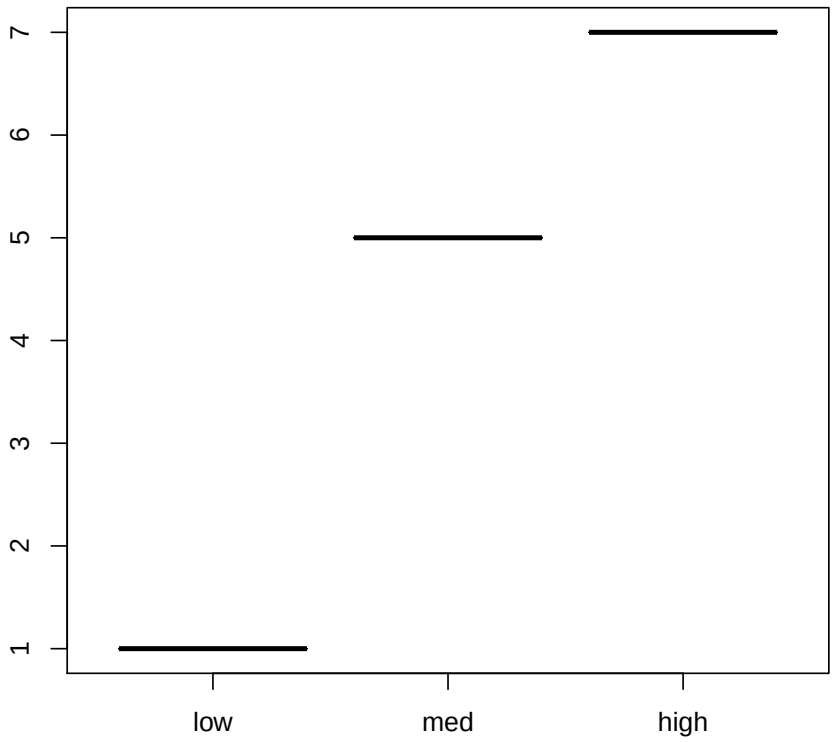


This is because internally, factors are stored as digits, 1, 2, etc. to the number of categories. The characters that you see are actually labels for the factor categories called "levels". Look at the help page for factor (`?factor`) and read the entry for levels. If nothing is specified for the order of the levels, then they are numbered alphabetically! This explains the weird behavior. So if you want the order of the levels to be a more logical "low, medium, high", you can do so by specifying the order of the levels explicitly when you create the factor:

```
> treat <- factor(treat, levels=c("low", "med", "high"))
> treat
```

```
[1] low  med  high
Levels: low med high
```

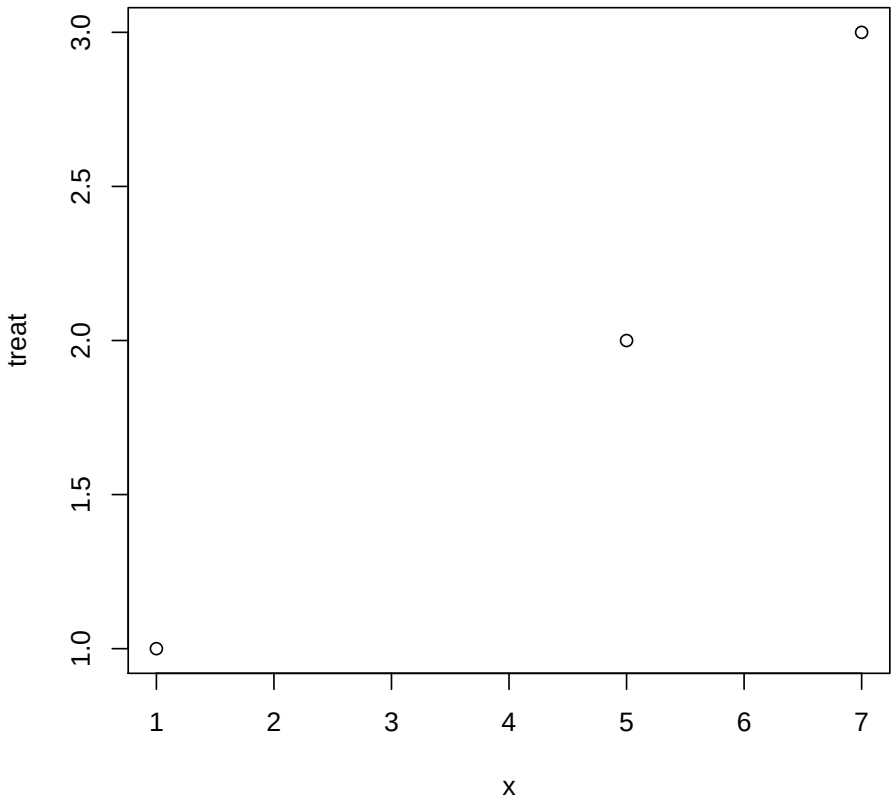
```
> plot(treat,x)
```



If you get frustrated because you're trying to treat a factor as a character, it's sometimes safer to manipulate factors by coercing to character first before doing anything else.

Here's another illustration of methods dispatch - The plot function by default accepts arguments in x, y order. If x is a factor as above, the plot assumes a categorial format for the x-axis. However, if you switch the order:

```
> plot(x,treat)
```



It is treated as a continuous x-axis. This is because methods dispatch for the plot function will be guided by the class of the objects that you supply. With x as a factor, R assumes you want a discrete x-axis, with the x as a continuous variable and y as the factor level, it will plot on the internal codes for the factor levels (if you want a horizontal bar plot use `barplot(..., horiz=TRUE)`. If you want histograms use `hist()`.

If you name a vector, the names are attached to each value:

```
> names(x) <- treatment
> x

 low  med high
   1    5    7

> names(x)

[1] "low"  "med"  "high"
```

3.2.4 Matrices

Data are often rectangular. For example, you might have height and weight columns for each subject (rows), or you might have a matrix describing the genetic relatedness between each pair of individuals in a family. Whether it is a square or a rectangular matrix, you will have a series of vectors of equal length. In fact, in R you can think of it as a single long vector with breaks at the end of each column.

Creating a matrix:

```
> xy <- cbind(x,y)  # column bind
> xy
```

```
     x           y
[1,] 1  1.21264406
[2,] 5  0.84095957
[3,] 7  0.08781878
[4,] 1  2.40779791
[5,] 5  1.16261456
[6,] 7  0.22951318
[7,] 1  0.14967345
[8,] 5 -1.80939979
```

```
> z <- matrix(1:6, nrow=2)  #create a matrix with 2 rows
> z
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
> matrix(1:6, nrow=2, byrow=2)
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```
> class(z)
```

```
[1] "matrix"
```

```
> mode(z)
```

```
[1] "numeric"
```

Names applied to a matrix will attach the names to each value. If you want to name the rows and columns of a matrix you must use `rownames()` or `colnames()`

```
> names(z) <- letters[1:6]
> z

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
attr(,"names")
[1] "a" "b" "c" "d" "e" "f"

> z["c"]

c
3

> z["f"]

f
6

> colnames(z) <- c("tom", "dick", "harry")
> z

     tom dick harry
[1,]   1    3     5
[2,]   2    4     6
attr(,"names")
[1] "a" "b" "c" "d" "e" "f"
```

Matrices have rows and columns, and are all of the same type whether it is numeric or character. There is a full suite of matrix math facilities in R to operate on numeric matrices.

```
> z+2

     tom dick harry
[1,]   3    5     7
[2,]   4    6     8
```

```
> z*2  # ordinary element-by-element multiplication
```

```
     tom dick harry
[1,]   2    6    10
[2,]   4    8    12
```

```
> x <- c(1,2,4)
> z%*%x  # matrix multiplication
```

```
     [,1]
[1,]   27
[2,]   34
```

3.2.5 Data Frames

Dataframes are superficially similar to matrices. They are rectangular. However, a major difference is that the vectors that comprise a dataframe can have vectors of different type. For example, one column can contain species names, whereas the other columns might contain numerical data.

```
> species = letters[1:3]
> y = rnorm(3)
> dat <- data.frame(species, x, y)
```

Internally, it is no longer a single vector but actually a list of vectors:

```
> class(dat)
```

```
[1] "data.frame"
```

```
> mode(dat)
```

```
[1] "list"
```

```
> names(dat)
```

```
[1] "species" "x"       "y"
```

```
> length(dat)
```

```
[1] 3
```

```
> dim(dat)
```

```
[1] 3 3
```

This is why the length of a dataframe is the number of columns. The names are the names of the columns, etc.

You can access the columns of dataframes using the $ operator, or using brackets, or by index number: and the name of the element, by the index number and double brackets, or by name and double bracket. Or a single value by using indexing on the row and column number separated by a comma. More on this later in the indexing chapter.

```
> dat$species
```

```
[1] a b c
Levels: a b c
```

```
> dat[1]
```

```
  species
1       a
2       b
3       c
```

```
> dat[1,3]
```

```
[1] -0.1890454
```

3.2.6 Lists

Lists are vectors of arbitrary objects. You can string any kind of objects together in a list. Sometimes the objects are identical, such as when you are doing a massive simulation study and each list element may be a simulated dataset in form of a dataframe. Alternatively, each element of a list may contain a different type of object. This is especially useful for model fitting output, for example, where you may want to store the input data in one list element, and in other list elements: the expression for the model, the fitted parameters, and the information criteria. All of these items are logically related, and if you name the list elements you can easily access the information wanted.

```
> mylist <- list(species,x,y,z)
> names(mylist) <- c("species", "x", "y", "z")
> class(mylist)

[1] "list"

> mode(mylist)

[1] "list"

> length(mylist)

[1] 4

> dim(mylist)

NULL
```

Note that lists have no "dimension" (because they are more flexible vectors - kinda like stretchy christmas stockings). However, if you have a matrix or a dataframe within a list those will have vectors. The object within lists have their own attributes.

You can access the z element of mylist in a number of ways, using the `$` operator and the name of the element, by the index number and double brackets, or by name and double bracket:

```
> mylist$z

     tom dick harry
[1,]   1    3     5
[2,]   2    4     6
attr(,"names")
[1] "a" "b" "c" "d" "e" "f"

> names(mylist$z)

[1] "a" "b" "c" "d" "e" "f"

> names(mylist[[4]])
```

```
[1] "a" "b" "c" "d" "e" "f"
```

```
> names(mylist[["z"]])
```

```
[1] "a" "b" "c" "d" "e" "f"
```

R has a rich collection of functions which are very helpful for creating and manipulating objects, so a bit of code can substitute for whole lot of typing. The tables below list some helpful functions. Look up help for anything you don't know. It will soon start making sense!

commands	actions
c(n1, n2, n3)	combines elements into an object
cbind(x, y)	binds objects together by column
rbind(x, y)	binds objects together by row

Table 3.2: Common combine functions used for creating data objects from existing objects

commands	actions
seq()	generate a sequence of numbers
1:10	sequence from 1 to 10 by 1
rep(x, times)	replicates x
sample(x, size, replace=FALSE)	sample size elements from x
rnorm(n, mean=0, sd=1)	draw n samples from normal distribution

Table 3.3: Functions used for creating sequences and sampling

commands	actions
vector()	create a vector
matrix()	create a matrix
data.frame()	create a data.frame
as.vector(x)	coerces x to vector
as.matrix(x)	coerces to matrix
as.data.frame(x)	coerces to data frame
as.character(x)	coerces to character
as.numeric(x)	coerces to numeric
factor(x)	creates factor levels for elements of x
levels()	orders the factor levels as specified

Table 3.4: Functions used for creating and coercing objects to new type/class

3.3 Simple plotting

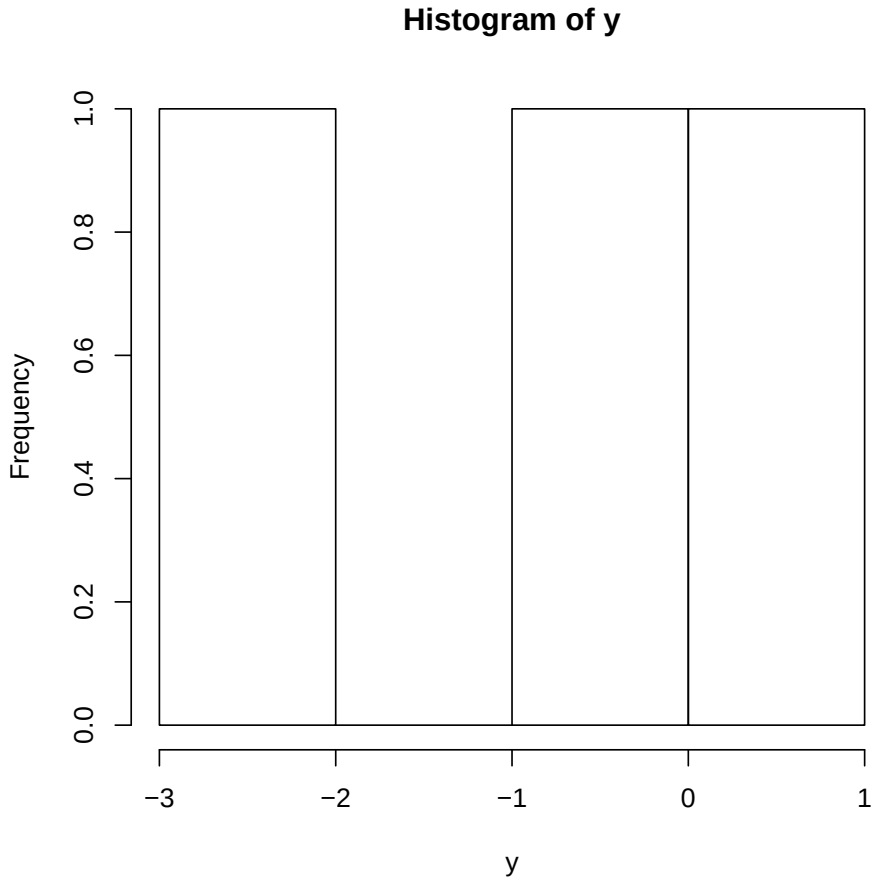
The generic function for plotting in R is `plot`. (NEED TO EDIT THIS)

Plot has a huge number of options for changing the symbols (see `?points`, color, size of symbols, axes. labels, adding regression lines or straight lines, etc. Creating multiple panels on a page, etc. Help pages you may want to visit include `?lines`, `?abline`, `?par`, `?axis`.

3.3.1 Univariate plot

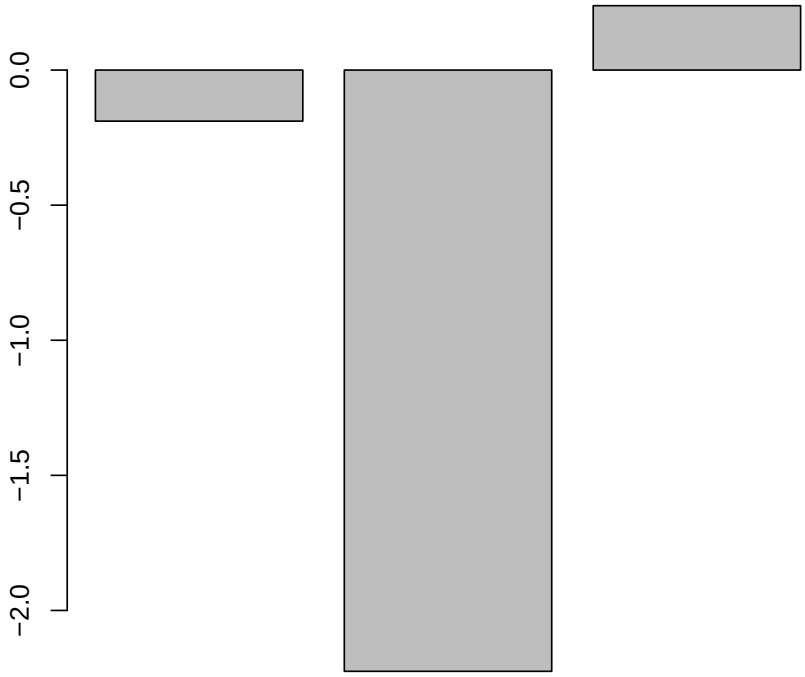
To plot a histogram, use:

```
> hist(y)
```



To plot a bar plot, use:

```
> barplot(y)
```



Practice

1. Create a dataset with simulated data using `rnorm()`.

   (a) Simulate 21 random data points drawn from a normal distribution (create a numeric vector), and save it in the variable "y". Create a second set of 21 points and save it as "y1".

```
> y <- rnorm(21)
> y1 <- rnorm(21)
> y
 [1]  0.42925522  0.93889973 -2.21735173 -0.01414596 -1.23276028  0.14569501
 [7]  0.66271863  0.71463989  0.68056766 -1.12142039  0.75404823 -0.05967583
[13]  0.74272919 -0.37372720 -0.29746154  0.84216124 -0.71524691  0.11090888
[19] -0.88330717  0.43067966 -0.44243766

> y1
```

```
 [1]  0.746261173  0.059111083 -0.229565663 -2.038115060  0.060646819
 [6] -0.103538466  0.001809016  0.891693909 -0.842844551  0.160416777
[11]  0.346901620  1.502334945 -1.079793730 -0.331938218 -1.331224920
[16] -0.716953042  1.651887457 -0.489992095 -0.629985532 -0.773549565
[21]  0.195006015
```

(b) Create a treatment vector with levels "low", "med", and "high", save it as a factor.

```
> treatment <- factor( c("low", "med", "high") )
> treatment
```

```
[1] low  med  high
Levels: high low med
```

(c) Our treatment has numeric values also, so create a numeric vector with the values 2, 4, 8, save it as x.

(d) Create a species vector with seven names.

(e) Create a matrix with y in the first column and x in the second column, save it as dat.matrix.

(f) Create a data frame with species, x, treatment, y and y1, save as dat. Why can't you make a matrix with these columns?

(g) Make a bivariate plot of the numeric value of the treatment (x) versus the response (y). You may want to check the help documentation for "plot". You will have to select the columns of the data frame.

(h) Make a plot on the treatment as factor versus the response. What is the difference between these two plots?

(i) Is the factor displayed in the plot in the order that makes sense? If not, fix this by applying factor to the treatment column of dat again, but this time specifying the levels vector with names of the levels in the order you want. You may want to look at the help page for factor. Plot it again.

(j) Let's make a scatterplot (`plot(y, y1)`) to see if there is any structuring in the data (eventually with respect to the treatment levels – the rest of this exercise is in the chapter on Workhorse Functions of Data Analysis). While we're at it, let's make it prettier. Change the symbols to solid circles by adding the optional parameter `pch=16`, and the points bigger by `cex=2`. Change the color to red using `col="red"`.

(k) Now let's make some data which should differ. For the "low" treatment, simulate y and y1 as normally distributed data with mean = -2 and sd=.5, and "high" as mean=5, and sd=3. Remake the dataframe.

(l) Now make two boxplots: treatment vs. y and treatment vs. y1.

(m) Make boxplots of species vs. y and species vs. y1. Why would you make this plot?

Chapter 4

Playing with R for the first time

4.1 Instructions

In this exercise, I want to introduce you to some of the built-in help facilities and documentation in R, and get you started with manipulating variables in R.

- If you haven't already done so, make a directory for this class. I would recommend naming it "Rclass" and putting it at some accessible location in your user directory (like at the top level of your User directory on a Mac, or at `C:/` on a windows machine). On my computer it would be like so: Fig. 4.1.

- Also within this directory, make another directory called "data". You will store all of your data files in there.

- Start up R.

- Move to your Rclass directory by using the `setwd("path to Rclass")` command.

  ```
  > setwd("~/Rclass")
  ```

  On a PC it will be something like:

  ```
  > setwd("C:/Rclass")
  ```

- Open the help facility using the command

  ```
  > help.start()
  ```

- Click on "An Introduction to R". The is *"the Bible"* for learning R.

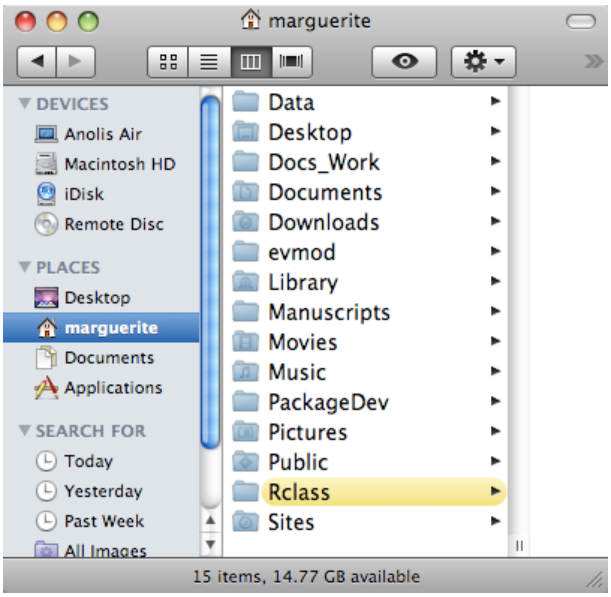


Figure 4.1: Rclass directory for saving course work. Make a folder in a convenient location on your computer, like at the top level of your user directory. When you are done with the course, you can move the whole folder to a permanent location with your other R code.

4.2 R session

Later, when you have more time, you will want to read and try out all of the section "Simple manipulations; numbers and vectors" (2.1 – 2.8). Please type the commands in yourself rather than cut-and-pasting. The typing helps develop "finger memory" which you will need to become proficient at programming.

4.2.1 Vectors

For now, let's try playing around with R. Create a variable or "object" named `height` and save a value of 10. The arrow means to put "10" into `height`:

```
> height <- 10
```

To see the value of `height`, type it and press return:

```
> height
```

```
[1] 10
```

Now let's create a vector, a variable with several elements or "observations":

```
> height  <- c(10, 12, 51, 24, 32)
> height
```

```
[1] 10 12 51 24 32
```

The `c()` function *combines values into a vector or a list.* You will use it a lot. Create a vector of weights:

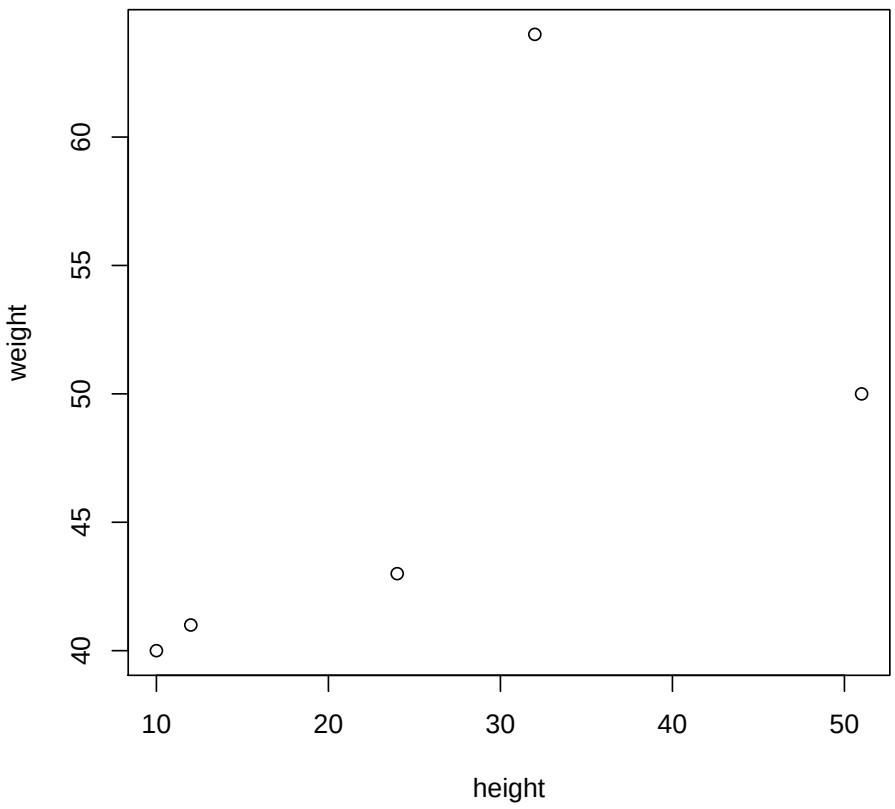
```
> weight <- c(40, 41, 50, 43, 64)
> weight
```

```
[1] 40 41 50 43 64
```

Whenever I am writing new code, I *ALWAYS* check to make sure the code produced the results I wanted. You should do the same. *Verify that EACH step worked correctly.* This means without errors!

Let's create our first plot. We'll use the `plot()` function, which is a generic function for just about any type of R object:

```
> plot(height, weight)
```



Voila!! Our first plot. Beautiful. Now suppose that we have males and females in the data, so we'd like some categorical variables for sex. In R, it's easy to do. Just create a character vector:

```
> sex <- "male"
> sex
```

```
[1] "male"
```

```
> sex <- c("male", "male")
> sex
```

```
[1] "male" "male"
```

4.3 Functions

We can create an object `sex` that contains one or more character strings in it, and use the generic `c()` function to create a vector of character strings. But it can get tedious typing the same thing over. So we can use the `rep()` function to repeat values:

```
> sex <- rep("male", 3)
> sex

[1] "male" "male" "male"

> sex <- c(sex, "female", "female")
> sex

[1] "male"   "male"   "male"   "female" "female"

> sex <- c( rep("male", 3), rep("female", 2) )
> sex

[1] "male"   "male"   "male"   "female" "female"
```

So what just happened? Why do the last two lines of code give the same result? *In R, as in most programming languages, the code is nested. The innermost function or bit of code is evaluated first, and whatever is returned is then the argument for the next outer bit of code.* So in the first line, we take three copies of "male" and shove it into `sex`. In the second line, we take the object `sex`, which is now a vector of three "male", and combine it with two copies of "female", into a vector of 5 elements. In the last line, first we create a vector of 3 males using the `rep` function, and then a vector of 2 females, and combine these two vectors together into a vector of 5 elements.

