Base R tutorial

Experiential Data science for Undergraduate Cross-disciplinary Education

Dr. Kim Dill-McFarland, U. of British Columbia

Contents

Base R tutorial

Learning objectives

- Execute commands in base R to:
  - Load tabular data
  - Access columns and rows within a data frame
  - Perform basic calculations on tabular data
  - Subset a data frame

Setup

If you would like to follow along, open a new RStudio session and create a Project. Download the data file `data.csv` using the following command in RStudio. If you would like to save code/notes, also start a new R script.

```
write.csv(
  read.csv("https://raw.githubusercontent.com/EDUCE-UBC/workshop_data/master/data.csv"),
  "data.csv", row.names=FALSE)
```

Not sure what a Project is? Be sure to include the "RStudio tutorial" in your materials!

If you would like to learn more about Saanich Inlet and these data, checkout our description.

Load data

Read in tabular data

To start analyzing data, we need to read it into RStudio. There a number of functions to do this; we will start with the most generic, `read.table()`. As with all R functions, you specify the function name and then enclose any specific parameters or arguments within parentheses.

```
Function(arugument1=..., argument2=..., ...)
```

For example, we load the file `data.csv` like so.

```
read.table(file="data.csv")
```

```
##        V1                            V2
## 1  Season ,"Depth_m","O2_uM","Add_data"
## 2    Fall              ,10,203.533,TRUE
## 3    Fall             ,20,183.787,FALSE
## 4    Fall             ,40,130.579,FALSE
## 5    Fall               ,60,91.115,TRUE
## 6    Fall              ,75,69.828,FALSE
## 7    Fall              ,85,26.972,FALSE
## 8    Fall               ,90,11.066,TRUE
## 9    Fall               ,97,8.997,FALSE
## 10   Fall              ,100,6.605,FALSE
## 11   Fall              ,110,5.933,FALSE
## 12   Fall               ,120,2.891,TRUE
## 13   Fall              ,135,2.766,FALSE
## 14   Fall             ,150,14.465,FALSE
## 15   Fall             ,165,24.239,FALSE
## 16   Fall             ,185,28.885,FALSE
## 17   Fall              ,200,26.766,TRUE
## 18 Summer              ,10,216.667,TRUE
## 19 Summer             ,20,159.672,FALSE
## 20 Summer             ,40,141.778,FALSE
## 21 Summer               ,60,97.894,TRUE
## 22 Summer              ,75,44.978,FALSE
## 23 Summer              ,85,25.807,FALSE
## 24 Summer               ,90,27.011,TRUE
## 25 Summer              ,97,34.436,FALSE
## 26 Summer             ,100,38.012,FALSE
## 27 Summer             ,110,27.557,FALSE
## 28 Summer              ,120,32.354,TRUE
## 29 Summer             ,135,20.446,FALSE
## 30 Summer                  ,150,0,FALSE
## 31 Summer                  ,165,0,FALSE
## 32 Summer                  ,185,0,FALSE
## 33 Summer                   ,200,0,TRUE
```

Here, we've told R our `file`. Since we are using a Project, R looks for this file in the Project directory. If we wanted to specify a file not in our Project folder, we would need to provide the full path like if the file were on my computer's Desktop.

```
read.table(file="/Users/kim/Desktop/data.csv")
```

In either case, we see the data appear in the R console. However, they are not formatted correctly for further analysis.

Add arguments

We can specify other features of the data using additional arguments in the `read.table()` function. In the case of these data, we want to use:

