An Introduction to R with Examples from Ecology and Evolution

Andy Rominger

2016-05-06

Contents

1 Getting Started

1.1 What to expect in this tutorial

This tutorial hopes to take a user who has never used R to a level of proficiency sufficient to preform many basic analytical tasks, such as importing data, manipulating data and extracting useful information from data stored in R, conducting basic statistical tests, and plotting data. Once these tasks have been mastered, it is hoped that a user can further explore through this tutorial how to conduct more complicate statistical procedures, analyze phylogenetic data, and harness geo-spatial data to use in other analyses. This tutorial makes use of several data sets which we summarize in Table [datasets]. In order to follow examples, it is advisable to first download these data.

This tutorial is aimed primarily at the computational side of ecological and evolutionary data analysis. Hence, while some explanation will be given for statistical inference techniques, a deeper understanding must be sought elsewhere.

Table 1: Data sets used in this tutorial

File name		
`GrasshopperAbundance`	`subfamily`	Grasshopper subfamily
	`species`	Grasshopper species
	`abund06`	Recorded abundance in 2006
	`abund07`	Recorded abundance in 2007

1.2 Installing R

Go to www.r-project.org.

In the side bar, click the CRAN link (under Download). This will direct you to a list of mirrors from which you can select a near-by host. Click the mirror link and then click on your appropriate OS. Follow the download instructions.

Further packages can be installed within the `R` environment using the `Packages & Data` pulldown, or using the function `install.packages`. `Packages & Data` allows you to search for packages and then install them. To use the function `install.packages` type directly into the console `install.packages("package")` where `package` is the name of the desired package. To then load the package into your current workspace, use the command `library(package)` or `require(package)`. Section [sec:commands] on commands in `R` will explain functions, and sections [sec:phylo] and [sec:geo] on phylogenetics and geographic data will provide examples of using auxiliary packages.

Table 2: Useful functions for import and set-up

Function		
`install.packages`	package name in quotes	imports the desired package
`library`	package name (no quotes)	loads package into memory
`require`	same as above	same as above

1.3 Using commands in `R`

Whatever the task to be completed in `R`, we do so using commands. Commands are given using operators (e.g. `*,/,+,-`) or functions (e.g. `mean(data)`). Most functions take arguments, which the user specifies, and which control how or what the function computes. Let's consider an example. Suppose we want to produce 10 random numbers drawn from a normal distribution with mean = 1 and variance = 4. We do so with the command

```
rnorm(n=10, mean=1, sd=2)
```

```
##  [1]  0.455942313  2.627715301  1.072953298 -0.774904035 -0.008773426
##  [6]  1.361582263  1.027045868  2.384229274 -2.375071650  3.006276460
```

Here the function is `rnorm` and the arguments are `n`, which specifies how many numbers we want to draw, `mean`, which specifies the mean, and `sd`, which specifies the standard deviation, i.e. the square root of the variance. For more on drawing samples from probability distributions see section [sec:distrib].

1.4 Getting help

No one can know the arguments of functions intuitively. To learn more about functions, there help pages, which we can find with the question mark (`?`). For example `?rnorm` provides us with information about the use of the function, its arguments, the details underlying its computation and its output (i.e. value) and, most help pages provide examples.

If we don't know the desired function there are many resources online. Providing Google with keywords such as "r sample normal distribution" or perhaps "r-project sample normal distribution" will typically produce an answer to "which function should I use?" `R` also provides built-in search options using keywords, accessed by a double question mark (`??`). For example `??normal` searches all `R` help pages for anything with "normal" as a keyword.

1.5 Assigning objects in R

Suppose we wanted to store a sample of 10 random normal numbers, rather than just print it to the screen as we did in the previous example. To do so we must tell `R` what name to store those values under, let's use the name `my.norm`.

```
my.norm <- rnorm(n=10, mean=1, sd=2)
```

This is called "assignment" and we would say "assign 10 random normal values to an object named `my.norm`." It uses a special assignment operator `<-`; it is equivilant to use an equal sign `=`, but the special assignment operator is prefered in `R`.

By again typing `my.norm` (and running it) we print those 10 numbers; note they will be different from the previous example because those were two different samples form `R`'s normal random number generator

```
my.norm
```

```
##  [1] -1.0865907  2.6306418  2.9200885  4.1274915  2.4453397 -0.1618248
##  [7] -1.0841880  1.3865220  0.4181742 -2.4511973
```

Now that these values are stored in the object `my.norm` we can use them for any ruther computation, for example, simple addition

```
my.norm + 2
```

```
##  [1]  0.9134093  4.6306418  4.9200885  6.1274915  4.4453397  1.8381752
##  [7]  0.9158120  3.3865220  2.4181742 -0.4511973
```

Let's assign that to a new object

```
my.norm2 <- my.norm + 2
my.norm2
```

```
##  [1]  0.9134093  4.6306418  4.9200885  6.1274915  4.4453397  1.8381752
##  [7]  0.9158120  3.3865220  2.4181742 -0.4511973
```

For the sake of really making this clear, let's recap: we made this object with the "assign" operator, i.e. the `<-` symbol, which means "assign what's on the right-hand-side to what's on the left-hand-side." `R` will also let you use `=` for this purpose, but it is good etiquette to reserve the assig operator `<-` for assigning values to objects and only use the equal operator `=` for specifying arguments in functions (see Table [basicOperators]).

Table 3: Basic operators and arithmetic functions. Many functions can be used with much finesse, which we can't summarize here. Consult the help pages, see section [sec:help].