In fact, the last line above is equivalent to the following (of course you would never actually write a line like the one below, the nested `c()` are completely unnecessary, this is just for demonstration):

```
> sex <- c(c("male", "male", "male"), c("female", "female"))
> sex

[1] "male"   "male"   "male"   "female" "female"
```

You can see that using functions like `rep()` and `seq()` for sequence can save you a lot of time, when you have lots of repeated data.

```
> sex <- c(rep("male", 50), rep("female", 50))
> sex
```

```
  [1] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "
 [12] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "
 [23] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "
 [34] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "
 [45] "male"   "male"   "male"   "male"   "male"   "male"   "female" "female" "female" "
 [56] "female" "female" "female" "female" "female" "female" "female" "female" "female" "
 [67] "female" "female" "female" "female" "female" "female" "female" "female" "female" "
 [78] "female" "female" "female" "female" "female" "female" "female" "female" "female" "
 [89] "female" "female" "female" "female" "female" "female" "female" "female" "female" "
[100] "female"
```

4.3.1 Generating Random Deviates

Now let's go back to our original height and weight variables and make up some larger samples. This time, let's use the random number generator function `rnorm()`, which generates random normal deviates. We can specify the mean and standard deviation as below. Let's make the males with larger mean, but same standard deviation. To save paper, I'm not going to display the object contents to the screen, but you should keep doing it.

```
> height_m <- rnorm(50, mean=55, sd=5)
> height_f <- rnorm(50, mean=45, sd=5)
```

How do we combine these into one vector?

```
> height <- c(height_m, height_f)
```

We could have also created the height vector in one step. While we're at it, let's also make up some data for weight. Let's pretend that this data is for children in inches and pounds:

```
> height <- c( rnorm(50, mean=55, sd=5), rnorm(50, mean=45, sd=5) )
> weight <- c( rnorm(50, mean=80, sd=10), rnorm(50, mean=65, sd=8) )
```

In general, it's best to keep your coding simple, especially when you are learning. Write clean code that is easy for you to understand. If it takes an extra line, it's not a big deal. The computer is VERY fast, you will not slow down your program this way. On the other hand, you can easily confuse yourself and make BIG MISTAKES by trying to be too clever.

To plot by sex, we need to do tell R that the object `sex` contains categories or "factors". We do this using the `factor()` function:

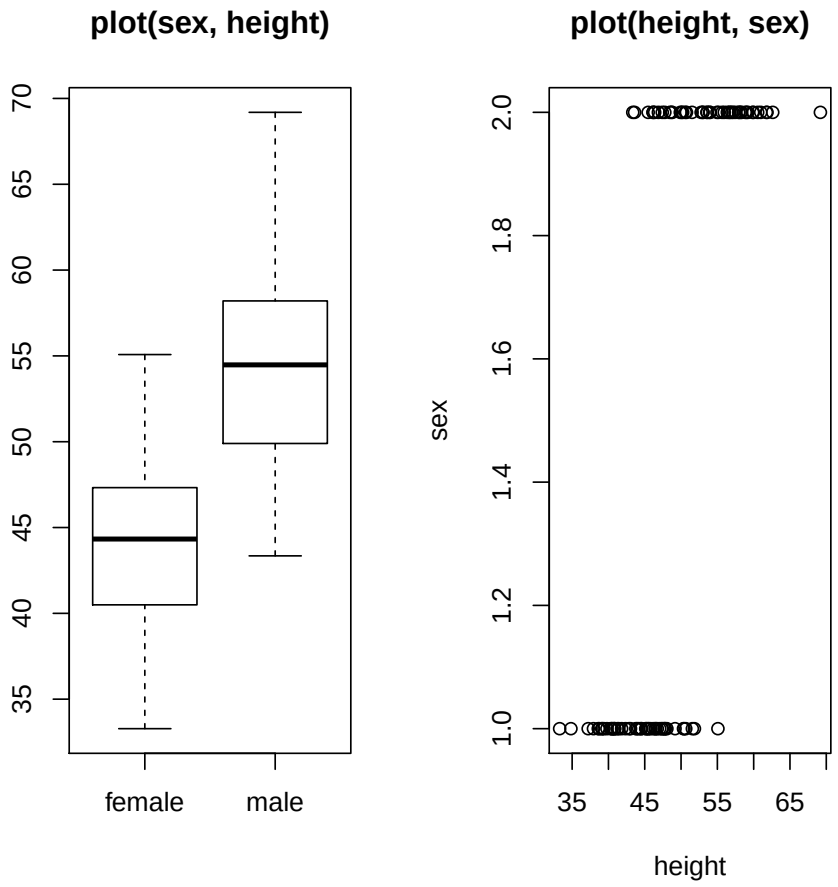
```
> sex <- factor(sex)
> sex
```

```
  [1] male   male   male   male   male   male   male   male   male   male   male   male
 [15] male   male   male   male   male   male   male   male   male   male   male   male
 [29] male   male   male   male   male   male   male   male   male   male   male   male
 [43] male   male   male   male   male   male   male   male   female female female femal
 [57] female female female female female female female female female female female femal
 [71] female female female female female female female female female female female femal
 [85] female female female female female female female female female female female femal
 [99] female female
Levels: female male
```

`sex` is now a factor, or categorical variable with two levels. Now we can plot with sex. Note that in R, when you make a bivariate plot where the first variable is a factor, it will create a barplot by default. If you put the quantitative variable first, you will get a scatterplot:

```
> plot(sex, height, main="plot(sex, height)")
> plot(height, sex, main="plot(height, sex)")
```

We added titles to the plots with the `main="mytitle"` argument, which is optional.



Now, let's run some statistics on our data. Is a child's weight related to height? We might want to run a linear regression, which we so using the `lm()` or linear model function. It produces a linear model object, let's save the output as lm.mf:

```
> lm.mf <- lm( weight ~ height )
```

There are several ways to give the linear model argument to `lm`, I prefer to use the formula representation `weight ~height`, which is read *weight as a function of height*. You can produce the a summary of the regression using `summary()`. Often, however, you want to see an anova table:

```
> summary(lm.mf)


Call:
lm(formula = weight ~ height)

Residuals:
```

```
    Min      1Q  Median      3Q     Max
-22.252  -6.624  -1.007   5.587  40.851

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  49.9385     7.6824   6.500 3.37e-09 ***
height        0.4568     0.1550   2.948    0.004 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 11.12 on 98 degrees of freedom
Multiple R-squared: 0.08146,        Adjusted R-squared: 0.07208
F-statistic: 8.691 on 1 and 98 DF,  p-value: 0.003998
```

```
> anova(lm.mf)

Analysis of Variance Table

Response: weight
          Df  Sum Sq Mean Sq F value   Pr(>F)
height     1  1074.6 1074.63  8.6907 0.003998 **
Residuals 98 12118.0  123.65
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Oh wow, there is a very significant effect. But wait! We have boys and girls in the dataset. We need to add in gender as a covariate:

```
> lm.mf <- lm(weight ~ sex + height)
> anova(lm.mf)

Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value    Pr(>F)
sex        1 4045.2  4045.2  44.264 1.707e-09 ***
height     1  282.8   282.8   3.094   0.08173 .
Residuals 97 8864.7    91.4
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

See what happened? Actually, males and females are significantly different, and there is no relationship between height and weight after accounting for sex. Does this make sense, given how we generated the data?

Note: Using the "+" between the parameters `sex` and `height` means to put them in as additive factors. If you want to include these as well as interactions, use "*". For interactions only (hardly ever done), use ":". Give it a try. For more explanation, see the formula help page:

```
> ?formula
```

4.3.2 Building a dataframe

The most typical data structure you will use is a dataframe. It is a "record format" type of layout, with the idea being one row per observation. You may have additional information or metadata that you want stored with your individual observations. For example, you may want a unique ID for each individual, and what city they are from, etc.

Let's create a unique ID. For some reason, we want each boy numbered from 1 to 50. Let's use the `seq()` function to create a sequence from 1 to 50, and the `paste()` function to combine them with "boy":

```
> seq(1,50)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
[34] 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Is the same as:

```
>  1:50
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
[34] 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Paste together with the word "boy":

```
>  boys <- paste("boy", 1:50, sep="")
>  girls <- paste("girl", 1:50, sep="")
>  ID <- c(boys, girls)
```

We separated the two parts of the paste with nothing, "". We could have separated with a "." or whatever we want.

Now let's create a city object. Suppose we collected 25 observations of each sex in Honolulu and Santa Barbara:

```
> city <- c( rep("Hon", 25), rep("SB", 25), rep("Hon", 25), rep("SB",25))
```

We could also repeat the repeat, since the 25 per city is a repeating pattern:

```
> city <- rep( c(rep("Hon", 25), rep("SB", 25)), times = 2)
```

This time we had to use the `times=` option, which means how many times to repeat the whole sequence. The other popular option is `each=`, which repeats element by element. To see the difference more clearly, try this simple example:

```
> rep(1:5, times=3)

 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

> rep(1:5, each=3)

 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

Now let's save all of our dataset together into one neat dataframe:

```
> dat <- data.frame(ID, sex, height, weight, city)

> dat
```

For really large datasets (saving paper here, you can print to screen), we can just look at the beginning and end of our dataframe:

```
> head(dat)

    ID  sex   height   weight city
1 boy1 male 50.74248 88.83133  Hon
2 boy2 male 46.14909 66.38892  Hon
3 boy3 male 53.56522 63.32075  Hon
4 boy4 male 46.36637 87.32131  Hon
5 boy5 male 50.58725 85.69922  Hon
6 boy6 male 62.62308 67.86794  Hon

> tail(dat)

        ID    sex   height   weight city
95  girl45 female 34.83214 68.87919   SB
96  girl46 female 43.03704 78.05635   SB
97  girl47 female 47.72882 71.00597   SB
98  girl48 female 40.86659 78.12755   SB
99  girl49 female 40.81870 73.01877   SB
100 girl50 female 41.40929 63.28541   SB
```

4.4 Save Your History

At the end of your session, save your session history. On a mac, press the little blue and yellow history icon and you will see the history sidebar appear. Click on "Save History." Give it a file name as below. On a PC, use the savehistory function, read the following help page :

```
> ?savehistory
```

You will want to save it with an informative file name like `VecPractice.history`. For example:

```
> savehistory(file="VecPractice.history")
```

4.5 Insert Comments

Outside of R, open up your history (in a text editor) and clean up your code. Add comments to help you remember what this code means. Place them in your history, just before the relevant section. Comments in R are indicated by the `#` symbol. Anything to the right of one or more `#` is considered a comment, and not executed by R.

4.6 Exercises

Work through the section "Simple manipulations; numbers and vectors" (2.1 – 2.8) in the *Introduction to R* – see top of this chapter for how to find it – and answer the questions below.

1. What is a numeric vector?
   Answer: `## A numeric vector is an ordered collection of numbers.`
2. Is ordinary arithmetic (+, -, *, /) on vectors in R done element-by-element or using matrix math? (to test an example, try or think about `x*y` where:
$$x = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \qquad y = \begin{pmatrix} 5 \\ 1 \end{pmatrix}$$
3. What is a sequence?
4. What is an logical value? What is a logical vector?
5. What is a missing value?

6. What is a character vector?
7. What is an index vector?

Chapter 5

Finding Help

R has great built-in help facilities. Once you get used to R's syntax (the form of R functions and data), you will find them incredibly useful.

Every object that comes with the R program is documented in some way – this means every function, internal dataset, as well as methods and classes (which we won't have time to cover).

5.1 When you know the name of the function

Say you want to find the mean of your data, so you guess that there is a function called `mean()`. Finding help is easy:

```
> ?mean
```

Will bring up the help page, and is equivalent to:

```
> help(mean)
```

Notice that as you type `help(` you start to see the function definition on the bottom of the console window. It shows you how to call the function (what variables it expects).

Looking at the help page, notice that there are sections (these are common to most help pages):

Description what it does

Usage the format for calling the function (making it run)

Arguments explanation for each of the arguments, their type, and what they represent

Details more explanation

Value what is returned from calling the function

Author

References

See Also other functions to check out

Examples Often the most valuable section, with examples that actually work. You can test them out by cutting and pasting into the R console.

There are also hyperlinks in many help documents, to related help pages, so you can "surf" you way through help.

5.2 Don't know the name of the function

But first to access the help for a specific function, you need to know what it is called.

Two good options are:

```
> help.start()
```

Which will bring up an an html browser, which you can browse. Click on "Packages", then "base" if you are looking for a basic function that should be in the base distribution of R. Click on the package name if you are looking for a function in a package. Browsing through the help is very useful for beginners.

```
> help.search("plot")
```

Will do a "fuzzy" search (i.e., will also match words close in spelling – not exact – to plot). Of course, replace "plot" with whatever you are looking for. This function searches through the full text of the help docs, so for a common word like plot, this will return a huge list, which you can look through package by package.

5.3 Package-specific help

Packages are generally a set of functions that are loaded from some (hidden) directory on your computer into active memory, so that you can use them by name. Now that you know the names of the functions, you can access specific help pages directly. Try the help page for independent contrasts:

```
> help(pic)
```

Here's a harder example. you might want to know more about the phylogeny plotting function in `ape`. If `tree` is a tree object in `ape`, you can use `plot(tree)` to call the function, so you might think that you can find the help page by using `help(plot)` or `?plot`. However, this brings up the generic plot function which doesn't say anything about the one you want (the tree plotting function in `ape`.

What is going on is that `ape` has a method set for plotting objects of the class `phylo`, so that you don't have to remember the specific function name. This is actually a wonderful feature of object-oriented programming, otherwise you would have to remember thousands of functions, all uniquely named.

So how do we find the one we want? You could try:

```
> help(plot, package="ape")
```

But you will see that this doesn't return anything. This means that the actual plotting function in `ape` is named something else, so that there is no function in ape named "plot" (R requires all named functions in packages to be documented).

Huh? How does `plot()` plot a phylogenetic tree when there is no function called `plot` in `ape`? This is an example of a *generic* function. The function `plot` is actually a generic, with different specific functions for different types of objects – R automatically chooses the correct one by looking at the objects `class`.

Anyway...

You have a couple more options (in addition to the general options above):

help(package="ape") will return the package's main help page, where you can see a list of functions, but they are not clickable. Once you locate the name of the function you can follow up with a help(plot.phylo).

methods(plot) will return all of the methods written for the generic plot call. Looking through it, you might guess that `plot.phylo` is the one you want. NOTE: this only works for S3 methods.

Chapter 6

What is it?

When working with a new package, you want to know what kind of object you are dealing with. Check its class and attributes.

To illustrate, let's make up some data:

```
> x <- 1:9
> x

[1] 1 2 3 4 5 6 7 8 9

> class(x)

[1] "integer"
```

`x` is a numeric vector. Vectors have length, but not dimension:

```
> length(x)

[1] 9

> dim(x)

NULL
```

However, we can easily change it into a matrix by giving it row and column dimensions. This has to be specified as a vector with number of rows, number of columns, here made with the combine function: `c(3,3)`

```
> dim(x) <- c(3,3)
> x
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
> class(x)
```

```
[1] "matrix"
```

Let's make a data frame, and name it `dat`

```
> species <- LETTERS[1:3]
> species
```

```
[1] "A" "B" "C"
```

```
> dat <- data.frame(species, x)
> dat
```

```
  species X1 X2 X3
1       A  1  4  7
2       B  2  5  8
3       C  3  6  9
```

Conveniently, the name "species" was correctly assigned to the first column of the dataframe, but the columns from x were given default names (because the columns of `x` did not have names to begin with). So let's replace these with more meaningful names:

```
> names(dat)
```

```
[1] "species" "X1"      "X2"      "X3"
```

```
> names(dat) <- c("species", "length", "width", "height")
> dat
```

```
  species length width height
1       A      1     4      7
2       B      2     5      8
3       C      3     6      9
```

Alternatively, we could have given column names to `x` and before making the dataframe:

```
> x

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> colnames(x) <- c("length", "width", "height")
> x

     length width height
[1,]      1     4      7
[2,]      2     5      8
[3,]      3     6      9

> dat <- data.frame(species, x)
> dat

  species length width height
1       A      1     4      7
2       B      2     5      8
3       C      3     6      9
```

We can see all attributes of `dat`

```
> attributes(dat)

$names
[1] "species" "length"  "width"   "height"

$row.names
[1] 1 2 3

$class
[1] "data.frame"
```

A vector has no dimension. So it's easy to turn `x` from a matrix back to a vector by getting rid of its dimensions. `NULL` is a special R variable.

```
> dim(x) <- NULL
> x
```

```
[1] 1 2 3 4 5 6 7 8 9
```

```
> class(x)
```

```
[1] "integer"
```

Another useful "What is it" function is `str` for structure, which nicely summarizes what it is:

```
> str(dat)
```

```
'data.frame':        3 obs. of  4 variables:
 $ species: Factor w/ 3 levels "A","B","C": 1 2 3
 $ length : int  1 2 3
 $ width  : int  4 5 6
 $ height : int  7 8 9
```

Chapter 7

Saving your work as R scripts

Chapter Topics:

- Building good scripts
- Running source code
- Debugging scripts
- Clearing your workspace

Skills: writing clean source code, verifying, using `print` and `cat`, using history files.

Because R is interactive, it is tempting to simply play with code until you get the results you want. The problem with this is that you may not be able to reproduce it. Also, you may have made many manipulations of your data, some of which you've lost track of, and so your data objects may not really be what you think they are. This makes it impossible to double-check your analysis.

A key part of any analysis is verification:

1. Did you do what you really think you did?
2. Was the input free of error?
3. Did the steps of your analysis work without error?
4. And perhaps most importantly – can you reproduce it?

To be able to accomplish these goals, you want to create clean scripts. Scripts are lines of code saved in an ordinary text file with a `.R` or `.r` ending. (Make sure it is plain text, and NOT a `.rtf`, or a `.doc`, etc file).

All good script follows the first three R's, as you increase along the path of R jedi-hood, you will add on the 4th R:

1. Readable – If you look at the script in a month or 6 months, will you be able to easily understand it?
2. Right – Does it run free of error, and does it produce correct results?
3. Repeatable – Can you reproduce your results from your input data?
4. Reusable – Is your coding modular and designed well so that your code can interact with other scripts, and/or use it for other purposes?

The mac interface has a very nice text editor. From the R menu, choose File > New Document (or command-N). Simply type or cut and paste your code from your history file into here. Let's make a script for the analyses we've done thus far.

7.1 Script template

First, make sure that you are in the directory that you want the script to execute from (`Rclass`). Start off with any packages that you wish to load, then begin to cut and paste your code. Make sure to add comments indicated by the `#` symbol so that you know what the code does:

Here is the basic structure of a script:

```
> require( ...addonpackage... )    # anything between ... needs to be changed
>                          # if none, then you don't need that line
>
> dat <- read.csv(..."your input file.csv"... )       # input data
>
> # Your lines of code to run analyses
> # You may have output or processed data that you want to save,
> # create an object for it and write it out to a csv file at the end
>
> # plot graphics
>
> write.csv(out, file="myoutput.csv")      # output data
```

And here is a simple example script that reads in data, calculates summary statistics, a linear regression, and a couple of figures.

```
> #require(stats)        # stats is part of the base package and doesn't need to be lo
>                 # but if you need an add-on package, you would require it here.
>
> dat <- read.csv ("Data/morphpre.csv")  # read in data
```

```
> lm.HLSVL <- lm(dat$HandL ~ dat$SVL)   # run a linear model
> summary(lm.HLSVL)        # get summary statistics
> str(lm.HLSVL)        # look at the linear model object
> coef(lm.HLSVL)[2]        # get the slope of the regression
> plot(dat$HandL ~ dat$SVL, cex=2)    # make a plot with big dots (cex controls size o
> abline(lm.HLSVL, col="red")             # plots the regression line, in red
>   title("Microhylid Hand Length vs Body Size")        # add a title
>   text(x=15, y=13, paste("slope = ", coef(lm.HLSVL)[2]))
>                   # add important info to the text
+
+ ###
+ # please insert your other lines of code here -- enough
+ #         to save a meaningful analysis
+ ###
```

Note that I have used spacing and indents to increase the "readability" of the code. Use it to set of blocks of code that accomplish one task, with indents to indicate heirarchy. We will talk more about this in the functions section.

Save the script file as "`testScript.R`" or an appropriate title in your `Rclass` folder. Now if you want to run the code, you simply type at the R console (from within your Rclass directory):

```
> source("testScript.R")
```

When I am trying to develop a script, I often work by having the script window open next to the R console, and once a bit of code is working, I cut and paste it directly into the script. Save the script and source it. Once you have a good amount of code, you can work by making changes to the script, saving, and sourcing, over and over again.

7.1.1 Writing pdf to file

If you'd like to print your pdf to a file instead of to the screen, you can add the following code into your script:

```
> pdf(file="MicrohylidHandLvsSize.pdf")        # open pdf device for printing
>   plot(dat$HandL ~ dat$SVL, cex=2)        # remake plot as before
>   abline(lm.HLSVL, col="red")
>     title("Microhylid Hand Length vs Body Size")
>     text(x=15, y=13, paste("slope = ", coef(lm.HLSVL)[2]))
> dev.off()        # turn off pdf device so future plots go back to screen
```

7.1.2 History file

Another handy feature of R is that it automatically saves a history file. That is, a file that has a list of every command you've executed in your sessions. It is saved by default as `.history` in your working directory. Because the file name begins with a period, it is not visible normally (although it is there – you can see it from the terminal by using the `ls -a` command). To save it explicitly with your own filename, either click on the history button on the R gui (box with yellow and blue lines), and click on "save history" at the bottom of the side window, or type the code:

```
> savehistory(file = "date_today.Rhistory")
```

This is an ordinary text file, which you can open up and edit (removing all the mistakes), and save as a `scriptname.R` file.

Another helpful tip when writing source code is to use `print` and `cat` functions to print out your output to the console. When you are using R in interactive mode, when you type the name of a variable, you get a print of its contents. However, when you source the same code, the variable does not print to the screen. You have to explicitly put a `print` or `cat` function around it.

Let's use a built-in dataset called `iris`, which is the famous Fisher iris dataset. Make a test script file and save it as `test.R`:

```
> names(iris)         # will not print to console when sourced
> spp <- unique(iris$Species)  # only unique values
> spp <- as.character(spp)     # factor -> character
> spp                # will not print to console when sourced
> print('Species names')  # will print
> print(spp)              # will print
> cat('\n', 'Species names =', spp)  # concatenate
>                 # \n is a carriage return character
> summary(iris)
```

Then test it by running:

```
> source("test.R")
```

```
[1] "Species names"
[1] "setosa"     "versicolor" "virginica"

 Species names = setosa versicolor virginica
```

You can see that `print` just makes a rough dump of the variables onto the screen. I added a character string so that we would know what variable was being printed to screen. `cat` makes a nicer, more customized display (it turns everything into a character vector, then pastes them together [i.e., concatenates them] before printing). They both do the same basic job, however. Notice also that `summary` does print to screen. Usually you only need to use these explicit print statements to see the contents of your variables as you are debugging.

7.2 Remember the workspace

Finally, remember that R is interactive, and the objects you create during a session are still around even after you've run your source code and forgotten about them. So to really check that your script is complete, you should shut down R (don't save the workspace), double click on the name of your script to restart R in the correct directory, and then source the program again. Does it work? Great!!

You could also try clearing all the objects from your workspace using the command:

```
> rm(list=ls())     # remove a list of objects consisting of the entire workspace
```

But this doesn't unload your packages, and there is still a danger that the script won't run in a fresh session. It's OK for minor incremental changes, but the best thing for a real test is to quit R and retry with a blank slate.

In general, most of my analyses are pretty quick in terms of computer time (not programming time!). So I never save my workspace, because I don't want to deal with any "ghost" objects I have forgotten about. Instead, I write a nice script that will generate the whole analysis. If it's a really big complex analysis, you can save intermediate output as r data files (more on this later).

Try to create a script file for all the analyses we've done so far (and for every session throughout the course).

7.3 Exercises

1. Create a script of the work we've done so far.

2. R has great diagnostic plots for linear models. Read about them in the help page for ?plot.lm and incorporate a multi-panel figure by adding two lines of code to the script you've already made:

```
> par(mfrow = c(2,2))   # set the plot environment to have two rows and two columr
> plot(lm.HLSVL)
```

3. Save output to a file.

4. Modify 'test.R' so that a summary of the iris data prints to the console when sourced.

5. Explore other datasets in R. At the R command prompt type `data()` to see what is available.

Chapter 8

Data Input and Output

So far, we have been working within R, either typing data in directly or using R's functions to generate data. In order to analyze your own data, you have to load data from an external file into R. Similarly, to save your work, you'll probably want to write files from R to your hard drive. Both of these require interacting with your computer's operating system. In this chapter, we're just going to do it. We'll talk more about what's going on in a later section on the R Environment.

8.1 Getting your data into R

The most convenient way to read data into R is using the `read.csv()` function. This requires that your data is saved in .csv format, which is possible from Microsoft Excel (save as... csv) or any spreadsheet format. It is a text format with data separated by commas. It is very nice because it is unambiguous, not easily corruptible, and non-proprietary. Thus it is readable by nearly every program that reads in data.

First, within your "Rclass" folder, create a folder named "Data". Copy the file "anolis.csv" and "Iguanamass.csv" into this folder.

Next, from within R, check which working directory you are in. You should be in your Rclass folder. If you are not, use `setwd()` to get there.

