layout: post title: "Lesson 02 - Basic Operations"

categories: jekyll

In this lesson we will focus on reading and handling data, and what you need to know in order to do so.

In this lesson, we will learn about:

- Data types
- Reading and writing data frames
- How to read help manuals for functions
- How to start writing scripts

Data types

1. Vectors
2. Matrices
3. Lists
4. Data frames

Vectors

A vector is a one-dimensional list of values. The values in this list have a **type**: either **numeric**, **character**, **factor**, or **logical**. All values in a vector will have the same type (whether you want them to or not).

Numeric vectors

All values in a vector are a number, or missing (`NA` or `NaN`).

```
> c(1, 2, 3, 4)
[1] 1 2 3 4
> c(1, 2, NA, 4.5)
[1] 1.0 2.0  NA 4.5
```

Character vectors

All values in a vector are a string (or missing).

```
> c('VA', 'NC', 'SC')
[1] "VA" "NC" "SC"
> c('VA', 'NC', 'SC', 200)
[1] "VA"  "NC"  "SC"  "200"
```

(Strings can look like numbers to the human eye. R doesn't care that the 200 *could* be a number, because it's with a bunch of other strings. You can coerce strings to become numbers with the function `as.numeric`, e.g. `as.numeric("200")`.)

Factors

Factors look like strings but have a specific function: to store different levels of a variable, such as experimental conditions.

```
> factor(c('M', 'F', 'M', 'F', 'M', 'F'))
[1] M F M F M F
Levels: F M
```

Logical vectors

Vectors can be made up of logicals: `TRUE` and `FALSE` values.

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

`TRUE` is not the same as `"TRUE"` or `"True"`. Special values such as `TRUE`, `FALSE`, `NA`, `NaN`, and `NULL` are case-sensitive and not placed in quotes.

Matrices

Matrices are two-dimensional vectors. Like vectors, they can only contain one type of data.

You can create matrices with functions such as `matrix`, `rbind` (binds vectors as rows), and `cbind` (binds vectors as columns).

```
> mat <- rbind(c(7, 8, 9, 10), c(1, 2, 3, 4), c(-1, -2, -3, -4))
> mat
     [,1] [,2] [,3] [,4]
[1,]    7    8    9   10
[2,]    1    2    3    4
[3,]   -1   -2   -3   -4
```

You can extract single values, rows, columns, or matrices from a matrix.

```
> mat[1, 4]
[1] 10
> mat[1, ]
[1]  7  8  9 10
> mat[, 4]
[1] 10  4 -4
> mat[1:2, 2:4]
     [,1] [,2] [,3]
[1,]    8    9   10
[2,]    2    3    4
```

Matrices have their uses but data frames can contain more information.

Lists

Lists are, well, lists of things. You can mix different data types in a list--you can have numeric values, strings, even data frames stored in a single list. They are versatile dumping grounds for information.

```
> list.example <- list(TRUE, 1, c(2, 3), 'four')
> list.example
[[1]]
[1] TRUE

[[2]]
[1] 1

[[3]]
[1] 2 3

[[4]]
[1] "four"
```

You can access elements of a list with double brackets, rather than single brackets such as for vectors and

matrices.

```
> list.example[[3]]
[1] 2 3
```

Data frames

Data frames store information!

For a data frame: Each row is an instance of an **observation** (such as a person, day, or flower), and each column is a **variable** (such as weight, temperature, or petal length). Each column has a name. Row names are optional (but personally, I do not like them).

R has some built-in data sets; let's look at one called `iris` now.

```
> 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
```

`head` gives you the first six rows of a data frame. Here, we see that there are four measurements for each flower, as well as an additional column for the species.

```
> 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 1 ...
```

`str` provides the structure of the data frame. The first four columns are numeric, and the last column is a factor.

Here, though, we can't see what the third level of the Species factor is! Fortunately, we can extract individual columns from a data frame. And unfortunately, there are multiple ways to do this.

