Introduction to R

Lucy Chang

August 25, 2015

1 Anatomy of R

1.1 The Program

Console
: The part of R that keeps track of what you do. You can't manipulate anything in its history, only read it. Anything after the > is something you've run.

Code
: What you tell R to run. A set of instructions. A recipe. This can be directly typed into the console (at the bottommost >), but it's better practice to write it in a *script* file so you can save your work and rerun it easily - good things if you update your data or your computer crashes. Spaces and tabs don't matter in R, unlike some other languages.

How to run your code
: There are two ways to run code in a script you've written: using the "run" button in the menu or pressing Ctrl+Enter (Cmd+Enter). If nothing is highlighted, doing either action will automatically run only the line of code your cursor is on. You can also highlight any amount of code and it will run only what is highlighted.

  The code you've run as well as any results R returns appear in the *console*.

1.2 The Code

`#comment`
: R does not run anything in a line that comes after at least one pound symbol. Use this to annotate your code!

```
> 2 + 2 ### will all this text mess up R's arithmetic?
[1] 4
```

`variable <- value`
: Variables are objects that hold information - just like variables in formulas

(ex: the x and y in x = 2*y), they are nicknames for some to-be-determined value. In R, variable names have no meaning until you assign a value to them and can be recognized as words with no quotation marks around them. For example, running

```
> grape
```

confuses R, because it has never seen that name before. However, you can run

```
> grape <- "red"
```

If you then ask R what `grape` is by entering it in the console like before, it will tell you that `grape` is a string of characters called `"red"`.

```
> grape
[1] "red"
```

You can overwrite variables with new values.

```
> grape <- "green"
> grape
[1] "green"
```

You can manipulate variables.

```
> grape <- 2 # grape is equivalent to the number 2
> grape*grape
[1] 4
> ten.grapes <- 5*grape
> ten.grapes
[1] 10
```

function(*argument*)

Functions perform actions. Function names can be recognized by the parentheses that follow them. Functions take arguments, which you write inside the parentheses. Arguments designate what information to perform the function's action on and any other settings the function has.

```
> log(3)
[1] 1.098612
> log(3, base=10)
[1] 0.4771213
> a <- log(3, base=10)  # assigning this function's value to a variable
> a
[1] 0.4771213
```

`library(`*`package`*`)`

`library()` is a special function that loads a package, which you name inside the parentheses. Packages are themed sets of functions developed by R users. You must first install a package on your computer before tyring to load it, either via the menu or by running a function called `install.packages("`*`package`*`")`. Once loaded, all the functions contained in the package can be used. You only need to load a package once each time you start a new session in R. So, good practice is to load all your packages at the top of your script to run first-thing and not think about again.

`?`*`function`* or `help(`*`function`*`)`

A question mark before a function name or the function `help()` with the function name as the argument brings up the help file for that function. *Extremely* useful for figuring out what arguments you need to feed into the function and what value the function gives back. Use it frequently!

When you come across documentation for a function you want to use, it will often start with *`function`*{*`package`*}. The contents of the squiggly brackets indicates what package you have to load to use the function. *`function`*{`base`} means it is always loaded in R.

Character vs Numeric vs Other Types of Data

There are a certain number of data types that R recognizes.

- *numeric*: numbers, which R can perform arithmetic on among other things
- *character*: words, always designated by quotation marks
- *logical*: like `TRUE` and `FALSE`, these words have special meaning to R
- Other ways of storing data (objects) you will frequently encounter include *matrix*, *data.frame*, and *array*.

Try these functions:

```
> is.numeric(2)

> is.numeric("two")

> class("two")

> class(is.numeric("two")) # a function inside of another function!
```

The type of data you are working with affects what functions can be performed on them. `2 + 2` means something to R, but `2 + "2"` does not. You can always use `class()` to discover what type of data you are working with.

`variable[index]`
: Ways of navigating data. Discussed in **Sections 2.1, 2.2**.