```
> getwd()
> setwd("~/Rclass")  # my folder is at the top level of my user directory
```

8.1.1 read.csv

Getting the file in is easy. If it is in csv format, you just use:

```
> read.csv("Data/anolis.csv")  # look for the file in the Data directory
```

This is an *Anolis* lizard sexual size dimorphism dataset. It has values of dimorphism by species for different ecomorphs, or microhabitat specialists.

To save the data, give it a name and save it:

```
> anolis <- read.csv("Data/anolis.csv")
```

It is a good practice to *always* check that the data were read in properly. If it is a large file, you'll want to at least check the beginning and end were read in properly:

```
> head(anolis)
```

```
  species   logSSD    ecomorph
1      oc -0.00512        twig
2      eq  0.08454 crown-giant
3      co  0.24703 trunk-crown
4     aln  0.24837 trunk-crown
5      ol  0.09844  grass-bush
6      in  0.06137        twig
```

```
> tail(anolis)
```

```
   species  logSSD     ecomorph
18      cr 0.39796 trunk-ground
19      st 0.15737  trunk-crown
20      cy 0.26024 trunk-ground
21     alu 0.08216   grass-bush
22      lo 0.13108        trunk
23      an 0.13547         twig
```

Voila! Now you can plot, take the mean, etc. Which prints out the first six and last six lines of the file.

R can read in many other formats as well, including database formats, excel native format (although it is easier in practice to save as .csv), fixed width formats, and scanning lines. For more information see the R manual "R Data Import/Export" which you can get from `help.start()` or at http://www.r-project.org.

8.2 Summary statistics on your data

Suppose you wanted to compute and save the means and standard deviations for the sexual size dimorphism values. A very convenient function for computing any function over groups in your dataframe (here, ecomorphs), is the function `aggregate` (look up help via `?aggregate`).

Calculate the mean by ecomorph group:

```
> aggregate(anolis$logSSD, by=list(anolis$ecomorph), mean)
```

```
       Group.1         x
1  crown-giant 0.1391750
2   grass-bush 0.1437525
3        trunk 0.1467167
4  trunk-crown 0.2626575
5 trunk-ground 0.3339650
6         twig 0.0848450
```

Notice we had to type `anolis$` in front of the variables we wanted. This is because these vectors are within the dataframe `anolis`. To be able to access `anolis`ś goodies, we need to tell R where to look (more on this later).

Notice that the argument to by, which groups we want the mean over, has to be a list, so we coerced the variable `anolis$ecomorph` into a list.

Calculate the mean and the sd by ecomorph group, and this time save them:

```
> anolis.mean <- aggregate(anolis$logSSD, by=list(anolis$ecomorph), mean)
> anolis.sd <- aggregate(anolis$logSSD, by=list(anolis$ecomorph), sd)
> anolis.sd
```

```
       Group.1          x
1  crown-giant 0.09909567
2   grass-bush 0.06924584
3        trunk 0.02136480
4  trunk-crown 0.09968872
5 trunk-ground 0.06966130
6         twig 0.07107131
```

Give the results of aggregate meaningful column names:

```
> names(anolis.mean)   # check that this is what we want to modify
```

```
[1] "Group.1" "x"
```

```
> names(anolis.mean) <- c("ecomorph", "mean")
> names(anolis.sd) <- c("ecomorph", "sd")
```

While we're at it, let's get the sample size so that we can calculate the standard error, which is the standard deviation divided by the square root of the sample size.

```
> anolis.N <- aggregate(anolis$logSSD, by=list(anolis$ecomorph), length)
> names(anolis.N) <- c("ecomorph", "N")
```

8.2.1 merge

It's not convenient to have so many data objects, what we'd really like is to have all summary statistics together in one data frame. So let's use the `merge` function.

Merge works two objects at a time, and merges by default on the common column names (here, ecomorph):

```
> merge(anolis.mean, anolis.sd)
```

```
      ecomorph      mean         sd
1  crown-giant 0.1391750 0.09909567
2   grass-bush 0.1437525 0.06924584
3        trunk 0.1467167 0.02136480
4  trunk-crown 0.2626575 0.09968872
5 trunk-ground 0.3339650 0.06966130
6         twig 0.0848450 0.07107131
```

Otherwise, you must specify `by=`. Or to be safe, you can specify it, it's good practice:

```
> out <- merge(anolis.mean, anolis.sd, by="ecomorph")
```

There is also options for `by.x=` and `by.y=` in case your columns have different names in the two objects – you can tell R which two columns to match.

Do it again to add the third object, N:

```
> out <- merge(out, anolis.N, by="ecomorph")
> out
```

```
      ecomorph      mean         sd N
1  crown-giant 0.1391750 0.09909567 4
2   grass-bush 0.1437525 0.06924584 4
3        trunk 0.1467167 0.02136480 3
4  trunk-crown 0.2626575 0.09968872 4
5 trunk-ground 0.3339650 0.06966130 4
6         twig 0.0848450 0.07107131 4
```

Now, it's easy to compute the standard error:

```
> out$se <- out$sd / sqrt(out$N)
> out
```

```
      ecomorph      mean         sd N         se
1  crown-giant 0.1391750 0.09909567 4 0.04954783
2   grass-bush 0.1437525 0.06924584 4 0.03462292
3        trunk 0.1467167 0.02136480 3 0.01233497
4  trunk-crown 0.2626575 0.09968872 4 0.04984436
5 trunk-ground 0.3339650 0.06966130 4 0.03483065
6         twig 0.0848450 0.07107131 4 0.03553565
```

8.3 write.csv

Writing out objects is even simpler. To write out a .csv file:

```
> write.csv(out, "anolis.summary.csv", row.names=FALSE)
```

The argument "row.names=" is optional, but I like to put it in or else you get row names added to your spreadsheet as an extra column. Leave it as TRUE (the default) only if the names are meaningful and useful.

8.4 save

You can also save the objects as R data files (.Rdat or .rda), which are R's binary format. The objects are saved directly, so you can just slurp up the .Rdata file and you will have your objects back. This is handy if you want to continue your analysis with your objects later.

```
> save( anolis, anolis.mean, anolis.sd, anolis.N, file="anolis.out.Rdata")
```

The command to load these back in is:

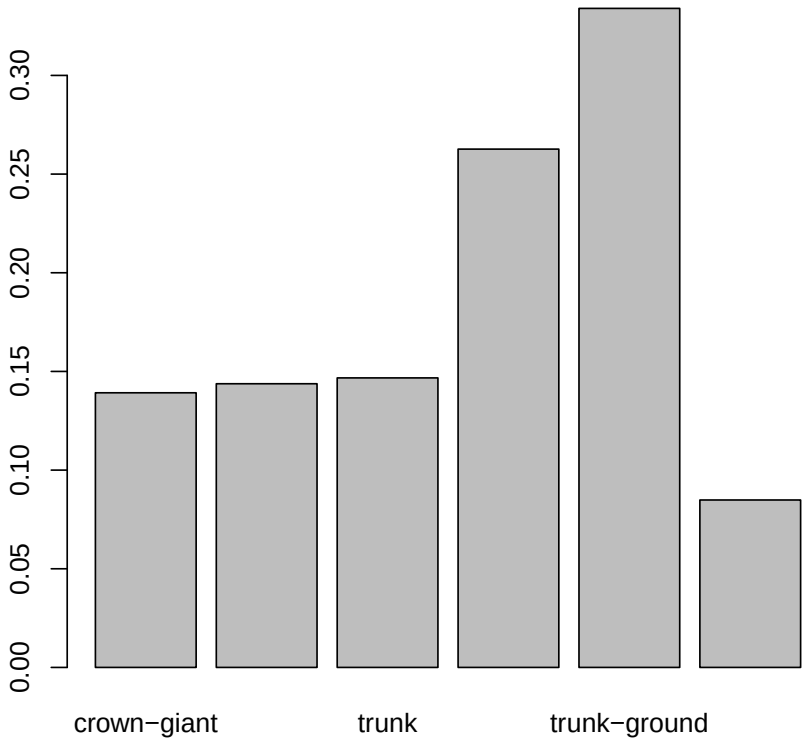
```
> load("anolis.out.Rdata")
```

Which will restore your objects.

8.5 Saving plots

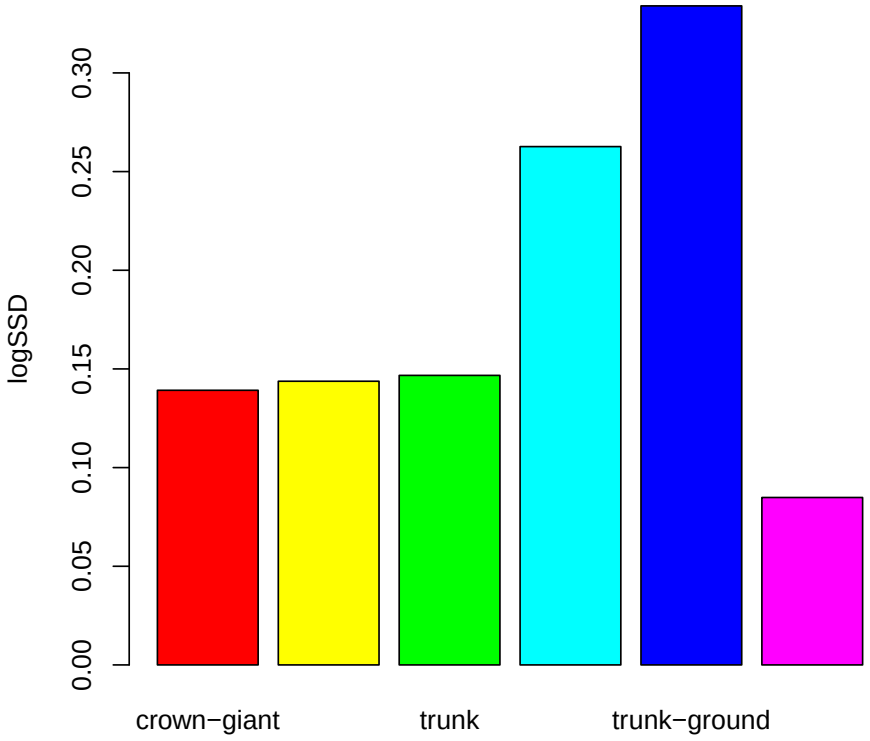
Let's make some plots to visualize SSD by ecomorph type. Recall that we can get box plots (median, quartiles, and range):

```
> barplot(out$mean, names.arg=out$ecomorph)
```



Let's add some color and a label for the y variable. Rainbow is a function which will generate a pallete of colors according to the number of colors you specify.

```
> barplot(out$mean, names.arg=out$ecomorph, col=rainbow(6), ylab="logSSD")
```

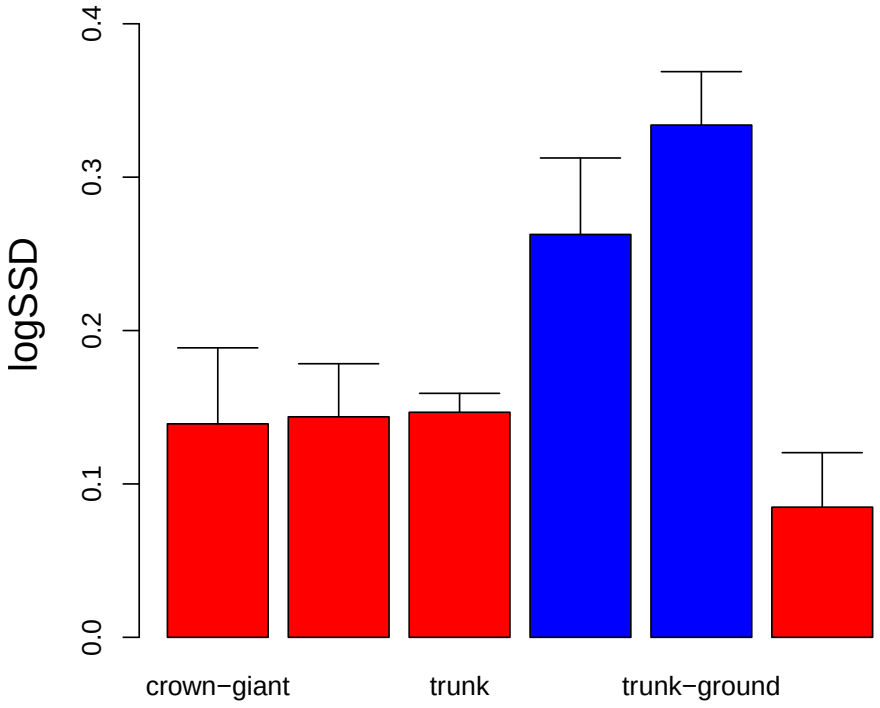


Alternatively, we may want to visually accentuate the "high" versus "low" dimorphism groups (for a talk for instance):

```
> bb <- barplot(out$mean, names.arg=out$ecomorph, col=c("red", "red", "red", "blue",
+  "blue", "red"), ylab="logSSD", cex.lab=1.5, ylim=c(0, max(out$mean)+.1))
> bb
```

```
     [,1]
[1,]  0.7
[2,]  1.9
[3,]  3.1
[4,]  4.3
[5,]  5.5
[6,]  6.7
```

```
> arrows(bb, out$mean, bb, out$mean+out$se, angle=90)
```



We've also made the y-axis label bigger using `cex.lab=1.5`, and finally added error bars by using the `arrows()` function. This function basically draws the error bars as line segments specified by the first four arguments. The `angle=90` tells the function to make the arrow heads flat, as in error bars. Read `?arrows` for more info. Finally, because the graph was not big enough to plot the highest error bar, I had to increase the y-limit using the `ylim` argument, which sizes the y-axis according to the lower and upper bounds given.

8.5.1 pdf

Now, if we are quite happy with our plot, we can save it as a pdf file. First we have to set the graphical devide to a pdf printer. Then plot the file, then turn the pdf device off (or it will keep writing to the same file every time you plot).

```
> pdf(file="anolisMeanSSD.pdf")  # turns on the pdf device for plotting
> barplot(out$mean, names.arg=out$ecomorph, col=c("red", "red", "red", "blue",
```

```
+ "blue", "red"), ylab="logSSD", cex.lab=1.5)
> dev.off()  # turns off pdf device for output


quartz
     2
```

8.6 Messier input files

The first example of a csv file was very easy to bring in to R. If it was hand-entered, you may have several issues including:

- extra delimiters in some rows (extra commas, etc.) so that some rows have extra columns
- extra header lines
- lots of missing values
- mixed character and numeric input

Any of these issues will cause problems because what you are reading in is a data frame. R expects columns to be of the same type, and the object is square, and etc.

Extra header lines are really easy to fix using the `skip=` option. However, the other issues will have to be fixed by editing your .csv file, or by writing code that reads in the lines one by one, makes the appropriate changes, and then writing out a "clean" .csv file. Which way to go should be determined by how much work it will be to hand-edit vs. program, which will depend a lot on how many problems the file contains, and whether they are unique or not. (Probably 80% or more of your R programming efforts are aimed at getting your input data into shape for analysis – which is why we cover these in the next section).

8.6.1 Input files generated by data loggers

An easier case to handle: files that are generated by computer. Take, for example, the file format generated from our hand-held Ocean Optics specroradiometer. It is very regular in structure, and we have tons of data files, so it is well worth the programming effort to code a script for automatic file input.

First, you can open the file below in a text editor. If you'd rather open it in R, you can use:

```
> readLines("Data/20070725_01forirr.txt")
```

Notice that there is a very large header, in fact the first 17 lines. Notice also that the last line will cause a problem. Also, the delimiter in this file is tab (backslash t).

```
> temp <- readLines("Data/20070725_01forirr.txt")
> head(temp)

[1] "SpectraSuite Data File"
[2] "++++++++++++++++++++++++++++++++++"
[3] "Date: Wed Jul 25 10:39:54 HST 2007"
[4] "User: guest"
[5] "Dark Spectrum Present: Yes"
[6] "Reference Spectrum Present: No"

> tail(temp)

[1] "888.38\t3.1306E-01"
[2] "888.54\t2.8153E-01"
[3] "888.71\t2.8245E-01"
[4] "888.87\t1.8988E-01"
[5] "889.04\t1.8988E-01"
[6] ">>>>>End Processed Spectral Data<<<<<"
```

We can solve these issues using the "skip" and the "comment.char" arguments of read.table to ignore both types of lines, reading in only the "good stuff". Also, the default delimiter in this function is the tab:

```
> dat <- read.table(file="Data/20070725_01forirr.txt", skip=17, comment.char=">")
> names(dat) <- c("lambda", "intensity")
> head(dat)

  lambda intensity
1 177.33         0
2 177.55         0
3 177.77         0
4 177.99         0
5 178.21         0
6 178.43         0

> tail(dat)
```

```
     lambda intensity
3643 888.21   0.29491
3644 888.38   0.31306
3645 888.54   0.28153
3646 888.71   0.28245
3647 888.87   0.18988
3648 889.04   0.18988
```

The file produces (useless) rows of data outside of the range of accuracy of the spectraradiometer. We can get rid of these by subsetting the data, selecting only the range 300-750nm:

```
> dat <- dat[dat$lambda >= 300, ]  # cut off rows below 300nm
> dat <- dat[dat$lambda <= 750, ]  #cut off rows above 750nm
```

Or do both at once:

```
> dat <- dat[dat$lambda >= 300 & dat$lambda <= 750,]
```

If we are going to be doing this subsetting over and over, we might want to save this as an index vector which tells us the position of the rows of data we want to keep in the dataframe (don't worry, we'll cover this again in the workhorse functions chapter).

```
> oo <- dat$lambda >= 300 & dat$lambda <= 750
> dat <- dat[oo, ]   # same as longer version above
```

We can now save the cleaned up version of the irradiance data:

```
> write.csv(dat, "20070725_01forirr.csv")
```

Chapter 9

The Workhorse Functions of Data Manipulation

Chapter Topics/Skills:

Indexing/Subsetting accessing particular elements of your data object

String Matching using grep, sub

Sorting ordering data

Matching using logical comparisons to index

Merging matching two data frames or matrices by a common column and merging into a new object

Reshaping R Objects changing the shape of matrices and dataframes, long-thin to short-fat formats

Attributes, Classes the characteristics of data objects and how to manipulate them

As a biologist, these data manipulation topics may seem dry, but they are really powerful and will allow you do to much more sophisticated analyses, and to do them with confidence. So it is well worth taking some time to learn how to use them well.

9.1 Indexing and subsetting

In general, accessing elements of vectors, matrices, or dataframes is achieved through *indexing* by:

inclusion a vector of positive integers indicating which elements of the vector to include

exclusion a vector of negative integers

logical values a vector of TRUE / FALSE values indicating which elements to include / exclude

by name a character vector of names of columns (only) or columns and rows

blank index take the entire column, row, or object

9.1.1 Vectors

The "index" of a vector is it's number in the order. Each and every element in any data object has at least one index (if vector, it's position along the vector, if a matrix or data frame, it's row and column number, etc.)

Let's create a vector:

```
> xx <- c(1, 5, 2, 3, 5)
> xx
```

```
[1] 1 5 2 3 5
```

Access specific values of `xx` by number:

```
> xx[1]
```

```
[1] 1
```

```
> xx[3]
```

```
[1] 2
```

You can use a function to generate an index. Get the last element (without knowing how many there are) by:

```
> xx[length(xx)]
```

```
[1] 5
```

Retrieve multiple elements of xx by using a vector as an argument:

```
> xx[c(1, 3, 4)]

[1] 1 2 3

> xx[1:3]

[1] 1 5 2

> xx[c(1, length(xx))]  # first and last

[1] 1 5
```

Exclude elements by using a negative index:

```
> xx

[1] 1 5 2 3 5

> xx[-1]  # exclude first

[1] 5 2 3 5

> xx[-2] # exclude second

[1] 1 2 3 5

> xx[-(1:3)] # exclude first through third

[1] 3 5

> xx[-c(2, 4)] # exclude second and fourth, etc.

[1] 1 2 5
```

Use a logical vector:

```
> xx[ c( T, F, T, F, T) ]  # T is the same as TRUE
```

```
[1] 1 2 5
```

```
> xx > 2
```

```
[1] FALSE  TRUE FALSE  TRUE  TRUE
```

```
> xx[ xx > 2 ]
```

```
[1] 5 3 5
```

```
> xx > 2 & xx < 5
```

```
[1] FALSE FALSE FALSE  TRUE FALSE
```

```
> xx[ xx>2 & xx<5]
```

```
[1] 3
```

Subsetting (picking particular observations out of an R object) is something that you will have to do all the time. It's worth the time to understand it clearly.

9.1.2 Matrices and Dataframes

Matrices and dataframes are both rectangular having two dimensions, and handled very similarly For indexing and subsetting. Let's work with a dataframe that is provided with the `geiger` package called `geospiza`. It is a list with a tree and a dataframe. The dataframe contains five morphological measurements for 13 species. First, let's clear the workspace (or clear and start a new R session):

If you have the package `geiger` installed, get the built-in dataset this way:

```
> rm(list=ls())
> require(geiger)
> data(geospiza)   # load the dataset into the workspace
> ls()               # list the objects in the workspace
```

```
[1] "geospiza"
```

Let's find out some basic information about this object:

```
> class(geospiza)

[1] "list"

> attributes(geospiza)

$names
[1] "geospiza.tree" "geospiza.data"
```

It is a list with two elements. Here we want the data

```
> geo <- geospiza$geospiza.data
> dim(geo)

[1] 13  5
```

You can also read it in as a .csv input file in the Data directory and proceed.

```
> geo <- read.csv("Data/geospiza_raw.csv")
> dim(geo)
```

It is a dataframe with 13 rows and 5 columns. If we want to know all the attributes of geo:

```
> attributes(geo)

$names
[1] "wingL"   "tarsusL" "culmenL" "beakD"   "gonysW"

$row.names
 [1] "magnirostris" "conirostris"  "difficilis"   "scandens"     "fortis"
 [6] "fuliginosa"   "pallida"      "fusca"        "parvulus"     "pauper"
[11] "Pinaroloxias" "Platyspiza"   "psittacula"

$class
[1] "data.frame"
```

We see that it has a ”names” attribute, which refers to column names in a dataframe. Typically, the columns of a dataframe are the variables in the dataset. It also has ”rownames” which contains the species names (so it does not have a separate column for species names).

Dataframes have two dimensions which we can use to index with: dataframe[row, column].

