Introduction to the R statistical environment

Devan Allen McGranahan & Carissa Wonkka

August 22, 2016

`R` is an environment designed for statistical computing, including data management, visualization, and analysis. It includes its own language in which developers write functions to perform tasks. Because of its free nature `R` users all over the world develop their own custom functions and can share them with other `R` users via packages. This expandability and the global network of users makes `R` appealing for researchers because one doesn't need to learn new programs or languages to perform new tasks. As a result, `R` functions are available for almost any type of data management or analysis task, from basic statistics to multivariate analysis, from GIS analysis and mapping to "big data" compilation and interactive visualization.

This chapter covers the basics of installing and interfacing with `R`, and provides an introduction to `R`'s data types and how to load them.

What is `R`?

One of the primary advantages of `R` is that it is free software, in both senses of the word[1]. Most users, especially beginners, appreciate that downloading, installing, and using `R` costs no money and requires no subscription or license fees. As one's use of—and reliance on—`R` develops, one appreciates how other users from around the world develop new packages with functions to perform a wide variety of tasks. Say you are confident in your ability to perform basic linear regression in `R` using the `lm()` function, but your professor (or a reviewer) says your data structure requires a mixed-effect linear regression; you'd definitely check out the `lmer()` function in the `lme4` package. Want to kick up the quality of your graphics? More and more `R` users are turning to `ggplot2`; soon you will be, too. Got rasterized spatial data? Better get package `raster`, or if you just want to make a cool map, the `sp` and `ggmaps` packages have what you need to bring your spatial data into the `ggplot` environment, all using `R`.

[1] In the software world, "free" can be thought of as "free beer" as well as "free" as in "free speech." `R` is both no-cost and open source.

R: A brief history

Sometimes it is helpful to know where `R` fits into the statistical computing universe. `R` has its roots in `S`, a proprietary language developed in the mid-1970's at Bell Labs. You will see this legacy in the `R` help files, with many citations to texts for the `S` language or hints on how `S` users should interpret `R` functionality. `R` emerged first from a collaboration between Ross Ihaka and Robert Gentleman at the University of Aukland, in New Zealand. The first version was established in 1993 and corresponding with the advent of the World Wide Web, `R` was shared with international collaborators as an open-source project in 1995. It became widely available with `R` version 1.0.0 in 2000[2].

[2] http://blog.revolutionanalytics.com/2016/03/16-years-of-r-history.html

WHEN COMPARING R TO OTHER STATISTICAL SOFTWARE, like `SAS`, `SPSS`, or `Stata`, users first think of the price tag. But there are substative differences that affect how one uses `R`, such as **object-oriented programming**, **graphics capacity**, and **vectorization**.

From here on out, `R` commands will be given in this heavier, italic type, preceded by `>`, which represents the prompt in the `R` command-line interface. You can copy and paste these commands into `R`; they were run by the program that made this `.pdf` file and automatically produced the non-italic output that follows. You will use a very similar technique for homework in this class.

Object-oriented programming is a key element of `R`, and refers to how `R` stores many different types of data, statistical results, and even graphics as *objects* represented by a word, phrase, or just a single character, which makes recalling these bits of information—and using them in subsequent analyses—quite simple.

For example, `R` can tell you that `2 + 2 = 4`:

```
> 2+2

[1] 4
```

`R` can also store the result, so you can refer to it later without having to re-calculate `2 + 2`:

```
> ans1 <- 2+2
> ans1

[1] 4

> ans2 <- ans1 * 2
> ans2

[1] 8

> sum(ans1, ans2)

[1] 12
```

Note the use of `sum()`, an `R` *function* pre-programmed to perform a specific task. In `R`, functions are always followed by paired parentheses, between which the *arguments* are supplied. Arguments take two forms: they are either the *objects* containing the data that fed into the function—in this case, all values of `d` are added together, or arguments deliver specific information to the function on how the function should operate or what the output should look like; see the examples in Fig. 1.