Logical operators
: Ways of filtering data. Discussed in **Section 2.1**.

2 Loading in and manipulating data

Files containing comma-separated values (.csv) are fairly standard for organizing data.

```
> read.csv(file="filedirectory/filename.csv")
```

Don't forget to save your data to a variable name. Else, all R does is look at it. You want to eventually be able to alter the dataset or refer to just the parts of it you want.

```
> dat <- read.csv(file="filedirectory/filename.csv")
```

If you're feeling lazy, you can replace the file name with `file.choose()`, which will open up a dialog box to browse for your file. `?read.csv` to see what other arguments the function takes.

You can get a sense of what your data looks like by viewing the variable that contains it.

```
> dat
```

You may find yourself working with datasets so large, it doesn't make sense for you to print out in the console everything (it might not even fit in the console window). To easily view the first few lines of your dataset, use:

```
> head(dat)
```

2.1 Working in one-dimension

Vectors are one-dimensional lists - just a line of values (each one called an *element*). You can create vectors using the concatenate function `c()`.

```
> a <- c(3, 6, 15, 20)
> a
[1]  3  6 15 20
> class(a)
[1] "numeric"
```

You can add new things to vectors.

```
> a <- c(a,"thing") # note, we're using a variable to rewrite itself!
> a
[1] "3"     "6"     "15"    "20"    "thing"
> class(a)
[1] "character"
```

Notice that by adding a different type of data (a word instead of another number), you've changed the class of your vector. In other words, R no longer thinks of 3, 6, 15, and 20 as numbers but as text.

What if you only want part of the vector? You designate what part you want in brackets like *`variable`*`[`*`index`*`]`, where *index* is the position (first, second, third, etc.) of the value you want.

```
> a <- c(3, 6, 15, 20) # make a vector of numeric values again
> a[3] # grab the 3rd element of your stored vector
[1] 15
> a[4]
[1] 20
> a[5] # what if you try to go outside the length of the vector?
[1] NA
> a[c(3, 4)]
[1] 15 20
```

What if I just want any values greater than or equal to 10? Try out some of these:

```
> a > 10 # greater than

> a >= 10 # greater than or equal to

> a[a >= 10]

> a[c(FALSE,FALSE,TRUE,TRUE)]
```

Now you're using logical operators. These basically pose a question (*ex: is 3 greater than or equal to 10?*), to which R lets you know if it's `TRUE` or if it's `FALSE` (*ex: FALSE*). These value can essentially be used to turn on and off elements. Common logical operators include:

`<`	less than	`<=`	less than or equal to
`>`	greater than	`>=`	greater than or equal to
`==`	equals	`!=`	does not equal
`|`	or	`&`	and

2.2 Working in two-dimensions

Most data are not one-dimensional. In the long run, you want to figure out how different sets of data are related to one another. `matrix` and `data.frame` objects are two-dimensional. The difference between the two is that the columns in a `data.frame` can store different types of data, while a `matrix` can only hold the same type. `array` objects are matrices that can be more than two dimensions.

Let's build a 3x4 matrix, filling the cells with the numbers 1 through 12. (Note: colons stand for "through", so 1:12 means 1 through 12.)

```
> b <- matrix(1:12, nrow=3)
> b
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Notice the rows and columns are labeled with brackets, where rows have numbers on the left side of the comma and columns have numbers on the right side. Instead of having only one number describing where an element is, now each element is defined by two positions (describint two dimensions). Each of those positions, however, is dealt with the same way as before.

```
> b[1,] # return what's in the first row
[1] 1  4  7 10
> b[,3] # return what's in the third column
[1] 7 8 9
> b[1,3]
[1] 7
> b[c(2,3),]
     [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
[2,]    3    6    9   12
```

You can add row and column names and refer to pieces of your matrix using those.