The best way is with the `$` operator: `iris$Species`.

```
> levels(iris$Species)
[1] "setosa"     "versicolor" "virginica"
```

Other ways to perform the same operation are `iris[['Species']]`, `iris[, 'Species']`, and `iris[, 5]`.

Examining data frames

We already learned the `head` function. Its corresponding function, `tail`, gives the last six rows of a data frame.

Functions `dim` (dimensions), `nrow` (number of rows), and `ncol` (number of columns) report the size of the data frame or matrix.

```
> dim(iris)
[1] 150   5
> nrow(iris)
[1] 150
> ncol(iris)
[1] 5
```

You can also examine the column names through the `names` and `colnames` function, and row names through `rownames`.

What can we do with data stored in a data frame?

Storing data in a data frame allows us to efficiently examine it, filter it, and run statistical analyses. For example, we can make some basic plots with this data:

```
> hist(iris$Petal.Length)
> boxplot(iris$Petal.Length ~ iris$Species)
```

We will learn more about plotting next week.

We can also filter data with the `subset` function (as well as logical vectors, which we will cover in Week 4). The subset allows us to select only certain rows from a data frame. For example, we can only select flowers that have a petal width greater than 2:

```
> subset(iris, Petal.Width > 2)
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
101          6.3         3.3          6.0         2.5 virginica
103          7.1         3.0          5.9         2.1 virginica
105          6.5         3.0          5.8         2.2 virginica
...
149          6.2         3.4          5.4         2.3 virginica
```

Reading data

You will probably want to load your own data of various types into R. Storing data in plain text files is a great way to share data, because then other users can use your data with any software.

Text file formats

You will likely encounter three different formats of plain-text data:

- Whitespace-delimited values
- Tab-separated values (tsv)
- Comma-separated values (csv)

All these files contain rows of observations with the same number of columns in each row. They only differ in the column delimiter.

These formats are not strict, however. You may find variations such as: whether or not a file includes headers (column names), whether it includes row names, what indicates a missing value, etc.

read.table

The R function `read.table` reads a plain-text file into a data frame.

The ? operator and reading help files

You can load the help for a function by using the `?` operator, e.g. `?read.table`. Try loading the help for read.table.

Reading the help manuals in R is a skill unto itself. They have all the information you need, but not necessarily clearly written.

First, there is a description of what the function does:

> Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Then, there is an example function:

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
           row.names, col.names, as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)
```

Each of the elements of the function call is an **argument**. Arguments within a function call are assigned with an equals sign. (They can also be assigned without the equals by their order.) The values of each argument listed here are the defaults.

This function has a ton of arguments! Fortunately, they are described later in the help manual. Some examples:

> `header` : a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: header is set to TRUE if and only if the first row contains one fewer field than the number of columns.
>
> `sep` : the field separator character. Values on each line of the file are separated by this character. If sep = "" (the default for read.table) the separator is 'white space', that is one or more spaces, tabs, newlines or carriage returns.
>
> `na.strings` : a character vector of strings which are to be interpreted as NA values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields.
>
> `stringsAsFactors` : logical: should character vectors be converted to factors? Note that this is overridden by `as.is` and `colClasses` , both of which allow finer control.

We also discover from the help manual the functions `read.csv` and `read.delim` , which read CSV files and TSV files, respectively. They are the same function as `read.table` , but with different default arguments.

And, at the bottom of the help page, there are examples of how the function can be used.

Working directory

R sessions exist in a working directory. You can find out which directory you're in with `getwd()`, and change your working directory with `setwd('DirectoryName')`. Within RStudio, you can also change your working directory with the menus: Session -> Set Working Directory -> Choose Directory.

Reading a CSV file

Can you read AirQuality.csv into your environment?

