A brief introduction to R

Sean Anderson, sean_anderson@sfu.ca

September 17, 2011

Contents

1 Why R? 2

2 The idea 2

3 Workflow 3

4 Object types 3

5 The workspace 5

6 Basic syntax 6

7 Helpful tools to look at your data 6

8 Subsetting and manipulation 8

9 Programming concepts 10

10 A matter of style 13

11 Reading data in 14

12 Getting data out 15

13 Packages 15

14 Getting help 16

15 Basic plotting 17

1 Why R?

R is scriptable, reusable, sharable. By using it you document your analyses for yourself and for others, which creates reproducible research. This is a *good thing*. R is open-source (meaning it's free to use) and has quickly become the standard for data analysis around the world. Additionally, R now tends to have the best and most current statistical packages available.

R has a steep initial learning curve but enormous longterm gains and your pace of learning will accelerate through time if you stick with it. You will find that there is always more to learn about R. The time, effort and quantity of code it will take you to accomplish tasks will hopefully dramatically lessen as you progress.

2 The idea

R is an "object-oriented programming language" (and as programming languages go, a relatively straightforward one!). The idea is that "objects" take on values. You can manipulate those objects as need be. You can add values to them, feed values between objects, and get values back out. Each object carries values of certain types (called a "class"). An object might represent a single number, a series of numbers, or a mixture words and numbers, or a function that does something like return an average of a series of numbers, for example.

These objects are given a name. In R, these object names must start with a letter of the alphabet and can then consist of letters, numbers, underscores, and periods.

Keep in mind that capitalization counts.

3 Workflow

There are two main ways you will interact with R: interactively and through scripts that you save (plain text files containing lines of code). Typically you will build code by trying it interactively and then save what works in a script. You'll probably find it easiest to code more complicated operations directly in a script. That script provides documentation of what you have done, can be modified and reused in the future, and can be passed on to others who want to repeat your analysis.

Generally, you'll find it easiest if you set up your desktop so you can view at least one script file and your "console" (where you type the interactive commands) on the screen at once. Since vertical space is usually at a premium, you might want to consider having one beside the other. Some text editors will let you open two or more files beside or on top of one another. You might find this helpful when your scripts get more complicated. Most good text editors will allow you to send lines of code directly to R from your script. If you're not already familiar with a program for working with R, RStudio is a good program to try.

The important message here is that if you care about what you're typing in the console and want to be able to repeat it then save it in a script!

4 Object types

R has a number of object classes that you'll come across frequently. Here are some of the basic classes that are good to know along with examples:

character `"hello"`

numeric `999`

logical `TRUE` or `FALSE`

vector multiple character, numeric, or logical values: `c("one", "two")` or `c(1, 2)` or `c(TRUE, FALSE)`

array or matrix Arrays are numeric vectors in multiple dimensions. A matrix (what you will most often see) is a two dimensional array.

```
> m <- matrix(data = c(1, 2, 3, 4), ncol = 2)
> m

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

data.frame Data frames can contain multiple vector types (e.g., numeric, character, logical) in column format. You can think of them as a matrix in which different columns can contain different classes of vectors. They are a common and useful object type.

```
> d <- data.frame(col1 = c(1, 2), col2 = c("species1", "species1"))
> d

  col1     col2
1    1 species1
2    2 species1
```

list Lists can contain any combination of object types — for example a data frame, a vector, and a character vector.

```
> l <- list()
> l[[1]] <- d
> l[[2]] <- c(1, 2, 3)
> l[[3]] <- "hello"
> l

[[1]]
  col1     col2
1    1 species1
2    2 species1

[[2]]
[1] 1 2 3

[[3]]
[1] "hello"
```

factor A factor is a vector of character objects with an underlying numeric value. They can be ordered or unordered. You might find these the most confusing at first. They can be useful for building models or plotting when the order of categorical values matters. (As straight character vectors they would be sorted alphabetically.)

```
> x <- c(1, 2, 3)
> factor.x <- factor(x, levels = 1:3)
> levels(factor.x) <- c("low", "medium", "high")
> as.numeric(factor.x)

[1] 1 2 3

> levels(factor.x)

[1] "low"    "medium" "high"
```

There are other standard object formats (such as timeseries, equations, and functions), but we won't go into these at this point. With the work we (ecologists, or maybe even most scientists) do, most of the time you'll find yourself working with data frames or vectors of numbers.

5 The workspace

In R, a "workspace" contains a group of objects. The most common workspace will be the global workspace. To see all the objects in your global workspace you can use the `ls()` command:

```
> ls()

[1] "d"        "factor.x" "l"        "m"        "x"
```

You can remove an object with the `rm()` command. You can remove all global objects with `rm(list = ls())`

