4: Coding Basics

Environmental Data Analytics | Sarah Ko Notes

Spring 2019

LESSON OBJECTIVES

1. Develop familiarity with the form and function of the RStudio interface.
2. Apply basic functionality of R
3. Evaluate how basic practies in R contribute to best management practices for data analysis

DATA TYPES IN R

R treats objects differently based on their characteristics. For more information, please see: https://www.statmethods.net/input/datatypes.html.

- **Vectors** 1 dimensional structure that contains elements of the same type.
- **Matrices** 2 dimensional structure that contains elements of the same type.
- **Arrays** Similar to matrices, but can have more than 2 dimensions. We will not delve into arrays in depth.
- **Lists** Ordered collection of elements that can have different modes.
- **Data Frames** 2 dimensional structure that is more general than a matrix. Columns can have different modes (e.g., numeric and factor). When we import csv files into the R workspace, they will enter as data frames.

Define what each new piece of syntax does below (i.e., fill in blank comments). Note that the R chunk has been divided into sections (# at beginning of line, —- at end)

```
# Vectors ----
vector1 <- c(1,2,5.3,6,-2,4) # numeric vector
vector1
```

```
## [1]  1.0  2.0  5.3  6.0 -2.0  4.0
```

```
vector2 <- c("one","two","three") # character vector
vector2
```

```
## [1] "one"   "two"   "three"
```

```
vector3 <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE) #logical vector
vector3
```

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

```
vector1[3] # called 'matrix subsetting'. pick vector 1, choose third item
```

```
## [1] 5.3
```

```
# Matrices ----
matrix1 <- matrix(1:20, nrow = 5,ncol = 4) # creates this vertically, then horizontally
matrix1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
```

```
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

```
matrix2 <- matrix(1:20, nrow = 5, ncol = 4, byrow = TRUE) # creates horizontally, then vertically
matrix2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20
```

```
matrix3 <- matrix(1:20, nrow = 5, ncol = 4, byrow = TRUE, # return after comma continues the line
                  dimnames = list(c("uno", "dos", "tres", "cuatro", "cinco"),
                                  c("un", "deux", "trois", "cat"))) #
matrix3 # this names the rows, then the columns
```

```
##        un deux trois cat
## uno     1    2     3   4
## dos     5    6     7   8
## tres    9   10    11  12
## cuatro 13   14    15  16
## cinco  17   18    19  20
```

```
matrix1[4, ] # grab row 4
```

```
## [1]  4  9 14 19
```

```
matrix1[ , 3] # grab column 3
```

```
## [1] 11 12 13 14 15
```

```
matrix1[c(12, 14)] # grab elements 12 and 14
```

```
## [1] 12 14
```

```
matrix1[c(12:14)] # grab elements 12 to 13
```

```
## [1] 12 13 14
```

```
matrix1[2:4, 1:3] # grab rows 2 to 4, and columns 1-3
```

```
##      [,1] [,2] [,3]
## [1,]    2    7   12
## [2,]    3    8   13
## [3,]    4    9   14
```

```
cells <- c(1, 26, 24, 68)
rnames <- c("R1", "R2")
cnames <- c("C1", "C2")
matrix4 <- matrix(cells, nrow = 2, ncol = 2, byrow = TRUE,
  dimnames = list(rnames, cnames)) # if you only specify rows and not columns, R assumes the number of
matrix4
```

```
##    C1 C2
## R1  1 26
## R2 24 68
```

```
# Lists ----
list1 <- list(name = "Maria", mynumbers = vector1, mymatrix = matrix1, age = 5.3); list1
```

```
## $name
## [1] "Maria"
##
## $mynumbers
## [1]  1.0  2.0  5.3  6.0 -2.0  4.0
##
## $mymatrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
##
## $age
## [1] 5.3
```

```
list1[[2]] #use double square brackets for lists
```

```
## [1]  1.0  2.0  5.3  6.0 -2.0  4.0
```

```
# Data Frames ----
d <- c(1, 2, 3, 4) # What type of vector? numeric
e <- c("red", "white", "red", NA) # What type of vector? char
f <- c(TRUE, TRUE, TRUE, FALSE) # What type of vector? logical
dataframe1 <- data.frame(d,e,f) #combine vectors
names(dataframe1) <- c("ID","Color","Passed"); View(dataframe1) # rename column names

dataframe1[1:2] # pulls columns 1&2. if you did matrix1[1:2] it would pull elements 1&2
```

```
##   ID Color
## 1  1   red
## 2  2 white
## 3  3   red
## 4  4  <NA>
```

```
dataframe1[,1:2] # same as the above line
```

```
##   ID Color
## 1  1   red
## 2  2 white
## 3  3   red
## 4  4  <NA>
```

```
dataframe1[1:2,] # pulls the rows
```

```
##   ID Color Passed
## 1  1   red   TRUE
## 2  2 white   TRUE
```

```
dataframe1[c("ID","Passed")] # pull columns by name
```

```
##   ID Passed
## 1  1   TRUE
## 2  2   TRUE
## 3  3   TRUE
## 4  4  FALSE
```