```
> air.qual <- read.csv('AirQuality.csv')
> str(air.qual)
'data.frame':   153 obs. of  6 variables:
 $ Ozone  : Factor w/ 68 levels "-","1","10","108",..: 36 31 9 16 1 24 21 17 59 1 ...
 $ Solar.R: Factor w/ 118 levels "-","101","112",..: 29 5 17 88 1 1 85 118 28 33 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
> head(air.qual)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5     -       - 14.3   56     5   5
6    28       - 14.9   66     5   6
> hist(air.qual$Ozone)
Error in hist.default(air.qual$Ozone) : 'x' must be numeric
```

Whoops! Because a hyphen was used as a missing value, and R wasn't expecting that, the `Ozone` and `Solar.R` columns were read in as factors, when they are numeric information and should be read in as numbers.

Here, it's a good thing we've learned about the `read.table` (and `read.csv`) options. How can we handle this situation to read in the table correctly?

Missing data

Most data sets have missing data points, and R has functions to handle them.

`is.na` returns a logical vector--elements that are `NA` return `TRUE`, and non-missing values return

`FALSE` --e.g., `is.na(air.qual$Ozone)`.

Based on what we learned with `subset`, how can we select only only rows from `air.qual` where data is missing for `Solar.R`?

Many functions return a missing value when given a data set with some missing data (e.g. `mean(air.qual$Ozone)`). If you think that removing missing data is scientifically justified, you can use the `na.rm` argument that many functions have.

```
> ?mean
> mean(air.qual$Ozone)
[1] NA
> mean(air.qual$Ozone, na.rm=TRUE)
[1] 42.12931
```

Scripting

We have learned a lot about what we can do interactively in R--but for reproducibility, we can save a series of commands in a *script*. This script can be saved, edited, and re-run as many times as you want.

To create a new script in RStudio, you can go to: File -> New File -> R Script. Type any commands you like. Finally, you can save the script by going to File -> Save.

You can run the whole script by clicking the "Source" button in the console. Alternatively, you can run specific lines by highlighting them and pressing Cmd+Enter (Mac) or Ctrl-Enter (Windows).

My typical workflow for analyzing data involves going back and forth between the console and the source code. I use the console to make sure I'm executing the steps correctly, and keeping the correct code in my source script.

Scripting also lets you annotate your process, so that your collaborators (or Future You) understands what you were trying to do at each step of your code. You can add comments to your code with the pound sign:

```
# This is a comment
head(air.qual)  # comment starts here
```

Everything after the pound sign (or "hashtag") is a comment, but anything before it will be executed.

Dealing with errors

You *will* have errors in your code, all the time. These can fall under certain categories:

- Things the computer recognizes as errors

```
> head(airqual)
Error in head(airqual) : object 'airqual' not found
> haed(air.qual)
Error: could not find function "haed"
```

- Things the computer thinks might be an error--so it gives a warning
- Things that the computer doesn't recognize as an error, but doesn't work as intended

```
hist(air.qual$Ozone)
abline(v = mean(air.qual$Ozone))
```

When you have an error in your code, follow these instructions:

1. Don’t panic.
2. Guess why it is happening.
3. Check if your guess is correct.
4. Repeat steps 1 through 3 as necessary.[1]

Google is your friend here! You can search for the error message, or what you're trying to do, and include "in R" or "Rstats" in your search terms.[2]

You can also restrict your search to StackOverflow, or use the Rseek search engine, which only searches on websites related to R.

Most of coding in R is (a) reading the R help and (b) searching for R help.

Nomenclature

Variables in R are typically lowercase, and can include periods: e.g. `example.variable`. (Most languages do not use periods in variable names, but R is an exception.) Functions can include periods but are also sometimes in "camel case": `example.function()` or `exampleFunction()`.

When copying + pasting a code example or output, using a fixed-width font, such as Courier New, indicates

that you are displaying code or plain-text output (like you see in these lesson plans).

Homework

- Find the mean and median of solar radiation levels from the air quality data set.
- Save the `iris` data set into a text-delimited format (hint: `?write.table`. What does it look like when opened with a text editor? What does it look like when opened with a spreadsheet program like Excel?

Resources

- R for cats
- Software Carpentry: R for reproducible scientific analysis
- Google R style guide

---

1. Source: http://www.burns-stat.com/documents/tutorials/impatient-r/more-r-errors-and-such/ ↩

2. One of the minor downsides of R is its name being a single character, which makes searches slightly more difficult. ↩