```
> rm(factor.x)
> rm(list = ls())
> ls()
```

```
character(0)
```

6 Basic syntax

The basic syntax to work with R is surprisingly simple. How you put it all together to do what you want is where it gets fun. Here are the symbols you'll need to know the meaning of:

`<-` This means assign a value to the object the arrow points to. There are other ways to assign values, but `<-` is generally accepted as the best practice. (e.g. `a <- 2`)

`=` Most often used when "calling" a function to tell the function what values to use. (e.g. `plot(1, 1, xlab = "My x-axis label")`)

`~` Most often used in equations to relate the dependent to the independent variables. (e.g. `lm(y ~ x)`)

`== < > >= <=` Equal to, less than, greater than, and so on.

`- + / * ^` Subtract, add, divide, multiply, raise to a power.

`$` Its most frequent use is to let you access a column in a data frame by its name. (e.g. `my.data.frame$column.name)`

`[]` Lets you access indexed values of a vector, matrix, or data frame. (e.g. `my.vector[1]`) We'll come back to this.

`[[]]` Lets you access indexes of a list. (e.g. `mylist[[1]]`)

7 Helpful tools to look at your data

head Shows you the first six rows of your data by default. Very useful when your data is long. Also see `tail()`.

```
> 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
```

names Gets the names of an object; most often this will be the column names of a data frame.

```
> names(iris)

[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

summary Summarizes information about your object. The output will vary depending on your object type.

```
> summary(iris[, 1:2])

  Sepal.Length    Sepal.Width
 Min.   :4.300   Min.   :2.000
 1st Qu.:5.100   1st Qu.:2.800
 Median :5.800   Median :3.000
 Mean   :5.843   Mean   :3.057
 3rd Qu.:6.400   3rd Qu.:3.300
 Max.   :7.900   Max.   :4.400
```

str Show a summary of the structure of your object.

```
> str(iris)

'data.frame':        150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1
```

unique Show the unique instances of values within an object.

```
> unique(iris$Species)

[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
```

max, min, range These do what you'd think they would.

```
> range(iris$Sepal.Length)

[1] 4.3 7.9
```

8 Subsetting and manipulation

R has many powerful functions for manipulating data. For now, we will focus on the basics.

First, let's create some data:

```
> d <- data.frame(species = c("A", "A", "B", "B", "C", "C"), count = c(9,
+     2, 5, 5, 7, 8))
```

Let's say we wanted to extract the counts for all of species A. Two basic ways we can do that are via indexing and via the `subset` function. For indexing we use square brackets. If we have rows and columns (as in a data frame), indexing takes the form of `[row, column]`. So, `my.data.frame[1, ]` would mean return the 1st row of the data frame. `my.data.frame[2, 3]` would mean return the value from the 2nd row and the 3rd column. Getting back to the species counts, here are two ways to start:

```
> d[d$species == "A", ]

  species count
1       A     9
2       A     2
```

```
> subset(d, species == "A")
```

```
  species count
1       A     9
2       A     2
```

Or, if we knew these were the first two rows, we could have done this:

```
> d[1:2, ]
```

```
  species count
1       A     9
2       A     2
```

Or we could have removed all other rows:

```
> d[-c(3:6), ]
```

```
  species count
1       A     9
2       A     2
```

```
> d[-which(d$species %in% c("B", "C")), ]
```

```
  species count
1       A     9
2       A     2
```

```
> subset(d, !species %in% c("B", "C"))
```

```
  species count
1       A     9
2       A     2
```

`transform` is an elegant way to create new columns in a data frame based on existing data. E.g., say we wanted to create a column which was equal to the log of the species count column and another that was the square root of species count.

```
> d <- transform(d, log.count = log(count), sqrt.count = sqrt(count))
> d
```

```
  species count log.count sqrt.count
1       A     9 2.1972246   3.000000
2       A     2 0.6931472   1.414214
3       B     5 1.6094379   2.236068
4       B     5 1.6094379   2.236068
5       C     7 1.9459101   2.645751
6       C     8 2.0794415   2.828427
```

Alternatively, we could have done this:

```
> d$log.count = log(d$count)
> d$sqrt.count = sqrt(d$count)
```

If we want to remove a column, we can set its value to `NULL`:

```
> d$sqrt.count <- NULL
> names(d)
```

```
[1] "species"   "count"     "log.count"
```

9 Programming concepts

A bit of familiarity with some standard programming concepts can help when getting started with R.