```
dataframe1$Color # pulls all elements in the defined column
```

```
## [1] red   white red   <NA>
## Levels: red white
```

QUESTION: How do the different types of data appear in the Environment tab?

ANSWER: in sections

QUESTION: In the R chunk below, write "dataframe1$". Press `tab` after you type the dollar sign. What happens?

ANSWER: displays column names to use

QUESTION: What happens when a comment in R is followed by "—-"?

ANSWER: makes it intos a section

Advanced: Sequential section headers can be created by using at least four -, =, and # characters.

PACKAGES

The Packages tab in the notebook stores the packages that you have saved in your system. A checkmark next to each package indicates whether the package has been loaded into your current R session. Given that R is an open source software, users can create packages that have specific functionalities, with complicated code "packaged" into a simple commands.

If you want to use a specific package that is not in your libaray already, you need to install it. You can do this in two ways:

1. Click the install button in the packages tab. Type the package name, which should autocomplete below (case matters). Make sure to check "intall dependencies," which will also install packages that your new package uses.
2. Type `install.packages("packagename")` into your R chunk or console. It will then appear in your packages list. You only need to do this once.

If a package is already installed, you will need to load it every session. You can do this in two ways:

1. Click the box next to the package name in the Packages tab.
2. Type `library(packagename)` into your R chunk or console.

2a. The command `require(packagename)` will also load a package, but it will not give any error or warning messages if there is an issue.

Tips and troubleshooting

- You may be asked to restart R when installing or updating packages. Feel free to say no, as this will obviously slow your progress. However, if the functionality of your new package isn't working properly, try restarting R as a first step.
- If asked "Do you want to install from sources the packages which needs compilation?", type `yes` into the console.
- You should only install packages once on your machine. If you store `install.packages` in your R chunks/scripts, comment these lines out, as below.
- Update your packages regularly!

```
# We will use the packages dplyr and ggplot2 regularly.
#install.packages("dplyr") # comment out install commands, use only when needed and re-comment
#install.packages("ggplot2")
```

```
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.5.2
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.5.2
```

```
#when items are masked, you can reinstall the packages, must do it in the correct order

# Some packages are umbrellas under which other packages are loaded
#install.packages(tidyverse)
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 3.5.2
```

```
## -- Attaching packages ----------------------------------- tidyverse 1.2.1 --
```

```
## v tibble  2.0.1     v purrr   0.2.5
## v tidyr   0.8.2     v stringr 1.3.1
## v readr   1.3.1     v forcats 0.3.0
```

```
## Warning: package 'tibble' was built under R version 3.5.2
```

```
## Warning: package 'tidyr' was built under R version 3.5.2
```

```
## Warning: package 'readr' was built under R version 3.5.2
```

```
## -- Conflicts -------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

What happens in the console when you load a package?

ANSWER:

FUNCTIONS

You've had some practice with functions with the simple commands you've entered in this lesson and the one previous. The basic form of a function is `functionname()`, and the packages we will use in this class will use these basic forms. However, there may be situations when you will want to create your own function. Below is a description of how to write functions through the metaphor of creating a recipe. Credit for this goes to Isabella R. Ghement (@IsabellaGhement on Twitter).

Writing a function is like writing a recipe. Your function will need a recipe name (functionname). Your recipe ingredients will go inside the parentheses. The recipe steps and end product go inside the curly brackets.

```
functionname <- function(){

}
```

A single ingredient recipe:

```
# Write the recipe
recipe1 <- function(x){
  mix <- x*2
  return(mix)
}

# Bake the recipe
simplemeal <- recipe1(5)

# Serve the recipe
simplemeal
```

```
## [1] 10
```

Two single ingredient recipes, baked at the same time:

```
recipe2 <- function(x){
  mix1 <- x*2
  mix2 <- x/2
  return(list(mix1 = mix1, #comma indicates we continue onto the next line
              mix2 = mix2))
}

doublesimplemeal <- recipe2(6)
doublesimplemeal
```

```
## $mix1
## [1] 12
##
## $mix2
## [1] 3
```

Two double ingredient recipes, baked at the same time:

```
recipe3 <- function(x, f){
  mix1 <- x*f
  mix2 <- x/f
  return(list(mix1 = mix1, #comma indicates we continue onto the next line
              mix2 = mix2))
}

doublecomplexmeal <- recipe3(x = 5, f = 2)
doublecomplexmeal
```

```
## $mix1
## [1] 10
##
## $mix2
## [1] 2.5
```

```
doublecomplexmeal$mix1
```

```
## [1] 10
```

Make a recipe based on the ingredients you have

```
recipe4 <- function(x) {
  if(x < 3) {
    x*2
  }
  else {
    x/2
  }
}

recipe5 <- function(x) {
  if(x < 3) {
    x*2
  }
  else if (x > 3) {
    x/2
  }
  else {
    x
  }
}
meal <- recipe4(4); meal
```

```
## [1] 2
```

```
meal2 <- recipe4(2); meal2
```

```
## [1] 4
```

```
meal3 <- recipe5(3); meal3
```

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

```
recipe6 <- function(x){
  ifelse(x<3, x*2, x/2)

}

meal4 <- recipe6(4); meal4
```

```
## [1] 2
```

```
meal5 <- recipe6(2); meal5
```

```
## [1] 4
```