This might not seem impressive, but imagine you had a lot of numbers you needed to manage:

```
 [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] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

You probably don't want to type out `1 + 2 + 3 ...` just to add them together. It becomes very convenient to store all these numbers in a single object, which can be referred to again and again as other functions are used to provide information about all those numbers:

```
> d <- 1:50
> d
```

```
 [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] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

```
> sum(d)  # Add all those numbers together
```

```
[1] 1275
```

```
> mean(d) # Calculate the mean
```

```
[1] 25.5
```

Results of functions being applied to objects can in turn be stored as new objects:

```
> mu.d <- mean(d)
> mu.d
```

```
[1] 25.5
```

In this way, even very large datasets can move through `R` with just a little bit of code, and through the re-naming of objects as changes are applied, one can always go back and use previous steps and apply new functions, or even combine them all, for example, in a graph like Figure 1.

```
> boxplot(d, las=1)
> points(mu.d, pch=17, cex=3, col="blue")
```

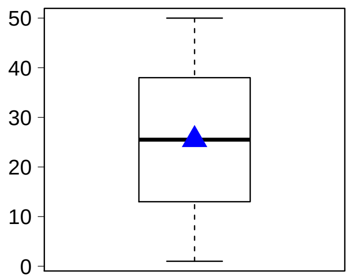

Figure 1: The `boxplot()` function creates a boxplot based on the 50 values stored in `d`, with some aesthetic improvement from the `las=1` argument, which rotates the y axis labels. Next, the `points()` function adds the mean of the 50 values—stored as `mu.d`—with arguments to specify the shape (`pch=`), size (`cex=`, for "character expansion" factor), and color (`col=`) of the point.

Graphics capacity As we've just seen in Fig. 1, `R` has a simple and robust capacity to create graphs. Even the default settings of functions in the built-in `graphics` package like `plot()` are good enough such that with few modifications—through arguments like those demonstrated in Fig. 1 or functions that add additional layers like `points()` or `legend()`—these quick graphs are near publication-ready and certainly suffice for your own visualization or presenting preliminary data to your professor. And as mentioned above, additional packages like `ggplot2` take `R` graphics to a whole new level (although some additional coding skills are required).

Even default `R` graphics are competitive with those of two main types of programs: conventional statistical programs like `SAS`, who long had `ASCII`-based graphic output

Vectorization This difference between R and other computer languages won't make a difference until you start writing some code of your own, and probably won't make sense unless you've had some experience with other languages, like maybe Java in a web programming class or C++ and the like in a computer science course. Vectorization refers to how R handles the information you give it, and how it performs the operations you request of it.

VECTORIZATION IS AT THE HEART of the object-oriented functionality that makes R easy to use. Let's take this simple example of addition[3], in which we have two sets of five numbers that we want added—not all together, but we want the first number of row 1 added to the first number of row 2, and the second of each added together, and so on:

[3] https://www.r-bloggers.com/how-to-use-vectorization-to-streamline-simulatio

Note that putting parentheses around a command tells R to both perform the operation and return the result automatically. It is the same as calling x right after the object is assigned.

```
> (x <- 1:5)

[1] 1 2 3 4 5

> (y <- 6:10)

[1]  6  7  8  9 10

> (z <- x + y)

[1]  7  9 11 13 15
```

How would this be done in a non-vectorized language? The script would have to include just what to do, how many times to do it, and what to do with the result. This little function tells the computer, for every case i, ordered one through five, take the first one from x and the first one from y, add them together, and store it in z:

An important point to make here is that though R is optimized for vectors, it doesn't need to use them. R runs all sorts of loops, and if you know how to write them, by all means use them.

```
> x <- 1:5
> y <- 6:10
> z <- NULL
> for (i in c(1:5)) {
+   z <- c(z, x[i] + y[i])
+ }
> z

[1]  7  9 11 13 15
```

Why does any of this matter? Brevity and speed. Vectorized functions are usually shorter; they take up fewer lines of code in your text window and can be easier to read/de-bug when you have errors (you will have errors!). In terms of speed, on modern computers the few miliseconds longer it takes to add a few digits with a loop amounts to nothing. But as datasets get larger and operations more complex, the speed gains from vectorized functions can add up to minutes or even hours.

How do we tell how long an operation takes, beyond timing R with our watch? R provides us with a function called `system.time` that returns the amount of time it spent chugging:[4]

[4] http://alyssafrazee.com/vectorization.html

```
> # Start with the default log function:
> log(3)
```

```
[1] 1.098612
```

```
> log(1:10)
```

```
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
```

```
> # a vector of 1 million random numbers between 1 and 10
> nums <- sample(1:10, size=1000000, replace=TRUE)
> # A custom function to call log on each vector element separately:
>  log_novec = function(n){
+    ret = rep(NA, length(n))
+    for(i in seq_along(n)){
+      ret[i] = log(n[i])
+    }
+   return(ret)
+ }
> # timing results:
> system.time(log_novec(nums)) # the for-loop function
```

```
   user  system elapsed
  1.944   0.028   2.144
```

```
> system.time(log(nums)) # the default, vectorized function
```

```
   user  system elapsed
  0.052   0.000   0.052
```

Result: The vectorized function is 100 times faster.