```
> geo     # the entire object, same as geo[] or geo[,]
> geo[c(1, 3), ]   # select the first and third rows, all columns
> geo[, 3:5]   # all rows, third through fifth columns
> geo[1, 5]  # first row, fifth column (a single number)
> geo[1:2, c(3, 1)]  # first and second row, third and first column (2x2 matrix)
> geo[-c(1:3, 10:13), ]  # everything but the first three and last three rows
> geo[ 1:3, 5:1]  # first three species, but variables in reverse order
```

To prove to ourselves that we can access matrices in the same way, let's coerce geo to be a matrix:

```
> geom <- as.matrix( geo )
> class(geom)

[1] "matrix"

> class(geo)

[1] "data.frame"

> geo[1,5]  # try a few more from the choices above to test

[1] 2.675983
```

Since `geo` and geom have row and column names, we can access by name (show that this works for `geom` too):

```
> geo["pauper", "wingL"]  # row pauper, column wingL

[1] 4.2325

> geo["pauper", ]  # row pauper, all columns

        wingL tarsusL culmenL  beakD gonysW
pauper 4.2325  3.0359   2.187 2.0734 1.9621
```

We can also use the names (or rownames) attribute if we are lazy. Suppose we wanted all the species which began with "pa". we could find which position they hold in the dataframe by looking at the rownames, saving them to a vector, and then indexing by them:

```
> sp <- rownames(geo)
> sp                              # a vector of the species names

 [1] "magnirostris" "conirostris"  "difficilis"   "scandens"     "fortis"
 [6] "fuliginosa"   "pallida"      "fusca"        "parvulus"     "pauper"
[11] "Pinaroloxias" "Platyspiza"   "psittacula"

> sp[c(7,8,10)]     # the ones we want are #7,8, and 10

[1] "pallida" "fusca"   "pauper"

> geo[ sp[c(7,8,10)], ]  # rows 7,8 and 10, same as geo[c(7, 8, 10)]

           wingL  tarsusL  culmenL    beakD   gonysW
pallida 4.265425 3.089450 2.430250 2.016350 1.949125
fusca   3.975393 2.936536 2.051843 1.191264 1.401186
pauper  4.232500 3.035900 2.187000 2.073400 1.962100
```

One difference between dataframes and matrices is that Indexing a data frame by a single vector (meaning, no comma separating) selects an entire column. This can be done by name or by number:

```
> geo[3]   # third column
> geo["culmenL"]  # same
> geo[c(3,5)]  # third and fifth column
> geo[c("culmenL", "gonysW")]  # same
```

Prove to yourself that selecting by a single index has a different behavior for matrices (and sometimes produces an error. Why? Because internally, a dataframe is actually a list of vectors. Thus a single name or number refers to the column, rather than a coordinate in a cartesian-coordinate-liek system. However, a matrix is actually a vector with breaks in it. So a single number refers to a position along the single vector. A single name could work, but only if the individual elements of the matrix have names (like naming the individual elements of a vector).

Another difference is that dataframes (and lists below) can be accessed by the `$` operator. It means indicates a column within a dataframe, so `dataframe$column`. This is another way to select by column:

```
> geo$culmenL

 [1] 2.724667 2.654400 2.277183 2.621789 2.407025 2.094971 2.430250 2.051843
 [9] 1.974420 2.187000 2.311100 2.331471 2.259640
```

An equivalent way to index is by using the `subset` function. Some people prefer it because you have explicit parameters for what to select and which variables to include. See help page `?subset`.

9.1.3 Lists

A list is like a vector, except that whereas a vector has the same type of data (numeric, character, factor) in each slot, a list can have different types in different slots. They are sort of like expandable containers, flexibly accommodating any group of objects that the user wants to keep together.

They are accessed by numeric index or by name (if they are named), but they are accessed by double square brackets. Also, you can't access multiple elements of lists by using vectors of indices:

```
> mylist <- list( vec = 2*1:10, mat = matrix(1:10, nrow=2), cvec = c("frogs", "birds")
> mylist

$vec
 [1]  2  4  6  8 10 12 14 16 18 20

$mat
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

$cvec
[1] "frogs" "birds"

> mylist[[2]]

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

> mylist[["vec"]]

 [1]  2  4  6  8 10 12 14 16 18 20

> # mylist[[1:3]]  # gives an error if you uncomment it
> mylist$cvec

[1] "frogs" "birds"
```

9.2 String Matching

A more useful feature is string matching. R has grep facilities, which can do partial matching of character strings. For example, we could directly search for species (the object or "x") names which contain "p" (the pattern):

```
> sp <- rownames(geo)
> grep(pattern = "p", x = sp)  # returns indices

[1]  7  9 10 12 13

> grep("p", sp, value=T)  # returns the species names which match

[1] "pallida"    "parvulus"   "pauper"     "Platyspiza" "psittacula"

> grep("p", sp, ignore.case=T, value=T)   # case-sensitive by default

[1] "pallida"      "parvulus"     "pauper"       "Pinaroloxias" "Platyspiza"
[6] "psittacula"

> grep("^P", sp, value=T)  # only those which start with (^) capital P

[1] "Pinaroloxias" "Platyspiza"
```

It is possible to use perl-type regular expressions, and the sub function is also available. Sub is related to grep, but substitutes a replacement value to the matched pattern. Notice that there are two species which have upper case letters. We can fix this with:

```
> sp <- rownames(geo)
> sub(pattern = "^P", replacement = "p", sp)

 [1] "magnirostris" "conirostris"  "difficilis"   "scandens"     "fortis"
 [6] "fuliginosa"   "pallida"      "fusca"        "parvulus"     "pauper"
[11] "pinaroloxias" "platyspiza"   "psittacula"

> rownames(geo) <- sub(pattern = "^P", replacement = "p", sp)   # to save changes
```

9.3 Ordering Data

Suppose we now want geo in alphabetical order. We can use the `sort` function to sort the rownames vector, then use it to index the dataframe:

```
> sort(rownames(geo))
> geo[ sort(rownames(geo)), ]
```

A better option for dataframes, though, is `order`:

```
> order(rownames(geo))   # the order that the species should take to be

 [1]  2  3  5  6  8  1  7  9 10 11 12 13  4

>                   #  sorted from a-z
> rbind(rownames(geo), order(rownames(geo)))  # to illustrate

     [,1]           [,2]          [,3]         [,4]       [,5]     [,6]
[1,] "magnirostris" "conirostris" "difficilis" "scandens" "fortis" "fuliginosa"
[2,] "2"            "3"           "5"          "6"        "8"      "1"
     [,7]      [,8]    [,9]       [,10]    [,11]          [,12]
[1,] "pallida" "fusca" "parvulus" "pauper" "pinaroloxias" "platyspiza"
[2,] "7"       "9"     "10"       "11"     "12"           "13"
     [,13]
[1,] "psittacula"
[2,] "4"

> oo <- order(rownames(geo))
> geo[oo,]   # sorted in alpha order

               wingL  tarsusL  culmenL    beakD   gonysW
conirostris  4.349867 2.984200 2.654400 2.513800 2.360167
difficilis   4.224067 2.898917 2.277183 2.011100 1.929983
fortis       4.244008 2.894717 2.407025 2.362658 2.221867
fuliginosa   4.132957 2.806514 2.094971 1.941157 1.845379
fusca        3.975393 2.936536 2.051843 1.191264 1.401186
magnirostris 4.404200 3.038950 2.724667 2.823767 2.675983
pallida      4.265425 3.089450 2.430250 2.016350 1.949125
parvulus     4.131600 2.973060 1.974420 1.873540 1.813340
pauper       4.232500 3.035900 2.187000 2.073400 1.962100
pinaroloxias 4.188600 2.980200 2.311100 1.547500 1.630100
```

```
platyspiza   4.419686 3.270543 2.331471 2.347471 2.282443
psittacula   4.235020 3.049120 2.259640 2.230040 2.073940
scandens     4.261222 2.929033 2.621789 2.144700 2.036944
```

Order can sort on multiple arguments, which means that you can use other columns to break ties. Let's trim the species names to the first letter using the substring function, then sort using the first letter of the species name and breaking ties by tarsusL:

```
> sp <- substring(rownames(geo), first=1, last=1)
> oo <- order(sp , geo$tarsusL) # order by first letter species, then tarsusL
> geot <- geo[oo,]["tarsusL"]   # ordered geo dataframe, take only the wingL column
> geo <- geo[oo,]
```

Note: using `geo["tarsusL"]` as a second index for order doesn't work, because it is a one column dataframe, as opposed to geo$tarsus which is a vector. It must match sp, which is a vector. Check the `dim` and `length` of each. vectors have length only, dataframes have dimension 2.

9.4 Matching

Matching is very easy in R, and is often used to create a logical vector to subset objects. Greater than and less than are as usual, but logical equal is "==" to differentiate from the assignment operator. Also >= and <=.

```
> geot > 3    # a logical index
```

```
             tarsusL
conirostris    FALSE
difficilis     FALSE
fuliginosa     FALSE
fortis         FALSE
fusca          FALSE
magnirostris    TRUE
parvulus       FALSE
pinaroloxias   FALSE
pauper          TRUE
psittacula      TRUE
pallida         TRUE
platyspiza      TRUE
scandens       FALSE
```

```
> geot == 3  # must match exactly 3, none do
```

```
             tarsusL
conirostris    FALSE
difficilis     FALSE
fuliginosa     FALSE
fortis         FALSE
fusca          FALSE
magnirostris   FALSE
parvulus       FALSE
pinaroloxias   FALSE
pauper         FALSE
psittacula     FALSE
pallida        FALSE
platyspiza     FALSE
scandens       FALSE
```

```
> geot[ geot > 3 ]   # use to get observations which have tarsus > 3
```

```
[1] 3.038950 3.035900 3.049120 3.089450 3.270543
```

```
> #  ii <- geot > 3    # these two lines of code accomplish the same
> #  geot[ii]
> cbind(geo["tarsusL"], geot > 3)  # check
```

```
              tarsusL tarsusL
conirostris  2.984200   FALSE
difficilis   2.898917   FALSE
fuliginosa   2.806514   FALSE
fortis       2.894717   FALSE
fusca        2.936536   FALSE
magnirostris 3.038950    TRUE
parvulus     2.973060   FALSE
pinaroloxias 2.980200   FALSE
pauper       3.035900    TRUE
psittacula   3.049120    TRUE
pallida      3.089450    TRUE
platyspiza   3.270543    TRUE
scandens     2.929033   FALSE
```

```
> geo[geot>3, ]["tarsusL"]  # what does this do?
```

```
              tarsusL
magnirostris 3.038950
pauper       3.035900
psittacula   3.049120
pallida      3.089450
platyspiza   3.270543
```

Matching and subsetting works really well for replacing values. Suppose we thought that every measurement that was less than 2.0 was actually a mistake. We can remove them from the data:

```
> geo [ geo<2 ] <- NA
```

Missing values compared to anything else will return a missing value (so NA == NA returns NA, which is usually not what you want). You must test it with `is.na` function. You can also test multiple conditions with and (&) and or (|)

```
> !is.na(geo$gonysW)
```

```
 [1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
[13]  TRUE
```

```
> geo[!is.na(geo$gonysW) & geo$wingL > 4, ]  # element by element "and"
```

```
               wingL  tarsusL  culmenL    beakD   gonysW
conirostris  4.349867 2.984200 2.654400 2.513800 2.360167
fortis       4.244008 2.894717 2.407025 2.362658 2.221867
magnirostris 4.404200 3.038950 2.724667 2.823767 2.675983
psittacula   4.235020 3.049120 2.259640 2.230040 2.073940
platyspiza   4.419686 3.270543 2.331471 2.347471 2.282443
scandens     4.261222 2.929033 2.621789 2.144700 2.036944
```

```
> geo[!is.na(geo$gonysW) | geo$wingL > 4, ]   # element by element "or"
```

```
               wingL  tarsusL  culmenL    beakD   gonysW
conirostris  4.349867 2.984200 2.654400 2.513800 2.360167
difficilis   4.224067 2.898917 2.277183 2.011100       NA
fuliginosa   4.132957 2.806514 2.094971       NA       NA
fortis       4.244008 2.894717 2.407025 2.362658 2.221867
magnirostris 4.404200 3.038950 2.724667 2.823767 2.675983
parvulus     4.131600 2.973060       NA       NA       NA
```

```
pinaroloxias 4.188600 2.980200 2.311100       NA       NA
pauper       4.232500 3.035900 2.187000 2.073400       NA
psittacula   4.235020 3.049120 2.259640 2.230040 2.073940
pallida      4.265425 3.089450 2.430250 2.016350       NA
platyspiza   4.419686 3.270543 2.331471 2.347471 2.282443
scandens     4.261222 2.929033 2.621789 2.144700 2.036944
```

```
> !is.na(geo$gonysW) && geo$wingL > 4   # vectorwise "and"
```

```
[1] TRUE
```

Matching works on strings also:

```
> geo[rownames(geo) == "pauper",]   # same as   geo["pauper", ]
> geo[rownames(geo) < "pauper",]
```

There are even better functions for strings, though. In the expression `A %in% B`, the `%in%` operator compares two vectors of strings, and tells us which elements of `A` are present in `B`.

```
> newsp <- c("clarkii", "pauper", "garmani")
> newsp[newsp  %in% rownames(geo)]     # which new species are in geo?
```

We can define the "without" operator:

```
> "%w/o%" <- function(x, y) x[!x %in% y]
> newsp  %w/o% rownames(geo)   # which new species are not in geo?
```

9.5 Merging

Merging is another powerful database function. The concept is simple. Given two objects with a common matching key, can we merge them together into one object? Usually, the matching key in comparative data is the species name.

A common task is to match a morphology dataset with an ecology dataset, or a tree file with a data file. Continuing our example, let's make an ecology field and add it to geot:

```
> geot$ecology <- LETTERS[1:nrow(geot)]   # A:M
```

Now, let's merge geo["tarsusL"] with the first five rows of geot:

```
>                     # only maches to both datasets are included
> merge(x=geo["tarsusL"], y=geot[1:5, ], by= "row.names")

    Row.names tarsusL.x tarsusL.y ecology
1 conirostris  2.984200  2.984200       A
2  difficilis  2.898917  2.898917       B
3      fortis  2.894717  2.894717       D
4  fuliginosa  2.806514  2.806514       C
5       fusca  2.936536  2.936536       E

>                     # all species in both datasets are included
> merge(x=geo["tarsusL"], y=geot[1:5,], by= "row.names", all=T)

      Row.names tarsusL.x tarsusL.y ecology
1   conirostris  2.984200  2.984200       A
2    difficilis  2.898917  2.898917       B
3        fortis  2.894717  2.894717       D
4    fuliginosa  2.806514  2.806514       C
5         fusca  2.936536  2.936536       E
6  magnirostris  3.038950        NA    <NA>
7       pallida  3.089450        NA    <NA>
8      parvulus  2.973060        NA    <NA>
9        pauper  3.035900        NA    <NA>
10 pinaroloxias  2.980200        NA    <NA>
11   platyspiza  3.270543        NA    <NA>
12   psittacula  3.049120        NA    <NA>
13     scandens  2.929033        NA    <NA>
```

The results of merge are sorted by default on the sort key. To turn it off:

```
> geo <- geo[rev(rownames(geo)), ]   # reverse the species order of geo
>                      # merge on geo first, then geot
> merge(x=geo["tarsusL"], y=geot[1:5, ], by= "row.names", sort=F)

    Row.names tarsusL.x tarsusL.y ecology
1       fusca  2.936536  2.936536       E
2      fortis  2.894717  2.894717       D
3  fuliginosa  2.806514  2.806514       C
4  difficilis  2.898917  2.898917       B
5 conirostris  2.984200  2.984200       A

>                      # geot first, then geo
> merge(x=geot[1:5,], y=geo["tarsusL"], by= "row.names", sort=F)
```

```
    Row.names tarsusL.x ecology tarsusL.y
1 conirostris  2.984200       A  2.984200
2  difficilis  2.898917       B  2.898917
3  fuliginosa  2.806514       C  2.806514
4      fortis  2.894717       D  2.894717
5       fusca  2.936536       E  2.936536
```

9.6 Reshaping R Objects

Internally, R objects are stored as one huge vector. The various shapes of objects are simply created by R knowing where to break the vector into rows and columns. So it is very easy to reshape matrices:

```
> vv <- 1:10  # a vector
> mm <- matrix( vv, nrow=2)  # a matrix
> mm

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

> dim(mm) <- NULL
> mm <- matrix( vv, nrow=2, byrow=T)  # a matrix, but cells are now filled by row
> mm

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10

> dim(mm) <- NULL
> mm  # vector is now n a different order because the collapse occurred by column

 [1]  1  6  2  7  3  8  4  9  5 10
```

Other means of "collapsing" dataframes are:

```
> unlist(geo)   # produces a vector from the dataframe
>             # the atomic type of a dataframe is a list
> unclass(geo)  # removes the class attribute, turning the dataframe into a
>             # series of vectors  plus any names attributes, same as setting
>             # class(geo) <- NULL
> c(geo)  # similar to unclass but without the attributes
```

Practice

1. Recall from the chapter on Data Objects that we were simulating data in different treatment groups, and wanting to visualize the groups. Now that we know how to index and subset, we can use the `points` function to add different colored points to the plot for different groups.

   (a) Now let's make some data which should differ. For the "low" treatment, simulate y and y1 as normally distributed data with mean = -2 and sd=.5, and "high" as mean=5, and sd=3. Remake the dataframe.

```
> species <- LETTERS[1:7]
> x <- c(2, 4, 8)
> y <- c(rnorm(7, mean=-2, sd=0.5), rnorm(7), rnorm(7, mean=5, sd=3))
> y1 <- c(rnorm(7, mean=-2, sd=0.5), rnorm(7), rnorm(7, mean=5, sd=3))
> y
 [1] -1.7324331 -1.6198601 -1.6943661 -1.5550680 -1.5659884 -2.6173233
 [7] -2.8677405  0.7436305  0.1378092 -0.5159335 -1.5854286 -0.1596901
[13]  1.6204093  1.8036682  5.4798608  2.8345419  5.9276164  4.8582151
[19]  3.1012095  8.3641030  0.1240776
> y1
 [1] -2.7783919 -2.0899730 -2.1505971 -2.0055296 -1.5302355 -2.0239404
 [7] -1.8948294  0.4669686 -0.3045481  0.0534605 -1.6229322 -1.6686932
[13]  0.2173391  0.4160709  3.1820160  6.2624768  5.0295437  5.3543262
[19]  1.9442462 11.2799850 -1.9427580
> dat <- data.frame(species, x, treatment=factor(rep(c("low", "med",
+  "high"), each=7), levels=c("low", "med", "high")), y, y1)
> dat
   species x treatment          y         y1
1        A 2       low -1.7324331 -2.7783919
2        B 4       low -1.6198601 -2.0899730
3        C 8       low -1.6943661 -2.1505971
4        D 2       low -1.5550680 -2.0055296
5        E 4       low -1.5659884 -1.5302355
6        F 8       low -2.6173233 -2.0239404
7        G 2       low -2.8677405 -1.8948294
8        A 4       med  0.7436305  0.4669686
9        B 8       med  0.1378092 -0.3045481
10       C 2       med -0.5159335  0.0534605
11       D 4       med -1.5854286 -1.6229322
12       E 8       med -0.1596901 -1.6686932
13       F 2       med  1.6204093  0.2173391
```

```
14       G 4       med  1.8036682  0.4160709
15       A 8      high  5.4798608  3.1820160
16       B 2      high  2.8345419  6.2624768
17       C 4      high  5.9276164  5.0295437
18       D 8      high  4.8582151  5.3543262
19       E 2      high  3.1012095  1.9442462
20       F 4      high  8.3641030 11.2799850
21       G 8      high  0.1240776 -1.9427580
```

(b) Let's differentially color the "high", "medium", and "low" points. First set up the plot window without any points by plotting y, y1 with the plot parameter `type="n"`. Then select only the "high" points by subsetting. You'll want to make an index vector to choose only the points you want. Then use the `points()` function (which has the same form as the `plot()` function, but only adds points to an existing plot. Choose three different colors for each treatment level and plot all the data. Is there any patterning in y, y1?

(c) Ooops! The data are actually supposed to be blocked by treatment (the first seven rows correspond to low, the second 7 correspond to med, etc.) Can you remake the dataframe keeping the y and y1 in the same position, but fixing the treatment?

(d) Make three plots: boxplot of treatment vs. y, treatment vs. y1, and three color scatterplot of y vs. y1 (treatments should be indicated by different colors).

2. Matrix reshaping and indexing

(a) Create a matrix with the values 1 through 20, filling four rows. Save it as "x". item What are the attributes of x?

(b) Change it to a matrix with 2 rows and 10 columns by changing its attribute. item Change x to a vector.

(c) Change x to a matrix with four rows, this time filling it by rows rather than by columns (you may want to check the help page).

(d) Coerce x to a vector again. Is it in the same order as the previous vector? What does this tell you about R's default behavior when flattening matrices to vector?

(e) Create the original x matrix again. Select only the 3rd row, 4th column. What is it?

(f) Select rows 3 and 4, columns 4 and 5. Print it to the console by using the `print(x)` function.

(g) Select the first and last rows, first and last columns. Print it.

3. Reading in Data and adding on

(a) Read in the external file `bimac.csv` in comma separated format. Save it as "`bimac`".

(b) This is a phylogenetic tree and data for the OUCH package. Without going into details for now, this method allows biologists to specify selective regimes on branches of the phylogeny, by specifying categories which correspond to alternative "niches". This is a body size evolution dataset, and "OU.LP" is a hypothesis with three size categories. We would like to make three additional hypotheses. Add additional columns to this dataframe: OU.1 which has values of "`global`" for all rows, OU.3 which is the same as OU.LP, except those rows with "`NA`" in the species names should get a value of "`medium`", and OU.4 which is again similar to OU.LP, except that those rows with "`NA`" in the species names get a value of "`anc`".

Chapter 10

Writing your own functions

We've learned how to write good scripts and debug at the console. You may have noticed that you sometimes have to do the same things over and over again. And you find yourself cutting and pasting bits of code and making minor changes to it. This is a situation where writing your own *function* is a big help.

Functions help in several ways. Once you perfect a bit of code, they help achieve these goals of good programming by writing code that is:

- Reusable and Generic
- Modular
- Easy to Maintain

10.1 Functions are wrappers for code that you want to reuse

Functions are just bits of code that you want to reuse. You can even build up your own function library in a script like `myfunctions.R` which you can source with every script you write. So in this way, you can save yourself a lot of trouble by designing and maintaining a "tight" function library.

A function is very easy to define. You need a name for your function, some arguments (the input), a valid R statement (i.e., some code to run), and output to return. You then put it together in this following format:

```
> my_function_name <- function( argument )  statement
```

The only things that are actually required are the name of your function, and the word `function` followed by parentheses. Arguments are optional (well so are the statements, but what would be the point of that?).

Here is a very simple function to calculate the square of a value:

```
> mysq <- function( x ) {   # function name is mysq
+         x*x      # the function will return the square of x
+ }
```

Once you run the function definition through the console, your function will be stored in RAM. Then you can then use your function in the normal way that you use functions:

```
> mysq(2)
```

```
[1] 4
```

You can have multiple lines of R code to run, and you can even have functions within functions.The output from the function (the return value) is the last value computed. It is often best practice to explicitly use the `return` function as in the code below.

```
> mysq <- function( x ) {
+         plot( x, x*x, ylab="Square of x")          # plot x and x*x on the y axis
+         return (x*x)           # return the square of x
+         }
> mysq(1:10)
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

10.2 Arguments

Generally speaking, arguments are included in functions because you might want to change them. Things that stay the same are usually hard-coded into the function. But what if you want to change it sometimes but not others?In our little example, what if you wanted to be able to change the label on the y-axis sometimes, but most of the time you wanted it to just say ”Square of x”?

```
> mysq <- function( x, yylab="Square of x" ) {   # default argument for the y-label
+
+         plot( x, x*x, ylab=yylab)
```

```
+         return (x*x)
+
+         }
> mysq(1:10, yylab="X times X")

 [1]   1   4   9  16  25  36  49  64  81 100
```

What happens if you just run the following:

```
> mysq(1:10)

 [1]   1   4   9  16  25  36  49  64  81 100
```

These arguments with default values are therefore *optional*. Because they will run just fine even if you don't put anything for them. Whenever I write functions, I try to make as many default arguments as I can so that I can run them with minimal brain power. When you look at it 6 months later, you don't want to have to reconstruct *why you wrote it in the first place just to make it go*.

Another cute trick is that if you don't want to have anything as your default value, but you still want to have the option to change it, set the default to `NULL`.

```
> mysq <- function( x, yylab=NULL) {   # default arg is no value for the y-label,
+
+         plot( x, x*x, ylab=yylab)        # but you can specify it if you want to.
+         return (x*x)
+
+         }
> mysq(1:10, yylab="X times X")

 [1]   1   4   9  16  25  36  49  64  81 100
```

10.3 Order of arguments

You may have noticed that you can run a function with or without naming the arguments. For example:

```
> mysq(2)

[1] 4
```

```
> mysq(x=2)
```

```
[1] 4
```

work just the same.

The reason is that R will assume that if you don't name the arguments, they are in *the same order* as in the function definition. Therefore,

```
> mysq( c(1, 3, 5, 7), "Squares of prime numbers")
```

```
[1]  1  9 25 49
```

```
> mysq( yylab = "Squares of prime numbers", x=c(1, 3, 5, 7))
```

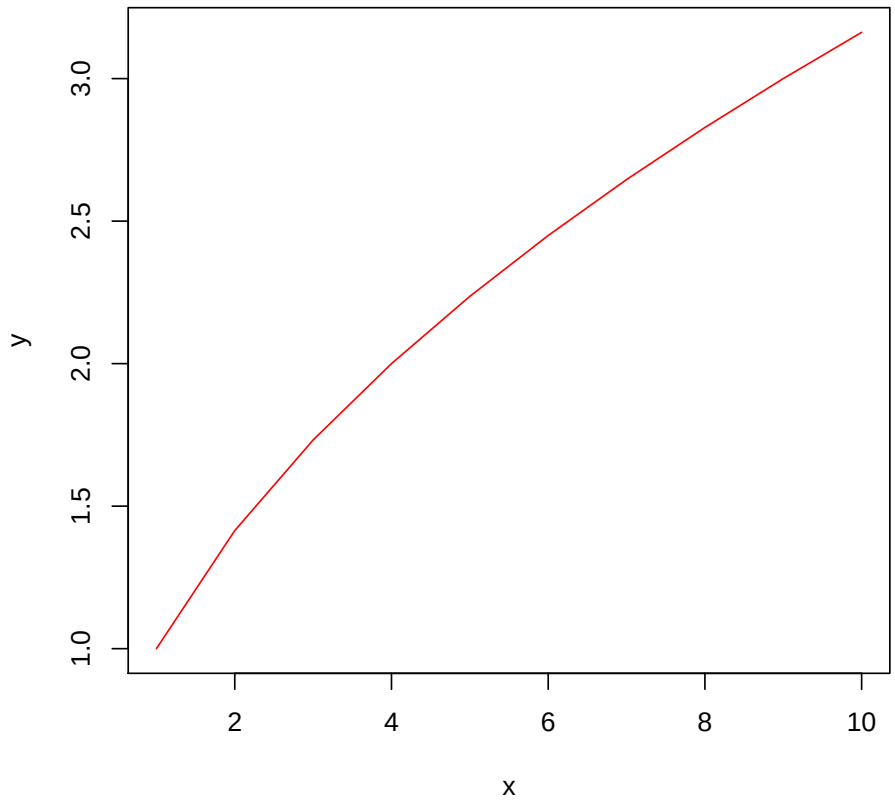
```
[1]  1  9 25 49
```

Are the same. Another way to put this, if you don't want to worry about the order that the arguments are defined in, *always use* the `names=`.

