**R Language**

**Sample data**

If you just want to play with some test data to see how they load and what basic functions you can run, the default installation of R comes with several data sets. Type:

data()

into the R console and you'll get a listing of pre-loaded data sets. Not all of them are useful (body temperature series of two beavers?), but these do give you a chance to try analysis and plotting commands. And some online tutorials use these sample sets.

One of the less esoteric data sets is mtcars, data about various automobile models that come from *Motor Trends*. (I'm not sure from what year the data are from, but given that there are entries for the Valiant and Duster 360, I'm guessing they're not very recent; still, it's a bit more compelling than whether beavers have fevers.)

You'll get a printout of the entire data set if you type the name of the data set into the console, like so:

mtcars

There are better ways of examining a data set, which I'll get into later in this series. Also, R does have a print() function for printing with more options, but R beginners rarely seem to use it.

**Existing local data**

R has a function dedicated to reading comma-separated files. To import a local CSV file named filename.txt and store the data into one R variable named mydata, the syntax would be:

mydata <- read.csv("filename.txt")

(Aside: What's that <- where you expect to see an equals sign? It's the R assignment operator. I said R syntax was a bit quirky. More on this in the section on R syntax quirks.)

And if you're wondering what kind of object is created with this command, mydata is an extremely handy data type called a data frame -- basically a table of data. A data frame is organized with rows and columns, similar to a spreadsheet or database table.

The read.csv function assumes that your file has a header row, so row 1 is the name of each column. If that's *not* the case, you can add header=FALSE to the command:

mydata <- read.csv("filename.txt", header=FALSE)

In this case, R will read the first line as data, not column headers (and assigns default column header names you can change later).

If your data use another character to separate the fields, not a comma, R also has the more general read.table function. So if your separator is a tab, for instance, this would work:

mydata <- read.table("filename.txt", sep="\t", header=TRUE)

The command above also indicates there's a header row in the file with header=TRUE.

If, say, your separator is a character such as | you would change the separator part of the command to sep="|"

**Examine your data object**

Before you start analyzing, you might want to take a look at your data object's structure and a few row entries. If it's a 2-dimensional table of data stored in an R data frame object with rows and columns -- one of the more common structures you're likely to encounter -- here are some ideas. Many of these also work on 1-dimensional vectors as well.

Many of the commands below assume that your data are stored in a variable called *mydata* (and not that *mydata* is somehow part of these functions' names).

*[This story is part of Computerworld's "Beginner's guide to R." To read from the beginning, check out* [*the introduction*](http://www.computerworld.com/s/article/9239625)*; there are links on that page to the other pieces in the series.]*

If you type:

head(mydata)

R will display mydata's column headers and first 6 rows by default. Want to see, oh, the first 10 rows instead of 6? That's:

head(mydata, n=10)

Or just:

head(mydata, 10)

Note: If your object is just a 1-dimensional vector of numbers, such as (1, 1, 2, 3, 5, 8, 13, 21, 34), head(mydata) will give you the first 6 items in the vector.

To see the *last* few rows of your data, use the tail() function:

tail(mydata)

Or:

tail(mydata, 10)

Tail can be useful when you've read in data from an external source, helping to see if anything got garbled (or there was some footnote row at the end you didn't notice).

To quickly see how your R object is structured, you can use the str() function:

str(mydata)

This will tell you the type of object you have; in the case of a data frame, it will also tell you how many rows (observations in statistical R-speak) and columns (variables to R) it contains, along with the type of data in each column and the first few entries in each column.

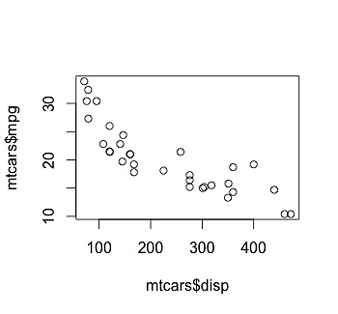
R's str functionResults of the str() function on the sample data set PlantGrowth.