Getting started with R

The first step is to download and install the latest version of R, from the Comprehensive R Archive Network, or CRAN: https://cran.r-project.org. The CRAN serves two main purposes: maintaining and updating the basic installation of R for everyone, and curating the ever-growing library of user-developed packages that expand on base R's functionality. R is hosted by various "mirrors" all over the world; we usually download from the one nearest us.

Operating R

Once R is on your machine you need to have a way to interact with it. Humans interact with computers in two main ways: either through the command line, where the only interaction is through code, or through a Graphical User Interface, or GUI, which has icons, buttons, toolbars, menus and pop-up windows. Almost all programs the average PC or Mac user interacts with is done through the GUI. In the stats world, the statistical program JMP is essentially a GUI for SAS; standard SAS code has been pre-programmed into various menu options so one can perform SAS analyses but not need to program SAS code.[5]

[5] R has a similar GUI called R commander that allegedly makes it possible to be all clicky-clicky with R, but we think that's lame.

Why not use a GUI all the time? First of all, one is stuck with the options given by the programmer of the GUI, which might not include all of the options available in the original program, although some programs allow the user to "go behind the curtain" and circumvent the GUI by adding custom code, such as writing Python scripts for ESRI products. Secondly, it becomes repetitious to perform the same sequence over and over again. It might be convenient to spend an afternoon clicking through your analysis and making a nice graph, but what if you need to do the same to different data—or, more likely, go back and fix a mistake—do you want to spend another afternoon trying to remember the same sequence of clicks, or simply edit a text file and hit "run" on the code you (hopefully) saved from the day before? *We use R to combine the powers of customization and repeatability.*

All that said, however, unless one is a hardcore LINUX command line user, one uses some level of GUI to tell R what to do. Opening the default installation of what is technically called the R environment opens in a basic GUI that includes the console—the command line where code goes in and output (and errors) are returned—and a toolbar of standard menu options: `File`, `Help`, and the like. These menus contain several frequently-used commands, such as those used for loading additional packages. In the default R GUI, one can click `Packages... > Install packages...`, select a mirror, and scroll through a list of available packages. The same process can be accomplished with a single line to the command line, as well:

```
> install.packages("ggplot2")
```

GUI options for R The default GUI is fully functional but it is pretty bare-bones. Several other solutions have emerged that mostly focus on making it more convenient for users to write and manage their code. The first two improvements users find useful are in general computer programmer applications that "know" the R language and color-code standard functions and help keep track of parentheses and brackets as they are opened and closed. These programs also allow users to have several script files open at once and manage them with tabs. Auto-saving features are also very nice! But these general programs—`Notepad++` is a great example—need extra add-ins to interface with `R`—in the case of `Notepad++` code is shuffled to `R` via a little background program called `npptoR`.

The solution that most `R` users migrate to sooner or later is `Rstudio` (`https://www.rstudio.com`), and we highly recommend it. Although `Rstudio` has all sorts of features designed to help app and package developers and big-data analysts—some of those features brought you this document and will help you with your homework assignments—the beginner will enjoy the simple layout that helps keep things from getting lost on the screen.

Loading data

This section summarizes the "R Data Import/Export" help manual available on CRAN at `https://cran.r-project.org/doc/manuals/r-release/R-data.html`. This and other extensive help manuals are also available through your `R` installation, just run `help.start()` in the console.

NOTHING IS A GREATER BARRIER TO USING `R` THAN NOT GETTING ONE'S DATA IN CORRECTLY, so we start with a review of the different kinds of data and file types `R` can load, the options users have for getting these data in to `R`, and how `R` handles these data types once they are loaded.