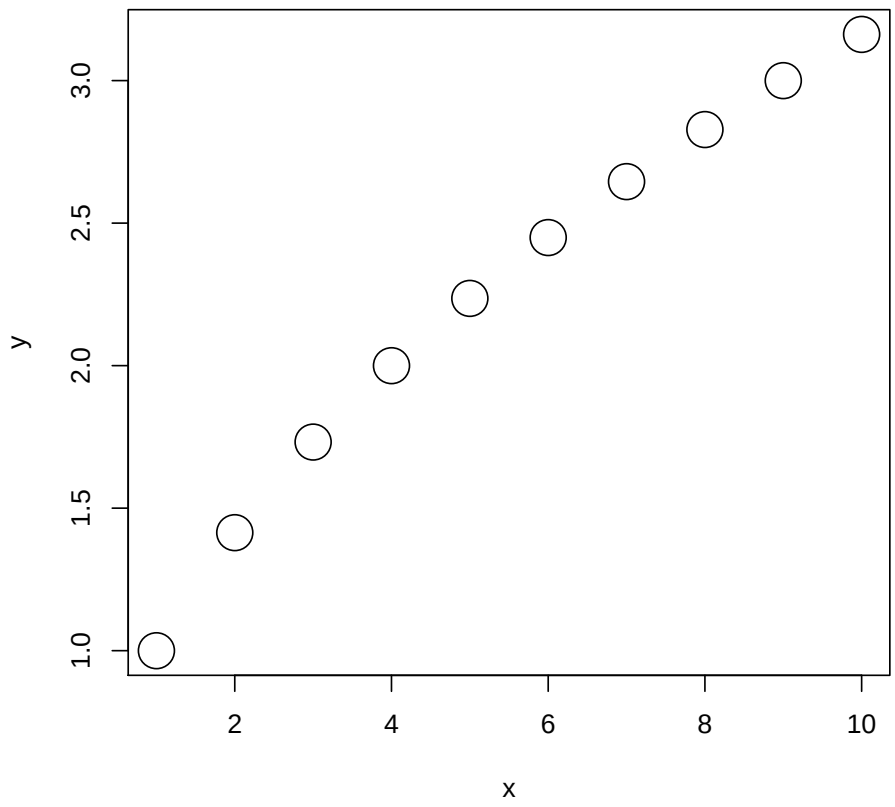
10.4 Arbitrary numbers of arguments

R is very flexible with its arguments. You can also have an arbitrary number of arguments by adding ... This is often used to pass additional arguments to `plot()`, which has many optional arguments, but it can be used for anything.

```
> myfun <- function(x, y,  ...) {
+         plot(x, y, ...)
+         }
> myfun( 1:10, sqrt(1:10), col="red", type="l")   # optional args color and line plot
```




```
> myfun( 1:10, sqrt(1:10), cex=3)   # optional arg for point size passed to plot()
```



Another common place where variable numbers of arguments comes up is in database queries, where you may want to run a search on a number of terms.

```
> query <- function( ... ) {
+         paste( ... )
+         }
> query( "cat", "dog", "rabbit")

[1] "cat dog rabbit"
```

Or any situation where you are just not sure how many inputs you will have. For example, you could have a list builder:

```
> addlist <- function( ... ) {
+         list( ... )
+         }
> metadat <- addlist ( dataset = "myeco", date="Jan 20, 2014")
> metadat
```

```
$dataset
[1] "myeco"

$date
[1] "Jan 20, 2014"
```

```
> dat <- addlist (ind=1:10, names=letters[1:10], eco=rnorm(10) )
> dat
```

```
$ind
 [1]  1  2  3  4  5  6  7  8  9 10

$names
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

$eco
 [1]  0.4781420 -0.6570550  2.0972989 -0.1178245 -0.8242408  1.8044553
 [7]  0.6412108 -1.4537969 -0.6516616 -2.0685793
```

These may seem like a silly example (and it is), but it is kept simple so you can see what's going on. These default arguments are very useful for making your functions flexible so that they can be more generic and reusable for many purposes.

10.5 Return value

As you have seen in the examples, R will return the last value computed (by default) or whatever you specify in the `return()` function. One thing that is a little peculiar to R is that you can only return **one** and **only one** object. So what do you do if you have several pieces of information you want returned? Use a list:

```
> mysq <- function( x, yylab="Square of x" ) {   # default argument for the y-label
+
+         plot( x, x*x, ylab=yylab)
+         output <- list( input=x, output=x*x )
+         return (output)
+
+         }
```

This is in fact what many model-fitting packages do. They return a list with the inputs, any fitted parameters, and model fit statistics, as well as characteristics of the objects such as names and factor levels, etc.

10.6 Looking inside R: functions that are inside packages

If you want to look at how any particular function in R is written – you can! R is open-source. You just type the name of the function with no parentheses:

```
> summary
```

```
function (object, ...)
UseMethod("summary")
<bytecode: 0x101fc6f08>
<environment: namespace:base>
```

You can see that it is a generic function, and that it uses different `methods` depending on the `class` of the object.

Here are all the methods that are defined for `summary`:

```
> methods('summary')
```

```
 [1] summary.aov             summary.aovlist         summary.aspell*
 [4] summary.connection      summary.data.frame      summary.Date
 [7] summary.default         summary.ecdf*           summary.factor
[10] summary.glm             summary.infl            summary.lm
[13] summary.loess*          summary.manova          summary.matrix
[16] summary.mlm             summary.nls*            summary.packageStatus*
[19] summary.PDF_Dictionary* summary.PDF_Stream*     summary.POSIXct
[22] summary.POSIXlt         summary.ppr*            summary.prcomp*
[25] summary.princomp*       summary.proc_time       summary.srcfile
[28] summary.srcref          summary.stepfun         summary.stl*
[31] summary.table           summary.tukeysmooth*

   Non-visible functions are asterisked
```

Here's how we find out what's inside `summary.factor` for example:

```
> summary.factor
```

```
function (object, maxsum = 100, ...)
{
    nas <- is.na(object)
```

```
    ll <- levels(object)
    if (any(nas))
        maxsum <- maxsum - 1
    tbl <- table(object)
    tt <- c(tbl)
    names(tt) <- dimnames(tbl)[[1L]]
    if (length(ll) > maxsum) {
        drop <- maxsum:length(ll)
        o <- sort.list(tt, decreasing = TRUE)
        tt <- c(tt[o[-drop]], `(Other)` = sum(tt[o[drop]]))
    }
    if (any(nas))
        c(tt, `NA's` = sum(nas))
    else tt
}
<bytecode: 0x10e80c238>
<environment: namespace:base>
```

Note: **Internal** functions are "hidden" inside the namespace of a package – the programmer has chosen to not make it available to the global environment. To find these, use `getAnywhere('functionname')` ha!

10.7 Scope

It is important to know that when you write a function, everything that happens inside the function is local in scope. It's like a big family secret – everything that is said in the family stays in the family. If you try to go talking about it to the outside world, no one will know what you are talking about. For example, suppose you wrote a function with some internal variables like this:

```
> myfunc <- function( fattony, littlejimmy) {
+
+         cannolis <- fattony*2 + littlejimmy
+         return(cannolis)
+ }
> myfunc( 5, 4 )

[1] 14
```

If we try type the following on the command line, we will get an error `'... object 'cannolis' not found'`.

```
> cannolis
```

Even though you ran the function, you can't *ask R how many cannolis you need* because *what's created in the function stays in the function.* When the function is over, poof! It's gone. That's because the objects used within the function are *local in scope* and not available to the global environment.

Of course, global variables are available to use inside of functions, just as family members are aware of what's going on in the outside world. So for example, it is perfectly valid to use `pi` or anything you've defined previously in the global environment inside a function:

```
> myfunc <- function( fattony, littlejimmy) {
+
+         cannolis <- fattony*pi + littlejimmy+littlebit
+         return( round(cannolis) )
+ }
> littlebit <- 1
> myfunc( 5, 4 )

[1] 21
```

The code above worked because `littlebit` was defined prior to running our function. But you can see that it's often a good idea to actually pass into a function anything that is needed to make it go.

So you may be wondering why it works this way? Well in general, in most advanced programming languages, the objects within functions are local in scope. This is to make it easier to program. If there is a clean separation between what goes on inside a function and what is outside of it, then you can write functions without worrying about every possibility regarding what could happen. You only have to worry about what is happening inside your little function. That's what helps to make it modular and extensible – so your functions can play nice with other codes.

10.8 Search Paths and Environment

Remember what we were saying about functions in R are objects? So if we look at our workspace, our functions should be there:

```
> ls()

 [1] "aa"              "addcols"         "addlist"         "dat"
 [5] "littlebit"       "ll"              "metadat"         "my_function_name"
```

```
 [9] "myfun"            "myfunc"           "mymean"           "myse"
[13] "mysq"             "ncentral"         "neast"            "oo"
[17] "query"            "readiir"          "readirr"          "regions"
[21] "regs"             "south"            "states"           "west"
```

And sure enough they are! As well as all of our data frames, lists, and other objects that we created. Now I should note that it is possible to write a function in R **with the same name** as a built-in function in R. For example, if for some crazy reason, we wanted to redefine the `mean` function, we can!

```
> mean <- function(...) {
+         return ("dirty harry")
+         }
> mean( 1:10 )
```

```
[1] "dirty harry"
```

What happened? Well we wrote our own function for `mean`. Why is R only returning our new function, an not the built-in one? Well, any object that we create (including our own functions) are in the **Global Environment**. Whereas functions in packages are in their further down the search path. R knows where things are by the order that they are attached. The global environment is first (containing any user-created objects), followed by attached packages:

```
> search()
```

```
 [1] ".GlobalEnv"        "tools:RGUI"        "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

The function `mean()` is in the `base` package, which is all the way at the end. So when we type `mean()` R will first look to see if there is any function by that name in our global environment, then in any of the other attached packages before finally finding it in `base`. Needless to say, it's very confusing (and potentially dangerous!) to name objects by the same name as R key words or built-in functions. Don't do it!

If you need to get rid of the custom build mean function, just type `rm(mean)` at the console.

```
> rm(mean)
> mean(1:10)
```

```
[1] 5.5
```

Whew! Or just shut down and restart R. It's a clean slate after that! (Don't worry, you can't break R ;).

10.9 Exercises

1. Write your own function for calculating a mean of a vector, using only the `sum()` and the `length()` functions. The input should be a vector, and the output is the mean.

2. Write your own function for calculating the standard error. You can use the `sd()`, `sqrt()`, and the `length()` functions. The input should be a vector of values.

3. Go back to Chapter 7 section 7.6.1. Write a function that will read in the irradiance data, trim it to wavelengths between 300 and 750 nm, and plot the data. Then use that function to read in files for the different directions: `up`, `for` (forward), `left`, and `right`: `20070725_01upirr.txt, 20070725_01forirr.txt, 20070725_01leftirr.txt, 20070725_01rightirr.txt`. Your function should take as input just the file name. Write a script that defines the function and then calls the function four times, once for each file.

4. Now take the function you just made, and add optional arguments for the cut off values 300 and 750. You may want to trim the data to different values. Try trimming it to different values and see what happens using your new function.

Chapter 11

All About Data

Goals:

- Handling raw data and preprocessing to R data files.
- Programming:

  Identify what do you want to do? (you can't do it if you can't articulate it exactly)

  Am I sure the raw data is free of error? (check data entry)

  Do I need to reshape my data? (check data processing)

  Am I sure that all of the programming steps are doing what I want them to do and free of error?

  Do the results make sense? Do I trust it?

Concepts:

- Reading in external files
- Saving r data files
- Flowcharting

11.1 Raw data to "curated" data

An important part of data analysis is checking for accuracy in your data transfer from your field notebook to the raw data files, to your processed data files. Here are some best practices:

- Set up a "data" and an "Rdata" directory for your raw data files and processed data files, respectively.

- Raw data is data that is the most original source that is on the computer. Whether it is manually entered it, or obtained from the literature. It is usually a text file or a spreadsheet. The most convenient format for R is ".csv" or comma separated format, but R can handle any delimiter or fixed-width file. Excel files should be saved as .csv if at all possible (it is possible to read in Excel, but it is a pain).

- Data entry is inherently error-prone. Keeping this in mind, take some steps to make it as easy as possible to enter accurately. Organize your field notebook in rows and columns, then type the data into your file exactly in the same order as in your notebook.

- It is important to check raw data for accurate data entry. Print out your text file, and check against your notebook line by line and number by number.

- Raw data should ONLY be corrected for typos on data entry. Numbers should not be altered because they are "outliers". Filling in missing data is not recommended.

- Protect the sanctity of your raw data. Keep your raw data files pristine and unaltered. Any changes, fixes, or cleaning up that you should be done in the analysis (i.e., via your R script), and not to your raw data file itself. If you permanently change your raw data file, and later have questions, you will never be able to reconstruct what you did. If it is just too difficult to fix the data with code, at least keep a copy of your original data file, and save a version as "edited" and keep notes on what was modified.

- Save your cleaned up, reogranized data as an R data file (.rda or .Rdata) in your Rdata folder.

- Save a script of all your data processing and analysis. If you send someone this script and the raw data file, they should be able to run your code.

- As with all programming projects, plan your steps from input (raw data and what shape is it in) to output ("curated data" and what shape do you want for our analyses).

R has great database (merging and matching), string manipulation, sorting, and data reshaping facilities. We'll illustrate some of these in a typical data "curation" step using a morphological dataset of *Anolis* lizards. We will start with the raw data, entered into the computer from a field notebook and save the "curated" product as an R data file.

11.1.1 Reading in fixed width format

Although `read.csv` is much easier, the data was intended for use with SAS and saved as a fixed-width format. R has a fixed-width reading function called `read.fwf` which requires as arguments the filename and a vector of widths. Note that if you have blank columns that you don't want read in, indicate the width of these with negative integers. For example, we are reading in the first 13 characters as the first variable, character 14 as the second variable, skipping character 15, etc.. So let's read in the files, assign names to the columns, and then check that the complete file was read in using the `head()` and `tail()` functions (which are especially useful for large datasets):

```
> read.fwf('data/94morphja.dat', widths=c(13, 1, -1, 5, 5, 5, -1, 1, 5, 5, 5,
+  3, -5, -5, -1, 1), as.is=T, strip.white=T) ->datja
> read.fwf('data/94morphpr.dat', widths=c(13, 1, -1, 5, 5, 5, -1, 1, 5, 5, 5,
+  3, -5, -5, -1, 1), as.is=T, strip.white=T) ->datpr
> names(datpr) <- names(datja) <- c('species', 'sex', 'svl', 'mass', 'tail', 'regen',
+ 'forel', 'hindl', 'headl', 'lamn', 'food')
> head(datja)
```

```
     species sex   svl mass  tail regen forel hindl headl lamn food
1    garmani   m 103.0 23.5 151.0     r  41.0  71.0     .   33    n
2     sagrei   m  54.0  3.8  93.5     r  23.0  39.0  13.5   18    n
3 lineatopus   m  62.5    .  82.0     r  28.0  47.0  19.0   20    y
4     sagrei   m  51.0  3.9 104.0     .  22.0  36.0  14.0   17    y
5     sagrei   m  50.0  3.2  65.0     r  21.5  37.0  13.5   18    n
6     sagrei   m  45.5  3.1     .     .     .     .     .    .    .
```

```
> tail(datja)
```

```
       species sex svl mass  tail regen forel hindl headl lamn food
52     garmani   f  75 10.5 156.0     .  30.0  53.0  19.5   28    y
53 valencienni   m  61  3.6  77.0     .  21.0  30.5  16.5   24    .
54     grahami   f  46    .     .     .     .     .     .    .    y
55     grahami   m  69    .     .     .     .     .     .    .    y
56      sagrei   m  51    .     .     .     .     .     .    .    y
57      sagrei   m  48    .     .     .     .     .     .    .    n
```

```
> head(datpr)
```

```
     species sex  svl mass  tail regen forel hindl headl lamn food
1  stratulus   m 43.5  1.9  44.0     r  21.0  32.0  11.5   19    y
```

```
2  evermanni   m 65.5  5.7  81.0     r  31.0  49.0  18.0   28    n
3      krugi   j 49.0  1.2 105.0     .  16.0  32.0  10.5   18    .
4      krugi   m 48.0  2.6 126.0     .  21.0  39.0  14.0   17    .
5 pulchellus   m 42.0  1.6 110.0     .  17.5  31.5  12.3   18    .
6  stratulus   j 32.0  1.3  58.5     r  18.0  28.5  10.5   21    .
```

```
> tail(datja)
```

```
       species sex svl mass  tail regen forel hindl headl lamn food
52     garmani   f  75 10.5 156.0     .  30.0  53.0  19.5   28    y
53 valencienni   m  61  3.6  77.0     .  21.0  30.5  16.5   24    .
54     grahami   f  46    .     .     .     .     .     .    .    y
55     grahami   m  69    .     .     .     .     .     .    .    y
56      sagrei   m  51    .     .     .     .     .     .    .    y
57      sagrei   m  48    .     .     .     .     .     .    .    n
```

These files are not too big, so display the datasets and take a look to make sure that all the data are in the proper columns of the dataframe. The data were saved as separate files for each island, so we want to add in island to each dataset. If we like, we can add year, etc.

```
> datpr$island <- "Puerto Rico"
> datja$island <- "Jamaica"
```

11.1.2 Combining the data into one file

We want to have one merged file to work with. A very simple thing to do would be to simply "row-bind" the data frames using `rbind()`. However, a danger here is that if the columns of one file is not in the same order as the columns of another file, we will have the wrong columns stacked on top of one another. However, we can check that the columns are identical before doing a row bind operation.

```
> names(datja)
```

```
 [1] "species" "sex"     "svl"     "mass"    "tail"    "regen"
 [7] "forel"   "hindl"   "headl"   "lamn"    "food"    "island"
```

```
> names(datpr)
```

```
 [1] "species" "sex"     "svl"     "mass"    "tail"    "regen"
 [7] "forel"   "hindl"   "headl"   "lamn"    "food"    "island"
```

We can see that they look good. If we had a huge number of columns, though, trusting your eye is sometimes risky. You can check that the names vectors match element-by-element by doing this:

```
> names(datja) == names(datpr)
```

```
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Or even adding up all the cases where the elements DO NOT match. Logical values in arithmetic operations have value `TRUE=1`, `FALSE=0`.

```
> sum(names(datja) != names(datpr))
```

```
[1] 0
```

OK, now we're sure, so let's combine the datasets:

```
> rbind(datja, datpr) -> dat
```

11.1.3 Adding variables to the data

Caribbean *Anolis* lizards have evolved convergently into distinct microhabitat specialists termed "ecomorphs". So we would like to add in the ecomorph designation, which is missing from the data files. To do that, we can make use of some very nice subsetting and string matching functions.

The `%in%` operator is a matching function for vectors of character strings. It returns which of those character strings to the left match the list of character strings on the right.

First we need to create vectors for each of the ecomorphs, containing the species names which belong to the ecomorph:

```
> spp <- unique(dat$species)
> spp                          # list of species in our sample
```

```
 [1] "garmani"      "sagrei"       "lineatopus"   "grahami"
 [5] "valencienni"  "stratulus"    "evermanni"    "krugi"
 [9] "pulchellus"   "cristatellus" "gundlachi"    "occultus"
[13] "cuvieri"
```

```
> tgspp <- c("cristatellus", "gundlachi", "sagrei", "lineatopus")
> tcspp <- c("stratulus", "evermanni", "grahami")
> cgspp <- c("cuvieri", "garmani")
> gbspp <- c("krugi", "pulchellus")
> twspp <- c("occultus", "valencienni")
```

Make sure that all of the species have been assigned to one and only one of the ecomorph groups. First combine all of the ecomorph species groups, into a vector. Test for duplicates:

```
> ecospp <- c(tgspp, tcspp, cgspp, gbspp, twspp)
> length(ecospp)

[1] 13

> length(spp)

[1] 13

> ecospp == unique(ecospp)  # if any are duplicated or missing, will get some FALSE

 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Make a new function called "%w/o%" which tells us what is in string A that is NOT in string B (A without B). There should be no spp which are not in ecospp and vice versa.

```
> "%w/o%" <- function(x, y) x[!x %in% y]
> ecospp %w/o% spp

character(0)

> spp %w/o% ecospp

character(0)
```

Now we are sure that we assigned everything to one and only one group.

(If you want to see what happens when we make a mistake go back and change the "4" in the list for "tcspp" to a "3" and see what happens – don't forget to reverse it and rerun the code correctly though). Now we can create an index vector for where to fill the (newly created) "ecomorph" variable with the correct ecomorph by searching for those rows which belong to the correct species.

```
> tgi <- dat$species %in% tgspp
> tci <- dat$species %in% tcspp
> cgi <- dat$species %in% cgspp
> gbi <- dat$species %in% gbspp
> twi <- dat$species %in% twspp
```

Now use these ecomorph indicies to add a variable called "ecomorph" to the dataset.

```
> dat$ecomorph[twi] <- "twig"
```

The above code fills the variable "ecomorph" with "twig" if the species name is one of "valencienni" or "occultus". You can check by looking at the following (I've only printed the first 10 lines here to save paper). You should have a "TRUE" everywhere that the species is "valencienni" or "occultus", and FALSE for all others. And for each of these the ecomorph should be "twig".:

```
> cbind(twi, dat[, c('species', 'ecomorph')])
```

```
      twi      species ecomorph
151 FALSE cristatellus     <NA>
152 FALSE        krugi     <NA>
153 FALSE        krugi     <NA>
154 FALSE    stratulus     <NA>
155 FALSE        krugi     <NA>
156 FALSE cristatellus     <NA>
157 FALSE    stratulus     <NA>
```

Let's do the rest:

```
> dat$ecomorph[tgi] <- "trunk-ground"
> dat$ecomorph[tci] <- "trunk-crown"
> dat$ecomorph[cgi] <- "crown-giant"
> dat$ecomorph[gbi] <- "grass-bush"
```

Check that everything looks OK (make sure that there are no "NA"s in the ecomorph field, etc.

```
> head(dat)
```

```
     species sex  svl mass  tail regen forel hindl headl lamn food
1    garmani   m  103 23.5 151.0     r  41.0  71.0     .   33    n
2     sagrei   m   54  3.8  93.5     r  23.0  39.0  13.5   18    n
3 lineatopus   m 62.5    .  82.0     r  28.0  47.0  19.0   20    y
4     sagrei   m   51  3.9 104.0     .  22.0  36.0  14.0   17    y
5     sagrei   m   50  3.2  65.0     r  21.5  37.0  13.5   18    n
6     sagrei   m 45.5  3.1     .     .     .     .     .    .    .
   island     ecomorph
1 Jamaica  crown-giant
2 Jamaica trunk-ground
3 Jamaica trunk-ground
4 Jamaica trunk-ground
5 Jamaica trunk-ground
6 Jamaica trunk-ground
```

```
> head(dat$ecomorph)
```

```
[1] "crown-giant"  "trunk-ground" "trunk-ground" "trunk-ground"
[5] "trunk-ground" "trunk-ground"
```

11.1.4 Sort by species and sex

First we create and order index "o" using the function `order`. Then we reorder the data frame rows by "o" and resave:

```
> o <- order(dat$species, dat$sex)
> dat <- dat[o, ]
```

11.1.5 Editing data into R format

Notice that missing values in this dataset were encoded with a ".". This is not recognized by R, so we should replace that with "NA"s. In fact, we actually read in the whole dataframe as "character". The `apply` repeats the `mode` function across the columns of dat:

```
> apply(dat, 2, mode)
```

```
    species         sex         svl        mass        tail
"character" "character" "character" "character" "character"
      regen       forel       hindl       headl        lamn
```

```
"character" "character" "character" "character" "character"
       food      island    ecomorph
"character" "character" "character"
```

Now we need to replace the "." and convert the the appropriate columns to mode numeric.

```
> dat[dat=='.'] <- NA
```

Also, we want to use only adult males and females, and not juveniles, so exclude anything that is not sex "m" or "f":

```
> dat <- dat[dat$sex %in% c('m', 'f'), ]
```

The "regen" column is an indicator variable for regenerated tails, which are not a good indication of adult tail size. So we want to look at regen, and if it is "r", we want to remove those tail measurements, replacing them with NA:

```
> dat$tail[ dat$regen == "r" ] <- NA
```

Check that we did what we wanted to do (only printing a few rows on paper):

```
> dat[c("regen", "tail")]
```

```
    regen tail
72      r <NA>
96   <NA> 77.0
105     r <NA>
113  <NA> 84.0
114     r <NA>
115  <NA> 82.0
```

We want the numeric data to have mode numeric so let's use:

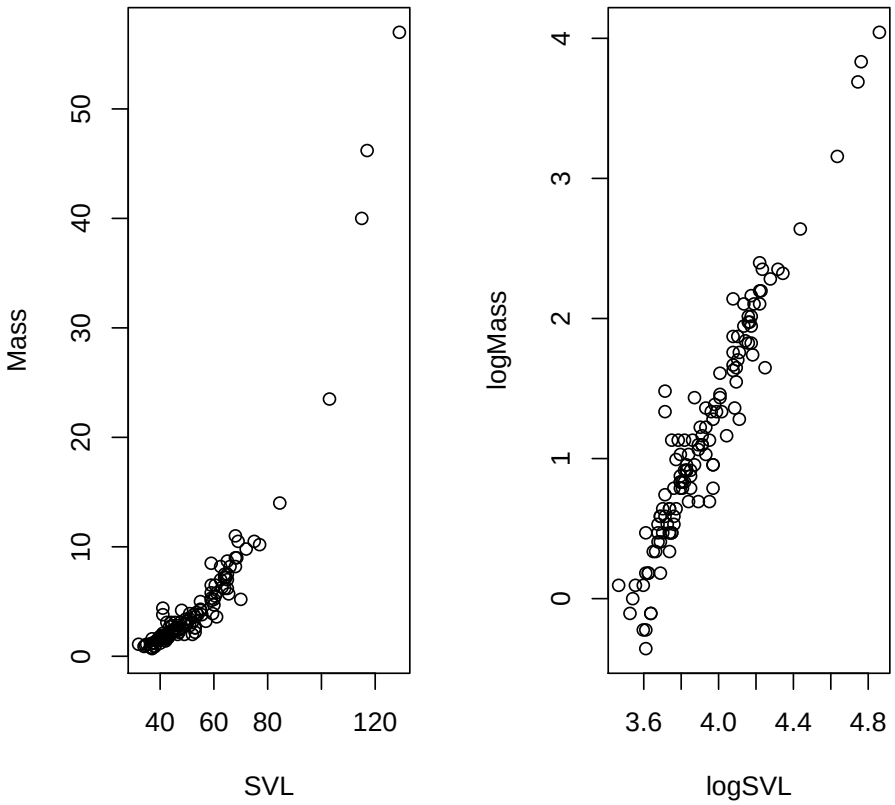
```
> dat.num <- dat[c('svl', 'mass', 'tail', 'forel', 'hindl', 'headl', 'lamn')]
> dat.num <- as.data.frame( apply(dat.num, 2, as.numeric) )
> apply(dat.num, 2, mode)
```

```
      svl      mass      tail     forel     hindl     headl      lamn
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
```

11.1.6 Getting statistics by species and sex

It is standard to use log-transformation in morphometrics. There are several reasons: (1) the data are usually not normally distributed (or, rather, the "errors" from any statistical model are not normally distributed), and log-transformation improves this. (2) Organisms tend to follow logistic growth curves, and species tend to follow non-allometric scaling patterns. Log transformation tends to linearize these non-linear trends, reducing the size of the very large and increase the size of the very small. (3) Some morphometric features scale multiplicatively with one another. For example, mass scales with the cube of length. Log-transformation makes these multiplicative relationships additive. Take a look at the plot of SVL versus MASS:

```
> op <- par(no.readonly = TRUE)
> par(mfrow=c(1,2))
> with(dat.num, plot(svl, mass, xlab="SVL", ylab="Mass"))
> with(dat.num, plot(log(svl), log(mass), xlab="logSVL", ylab="logMass"))
> par(op)
```



Let's log-transform the data, and then get the means, sample size, and standard deviations by species and sex. The `aggregate` function is an apply method, that is it operates on the matrix dat.num all at once, but it groups observations by some grouping variable first. In this case, species and sex.