For a vector, str() tells you how many items there are -- for 8 items, it'll display as [1:8] -- along with the type of item (number, character, etc.) and the first few entries.

One of the most appealing things about R is its ability to create data visualizations with just a couple of lines of code.

For example, it takes just one line of code -- and a short one at that -- to plot two variables in a scatterplot. Let's use as an example the mtcars data set installed with R by default. To plot the engine displacement column *disp* on the x axis and *mpg* on y:

plot(mtcars$disp, mtcars$mpg)

Default scatterplot in R.

You really can't get much easier than that.

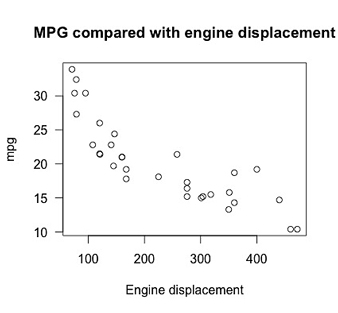
*[This story is part of Computerworld's "Beginner's guide to R." To read from the beginning, check out* [*the introduction*](http://www.computerworld.com/s/article/9239625)*; there are links on that page to the other pieces in the series.]*

Of course that's a pretty no-frills graphic. If you'd like to label your x and y axes, use the parameters xlab and ylab. To add a main headline, such as "Page views by time of day," use the parameter main:

plot(mtcars$disp, mtcars$mpg, xlab="Engine displacement", ylab="mpg", main="MPG compared with engine displacement")

If you find having the y-axis labels rotated 90 degrees annoying (as I do), you can position them for easier reading with the las=1 argument:

plot(mtcars$disp, mtcars$mpg, xlab="Engine displacement", ylab="mpg", main="MPG vs engine displacement", las=1)

Adding a main headline and axes labels to an R plot.

What's las and why is it 1? las refers to **la**bel **s**tyle, and it's got four options. 0 is the default, with text always parallel to its axis. 1 is always horizontal, 2 is always perpendicular to the axis and 3 is always vertical. For much more on plot parameters, run the help command on par like so:

?par

In addition to the basic dataviz functionality included with standard R, there are numerous add-on packages to expand R's visualization capabilities. Some packages are for specific disciplines such as biostatistics or finance; others add general visualization features.

Why use an add-on package if you don't need something discipline-specific? If you're doing more complex dataviz, or want to pretty up your graphics for presentations, some packages have more robust options. Another reason: The organization and syntax of an add-on package might appeal to you more than do the R defaults

**Assigning values to variables**

In pretty much every other programming language I know, the equals sign assigns a certain value to a variable. You know, x = 3 means that x now holds the value of 3.

Not in R. At least, not necessarily.

In R, the primary assignment operator is <- as in:

x <- 3

But not:

x = 3

To add to the potential confusion, the equals sign actually *can* be used as an assignment operator in R -- but not all the time. When can you use it and when can you not?

The best way for a beginner to deal with this is to use the preferred assignment operator <- and forget that equals is ever allowed. Hey, if it's good enough for [Google's R style guide](http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html#assignment) -- they advise not using equals to assign values to variables -- it's good enough for me.

(If this isn't a good enough explanation for *you*, however, and you really really want to know the ins and outs of R's 5 -- yes, count 'em, 5 -- assignment options, check out the [R manual's Assignment Operators page](http://stat.ethz.ch/R-manual/R-patched/library/base/html/assignOps.html).)

One more note about variables: R is a case-sensitive language. So, variable x is not the same as X. That applies to pretty much everything in R; for example, the function subset() is not the same as Subset().