Data types and file formats "File format" basically means "encoding," which describes how the program that created the original file mapped human-readable information to computer-readable information. A very simple encoding is basic ASCII characters, found in very basic text files with no formatting and old-school, type-writer like fonts.[6] As programs add complexity to files—think of the difference between a basic text file and a MS Word document with fonts, pictures, double-spacing, margins, etc.—the encoding necessary to convey all that information to the computer becomes more elaborate, program-specific, and often proprietary.

[6] ASCII stands for American Standard Code for Information Interchange, and is simply a means of creating numerical, computer-readable mapping, or encoding, of keyboard inputs like letters and punctuation.

There are two basic strategies to get `R` to read your data:

- Convert the file from the "native" encoding—the format it was created in by the original program—to one familiar to `R`,
- Use a function in `R` specifically designed to read the file's encoding and convert it to an `R` object.[7]

[7] There are several of these built to not only read files created in other stats programs like `SPSS` and `SAS`, but also to read complex data files like `.shp` GIS shapefiles, etc. We'll get into those later.

Not surprisingly, a basic text file comprised of ASCII characters is the simplest file format read by `R` and also one of the most reliable. While "basic text file" often brings to mind files with a `.txt` extension, when we talk data, we talk `.csv` files. These are files comprised of "comma-separated values" in which rows are individual records and values for each column are kept separate by commas. When `R` reads a `.csv` file, `R` knows to line up columns by what's within the commas.[8]

[8] There are of course some quirks that can arise, especially if values contain commas, like in many European locales where they use a comma instead of a decimal point! But this can be handled by the encoding and as an international language, `R` has functions that handle these encodings.

> *The `.csv` file is the average `R` user's basic currency of data input and output.*

Moving data from Excel to `R` Most graduate students, researchers, and almost everyone uses MS Excel at some point in their data workflow, for entry, storage, sorting and summarizing, and even —shudder—graphing. We will work quickly to rescue your graphs from Excel and will provide you with tools to move completely out of Excel, but the fact remains that Excel is popular (or at least widely used) and is actually more powerful than a lot of folks give it credit for. So you will likely put your data into Excel, and if someone sends you data, it will come in an Excel file.

While `R` has some features that can go into native Excel formats and fetch your data, standard practice is to export a single Excel sheet as a `.csv` file before feeding it to `R`.[9] Remember that although you see your data as a table of rows and columns in Excel, this interaction is a result of Excel's GUI. An Excel file is not simply a spreadsheet, it is a workbook of several spreadsheets each with formatting and functions and macros that `R` doesn't have any idea what to do with.

[9] Prior to this step you will likely clean up and arrange your data in Excel using the menus and options provided by Excel's GUI rather than try to manipulate your data via code in `R`, but as you become more familiar with `R` you'll find that you can, and will want to, get your data out of Excel and into `R` as quickly as possible.

It is easy to get your Excel data into `.csv` format:

- In Excel, once you have your rows and columns as you want them in your spreadsheet, click `File...` > `Save as...`
- Navigate to the appropriate folder in which you want to save your data for `R`.
- Select the file type drop-down list arrow. Scroll down until you find "`.csv`," and select it.
- Give it a useful file name and save it. Excel will ask you if you really want to do this, with a warning about losing formatting and other features. Try not to feel bad for Excel, but remember, all of Excel's additional crap is exactly why we need to use the `.csv` format!

As you begin to learn `R`, we highly recommend creating a folder called "R" on your desktop with three sub-folders: "Data", "Code", and "Figures". As you will see shortly, this standardization makes adopting our code here much easier.

It is just as easy to get your data into `R` once it has been saved as `.csv`. However, there are two options: One can either tell `R` exactly where the file is by providing the whole path...[10]

```
> ed <- read.csv("C:/user/Desktop/R/data/export_data.csv")
```

...or one can prompt `R` to launch a little GUI window that lets one navigate to where the file is saved. *While this is convenient, remember, like all GUI features, it isn't repeatable*, whereas the code above can be run through the console over and over again.

[10] To get the whole path, go to where you saved your file and open the Properties window by right-clicking on the file, or whatever the Mac equivalent of right-clicking is. **Note slash direction**; `R` will either accept forwardslashes / or double backslashes \\ but not single backslashes \, the Windows default.

```
> ed <- read.csv(file.choose())
```

The `read.csv()` function is the average R user's workhorse for data input and output.