- header: tells R that the first row is column (variable) names, not data
- sep: tells R that the data are comma-delimited `","`. If you had a tab-delimited file, you would use `sep="\t"`.

```
read.table(file="data.csv", header=TRUE, sep=",")
```

```
##    Season Depth_m   O2_uM Add_data
## 1    Fall      10 203.533     TRUE
## 2    Fall      20 183.787    FALSE
## 3    Fall      40 130.579    FALSE
## 4    Fall      60  91.115     TRUE
## 5    Fall      75  69.828    FALSE
## 6    Fall      85  26.972    FALSE
## 7    Fall      90  11.066     TRUE
## 8    Fall      97   8.997    FALSE
## 9    Fall     100   6.605    FALSE
## 10   Fall     110   5.933    FALSE
## 11   Fall     120   2.891     TRUE
## 12   Fall     135   2.766    FALSE
## 13   Fall     150  14.465    FALSE
## 14   Fall     165  24.239    FALSE
## 15   Fall     185  28.885    FALSE
## 16   Fall     200  26.766     TRUE
## 17 Summer      10 216.667     TRUE
## 18 Summer      20 159.672    FALSE
## 19 Summer      40 141.778    FALSE
## 20 Summer      60  97.894     TRUE
## 21 Summer      75  44.978    FALSE
## 22 Summer      85  25.807    FALSE
## 23 Summer      90  27.011     TRUE
## 24 Summer      97  34.436    FALSE
## 25 Summer     100  38.012    FALSE
## 26 Summer     110  27.557    FALSE
## 27 Summer     120  32.354     TRUE
## 28 Summer     135  20.446    FALSE
## 29 Summer     150   0.000    FALSE
## 30 Summer     165   0.000    FALSE
## 31 Summer     185   0.000    FALSE
## 32 Summer     200   0.000     TRUE
```

With these added arguments, we see that our data are now correctly formatted in multiple columns with column (variable) names.

Argument names and order

You will see in the preceding command that for every argument, we provide the name, `=`, and then the necessary input for that argument. However, if you write arguments in the function's default order, you can leave out the argument name.

```
read.table("data.csv", TRUE, ",")
```

Default orders are listed in a function's help page (accessed with `?`). Removing argument names should be done *with caution* as it makes the code less readable and runs the risk of breaking if a function updates or changes in the future. And if you provide arguments out of order, the argument name *must* be provided.

For example, this gives an error because `","` is not an allowable input for the second default argument `header` and the same goes for `TRUE` and the third argument `sep`.

```
read.table("data.csv", ",", TRUE)
```

```
## Error in read.table("data.csv", ",", TRUE): invalid 'sep' argument
```

In contrast, specifying the argument names allows you to write them out of order.

```
read.table("data.csv", sep=",", header=TRUE)
```

Other `read.` functions

If you look at the `read.table` help page, you will see several other similar functions. These allow you to read in data using fewer arguments. For example, `read.csv` reads in a comma-delimited file without the need to specify `sep=","`.

Save data to the Environment

All of the examples above simply print the data table into the console. This does not allow us to work further with these data! So, we need to name the data as an object in R so that it is saved in the local Environment and used in subsequent analyses.

We do this by prefacing `read.table` with the name we want to give the table and either `=` or `<-` to assign that name to whatever `read.table` reads in.

```
dat = read.table(file="data.csv", header=TRUE, sep=",")

#Which is equivalent to

dat <- read.table(file="data.csv", header=TRUE, sep=",")
```

Now, we see the `dat` object show up in our Environment in the upper right. If we click on that object in the Environment pane, we can see the table. Row and columns names are in grey boxes and data are in white boxes.

Since we are using an R project, we can close RStudio, save our `RScript` and `RData`, and when we re-open using the `.Rproj` file, our data will still be in the Environment! Try it and see!

Access data

Our `dat` data frame consists of 32 rows (observations) and 4 columns (variables).

Columns

We can extract columns from data frames using the `$` operator. For example, this accesses the micromolar (uM) oxygen measurements.

```
dat$O2_uM
```

```
##  [1] 203.533 183.787 130.579  91.115  69.828  26.972  11.066   8.997
##  [9]   6.605   5.933   2.891   2.766  14.465  24.239  28.885  26.766
## [17] 216.667 159.672 141.778  97.894  44.978  25.807  27.011  34.436
## [25]  38.012  27.557  32.354  20.446   0.000   0.000   0.000   0.000
```

Note that this variable is a capital O as in Oxygen, not a 0 as in zero. Tab-completion can help you here!