**c is for combine (or concatenate, and sometimes convert/coerce.)**

When you create an array in most programming languages, the syntax goes something like this:

myArray = array(1, 1, 2, 3, 5, 8);

Or:

int myArray = {1, 1, 2, 3, 5, 8};

Or maybe:

myArray = [1, 1, 2, 3, 5, 8]

In R, though, there's an extra piece: To put multiple values into a single variable, you need the c() function, such as:

my\_vector <- c(1, 1, 2, 3, 5, 8)

If you forget that c, you'll get an error. When you're starting out in R, you'll probably see errors relating to leaving out that c() a *lot*. (At least I certainly did.)

And now that I've stressed the importance of that c() function, I (reluctantly) will tell you that there's a case when you can leave it out -- if you're referring to consecutive values in a range with a colon between minimum and maximum, like this:

my\_vector <- (1:10)

I bring up this exception because I've run into that style quite a bit in R tutorials and texts, and it can be confusing to see the c required for *some* multiple values but not others. Note that it won't *hurt* anything to use the c with a colon-separated range, though, even if it's not required, such as:

my\_vector <- c(1:10)

One more very important point about the c() function: It assumes that everything in your vector is of the same data type -- that is, all numbers or all characters. If you create a vector such as:

my\_vector <- c(1, 4, "hello", TRUE)

You will *not* have a vector with two integer objects, one character object and one logical object. Instead, c() will do what it can to convert them all into all the same object type, in this case all character objects. So my\_vector will contain "1", "4", "hello" and "TRUE". In other words, c() is also for "convert" or "coerce."

To create a collection with multiple object types, you need a *list*, not a vector. You create a list with the list() function, not c(), such as:

My\_list <- list(1,4,"hello", TRUE)

Now you've got a variable that holds the number 1, the number 4, the character object "hello" and the logical object TRUE.

**Loopless loops**

Iterating through a collection of data with loops like "for" and "while" is a cornerstone of many programming languages. That's not the R way, though. While R does have [for, while and repeat loops](http://cran.r-project.org/doc/manuals/r-release/R-lang.html#Looping), you'll more likely see operations applied to a data collection using *apply()* functions or by using the *plyr()* add-on package functions.

But first, some basics.

If you've got a vector of numbers such as:

my\_vector <- c(7,9,23,5)

and, say, you want to multiply each by 0.01 to turn them into percentages, how would you do that? You don't need a for, foreach or while loop. Instead, you can create a new vector called my\_pct\_vectors like this:

my\_pct\_vector <- my\_vector \* 0.01

Performing a mathematical operation on a vector variable will automatically loop through each item in the vector.

Typically in data analysis, though, you want to apply functions to subsets of data: Finding the mean salary by job title or the standard deviation of property values by community. The apply() function group and plyr add-on package are designed for that.

There are more than half a dozen functions in the apply family, depending on what type of data object is being acted upon and what sort of data object is returned. "These functions can sometimes be frustratingly difficult to get working exactly as you intended, especially for newcomers to R," says a [blog post at Revolution Analytics](http://blog.revolutionanalytics.com/2009/12/why-use-plyr.html), which focuses on enterprise-class R.

Plain old apply() runs a function on either every row or every column of a 2-dimensional matrix where all columns are the same data type. For a 2-D matrix, you also need to tell the function whether you're applying by rows or by columns: Add the argument 1 to apply by row or 2 to apply by column. For example:

apply(my\_matrix, 1, median)

returns the median of every row in my\_matrix and

apply(my\_matrix, 2, median)

calculates the median of every column.

Other functions in the apply() family such as lapply() or tapply() deal with different input/output data types. Australian statistical bioinformatician Neal F.W. Saunders has a nice [brief introduction to apply in R](http://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/) in a blog post if you'd like to find out more and see some examples. (In case you're wondering, bioinformatics involves issues around storing, retrieving and organizing biological data, not just analyzing it.)

Many R users who dislike the the apply functions don't turn to for-loops, but instead install the plyr package created by Hadley Wickham. He uses what he calls the "split-apply-combine" model of dealing with data: Split up a collection of data the way you want to operate on it, apply whatever function you want to each of your data group(s) and then combine them all back together again.