Command		
`<-`	Assignment of value to object memory	`x <- 5,`
`=`	Specifies values of arguments in functions	`rnorm(n=10)`
`+ - * /`	Preforms addition, subtraction, etc.	`4 + 3`
`^`	Raises a number to a power	`2^2`
`exp`	Raise e to a power	`exp(0), exp(1), exp(2)`
`sqrt`	takes square root	`sqrt(4)`
`mean, median var, sd`	computes mean, median, etc.	`mean(y), sd(y)`

1.6 A note on objects and classes in R

`R` is an object-oriented programming language, meaing it operates on objects (values which we've assigned a name to in memory) and those objects additionally are automaticallly characterized by their class. Not knowing the details of what that means is not a major limitation; however it is helpful to have a basic understanding. All objects belong to a class and this class identity determines what can be done to or with a given object. A silly abstract example is the difference between an object that is of class `wall` and an object that is of class `human`. You can talk to a human, but you shouldn't talk to a wall. Similarly, a human can be used to build a wall (perhaps using some function `build.wall(method='human')`), but no function can use a wall to build a human. `R` has a huge number classes and we'll learn some key ones as we go. Some examples are `numeric`, which is the class of our `my.norm` object, and `data.frame`, which is the class of spread sheet like objects in `R` and the default class for any data read into `R` using `read.csv` or `read.table` (see section [sec:reading]).

1.7 Data structures in R

Before going any further we summarize some important structures pertaining to working with your data in `R`.

1.7.1 Vector

This is the most basic data structure. A vector can be composed of numbers (class `numeric` or `integer`), characters (class `character`), or booleans (`TRUE`/`FALSE` values; class `logical`). Vectors can be created using various functions, for example we create a vector of three zeros in four different ways:

```
x1 <- c(0, 0, 0)
x2 <- rep(0, times = 3)
x3 <- numeric(length=3)
x4 <- vector(mode = "numeric", length = 3)
```

The `c` function can also be conveniently used to create vectors of unique numbers, characters and booleans, for example:

```
x <- c(1, 2.3, 3.1)
y <- c("a", "b", "c")
z <- c(TRUE, TRUE, FALSE)
```

To determine the number of elements in a vector, we use the function `length`

```
length(x)
```

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

We can also easily make vectors of consecutive integers using a colon

```
x.seq <- 1:4
x.seq
```

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

Similarly, we could use the `seq` function, which can also be used to make more complicated sequences of numbers

```
x.seq <- seq(from=1, to=4)
x.seq
```

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

```
x.seq1 <- seq(from=1, to=4, by=0.2)
x.seq1
```

```
##  [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
```

```
x.seq2 <- seq(from=1, by=0.2, length.out=16)
x.seq2
```

```
##  [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
```

Vectors are commonly used in `R` functions, such as in conducting a t-test (testing the means of two *vectors*) or in plotting data points (plotting two *vectors* in a Cartesian coordinate system).

We can access different parts (i.e. "elements") of a vector using indices set off by square brackets `[ ]`. For example, suppose we wanted only the second element of the vector `x` (recall that the values of `x` are `1, 2.3, 3.1`), we would simply type:

```
x[2]
```

```
## [1] 2.3
```

We can also use multiple indices to extract multiple elements of a vector

```
x[1:2]
```

```
## [1] 1.0 2.3
```

```
x[c(1, 3)]
```

```
## [1] 1.0 3.1
```

We could equally well use `TRUE`'s and `FALSE`'s to indicate which elements we desire (recall the vector `z` composed of the booleans `TRUE, TRUE, FALSE`):

```
x[z]
```

```
## [1] 1.0 2.3
```

In this case we've extracted every value of `x` for which `z` is `TRUE`.

We can also use indices to help us change the value of specific elements in a vector, for example we can change the first element of `y` to a capital `A`.

```
y
```

```
## [1] "a" "b" "c"
```

```
y[1] <- "A"
y
```

```
## [1] "A" "b" "c"
```

Numeric vectors can be added, subtracted, multiplied and divided[1]. `R` carries out these operations element-wise, meaning that the operation is applied to each element of the vector independently. This can be both helpful and confusing because it does not conform with some operations in linear algebra[2]. Examples will help illuminate

```
my.vector <- 1:4
ur.vector <- 5:8
my.vector + 1
```

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

```
my.vector * 2
```

```
## [1] 2 4 6 8
```

```
my.vector * ur.vector
```

```
## [1]  5 12 21 32
```

In the last line, we notice that the result is four elements long, the first element corresponding to `my.vector[1]*ur.vector[1]`, and the second element to `my.vector[2] *ur.vector[2]`, and so on. This is often the desired result, unless we wanted to take the dot product of the two vectors, that will come later.

[1] Note: this is an example where classes determine the possible operations that can be done on a object; a vector of class `character` cannot be operated on by `+, - *`, etc.

[2] See in section [sec:matrix] for linear algebra operations

Table 4: Useful functions for vectors and factors

Function		
`c`	several values (all of the same class)	vector containing the given values
`rep`	`x`: what will be replicated `times`: number of replicates	vector containing the value(s) replicated
`seq`	many, see text	a vector containing the desired sequence
`factor`	a vector, typically of characters	a factor object with alphabetically ordered levels
`levels`	a factor	a vector containing all the unique entries of the factor object given
`length`	a vector or factor	the number of elements
`names`	a vector or factor (other structures will also work)	the names of the object's elements, or can be used to assign new names

A complete argument list is not feasible in this space, refer to the help documentation, see section [sec:help]

2 Common Probability Distributions

3 Graphics

4 Simulation and Permutation

5 Writing Your Own Functions