```
> dat.mean <- aggregate( log(dat.num), list(species=dat$species, sex=dat$sex),
+ mean, na.rm=T)
> dat.sd <- aggregate( log(dat.num), list(species=dat$species, sex=dat$sex),
+ function(x) {if (all(is.na(x))) return(NA) else return(sd(x, na.rm=T))})
```

If we are interested in sexual dimorphism, then we will need to reshape our data. We want male and female observations to be matched by secies, so that we can take the ratio of male size to female size, for example.

```
> dd <- split(dat.mean, dat.mean$sex)
> fems <- dd[[1]]
> mals <- dd[[2]]
> sexes_sf <- merge(fems, mals, by="species", suffixes=c(".f", ".m"))
```

`sexes_sf` is a short-fat representation of the male and female data. We could then go through and subtract svl.m - svl.f, and so on, variable by variable. Alternatively, since R by default performs performs matrix operations element by element, and we know that the two matrices have exactly the same structure, we can simply subtract:

```
> sexdim <- mals[-(1:2)] - fems[-(1:2)]   # computes male larger dimorphism value
```

Limit to the five main variables.

Workarounds for broken code

- write work-around code and place it before every instance of the broken function
- `fix()`
- write a work-around function in the global environment by the same name

In R version 2.6.2, the `na.rm` option in `sd` used to return an `NA` if there were no observations in a particular group (since sd is not defined for zero observations). However, in 2.7.1, the behavior has changed and we get an error that halts execution. Go ahead and try:

```
> dat.sd <- aggregate(dat.num, list(species=dat$species, sex=dat$sex), sd, na.rm=T)
```

Now try:

```
> dat.sd <- aggregate(dat.num, list(species=dat$species, sex=dat$sex),
+ function(x) {if (all(is.na(x))) return(NA) else return(sd(x, na.rm=T))})
```

The function that I wrote above is a work-around. If all observations are missing, the function simply returns an NA without sending it to `sd`. This is an example of placing a fix right in front of the function call. Not very convenient if we need to use `sd` again. We could also use a handy function called `fix`, which allows us to edit code within a function, but it is only present for the current session, and disappears after.

```
> fix(sd)
```

When a source code window pops up, make the following edits, then close the window again:

Try the line of code should now work:

```
> dat.sd <- aggregate(dat.num, list(species=dat$species, sex=dat$sex), sd, na.rm=T)
```

Another option is to take the square root of the variance, and use a feature of the variance function to eliminate NA's:

```
> sd <- function (x, na.rm=FALSE)  sqrt(var(x,na.rm=na.rm,
+ use='pairwise.complete.obs'))
> dat.sd <- aggregate(dat.num, list(species=dat$species, sex=dat$sex),
+  sd, na.rm=T)
```

Whether you manually remove NA observations or use the square root and variance option, re-writing the `sd` function is probably the best option. It will reside in your global environment, so that will take precedence over the one in the `stats` package. And you can source this code (run it) without any manual inputs. `fix` is useful though, because it brings up a copy of the code you need to fix and allows you to make edits to it.

While `mean()` and `sd()` have options to remove NA's (although they may not work perfectly!), length does not. So we have to write a little function to do this manually before applying length.

```
> dat.N <- aggregate(dat.num, list(species=dat$species, sex=dat$sex),
+ function(x) { x<- x[!is.na(x)]; length(x)})
```

Or more compactly:

```
> dat.N <- aggregate(dat.num, list(species=dat$species, sex=dat$sex),
+  function(x) { length(x[!is.na(x)])})
```

Notice that with apply-like functions, which expect the name of a function to apply as a parameter, if you need to modify the input in any way or to do more than one step, you must write a little function.

Save all of our cleaned up data files in an external R dataset:

```
> save(dat, dat.mean, dat.sd, dat.N, file="Rdata/anolis_dat.rda")
```

Chapter 12

A Small Tour of Some Multivariate Methods in R

Note: You will need to install the package `candisc` to do the analyses at the end of the chapter.

Most of us have some multivariate data that we would like to explore. After we have gone through the task of making bivariate plots, checking for errors in the data, and finalizing the raw data, it is time to start looking for patterns and exploring.

The first step you will often think about is do you have to log-transform your data? Or do some other transformation? You may need to, for example, if you are doing morphometrics and have a lot of size variation. You may also expect your data to follow a power law, in which case a log-transformation will make the data linear. For example, things that scale with body size tend to have the form:

$$\begin{aligned} Y &= aMass^b \\ log(Y) &= log(a) + b \times log(Mass) \end{aligned}$$

Will you want to do an analysis of the data along with a size-corrected dataset? If shape variation is interesting for your data (i.e., do they differ in shape when we control for differences in size, or are they *relatively* larger or smaller?), then you may want to find some sort of size-adjustment. Popular methods include regressing against size, PCA analysis excluding PC1, shear or Procrustes methods, and ratios with size. There is a huge wealth of literature on size and how to analyze shape.

Now you are ready to begin. Often we have many variables measured and we suspect that many of the variables are collinear (or correlated) so that many of them contain the same or similar information. To summarize the variation, we may want to do a Principal Components Analysis.

12.1 Principal Components Analysis

PCA is an ordination method that is useful to explore patterns of variation in the data. When variables are correlated (or non-independent), PCA finds linear combinations of the original data that summarize *most* of the variation. It is therefore very useful for reducing the number of variables to a few most important axes of variation.

It produces a number of Principal Component axes (the same number as the number of original variables). The first PC axis is along the direction of greatest variation in the data. The second PC axis is orthogonal (perpendicular) to the first, and in the direction of the next greatest source of variation in the data. The third is orthogonal to the first and second, etc. and so on. Because all of the axes are orthogonal to one another, they summarize independent variation.

Some things to look for in PC analysis: The **loadings** of the variables on the PC axes show how much each variable is correlated with that PC axis. The magnitude of the loading indicates how strong the correlation is, and the sign indicates the direction. The sign of the loading is only informative if variables load with different signs on the same PC axis. For example if variable A and B load positively with PC 2, and variable C loads negatively, this is often interpreted as varying along PC2 in an increasing direction indicating larger A and B but smaller C. In a morphological analysis, the first PC axis often loads positively and nearly equally on all variables, and is therefore considered to indicate size. PC1 also typically explains a large fraction of the variation.

The amount of variation each PC axis explains is called the proportion of variance explained. It is usually expressed as a percent or a fraction. It is not uncommon in morphological analysis for PC1 to explain 90% of the variation in the data.

It is important to note, however, that the amount of variation does not necessarily indicate it's importance. Many ecological associations or functionally significant variation is reflected in shape variation, which as we said may be only 10% of the variation. However, this might be very functionally relevant. Size may vary a lot, but it might be whether or not you have very long legs relative to your size that tells us if you are a good runner. Long legs (in an absolute sense) may not make you a great runner if you are actually huge in size, so that relative to your body length, your legs are actually relatively short. So one thing to keep in mind is that you often will use only 3 axes, even though you have 10 or more variables. If you have managed to capture 90 or 95% of the variation with the first three variables (sometimes even more), you're probably in great shape. It's a tradeoff between keeping the analysis and interpretation manageable, and keeping all the variation in the data. Usually the minor axes have less than 1% of the variation, and are usually not interesting even if you were to keep them. Anyway, to conclude this paragraph, you may want to do a PC analysis on the data with size included, and then do a second analysis on the size-adjusted data (shape). Another strategy is to do a PC analysis on the data with size, and then leave out PC1 in downstream analyses of "shape".

Let's do a PC analysis on Fisher's Iris data, which is a famous multivariate dataset build into R.

```
> head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

There are four measurements of sepal and petal length and width, along with species for three species of iris. Fifty individuals are measured for each iris species.

PC analysis in R uses the `princomp` function. You can either specify the columns of the data to do a PC analysis, or use the formula representation (with no response variables). A tilde with a dot indicates all variables, and you can exclude columns with a minus sign as usual. You should only put continuous variables into a PC analysis. So in the iris data, we can exclude species with the minus sign:

```
> pc.iris <- princomp (~ .-Species, data=iris, scores=T)
```

We see the first two PC axes explain more than 97% of the variation, with PC1 explaining 92%, and PC2 5%. Because there are four original variables, we have two more PC axes but they only explain 1.7% and 0.5% of the variation and we will ignore them.

```
> summary(pc.iris)
```

```
Importance of components:
                          Comp.1     Comp.2     Comp.3      Comp.4
Standard deviation     2.0494032 0.49097143 0.27872586 0.153870700
Proportion of Variance 0.9246187 0.05306648 0.01710261 0.005212184
Cumulative Proportion  0.9246187 0.97768521 0.99478782 1.000000000
```

We can see how the PC axes reflect the original variables by using the loadings accessor function:

```
> loadings(pc.iris)
```

```
Loadings:
             Comp.1 Comp.2 Comp.3 Comp.4
Sepal.Length  0.361 -0.657 -0.582  0.315
Sepal.Width         -0.730  0.598 -0.320
Petal.Length  0.857  0.173        -0.480
Petal.Width   0.358         0.546  0.754

               Comp.1 Comp.2 Comp.3 Comp.4
SS loadings      1.00   1.00   1.00   1.00
Proportion Var   0.25   0.25   0.25   0.25
Cumulative Var   0.25   0.50   0.75   1.00
```

PC1 reflects variation primarily in petal length as evidenced by it's high correlation (0.857), and to a lesser extent by sepal length and petal width. PC2 indicates variation in sepal width, as well as additional variation in sepal length. Petal length loads in the opposite direction and thus would decrease as sepal length and width increased, but the correlation value is rather low (0.173). PC3 actually shows a trade-off between sepal length and both sepal width and petal width (with all loadings having similar magnitudes), but it explains very little of the overall variation.

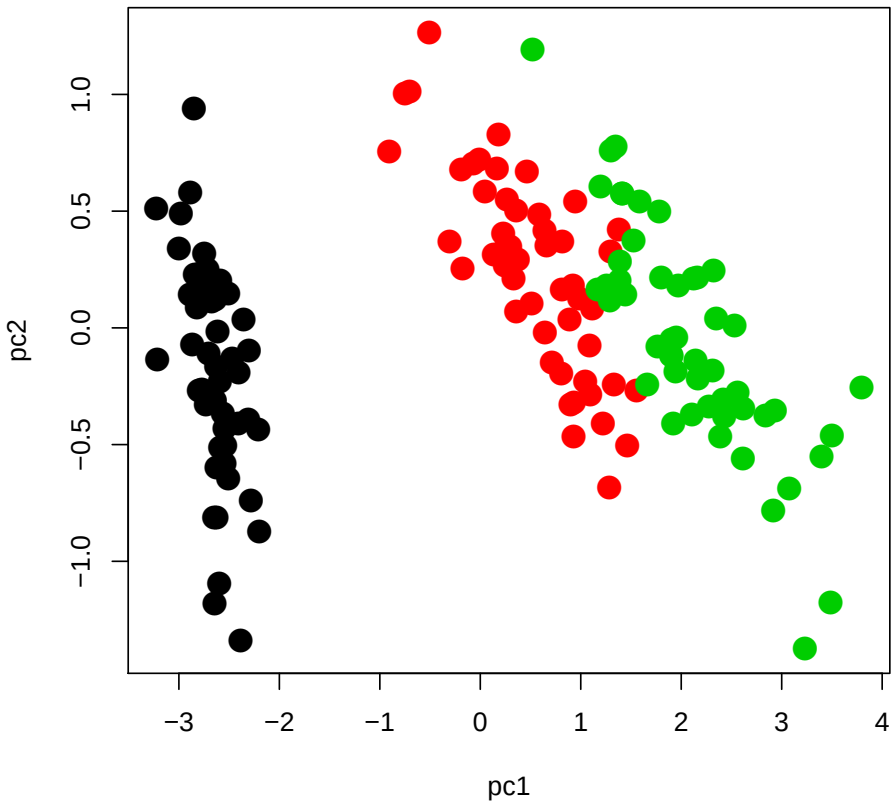
The scores are the values of each datapoint along the PC axes in PC space. It is an element of `pc.iris$scores`

```
> head(pc.iris$scores)
```

```
     Comp.1     Comp.2      Comp.3       Comp.4
1 -2.684126 -0.3193972 -0.02791483  0.002262437
2 -2.714142  0.1770012 -0.21046427  0.099026550
3 -2.888991  0.1449494  0.01790026  0.019968390
4 -2.745343  0.3182990  0.03155937 -0.075575817
5 -2.728717 -0.3267545  0.09007924 -0.061258593
6 -2.280860 -0.7413304  0.16867766 -0.024200858
```

We can access the PC scores, but it's long. So for convenience let's save it as something shorter.

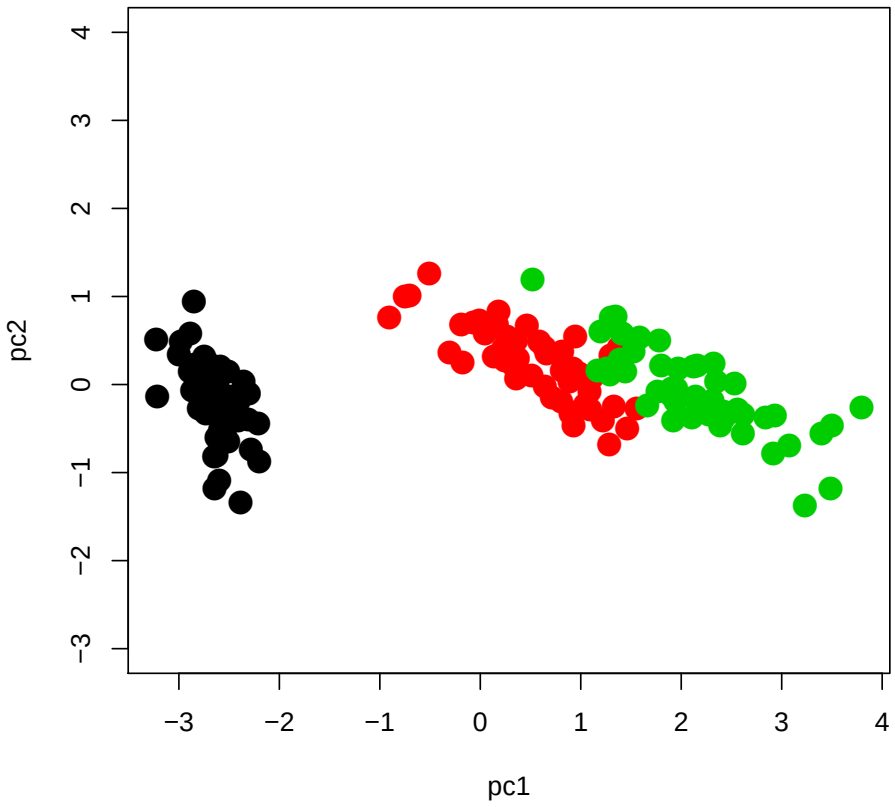
```
> pc1 <- pc.iris$scores[,1]
> pc2 <- pc.iris$scores[,2]
> pc3 <- pc.iris$scores[,3]
> plot(pc2 ~ pc1, col=iris$Species, cex=2, pch=16)
```



We can see that there are three distinct groups of points. PC1 seems to contribute the most to separating the species, whereas PC2 seems to reflect variation within species.

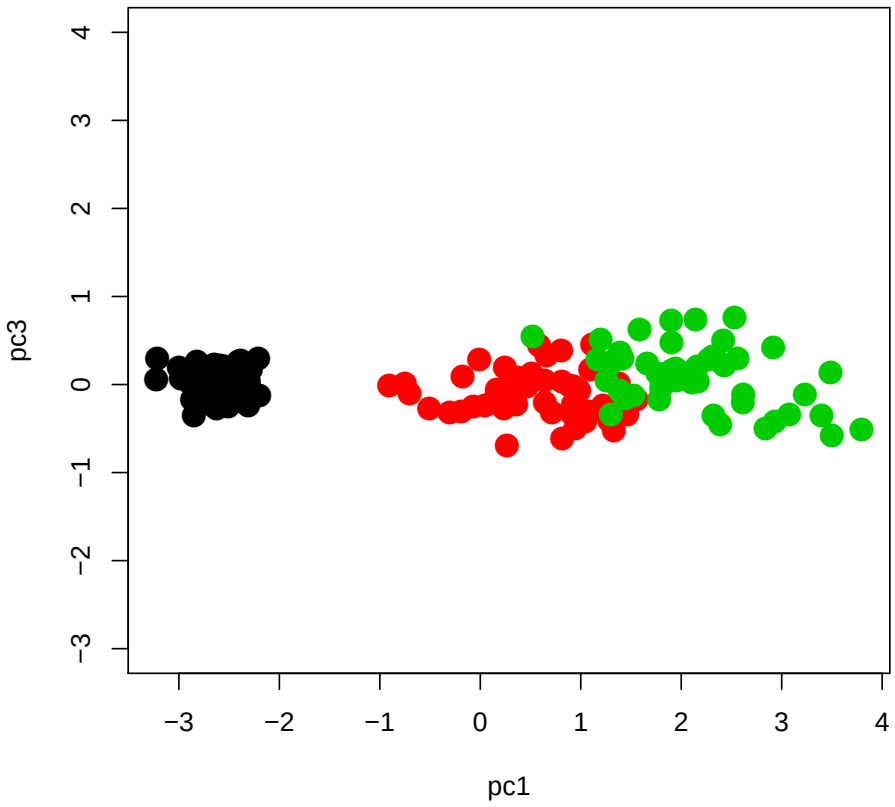
One important point to note is that the axes should be scaled to match each other. One unit on the X axis should occupy the same length of graph as one unit on the Y axis, otherwise the plot will be visually deceptive with regard to how much variation each axis has. Here is how we should scale the Y-axis in order to match the X-axis. Now you can see what the meaning of 97% of the variation being along PC1 is!

```
> plot(pc2 ~ pc1, col=iris$Species, cex=2, pch=16, ylim=c(-3, 4))
```

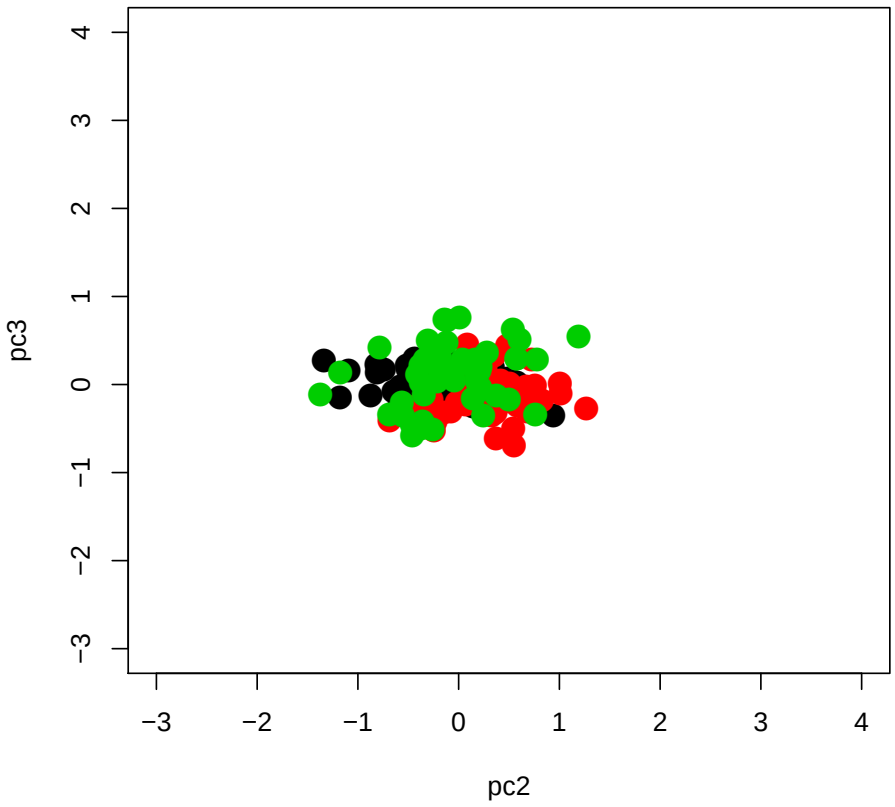


What about PC3? It had only 1.7% of the variation.

```
> plot(pc3 ~ pc1, col=iris$Species, cex=2, pch=16, ylim=c(-3, 4))
```




```
> plot(pc3 ~ pc2, col=iris$Species, cex=2, pch=16, ylim=c(-3, 4), xlim=c(-3, 4))
```



So we see that most of the separation is achieved along PC1, and we can separate the groups pretty well if we just use PC1 with PC2 or PC3. We can see if the separation is significant using MANOVA:

```
> manova.iris <- manova( cbind(pc1, pc2, pc3) ~ Species, data=iris)
```

We obtain the multivariate significance test (differences amongst species) using:

```
> summary(manova.iris)
```

```
           Df Pillai approx F num Df den Df    Pr(>F)
Species     2 1.1756   69.402      6    292 < 2.2e-16 ***
Residuals 147
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Or we can also see results using Wilks' lambda:

```
> summary(manova.iris, test="Wilks")

           Df    Wilks approx F num Df den Df    Pr(>F)
Species     2 0.024809   258.53      6    290 < 2.2e-16 ***
Residuals 147
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

And get univariate statistics here:

```
> summary.aov(manova.iris)

 Response pc1 :
             Df Sum Sq Mean Sq F value    Pr(>F)
Species       2 585.77 292.886  973.27 < 2.2e-16 ***
Residuals   147  44.24   0.301
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

 Response pc2 :
             Df Sum Sq Mean Sq F value    Pr(>F)
Species       2  4.986 2.49300  11.756 1.835e-05 ***
Residuals   147 31.172 0.21205
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

 Response pc3 :
             Df  Sum Sq Mean Sq F value    Pr(>F)
Species       2  1.2578 0.62892  8.8934 0.0002259 ***
Residuals   147 10.3954 0.07072
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

By comparing results we see that they are actually pretty well separated along each PC axis, with no real improvement gained by using a multivariate method. I should find another example!

Note: You should decide whether to do PCA using covariance matrices or correlation matrices. The default in R is covariance matrices, and this will preserve the original scale of the data. If we use correlation matrices, then each variable is allowed to contribute equally, irrespective of how wide the range of values within each variable (it's like they're all standardized first). So you should only use covariances when all measurements are in the same units, for example all lengths in same units, and it makes sense to relate the magnitude of variation in one to another. Otherwise use correlation matrices.