The plyr package is probably a step beyond this basic beginner's guide; but if you'd like to find out more about plyr, you can head to [Wickham's plyr website](http://plyr.had.co.nz/). There's also a [useful slide presentation on plyr](http://www.stat.cmu.edu/~cshalizi/statcomp/11/lectures/14/lecture-14.pdf) in PDF format from Cosma Shalizi, an associate professor of statistics at Carnegie Mellon University, and Vincent Vu. Another [PDF presentation on plyr](http://streaming.stat.iastate.edu/workshops/r-intro/lectures/6-advancedmanipulation.pdf) is from an [introduction to R workshop](http://streaming.stat.iastate.edu/workshops/r-intro/) at Iowa State University.

**R data types in brief (very brief)**

Should you learn about *all* of R's data types and how they behave right off the bat, as a beginner? If your goal is to be an R ninja then, yes, you've got to know the ins and outs of data types. But my assumption is that you're here to try generating quick plots and stats before diving in to create complex code.

So, to start off with the basics, here's what I'd suggest you keep in mind for now: R has multiple data types. Some of them are especially important when doing basic data work. And some functions that are quite useful for doing your basic data work require your data to be in a particular type and structure.

More specifically, R has the "Is it an integer or character or true/false?" data type, the basic building blocks. R has several of these including integer, numeric, character and logical. Missing values are represented by NaN (if a mathematical function won't work properly) or NA (missing or unavailable).

As mentioned in the prior section, you can have a vector with multiple elements of the same type, such as:

1, 5, 7

or

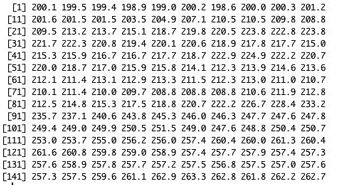
"Bill", "Bob", "Sue"

>

A single number or character string is also a vector -- a vector of 1. When you access the value of a variable that's got just one value, such as 73 or "Learn more about R at Computerworld.com," you'll also see this in your console before the value:

[1]

That's telling you that your screen printout is starting at vector item number one. If you've got a vector with lots of values so the printout runs across multiple lines, each line will start with a number in brackets, telling you which vector item number that particular line is starting with. (See the screen shot, below.)

If you've got a vector with lots of values so the printout runs across multiple lines, each line will start with a number in brackets, telling you which vector item number that particular line is starting with.

If you want to mix numbers and strings or numbers and TRUE/FALSE types, you need a list. (If you don't create a list, you may be unpleasantly surprised that your variable containing (3, 8, "small") was turned into a vector of characters ("3", "8", "small") ).

And by the way, R assumes that 3 is the same class as 3.0 -- numeric (i.e., with a decimal point). If you want the *integer* 3, you need to signify it as 3L or with the as.integer() function. In a situation where this matters to you, you can check what type of number you've got by using the class() function:

class(3)

class(3.0)

class(3L)

class(as.integer(3))

There are several as() functions for converting one data type to another, including as.character(), as.list() and as.data.frame().

R also has special vector and list types that are of special interest when analyzing data, such as matrices and data frames. A matrix has rows and columns; you can find a matrix dimension with dim() such as

dim(my\_matrix)

A matrix needs to have all the same data type in every column, such as numbers everywhere.

Data frames are like matrices except one column can have a different data type from another column, and each column must have a name. If you've got data in a format that might work well as a database table (or well-formed spreadsheet table), it will also probably work well as an R data frame.

In a data frame, you can think of each row as similar to a database record and each column like a database field. There are lots of useful functions you can apply to data frames, some of which I've gone over in earlier sections, such as summary() and the psych package's describe().

And speaking of quirks: There are several ways to find an object's underlying data type, but not all of them return the same value. For example, class() and str() will return *data.frame* on a data frame object, but mode() returns the more generic *list*.