Using working directories One ends up using `read.csv()`, its sibling `write.csv()`, and several other functions (e.g. `load()`, `save()`, `jpeg()`, etc.) so frequently that it can be a pain to enter complete file pathways over and over again, especially if it all uses the same folder. Thus one can set a *working directory* once, which shortens future commands:[11]

[11] Note how the directory is replaced by a period in subsequent commands.

```
> setwd("C:/user/Desktop/R/")
> ed <- read.csv("./data/export_data.csv")
> save(ed, file="./data/ed.Rdata")
> load(file="./data/ed.Rdata")
```

If you ever need to know what your working directory is, simply run

```
> getwd()
```

Working with data in R

Once your data have successfully been imported into R, you need to know how R handles them. Once in R, your data become an *object*, which you should recognize from earlier in the chapter when we discussed how R is *object-oriented*; whereas at the beginning of the chapter we created objects by assigning values generated in the R environment—`answer <- 2 + 2`—here we've imported data and called it `ed`.

R comes pre-loaded with a lot of different datasets that we use for examples and you can use to practice on. To see all the different data available, enter `data()` in the console.

We can explore objects to ensure we brought in the correct file, or that the import process was successful. Here we use an example dataset that comes with R to:

Confirm the dimensions of the dataset `cars`:

```
> dim(cars)
```

```
[1] 50  2
```

The dataset has 50 rows and 2 columns. Let's look at the column names:

```
> names(cars)
```

```
[1] "speed" "dist"
```

Check out the first five rows of data:

```
> head(cars)
```

```
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
6     9   10
```

Check out the last five rows:

```
> tail(cars)
```

```
   speed dist
45    23   54
46    24   70
47    24   92
48    24   93
49    24  120
50    25   85
```

Objects and their classes Each object is assigned to a certain *class* that dictates how various `R` functions treat the object. Frequent classes `R` users deal with include `data.frame`, `matrix`, and `ts` for time series data. There are others, and some speciality functions have their own object classes, but most `R` users use data in class `data.frame` or convert to a `data.frame` before proceeding.

Class awareness is important because sometimes one function will return an object of one class while the next function you run expects an object of a different class, and you will not get the results you want no matter how many times you check spelling, parentheses, etc. If you think you coded something correctly and still get weird errors, check your object class!

The function `class()` tells you what class an object is and functions that begin with `as.` convert between classes (when possible to do so):

```
> class(cars)
```

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

```
> cars.m <- as.matrix(cars)
> class(cars.m)
```

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

```
> (example.matrix <- matrix(
+     c(2, 4, 3, 1, 5, 7),
+     nrow=3, ncol=2))
```

```
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
```

```
> class(example.matrix)

[1] "matrix"

> (ex.m.d <- as.data.frame(example.matrix))

  V1 V2
1  2  1
2  4  5
3  3  7

> class(ex.m.d)

[1] "data.frame"
```

Classes within objects This section is essential to understand prior to graphing data and performing statistical tests. Because `R` is primarily a statistical language, it makes sense that the identities of data within objects relate to how they are treated in statistical models. Also, because most objects you will use are of class `data.frame` or you'll convert them to be so, we focus here on classes within the `data.frame` object.[12]

[12] The official introduction to `R` describes `data.frame` as a type of list, and describes classes within list objects, but here we are going to be more clear, and perhaps technically incorrect, by focusing on `data.frame`. https://cran.r-project.org/doc/manuals/r-release/R-intro.html#Data-frames

`R` stores objects as ordered collections of items known as *components*. This ordering allows items to be referred to by number. So in a `data.frame` every row, column, or individual item—think a cell in a spreadsheet—can be referred to specifically, or as a group, by a number or range of numbers, respectively. Specific operators are assigned to these tasks and contribute to the syntax of the `R` language (See Table 2 for some of the common `R` operators and their uses). For example, columns within a `data.frame` can be called either by column name with a dollar sign (`$`), or by column number, with a bracketed numeral:

```
> head(cars) # Top six rows of both columns

  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
6     9   10

> head(cars$speed) # Top six rows of column "speed"

[1] 4 4 7 7 8 9

> head(cars[1]) # Top six rows of column "speed"
```

```
  speed
1     4
2     4
3     7
4     7
5     8
6     9
```

We can refer to a specific row, or range of rows, as well, although the syntax is slightly more awkward.[13] We need to map to the x dimension with a comma, as if we were thinking of the `data.frame` as a spreadsheet or map with x,y coordinates. This coordinate mapping syntax extends in both directions:

[13] This is probably because one rarely needs to do it. Rows represent units of observations, and if a group of observations needs to be extracted (to be focused on) or removed (to not be used at all), this is best done by defining a subset of an identifying variable with its own column.

```
> cars[1,] # Display row 1
```

```
  speed dist
1     4    2
```

```
> cars[1:3,] # Display rows 1 through 3
```

```
  speed dist
1     4    2
2     4   10
3     7    4
```

```
> cars[3:5,] # Display rows 3 through 5
```

```
  speed dist
3     7    4
4     7   22
5     8   16
```

```
> cars[5,2] # Display component located in row 5, column 2
```

```
[1] 16
```

The `data.frame` is essentially an object made up of one or more objects, each with their own components. This isn't confusing, it is convenient, because it allows for the *combination of different classes of objects* into one object—the `data.frame`. The different classes within an `R` object relate generally to the nature of the components—say, text *vs.* numbers—and more specifically to the type of variable those data might be treated as in a statistical model—say, categorical *vs.* continuous variables (Table 1).

Identifying classes within an object works the same as with the objects themselves, but the `str()` function is an even better choice because it gives us all the information we've requested individually from functions like `dim`, `head`, and `class`:

```
> class(cars$speed)
```

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

```
> str(cars)
```

```
'data.frame':        50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

`R` assigns classes upon import and seems to base assignments on whether the data are text, whole numbers, or numbers with decimals—class `factor`, `integer`, or `numeric`, respectively. But sometimes the assignment doesn't match the information carried in the data, for example, if one entered 1 or 0 for presence/absence, or on/off, etc, `R` will read these as whole numbers and assign them to class `integer` when one really wants them treated at categorical variables of class `integer`. Let's look at examples from a new dataset, `mtcars`:

```
> str(mtcars)
```

```
'data.frame':        32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

All are assigned as numeric values, which makes sense because they are numbers, but doesn't necessarily make sense in terms of how we think of the data. For example, the number of cylinders in a car engine is neither a continuous variable—one cannot have 5.5 cylinders—nor is it truly a discrete variable because it is fairly limited in

`R` class	Variable type	Description
`numeric`	continuous	Measurements like mass or length
`integer`	discrete	Counts of things
`factor`	categorical	Non-numerical groups; On/Off, Male/Female
`character`	n/a	Text

Table 1: Summary of `R` component classes matched to type of variable often used in statistics.

possible range—most internal combustion engines in consumer vehicles are confined to four, six, or eight cylinders, with a few 10-cylinder pick-ups and of course 12-cylinder Ferraris. So `cyl` is best handled as a categorical variable of class `factor`. The same applies to the transmission type (`am`), number of gears in the transmission (`gear`), and number of carburetors (`carb`).[14] Changing classes within objects is the same as before, with `as.` functions:

[14] Information about these data are available in default `R` documentation via `?mtcars` in the console.

```
> mtcars$cyl <- as.factor(mtcars$cyl)
> mtcars$vs <- as.factor(mtcars$vs)
> mtcars$am <- as.factor(mtcars$am)
> mtcars$gear <- as.factor(mtcars$gear)
> mtcars$carb <- as.factor(mtcars$carb)
> str(mtcars)
```

```
'data.frame':        32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : Factor w/ 3 levels "4","6","8": 2 2 1 2 3 2 3 1 1 2 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : Factor w/ 2 levels "0","1": 1 1 2 2 1 2 1 2 2 2 ...
 $ am  : Factor w/ 2 levels "0","1": 2 2 2 1 1 1 1 1 1 1 ...
 $ gear: Factor w/ 3 levels "3","4","5": 2 2 2 1 1 1 1 2 2 2 ...
 $ carb: Factor w/ 6 levels "1","2","3","4",..: 4 4 1 1 2 1 4 2 2 4 ...
```

Table 2: Some `R` operators used to refer to object components.

`R` operator	Description	Example
`#`	Comment ignored by R	`# Hidden from R`
`?`	When followed by a function name, pulls up function's help file	`?data.frame`
`=` & `<-`	Assigns the right-hand side to an object named in the lefthand side	`my.df <- data.frame(column1= c('A','B'), column2=c(1:2))`
`$`	Refer to a column by name	`my.df$column1`
`[ ]`	Refer to a column or row by number	`my.df[2,]` `my.df[,2]`