```
> pc.iris.cor <- princomp (~ .-Species, data=iris, scores=T, cor=T)
> summary(pc.iris.cor)

Importance of components:
                          Comp.1    Comp.2     Comp.3      Comp.4
Standard deviation     1.7083611 0.9560494 0.38308860 0.143926497
Proportion of Variance 0.7296245 0.2285076 0.03668922 0.005178709
Cumulative Proportion  0.7296245 0.9581321 0.99482129 1.000000000

> loadings(pc.iris.cor)

Loadings:
             Comp.1 Comp.2 Comp.3 Comp.4
Sepal.Length  0.521 -0.377  0.720  0.261
Sepal.Width  -0.269 -0.923 -0.244 -0.124
Petal.Length  0.580        -0.142 -0.801
Petal.Width   0.565        -0.634  0.524

               Comp.1 Comp.2 Comp.3 Comp.4
SS loadings      1.00   1.00   1.00   1.00
Proportion Var   0.25   0.25   0.25   0.25
Cumulative Var   0.25   0.50   0.75   1.00
```

You can see that the picture we get is a little different. Now that we use correlation matrices, the dominance of the lengths are reduced (the petals and sepals are much more variable in length than in width). You should go through the analysis yourself.

12.2 Canonical Discriminant Analysis

The PCA does not at all account for group structure. When we have multiple groups, we sometimes want to control for within-group structure. One reason is that if we adjust for within-group structure, the groups will be better separated. Another is that the directions of within-group differences might be slightly different than the between-group differences, and we don't want to blend them together because they have different biological meanings. (In this view, the within-group variance is usually the less-interesting portion, and we really are interested in studying the between-group variance).

Another reason to do canonical discriminant analysis is to obtain scores as in PCA to do further analysis. It just produces scores accounting for group structure, so it is like a multi-group PCA. For example, I used this technique to produce a multivariate multi-group ordination in my studies of sexual dimorphism amongst multiple species. I allowed

each species and sex to be a different group, and then tested to see if ecomorphs clustered together.

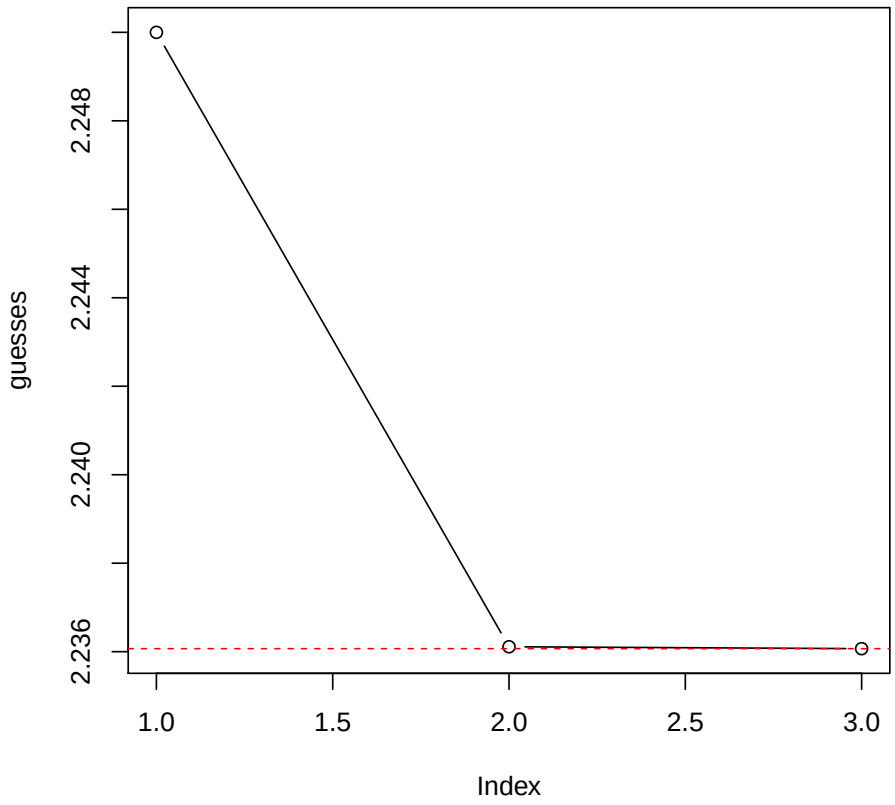
To do this analysis, you need to install the package `candisc`. And produce a multivariate model as in the manova example above.

```
> require(candisc)
> iris.multiv <- lm( cbind(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width) ~ Spe
> iris.can <- candisc( iris.multiv, term="Species")
> iris.can
```

```
Canonical Discriminant Analysis for Species:

   CanRsq Eigenvalue Difference  Percent Cumulative
1 0.96987   32.19193     31.907 99.12126     99.121
2 0.22203    0.28539     31.907  0.87874    100.000

Test of H0: The canonical correlations in the
current row and all that follow are zero

  LR test stat approx F num Df den Df   Pr(> F)
1      0.02344   403.82      4    292 < 2.2e-16 ***
2      0.77797    41.95      1    147  1.32e-09 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

```
> plot(iris.can, col=iris$Species)
```

```
Vector scale factor set to  8
```



So you can see that by accounting for within-species variation, we get better separation of species. Again, we see the greatest separation along Can1. But interestingly, we see that three of the variables are contributing the most to Can1 (but all are highly correlated).

You can see the loadings on the original variables, as well as the percent variance explained using the following code:

```
> iris.can$structure
```

```
                   Can1       Can2
Sepal.Length  0.7918878 0.21759312
Sepal.Width  -0.5307590 0.75798931
Petal.Length  0.9849513 0.04603709
Petal.Width   0.9728120 0.22290236
```

```
> iris.can$eigenvalues
```

```
[1] 3.219193e+01 2.853910e-01 8.276011e-16 7.355228e-16
```

```
> iris.can$pct

[1] 9.912126e+01 8.787395e-01 2.548243e-15 2.264727e-15
```

Ninetly-nine percent of the variance is explained by Can1, which is itself dominated by Petal length and width (.98 and .97), and also with a strong contribution by sepal length (.79). Sepal width provides a contribution in the opposite direction but its correlation value is less (-0.53). We can get most of the separation between species along simply Can1. The scores of each individual in Can space is given in `iris.can$scores`.

Chapter 13

Answers to Exercises – Creating Data Objects

Practice

1. Create a dataset with simulated data using `rnorm()`.

   (a) Simulate 21 random data points drawn from a normal distribution (create a numeric vector), and save it in the variable "y". Create a second set of 21 points and save it as "y1".

   ```
   > y <- rnorm(21)
   > y1 <- rnorm(21)
   > y
   ```

   ```
    [1]  0.29083371 -0.73099247  1.87604553  0.03029000 -0.43811617  0.27508761
    [7]  1.54777706  0.16939718 -1.04787823  1.22163054  1.29919395 -1.11229364
   [13] -1.08301087 -1.38131233 -0.74087168  1.10387956  1.43960280  3.05686922
   [19] -0.51158102 -0.46746532  0.45442855
   ```

   ```
   > y1
   ```

   ```
    [1]  0.524493420 -0.506056725  1.421076163  1.427837349 -0.001765432
    [6]  0.003686001  1.833804669 -0.152058484  1.268763349  0.145833684
   [11] -1.515643626  0.023458920 -0.636832812  0.117446194 -1.488938571
   [16]  0.898151084 -0.212672558 -0.118243333 -0.477678434 -1.739092961
   [21]  1.402590073
   ```

   (b) Create a treatment vector with levels "low", "med", and "high", save it as a factor.

   ```
   > treatment <- factor( c("low", "med", "high") )
   > treatment
   ```

```
[1] low  med  high
Levels: high low med
```

(c) Our treatment has numeric values also, so create a numeric vector with the values 2, 4, 8, save it as x.

```
> x <- c(2, 4, 8)
> x
```

```
[1] 2 4 8
```

(d) Create a species vector with seven names.

```
> species <- LETTERS[1:7]
> species
```

```
[1] "A" "B" "C" "D" "E" "F" "G"
```

(e) Create a matrix with y in the first column and x in the second column, save it as dat.matrix.

```
> dat.matrix <- cbind( y, x )
> dat.matrix
```

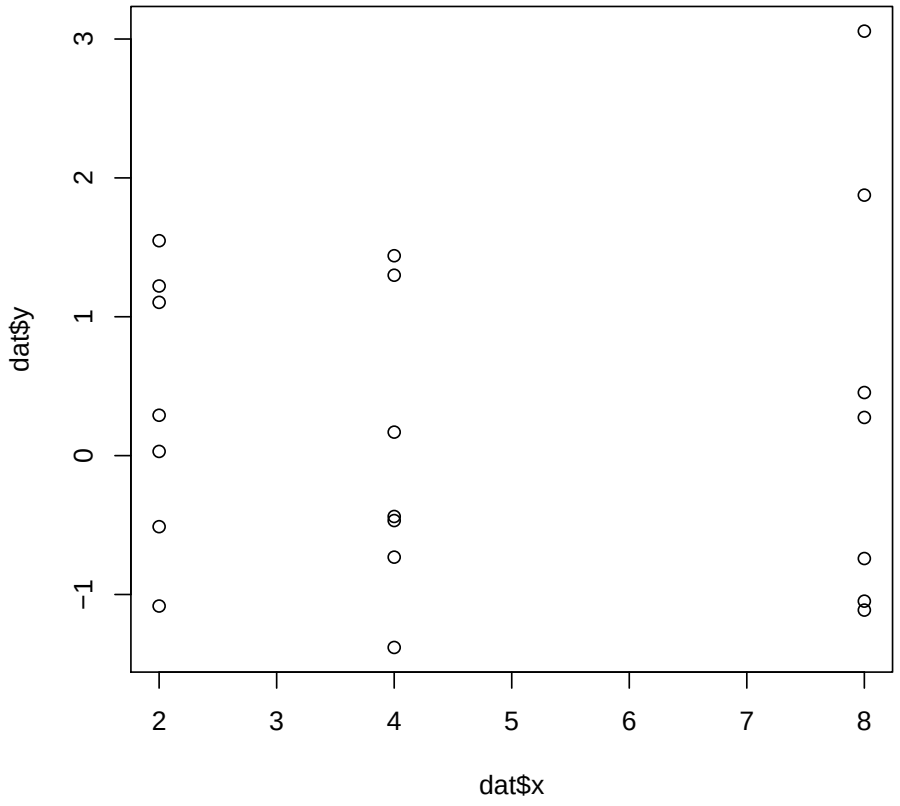
```
                y x
 [1,]  0.29083371 2
 [2,] -0.73099247 4
 [3,]  1.87604553 8
 [4,]  0.03029000 2
 [5,] -0.43811617 4
 [6,]  0.27508761 8
 [7,]  1.54777706 2
 [8,]  0.16939718 4
 [9,] -1.04787823 8
[10,]  1.22163054 2
[11,]  1.29919395 4
[12,] -1.11229364 8
[13,] -1.08301087 2
[14,] -1.38131233 4
[15,] -0.74087168 8
[16,]  1.10387956 2
[17,]  1.43960280 4
[18,]  3.05686922 8
[19,] -0.51158102 2
[20,] -0.46746532 4
[21,]  0.45442855 8
```

(f) Create a data frame with species, x, treatment, y and y1, save as dat. Why can't you make a matrix with these columns?

```
> dat <- data.frame(species, x, treatment, y, y1)
```

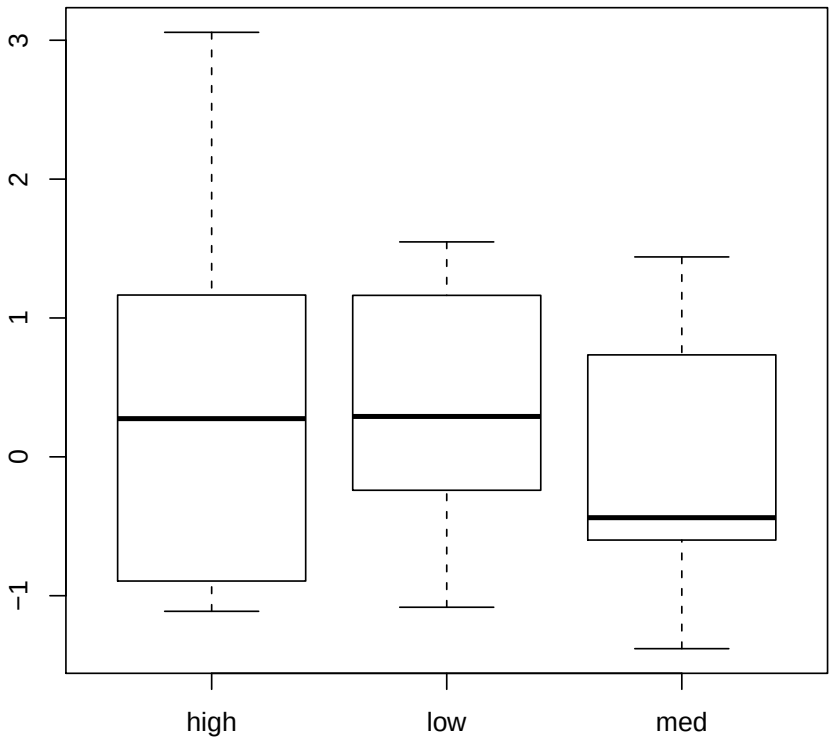
(g) Make a bivariate plot of the numeric value of the treatment (x) versus the response (y). You may want to check the help documentation for "plot". You will have to select the columns of the data frame.

```
> plot( dat$x, dat$y )
```



(h) Make a plot on the treatment as factor versus the response. What is the difference between these two plots?

```
> plot( dat$treatment, dat$y )  # scatterplot vs boxplot
```

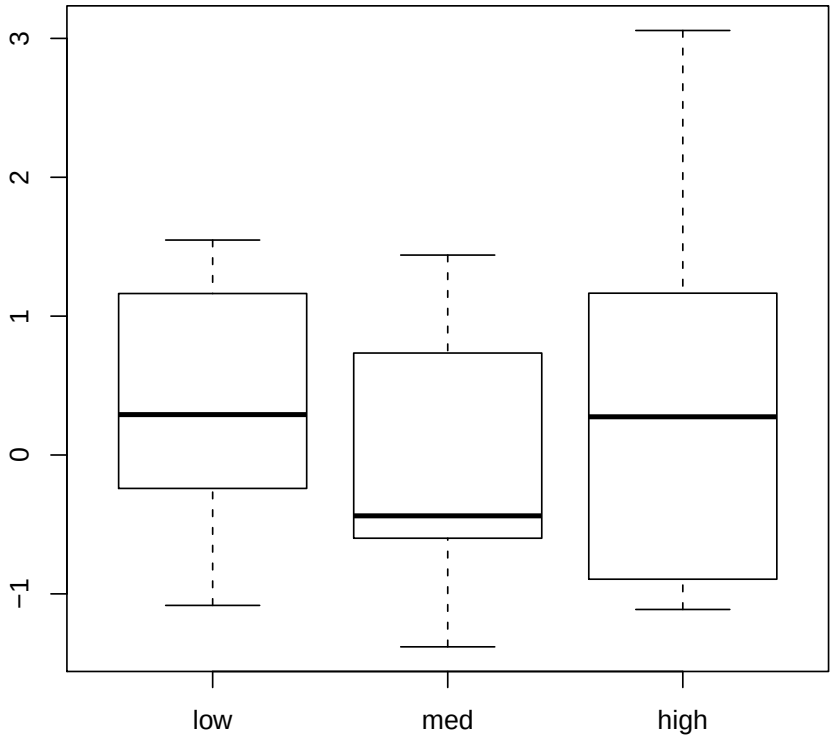


(i) Is the factor displayed in the plot in the order that makes sense? If not, fix this by applying factor to the treatment column of dat again, but this time specifying the levels vector with names of the levels in the order you want. You may want to look at the help page for factor. Plot it again.

```
> dat$treatment <- factor(dat$treatment, levels=c("low", "med", "high"))
> plot( dat$treatment, dat$y )
> dat
```

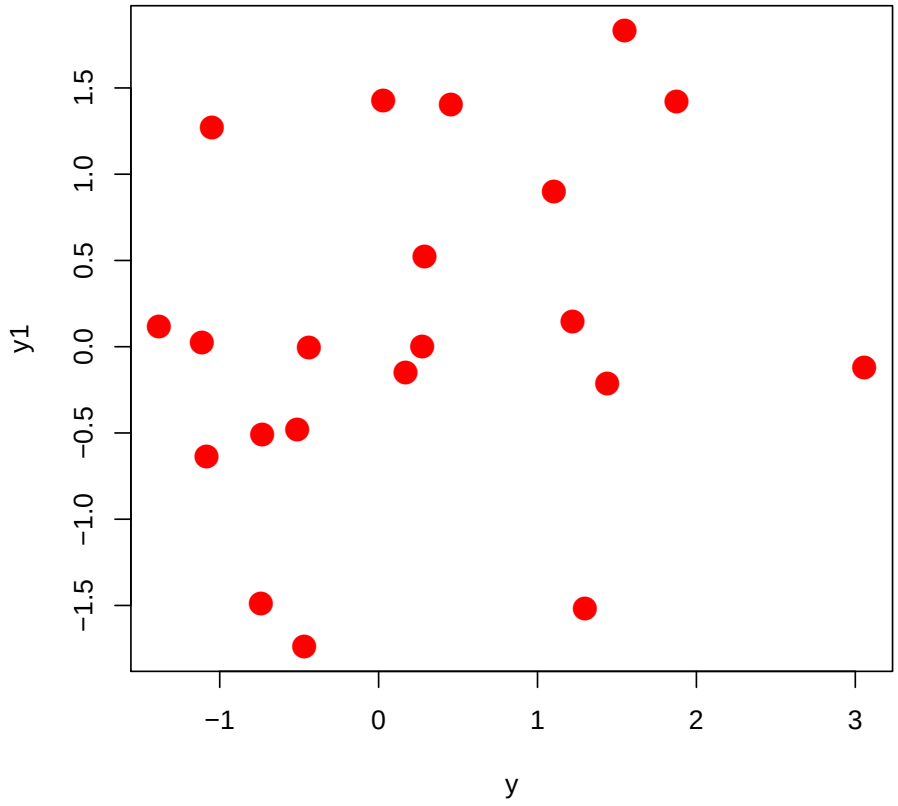
```
   species x treatment           y           y1
1        A 2       low  0.29083371  0.524493420
2        B 4       med -0.73099247 -0.506056725
3        C 8      high  1.87604553  1.421076163
4        D 2       low  0.03029000  1.427837349
5        E 4       med -0.43811617 -0.001765432
6        F 8      high  0.27508761  0.003686001
7        G 2       low  1.54777706  1.833804669
8        A 4       med  0.16939718 -0.152058484
9        B 8      high -1.04787823  1.268763349
10       C 2       low  1.22163054  0.145833684
```

```
11       D 4       med  1.29919395 -1.515643626
12       E 8      high -1.11229364  0.023458920
13       F 2       low -1.08301087 -0.636832812
14       G 4       med -1.38131233  0.117446194
15       A 8      high -0.74087168 -1.488938571
16       B 2       low  1.10387956  0.898151084
17       C 4       med  1.43960280 -0.212672558
18       D 8      high  3.05686922 -0.118243333
19       E 2       low -0.51158102 -0.477678434
20       F 4       med -0.46746532 -1.739092961
21       G 8      high  0.45442855  1.402590073
```



(j) Let's make a scatterplot (`plot(y, y1)`) to see if there is any structuring in the data (eventually with respect to the treatment levels – the rest of this exercise is in the chapter on Workhorse Functions of Data Analysis). While we're at it, let's make it prettier. Change the symbols to solid circles by adding the optional parameter `pch=16`, and the points bigger by `cex=2`. Change the color to red using `col="red"`.

```
> plot(y, y1, pch=16, cex=2, col="red")
```



(k) Now let's make some data which should differ. For the "low" treatment, simulate y and y1 as normally distributed data with mean = -2 and sd=.5, and "high" as mean=5, and sd=3. Remake the dataframe.

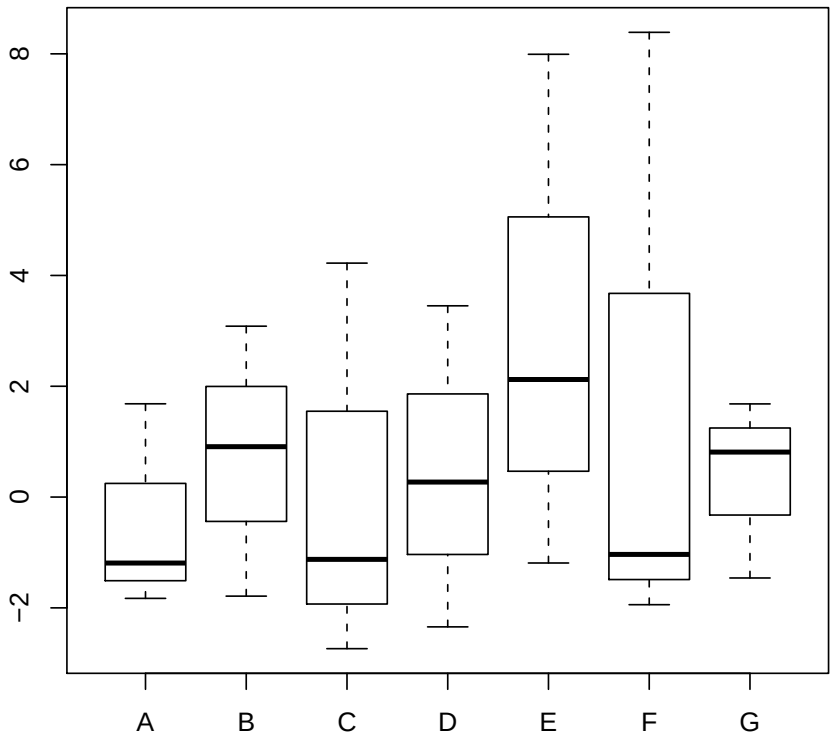
```
> y <- c(rnorm(7, mean=-2, sd=0.5), rnorm(7), rnorm(7, mean=5, sd=3))
> y1 <- c(rnorm(7, mean=-2, sd=0.5), rnorm(7), rnorm(7, mean=5, sd=3))
> y
 [1] -1.8288029 -1.7884820 -2.7364176 -2.3429563 -1.1901773 -1.9428021
 [7] -1.4607354 -1.1911728  0.9092057 -1.1247820  0.2706827  2.1220359
[13] -1.0355536  0.8118809  1.6833277  3.0841090  4.2227286  3.4519807
[19]  7.9920123  8.3880632  1.6813836

> y1
 [1] -1.8899794 -1.6375741 -1.6448559 -1.9921662 -1.7377710 -1.3265830
 [7] -1.7270497  1.6028813  1.0228623 -1.3234996  0.4263121 -0.0610781
[13] -0.4517876 -0.7372885  7.7446775  9.7606643  7.2982633  7.8386139
[19]  1.0057862  2.5540743  9.4384401
```

```
> dat <- data.frame(species, x, treatment=factor(rep(c("low", "med",
+  "high"), each=7), levels=c("low", "med", "high")), y, y1)
> dat
```

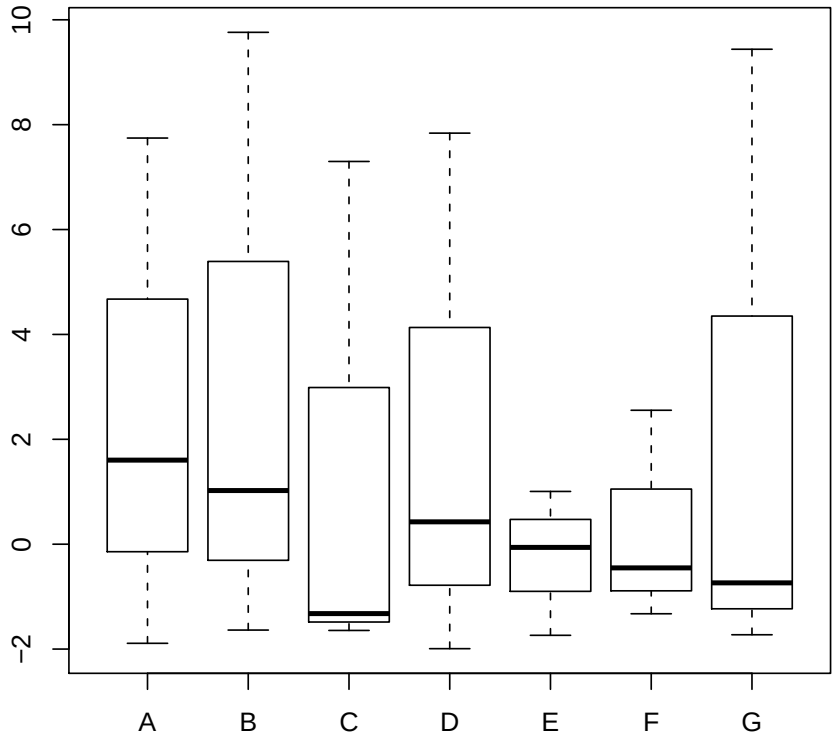
```
   species x treatment          y         y1
1        A 2       low -1.8288029 -1.8899794
2        B 4       low -1.7884820 -1.6375741
3        C 8       low -2.7364176 -1.6448559
4        D 2       low -2.3429563 -1.9921662
5        E 4       low -1.1901773 -1.7377710
6        F 8       low -1.9428021 -1.3265830
7        G 2       low -1.4607354 -1.7270497
8        A 4       med -1.1911728  1.6028813
9        B 8       med  0.9092057  1.0228623
10       C 2       med -1.1247820 -1.3234996
11       D 4       med  0.2706827  0.4263121
12       E 8       med  2.1220359 -0.0610781
13       F 2       med -1.0355536 -0.4517876
14       G 4       med  0.8118809 -0.7372885
15       A 8      high  1.6833277  7.7446775
16       B 2      high  3.0841090  9.7606643
17       C 4      high  4.2227286  7.2982633
18       D 8      high  3.4519807  7.8386139
19       E 2      high  7.9920123  1.0057862
20       F 4      high  8.3880632  2.5540743
21       G 8      high  1.6813836  9.4384401
```

(l) Make boxplots of species vs. y and species vs. y1. Why would you make this plot?

```
> plot(dat$species, dat$y) # differences among species?
```




```
> plot(dat$species, dat$y1)
```


10
8
6
4
2
0
−2
A
B
C
D
E
F
G

Chapter 14

Answers to Exercises – The Workhorse Functions of Data Manipulation

Practice

1. Recall from the chapter on Data Objects that we were simulating data in different treatment groups, and wanting to visualize the groups. Now that we know how to index and subset, we can use the `points` function to add different colored points to the plot for different groups.

   (a) Now let's make some data which should differ. For the "low" treatment, simulate y and y1 as normally distributed data with mean = -2 and sd=.5, and "high" as mean=5, and sd=3. Remake the dataframe.