```
> rownames(b) <- c("how","are","you")
> colnames(b) <- c("I","am","doing","fine")
> b[,c("doing","fine")]
    doing fine
how     7   10
are     8   11
you     9   12
```

Notice that when you fix one dimension (ex: only return column 3 and no other columns), you get back a one-dimensional list of values. A vector!

2.2.1 A special way to navigate `data.frame` objects

If you are working with a `data.frame` object, then things get slightly easier. You can actually refer to columns by their column names if they have them, instead of what number column it is. Instead of brackets, you call up `variable$column`. What major advantage is there to referring to columns by their names?

2.3 More dimensions?

It is possible to have `array` objects of many dimensions. Just keep adding those commas inside the brackets. For example, `variable[,,,,,]` has five dimensions (that is, you need five independent pieces of information to locate any one element of data).

2.4 Exercises

- Read in the example dataset patientData.csv and save it to a variable called `patients`. What type of object is `patients`?
- Find the mean (`mean()`) age of all the patients.
- Combine (`cbind()`) only the age and BMI columns into a new matrix called `patients2`.
- Reduce `patients2` down to only those patients who are older than 30 but younger than 50.

3 Making plots

There are many packages out there that allow you to make pretty plots - one of the advantages that R has over other programming languages. The most basic function you can use is `plot(x,y)` where `x` represents a vector (remember `c()`?) of all your x-values and `y` represents a vector of all the y-values associated with them

```
> plot(c(1,2,3,5),c(9,7,4,3))
```

From here you can build on your basic plot. Remember, you can change the settings of your basic plot by including arguments (`?plot`). Also check out functions like `title()`, `points()`, `abline()`, and others.

3.1 Exercises

- Using the patients dataset we used before, generate a plot where your x-values are patient weights and your y-values are BMI.
- Make the plot again, but label the axes "weight" and "BMI", respectively, and color the dots orange.
- Generate a histogram of the patient ages.

4 Running analyses

Because data is vast and nuanced, we use numbers to summarize the gist of the data and interpret them - statistics. That is, we can reduce lots of numbers (say, the heights of 500 people) into one descriptive number (the average height) usually at the expense of accuracy. We can also test how unusual these descriptive numbers are.

There are several key aspects of your data you need to figure out before you use a statistical test.

- Are you working with *continuous* data, *categorical* data, or a mix of the two?
  - Can it be any number (decimals and all) or does it fall into bins that you can describe?
- Are your variables independent?
  - If you change one variable, does your other one *necessarily* have to change?
- Most important of all, what *exactly* do you want to test?
  - The more explicit you can be with this (the more you can boil it down to one or a few very descriptive metrics) - the easier it will be to find the right test for your question.
  - Are means of these two sets of values actually different? Does one thing occur more frequently than you'd expect if the world behaved randomly? (What would a randomly behaving world look like?) How strong of a relationship is there between two variables?

For starters, peruse some basic statistical functions described at:
`http://www.r-tutor.com/elementary-statistics/`
`http://www.ats.ucla.edu/stat/mult_pkg/whatstat/`

4.1 Exercises

- How correlated are patient weights and BMIs?
- Are the means of patient ages and BMIs significantly different? (They should be!)

5 What now?

Once you identify the characteristics of your data and your question, functions exist in R to do almost anything (and for the more advanced, you write the functions you need!).

- How similar are two faunal lists recorded from different times or places?
  → `vegdist{vegan}`
- What percent area of the United States does a species occupy, as calculated using the area of counties where observations have been made?
  → `area.map{maps}`
- How does the rate of shape change differ along growth trajectories between two organisms?
  → `trajectory.analysis{geomorph}`

GOOGLE LIBERALLY.

Use it to look around for a function or a suite of functions (like in a package) that gets at, or at least part of, what you're trying to test. Use it to look up help for using functions or manipulating data. Use it to look up terminology and ask dumb questions.

Commonly used functions for R can be found at
`http://cran.r-project.org/doc/contrib/Short-refcard.pdf`