loops Loops repeat a section of code multiple times. They can be ended after a certain number of times or when a certain condition becomes true. Unlike many other languages, loops are generally discouraged in R since there are often more elegant and faster options available. Still, sometimes they are necessary, and sometimes they are simpler to code. Here's a basic "for loop":

```
> x <- 0
> for (i in 1:3) {
+     i <- i + 1
+     print(i)
+ }

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

This is probably a good time to point out that R excels at vector and matrix operations but can be exceedingly slow at loops, especially if you are subsetting or manipulating your data with each loop. People will often suggest that you "vectorize" what you're doing. This means remove the loop and operate on the vector instead. Here's a simple example. `system.time` is just a function that times how long something takes. These both do the same thing but the first attempt is over a 1000 times slower. This can make a huge difference with long loops or big datasets. For example, here is the same operation in a loop and vectorized format:

```
> x <- runif(2e+05)
> system.time(for (i in 1:2e+05) {
+     x[i] <- x[i] * 2
+ })

   user  system elapsed
  0.576   0.002   0.578

> system.time(x * 2)

   user  system elapsed
      0       0       0
```

logical statements Operators such as `<, >, <=, >=, ==` can be combined with "and" operators & and/or "or" operators | to select data or check the values of an object. Use brackets to group logical statements. Another useful operator is `%in%` which checks which values from the first object are found in the second object.

```
> x <- c(1, 2, 3)
> y <- c(2, 3, 4)
> x < y

[1] TRUE TRUE TRUE

> x[x < y]

[1] 1 2 3

> x %in% y

[1] FALSE  TRUE  TRUE

> x[x %in% y]

[1] 2 3

> x[x %in% y & x <= 2]

[1] 2

> x[x %in% y | x <= 2]

[1] 1 2 3

> x[x %in% y | (x <= 2 & x > 1)]

[1] 2 3
```

if else statements If else statements check is something is true. If the statement is true, they do one thing. If the statement if false, they do something else.

```
> a <- 4
> if (a < 3) {
+     a.is.small <- TRUE
+ } else {
+     a.is.small <- FALSE
+ }
> a.is.small
```

```
[1] FALSE
```

Keep in mind that you don't necessarily need an `else` statement. Also, if there is only one line after an `if` or `else` statement then you don't need the `{}`s.

functions Functions are chunks of code that typically fulfill one purpose. They take input values, do something to them or plot something, and can return values. This may seem abstract at this point, but functions are immensely useful. Essentially, they let you reuse bits of code and make your code more readable by making your intentions explicit. Almost any analysis in R that you do will at least make use of functions from packages. If your analysis gets slightly more complicated, you will quickly find that writing your own functions will simplify your work in the long run. You won't end up copying and pasting code and it will therefore be easier to maintain if you want to make changes to it.

As an example, we might rewrite a simplified version of the `mean` function like this:

```
> my.mean <- function(x) sum(x)/length(x)
> my.mean(c(1, 6, 2, 9, 4))
```

```
[1] 4.4
```

10 A matter of style

Following a consistent style can make your code easier for you and others to read and, most importantly, reduce mistakes.

There is no official style guide for coding in R, but the most frequently used guide is probably the one by Google: http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html

As an alternative, see Hadley Wickham's: http://had.co.nz/stat405/resources/r-style-guide.html

11 Reading data in

Before you read data in, you'll want to make sure R knows what directory you are working in. On a Mac (or any Unix system) you can set the "working directory" like this (for Windows the directory name will start with something like `c:/`):

```
> setwd("~/Dropbox/presentations/r-intro/")
```

You can always check your current working directory with `getwd()`. Note that on a Mac (or any Unix system) `~` is an abbreviation for your home directory.

```
> getwd()
```

```
[1] "/Users/seananderson/Dropbox/presentations/r-intro"
```

There are many ways to read in your data. Here are some common ways:

`read.csv` Reads your data from a .csv file.

`read.table` Reads your data from any plain text based format.

`load` Loads data from a .rda or .Rdata file that was previously saved.

`scan` The most basic way to import your data. Still useful for text files with one column of data.

`read.csv` is a special version of `read.table` that has defaults that work well for .csv files. A common question is whether you can read Excel files into R. The answer is yes (via a package such as `gdata`), but it's usually not advisable. File formats change

through the years (and even from computer to computer) and you want your data in as simple and longterm a format as possible. Plus, formatting in spreadsheets can have strange consequences when read directly into R. So, save your spreadhsheet as a .csv or tab delimited file first.

12 Getting data out

`write.csv` Writes your data to a .csv file.

`write.table` Writes your data to whatever format you'd like.

`save` Saves an object to an R data file. Conventional file extensions are .rda or .Rdata.

`save.image` Saves all objects in memory to a file named `.Rdata` unless you specify otherwise.

13 Packages

Packages contain groups of functions, usually for a specific purpose. All the functions come with help files. Sometimes the package comes with a vignette, which is a fancy name for a helpful manual written by the package author or authors.

To install a package you can do so via the R or RStudio menu system or type something like the following, in this case to install the package `beanplot`:

```
install.packages("beanplot")
```

After you've installed a package, you can load it with the `library` or `require` commands. These do nearly the same thing, read the help files if you're curious about the difference.