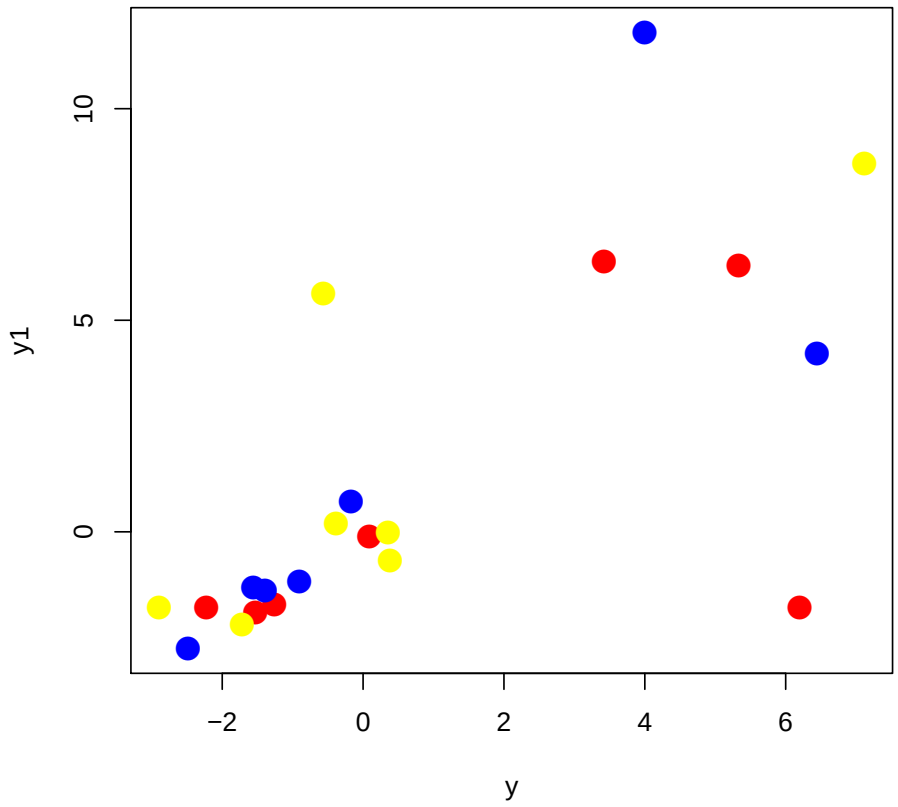
```
> species <- LETTERS[1:7]
> x <- c(2, 4, 8)
> y <- c(rnorm(7, mean=-2, sd=0.5), rnorm(7), rnorm(7, mean=5, sd=3))
> y1 <- c(rnorm(7, mean=-2, sd=0.5), rnorm(7), rnorm(7, mean=5, sd=3))
> y
 [1] -1.39039058 -1.72151685 -2.22420890 -1.55894636 -2.89591841 -1.52939747
 [7] -2.48569947  0.35565222  0.08676244 -0.17269100  0.38304673 -1.25836260
[13] -0.90697300 -0.38370823  3.42139889  3.99650326  7.11672665  6.20038054
[19]  6.44302123 -0.56430891  5.33022433
> y1
 [1] -1.38024429 -2.19688465 -1.78252098 -1.32358065 -1.78243836 -1.91741911
 [7] -2.76144918 -0.01531455 -0.10303655  0.71433332 -0.68143340 -1.72629590
[13] -1.16873921  0.18668455  6.38568403 11.80470115  8.70903520 -1.78316658
[19]  4.20823905  5.63218438  6.30643576
```

```
> dat <- data.frame(species, x, treatment=factor(rep(c("low", "med",
+  "high")), levels=c("low", "med", "high")), y, y1)
> dat
```

```
   species x treatment           y          y1
1        A 2       low -1.39039058 -1.38024429
2        B 4       med -1.72151685 -2.19688465
3        C 8      high -2.22420890 -1.78252098
4        D 2       low -1.55894636 -1.32358065
5        E 4       med -2.89591841 -1.78243836
6        F 8      high -1.52939747 -1.91741911
7        G 2       low -2.48569947 -2.76144918
8        A 4       med  0.35565222 -0.01531455
9        B 8      high  0.08676244 -0.10303655
10       C 2       low -0.17269100  0.71433332
11       D 4       med  0.38304673 -0.68143340
12       E 8      high -1.25836260 -1.72629590
13       F 2       low -0.90697300 -1.16873921
14       G 4       med -0.38370823  0.18668455
15       A 8      high  3.42139889  6.38568403
16       B 2       low  3.99650326 11.80470115
17       C 4       med  7.11672665  8.70903520
18       D 8      high  6.20038054 -1.78316658
19       E 2       low  6.44302123  4.20823905
20       F 4       med -0.56430891  5.63218438
21       G 8      high  5.33022433  6.30643576
```

(b) Let's differentially color the "high", "medium", and "low" points. First set up the plot window without any points by plotting y, y1 with the plot parameter `type="n"`. Then select only the "high" points by subsetting. You'll want to make an index vector to choose only the points you want. Then use the `points()` function (which has the same form as the `plot()` function, but only adds points to an existing plot. Choose three different colors for each treatment level and plot all the data. Is there any patterning in y, y1?

```
> plot(y, y1, type="n")
> points( y[dat$treatment=="high"], y1[dat$treatment=="high"],
+ pch=16, cex=2, col="red")
> points( y[dat$treatment=="med"], y1[dat$treatment=="med"],
+ pch=16, cex=2, col="yellow")
> points( y[dat$treatment=="low"], y1[dat$treatment=="low"],
+ pch=16, cex=2, col="blue")
```



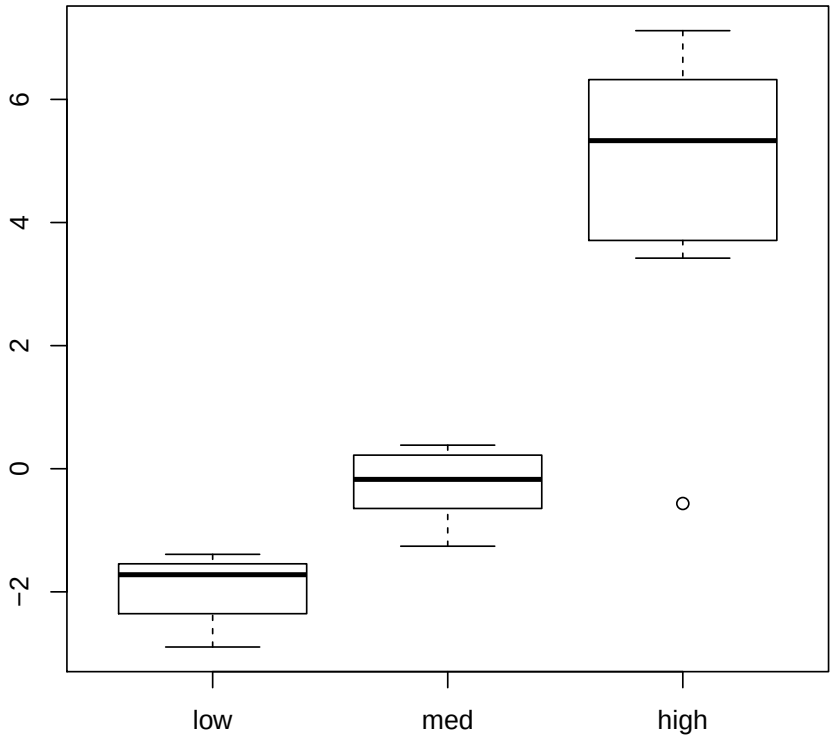
(c) Ooops! The data are actually supposed to be blocked by treatment (the first seven rows correspond to low, the second 7 correspond to med, etc.) Can you remake the dataframe keeping the y and y1 in the same position, but fixing the treatment?

```
> dat <- data.frame(species, x, treatment=factor(rep(c("low", "med",
+ "high"), each=7), levels=c("low", "med", "high")), y, y1)
> dat
   species x treatment           y          y1
1        A 2       low -1.39039058 -1.38024429
2        B 4       low -1.72151685 -2.19688465
3        C 8       low -2.22420890 -1.78252098
4        D 2       low -1.55894636 -1.32358065
5        E 4       low -2.89591841 -1.78243836
6        F 8       low -1.52939747 -1.91741911
7        G 2       low -2.48569947 -2.76144918
8        A 4       med  0.35565222 -0.01531455
9        B 8       med  0.08676244 -0.10303655
10       C 2       med -0.17269100  0.71433332
```

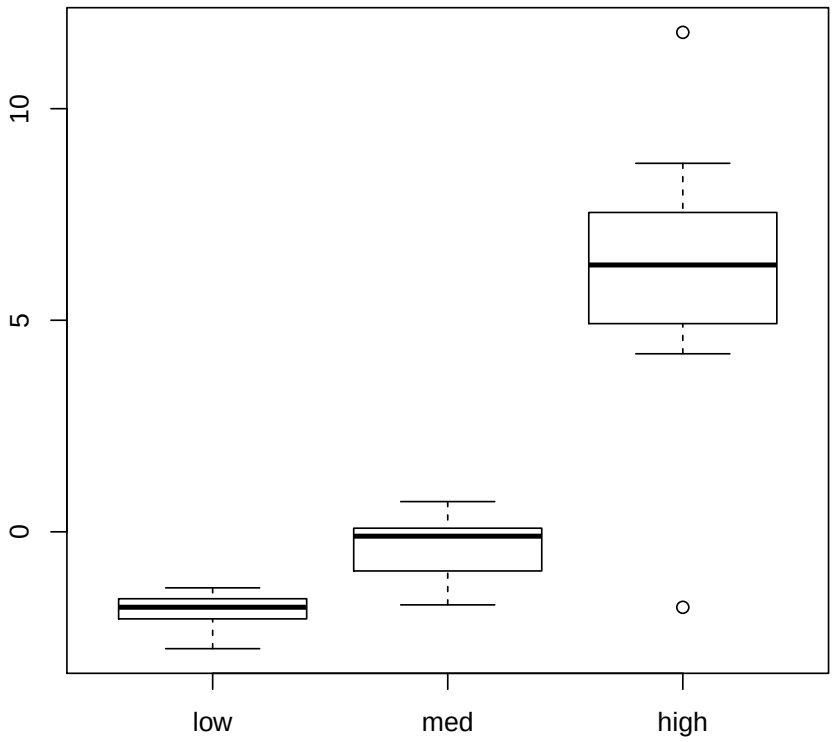
```
11       D 4       med  0.38304673 -0.68143340
12       E 8       med -1.25836260 -1.72629590
13       F 2       med -0.90697300 -1.16873921
14       G 4       med -0.38370823  0.18668455
15       A 8      high  3.42139889  6.38568403
16       B 2      high  3.99650326 11.80470115
17       C 4      high  7.11672665  8.70903520
18       D 8      high  6.20038054 -1.78316658
19       E 2      high  6.44302123  4.20823905
20       F 4      high -0.56430891  5.63218438
21       G 8      high  5.33022433  6.30643576
```

(d) Make three plots: boxplot of treatment vs. y, treatment vs. y1, and three color scatterplot of y vs. y1 (treatments should be indicated by different colors).

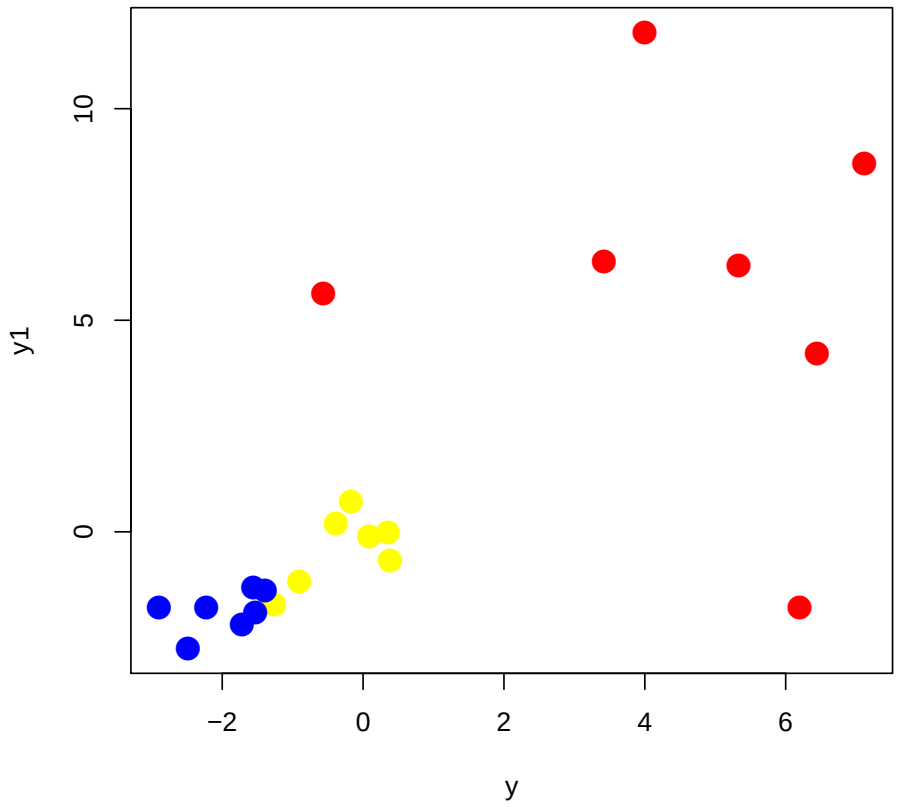
```
> plot(dat$treatment, dat$y)
```




```
> plot(dat$treatment, dat$y1)
```




```
> plot(y, y1, type="n")
> points( y[dat$treatment=="high"], y1[dat$treatment=="high"],
+ pch=16, cex=2, col="red")
> points( y[dat$treatment=="med"], y1[dat$treatment=="med"],
+ pch=16, cex=2, col="yellow")
> points( y[dat$treatment=="low"], y1[dat$treatment=="low"],
+ pch=16, cex=2, col="blue")
```



2. Matrix reshaping and indexing – This was given for homework so no answers. Sorry!

   (a) Create a matrix with the values 1 through 20, filling four rows. Save it as "x".

   (b) What are the attributes of x?

   (c) Change it to a matrix with 2 rows and 10 columns by changing its attribute.

   (d) Change x to a vector.

   (e) Change x to a matrix with four rows, this time filling it by rows rather than by columns (you may want to check the help page).

   (f) Coerce x to a vector again. Is it in the same order as the previous vector? What does this tell you about R's default behavior when flattening matrices to vector?

   (g) Create the original x matrix again. Select only the 3rd row, 4th column. What is it?

   (h) Select rows 3 and 4, columns 4 and 5. Print it to the console by using the `print(x)` function.

(i) Select the first and last rows, first and last columns. Print it.

3. Reading in Data and adding on

(a) Read in the external file `bimac.csv` in comma separated format. Save it as "`bimac`".

```
> bimac <- read.csv("Data/bimac.csv")
> bimac
```

	node	species	size	ancestor	time	OU.LP	OU.1	OU.3	OU.4
1	1	<NA>	NA	NA	0	medium	global	medium	anc
2	2	<NA>	NA	1	12	medium	global	medium	anc
3	3	<NA>	NA	2	32	small	global	medium	anc
4	4	<NA>	NA	3	34	small	global	medium	anc
5	5	<NA>	NA	4	36	small	global	medium	anc
6	6	<NA>	NA	3	36	small	global	medium	anc
7	7	<NA>	NA	1	8	medium	global	medium	anc
8	8	<NA>	NA	7	13	medium	global	medium	anc
9	9	<NA>	NA	8	18	large	global	medium	anc
10	10	<NA>	NA	9	23	large	global	medium	anc
11	11	<NA>	NA	10	28	large	global	medium	anc
12	12	<NA>	NA	9	28	large	global	medium	anc
13	13	<NA>	NA	8	22	medium	global	medium	anc
14	14	<NA>	NA	13	26	medium	global	medium	anc
15	15	<NA>	NA	14	34	medium	global	medium	anc
16	16	<NA>	NA	15	36	medium	global	medium	anc
17	17	<NA>	NA	7	28	medium	global	medium	anc
18	18	<NA>	NA	17	30	medium	global	medium	anc
19	19	<NA>	NA	18	34	medium	global	medium	anc
20	20	<NA>	NA	19	36	medium	global	medium	anc
21	21	<NA>	NA	20	37	medium	global	medium	anc
22	22	<NA>	NA	19	36	medium	global	medium	anc
23	23	po	13.5	2	38	small	global	small	small
24	24	se	14.3	4	38	small	global	small	small
25	25	sc	14.3	5	38	small	global	small	small
26	26	sn	14.2	5	38	small	global	small	small
27	27	wb	14.5	6	38	small	global	small	small
28	28	wa	14.9	6	38	small	global	small	small
29	29	be	23.6	10	38	large	global	large	large
30	30	bn	27.1	11	38	large	global	large	large
31	31	bc	27.9	11	38	large	global	large	large
32	32	lb	28.6	12	38	large	global	large	large
33	33	la	28.8	12	38	large	global	large	large
34	34	nu	21.1	13	38	medium	global	medium	medium
35	35	sa	18.3	14	38	medium	global	medium	medium

```
36   36      gb 19.7       15   38 medium global medium medium
37   37      ga 18.8       16   38 medium global medium medium
38   38      gm 19.6       16   38  large global  large  large
39   39      oc 22.3       17   38 medium global medium medium
40   40      fe 28.4       18   38 medium global medium medium
41   41      li 18.7       20   38 medium global medium medium
42   42      mg 18.9       21   38 medium global medium medium
43   43      md 19.9       21   38 medium global medium medium
44   44      t1 21.3       22   38 medium global medium medium
45   45      t2 21.5       22   38 medium global medium medium
```

(b) This is a phylogenetic tree and data for the OUCH package. Without going into details for now, this method allows biologists to specify selective regimes on branches of the phylogeny, by specifying categories which correspond to alternative "niches". This is a body size evolution dataset, and "OU.LP" is a hypothesis with three size categories. We would like to make three additional hypotheses. Add additional columns to this dataframe: OU.1 which has values of "`global`" for all rows, OU.3 which is the same as OU.LP, except those rows with "`NA`" in the species names should get a value of "`medium`", and OU.4 which is again similar to OU.LP, except that those rows with "`NA`" in the species names get a value of "`anc`".

```
> bimac$OU.1 <- "global"
> bimac$OU.3 <- bimac$OU.LP
> bimac$OU.3[1:22] <- "medium"  # or the next way
> bimac$OU.3[ is.na(bimac$species) ] <- "medium"
> bimac$OU.4 <- as.character( bimac$OU.LP )
> bimac$OU.4[1:22] <- "anc"  # or the next way
> bimac$OU.4 <- factor(bimac$OU.4)  # to make it a factor again
> bimac
   node species size ancestor time  OU.LP   OU.1   OU.3   OU.4
1     1    <NA>   NA       NA    0 medium global medium    anc
2     2    <NA>   NA        1   12 medium global medium    anc
3     3    <NA>   NA        2   32  small global medium    anc
4     4    <NA>   NA        3   34  small global medium    anc
5     5    <NA>   NA        4   36  small global medium    anc
6     6    <NA>   NA        3   36  small global medium    anc
7     7    <NA>   NA        1    8 medium global medium    anc
8     8    <NA>   NA        7   13 medium global medium    anc
9     9    <NA>   NA        8   18  large global medium    anc
10   10    <NA>   NA        9   23  large global medium    anc
11   11    <NA>   NA       10   28  large global medium    anc
12   12    <NA>   NA        9   28  large global medium    anc
13   13    <NA>   NA        8   22 medium global medium    anc
14   14    <NA>   NA       13   26 medium global medium    anc
```

15	15	<NA>	NA	14	34	medium	global	medium	anc
16	16	<NA>	NA	15	36	medium	global	medium	anc
17	17	<NA>	NA	7	28	medium	global	medium	anc
18	18	<NA>	NA	17	30	medium	global	medium	anc
19	19	<NA>	NA	18	34	medium	global	medium	anc
20	20	<NA>	NA	19	36	medium	global	medium	anc
21	21	<NA>	NA	20	37	medium	global	medium	anc
22	22	<NA>	NA	19	36	medium	global	medium	anc
23	23	po	13.5	2	38	small	global	small	small
24	24	se	14.3	4	38	small	global	small	small
25	25	sc	14.3	5	38	small	global	small	small
26	26	sn	14.2	5	38	small	global	small	small
27	27	wb	14.5	6	38	small	global	small	small
28	28	wa	14.9	6	38	small	global	small	small
29	29	be	23.6	10	38	large	global	large	large
30	30	bn	27.1	11	38	large	global	large	large
31	31	bc	27.9	11	38	large	global	large	large
32	32	lb	28.6	12	38	large	global	large	large
33	33	la	28.8	12	38	large	global	large	large
34	34	nu	21.1	13	38	medium	global	medium	medium
35	35	sa	18.3	14	38	medium	global	medium	medium
36	36	gb	19.7	15	38	medium	global	medium	medium
37	37	ga	18.8	16	38	medium	global	medium	medium
38	38	gm	19.6	16	38	large	global	large	large
39	39	oc	22.3	17	38	medium	global	medium	medium
40	40	fe	28.4	18	38	medium	global	medium	medium
41	41	li	18.7	20	38	medium	global	medium	medium
42	42	mg	18.9	21	38	medium	global	medium	medium
43	43	md	19.9	21	38	medium	global	medium	medium
44	44	t1	21.3	22	38	medium	global	medium	medium
45	45	t2	21.5	22	38	medium	global	medium	medium

Chapter 15

Answers to Exercises – Writing your own functions

15.1 Exercises

1. Write your own function for calculating a mean of a vector, using only the `sum()` and the `length()` functions. The input should be a vector, and the output is the mean.

```
> mymean <- function(x) {  # version 1
+
+         xbar <- sum(x) / length(x)
+         return (xbar)
+ }
> mymean <- function(x) {  # version 2 don't get so short that you confuse yoursel
+
+         return (sum(x) / length(x))
+ }
> mymean( 1:5 )
```

```
[1] 3
```

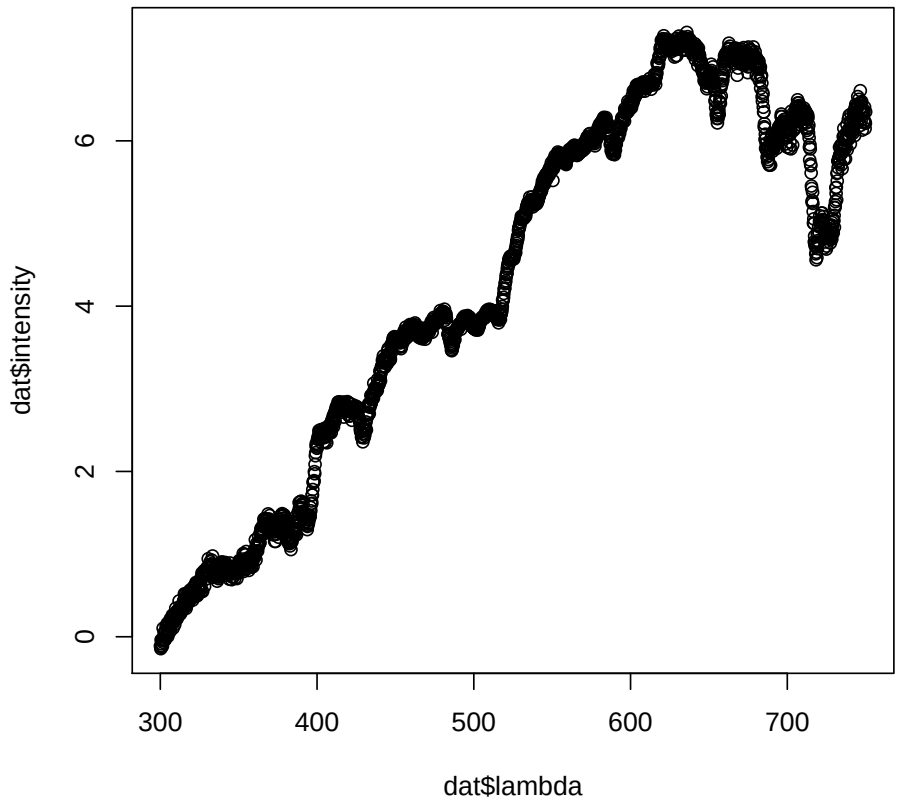
2. Write your own function for calculating the standard error. You can use the `sd()`, `sqrt()`, and the `length()` functions. The input should be a vector of values.

```
> myse <- function(x) {
+
+         se <- sd(x) / sqrt(length(x))
+         return (se)
+ }
> myse( 1:5 )
```

```
[1] 0.7071068
```

3. Go back to Chapter 7 section 7.6.1. Write a function that will read in the irradiance data, trim it to wavelengths between 300 and 750 nm, and plot the data. Then use that function to read in files for the different directions: `up`, `for` (forward), `left`, and `right`: `20070725_01upirr.txt`, `20070725_01forirr.txt`, `20070725_01leftirr.txt`, `20070725_01rightirr.txt`. Your function should take as input just the file name. Write a script that defines the function and then calls the function four times, once for each file.

```
> readirr <- function ( ifile = "Data/20070725_01upirr.txt") {
+         # I like to put in a default arguments so it will "go" by itself
+
+         dat <- read.table(file=ifile, skip=17, comment.char=">")  # read in
+         names(dat) <- c("lambda", "intensity")  #add names
+
+         dat <- dat[dat$lambda >= 300 & dat$lambda <= 750,]  # subset wavelengths
+         plot( dat$lambda, dat$intensity)
+         }
> readirr()
> readirr("Data/20070725_01forirr.txt")
> readirr("Data/20070725_01leftirr.txt")
> readirr("Data/20070725_01rightirr.txt")
```



4. Now take the function you just made, and add optional arguments for the cut off values 300 and 750. You may want to trim the data to different values. Try trimming it to different values and see what happens using your new function.

```
> readirr <- function ( ifile = "Data/20070725_01upirr.txt",
+           start=300, stop=750) {  # I like to put in a default so it will "go" b
+
+         dat <- read.table(file=ifile, skip=17, comment.char=">")  # same
+         names(dat) <- c("lambda", "intensity")
+
+         dat <- dat[dat$lambda >= start & dat$lambda <= stop,]  # add start stop
+         plot( dat$lambda, dat$intensity)
+         }
> readirr( start=300, stop=350 )   # only looking in the UV
```