Rows and/or columns

We can also access the data frame within the syntax `dat[rows, columns]`. This works with the row/column number (also called its index value) or the row/column name. In R, indexing starts at 1 meaning the first measurement is 1, second is 2, etc. Thus, to access the 3rd row, 4th column in the `dat` data, you use this:

```
dat[4, 3]
```

```
## [1] 91.115
```

Or you can use the following. We do not have row names in these data, so we will only use the column name here.

```
dat[4, "O2_uM"]
```

```
## [1] 91.115
```

This can also be combined with the `$` notation since a column is a 1 dimensional vector as `dat$column[row]`.

```
dat$O2_uM[4]
```

```
## [1] 91.115
```

Basic calculations

Since we can access pieces of data within the data frame `dat`, we can also run various calculations on those data. Being a statistical software, base R comes with many basic functions like

```
# Compute mean Oxygen
mean(dat$O2_uM)
```

```
## [1] 53.28247
```

```
# Compute variance for Oxygen
var(dat$O2_uM)
```

```
## [1] 4140.459
```

You can find a list of other common statistical functions in the stats package that comes pre-installed with R.

Subset data

Conditional statements

A conditional statement queries data and returns TRUE or FALSE based on the statement. It is important to remember that these statements give answers of TRUE/FALSE. They *do not* subset the data to all the TRUE observations.

Some common statements include:

- `a == b` for 'a is equal to b'
  - Note that this is different from the single `=` used in functions!
- `a != b` for 'a is not equal to b'
- `a > b` for 'a is greater than b'
- `a >= b` for 'a is greater than or equal to b'
  - Similarly for `<` and `<=`
- `a %in% b` for 'a is in b'
- `is.na()` for 'is missing data'
- `!is.na()` for 'is not missing data'

- Note that the `!` is often used to be a statement into its negative form such as `!=` and `!is.na()`

For example, you can determine which depth values in `dat` are equal to 200.

```
dat$Depth_m == 200
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

Or greater than 100.

```
dat$Depth_m > 100
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Or if a depth of 80 exists in the data.

```
80 %in% dat$Depth_m
```

```
## [1] FALSE
```

Or if any of the depth values are missing.

```
is.na(dat$Depth_m)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

And so on...

Logical operators

You can string together multiple conditional statements using logical operators. These most commonly include:

- `&` for 'and'
- `|` for 'or'

For example, you can determine which depth values are between 50 and 150 (not inclusive).

```
dat$Depth_m > 50 & dat$Depth_m < 150
```

```
##  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
## [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

Or the opposite, which depth values are outside 50 and 150 (not inclusive).

```
dat$Depth_m < 50 | dat$Depth_m > 150
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Or you can string together even more statements to ask if deep waters below 150 meters ever experience anoxic (*e.g.* no oxygen) conditions in the Fall.

```
dat$Season == "Fall" & dat$Depth_m > 150 & dat$O2_uM == 0
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Use conditional statements and logical operators to subset data

Since conditional statements return TRUE/FALSE, they can be used to subset data to the TRUE observations. Since R returns the TRUE subset, it is important to always craft your statement(s) to return TRUE for the data you want to keep.

For example, let's return to our first example statement.

```
dat$Depth_m == 200
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

We can use this vector of TRUE/FALSE is the normal data frame syntax `dat[rows, columns]` to ask R to subset `dat` to the rows where depth is equal to 200.

```
dat[dat$Depth_m == 200, ]
```

```
##    Season Depth_m  O2_uM Add_data
## 16   Fall     200 26.766     TRUE
## 32 Summer     200  0.000     TRUE
```

A common way to clean-up code and make it more readable is to save your statement(s) as an object in R and then use it to subset. For example, if we take our last statement example and save it as a vector named `subset_statement`

```
subset_statement <- dat$Season == "Fall" & dat$Depth_m > 150 & dat$O2_uM == 0
```

We can then use this vector

```
subset_statement
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

to subset dat

```
dat[subset_statement, ]
```

```
## [1] Season   Depth_m  O2_uM    Add_data
## <0 rows> (or 0-length row.names)
```

which returns an empty data frame since in this data-set, deep waters are never anoxic in the Fall.

While many complex subsets are possible using base R, the code can quickly become messy and difficult to read. Thus, we recommend the `tidyverse` packages to work with more complex data.