```
require(beanplot)
```

R comes with many packages that the maintainers of R have deemed general and useful enough to be loaded automatically. For example, `plot` is part of the `graphics` package that is automatically loaded when you start R.

Want to check for a vignette?

```
vignette(beanplot)
```

14 Getting help

Knowing how to get help on R functions is vital. It is expected that you will read the help files and they are often surprisingly detailed and helpful — particularly the examples at the bottom.

The most basic way to get help is with the `?` before a function name. E.g., `?plot` will bring up the help screen for the function `plot`. The question mark is your best friend in R!

Other tricks you may find helpful:

`RSiteSearch` This searches the R-help listserve in a browser. You may want to refine what aspects of the listserve are searched once the results appear by ticking off additional boxes.

`sos` **package and** `???` If you're trying to find a package or function to do something, odds are someone's already written it. Try `???` and you'll often be amazed.

```
require(sos)
???"multimodel averaging"
```

websites There are lots. One favourite is http://www.rseek.org/. http://stackoverflow.com/ and the related http://stats.stackexchange.com/ are also very good.

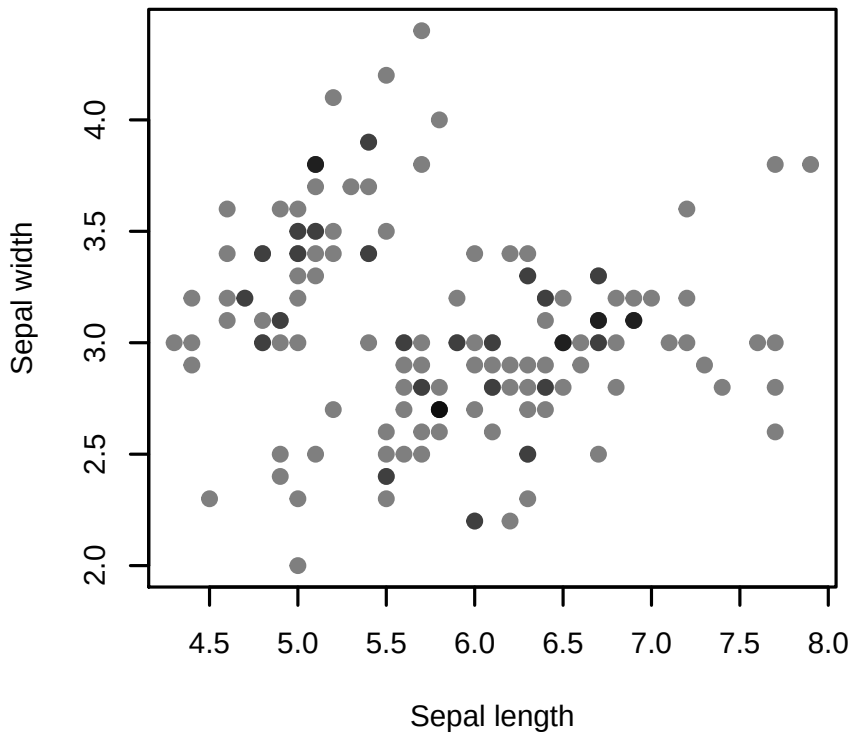
books There are lots of good books. All recent Springer books are available as PDF downloads on the SFU network http://springerlink.com/content/ Peter Dalgaard's *Introductory Statistics with R* is a classic starting point. Venables

and Ripley's *Modern Applied Statistics with S* is nearly essential. Both of these books are surprisingly readable. As is Zuur, Ieno, and Meesters's *A Beginner's Guide to R*.

15 Basic plotting

Personally, I find visualizing data where R really shines. There are almost no limits to what you can create visually via R, although some of the syntax and tricks to achieve what you want may seem arcane at first. I will save most plotting for another day, but to get you started, let's use the `plot` function and explain a few of the options.

```
> par(cex = 0.6)
> plot(iris$Sepal.Length, iris$Sepal.Width, pch = 19, col = "#00000080",
+     xlab = "Sepal length", ylab = "Sepal width")
```



I first set the "character expansion factor" `cex` to a smaller size than the default value of one to make the text, axes, and symbols a reasonable size for the plot. Then I gave the function `plot` a series of numbers for the x-values and a series of numbers for the y-values. Be default, R will try to come up with some reasonable axis limits and labels. I chose to specify the axis labels myself. I told it to use filled in circles `pch = 19` and gave it a colour of black with 80% opacity `#00000080` to show overlapping points. Everything you see is customizable. See `?plot.default` and `?par` for many of the options.