R: An Introduction

Part 2 of Week 1, using RMarkdown.

Introduction to RStudio

This section introduces the R environment and some of the most basic funcionality aspects of R that are used through the remainder of the class. This section assumes little to no experience with statistical computing with R. We will introduce the R statistical computing environment, RStudio, and the dataset that we will work with for the remainder of the lesson. We will cover very basic functionality in R, including variables, functions, and importing/inspecting data frames.

R is the underlying statistical computing environment RStudio sits on top of R and makes writing code a lot easier.

R Studio Panes

On the top left is the script or editor window. This is where we are going to write all of our code. On the lower left there's the console window. This is where R tells us what it thinks we told it and then the answer. This is basically the same kind of interface as the terminal The top right has the environment and history tabs … environment is a list of all objects that are saved in memory … history is the history of all commands that have been run … On the bottom right hand side there's a window with Files / Plots / Help

Creating projects

Projects provide a way for the user to organize their data, code, and any other documents in one place. To create a project you go to File -> New Project and then create a new project which will create a new folder for your data/script, etc. or you can choose an existing project which will allow you to point to a folder that already has all of your resources.

Comments

Comments are an essential part in understanding what your code does and why something was coded a specific way. Comments are notes to yourself or to others that are made to explain what is going on in the code. They are not read by R and will only be seen when looking at the script.

```
#Intro to R
#-----------------------------------------------------
#this is a comment and will not be run as code
```

Basics of R

R can be used to do very advanced calculations, statistics, and data wrangling, however, it can also be used to do some very basic calculations and functions. R can act like a typical calculator, but it can also store the values that are run for later. To run lines of code in R you can put the cursor on the line of code you wish to run and hit the "Run" button. Alternatively you can highlight the line or lines of code you want to run and hit the "Run" button. You can also use the keyboard shortcut of `Ctrl+Enter` on Windows or `CMD+Enter` on a MAC.

```
# R can be used as a calculator
2 + 2
5 * 3
2 ^ 5
```

```r
# R knows order of operations
2 + 3/4 * 4 * (4+4)/4^2
```

As mentioned before, to do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` and the value we want to give it:

```r
# Assign objects using <-
weight_kg <- 55
```

`<-` is the assignment operator. Assigns values on the right to objects on the left, it is like an arrow that points from the value to the object. Mostly similar to `=` but not always. Learn to use `<-` as it is good programming practice. Using `=` in place of `<-` can lead to issues down the line. The keyboard shortcut for inserting the `<-` operator is `Alt-dash`.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names, which we'll get into shortly (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). In doubt check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. It is also recommended to use nouns for variable names, and verbs for function names.

When assigning a value to an object, R does not print anything. You can force to print the value by typing the name:

```r
weight_kg

# R is case sensitive
Weight_kg <- 60
Weight_kg <- 30


#we can do arithmetic with R objects
2.2 * weight_kg

#modify R object by reassigning object to new number
weight_kg <- 57.5
2.2 * weight_kg

#we can re-name 2.2*weight_kg as weight_lb
weight_lb <- 2.2 * weight_kg

#change value of weight_kg
weight_kg <- 100
```

Working with the Environment

You can see what objects (variables) are stored by viewing the Environment tab in Rstudio. You can also use the `ls()` function. You can remove objects (variables) with the `rm()` function. You can do this one at a time or remove several objects at once. You can also use the little broom button in your environment pane to remove everything from your environment.

```r
#list objects in environment
ls()

#remove objects from environment
```

```r
rm(weight_lb, weight_kg)
ls()

weight_lb <- 20
weight_lb

#Removes all objects from the environment
rm(list = ls())
```

EXERCISE 1

1. You have a patient with a height (inches) of 73 and a weight (lbs) of 203. Create r objects labeled 'height' and 'weight'.

```r
height <- 73
weight <- 203
```

2. Convert 'weight' to 'weight_kg' by dividing by 2.2. Convert 'height' to 'height_m' by dividing by 39.37

```r
weight_kg <- weight /2.2
height_m <- height/39.37
```

3. Calculate a new object 'bmi' where BMI = weight_kg / (height_m*height_m)

```r
bmi <- weight_kg / (height_m * height_m)
bmi <- weight_kg / (height_m ^ 2)
```

Built-in Functions

R has built-in functions that allow you to do more than simple math. Functions take in arguments within the parenthesis and return a value of a specific type.

```r
sqrt(144)
log(1000)
```

Get help with function

To get help with a function (or anything in R) you can use the `?` mark followed by the name of the function.

```r
?log
```

Note syntax highlighting when typing this into the editor. Also note how we pass *arguments* to functions. The `base=` part inside the parentheses is called an argument, and most functions use arguments. Arguments modify the behavior of the function. Functions some input (e.g., some data, an object) and other options to change what the function will return, or how to treat the data provided. Finally, see how you can *next* one function inside of another (here taking the square root of the log-base-10 of 1000).

```r
log10(1000)
log(1000, base = 10)
#The base= part inside the parentheses is called an argument. Arguments are the inputs to functions
```

We can write functions without labeling arguments as long as they come in the same order as in the help file

```r
log(1000, 10)
```

Nesting Functions

Oftentimes you will want to do several functions at once, which allows you to use less lines of code. Nesting functions allows you to use the output of one function as the input of the next function. This works as long as the type of output returned from the first function is the same as the type of input allowed from the second function. (e.g. both are numeric, both are characters, etc.)

```r
sqrt(log(1000, 10))
#Because sqrt() takes a number and because log() outputs a number we can nest the two together

#create intermediate object to make nesting easier to decipher
myval <- log(1000, 10)
sqrt(myval)
```

EXERCISE 2

See `?abs` and calculate the square root of the log-base-10 of the absolute value of `-4*(2550-50)`. Answer should be `2`.

```r
sqrt(log(abs(-4*(2550 - 50)), 10))
```

Vectors

Vectors are one of the most basic data types in R. They are a sequence of data elemnets of the same type. This can be numeric (2, 3.2, etc.), character ("a", "a sentence is here", "548-4241"), logical ("TRUE", "FALSE"), etc, they just all have to be of the same type.

Using the `:` operator allows you to create a consecutive sequence of numbers.

```r
1:5
6:10
1:5 + 6:10

#Careful, they must be multiples of each other
#1:5 + 6:8 #(Will not work)
1:6 + 6:8

#This will add 3 to each element of the vector
1:5 + 3
```

To create vectors that not necessilary consecutive you use the c() function (concatenate / combine). This allows you to enter in as many different values into a vector.

```r
c(1, 5, 6)
c(1:5, 1:6)

#What if we wanted to created a vector from to 2 to 200 by 4s?
```

The seq function allows you to create a vector of values that are increase from one value to another by a designated increment.

```r
?seq

seq(from = 2, to = 200, by = 4)
```

Just like other objects, you can assign a vector a name.

```r
#assign vectors to object name
animal_weights <- c(50, 60, 66)
animal_weights

#vectors can also be character type
animals <- c("mouse", "rat", "dog")
animals
```

When creating vectors or looking at data from other sources, you may want to inspect the vector more closely. You will want to know things like how long vector is, the number elements, which type of object it is, etc. There are a series of functions that can be used to inspect vectors.

```r
#Inspecting vectors
#Length provides the number of elements
length(animals)

#Class shows the type of vector
class(animals)
class(animal_weights)

#str shows the structure of the vector
str(animals)
str(animal_weights)
```

Oftentimes we will want to add to a vector. To do this, we can again use the c() function.

```r
#appending to vectors using c()
animal_weights <- c(animal_weights, 80)
animal_weights <- c(49, animal_weights)
animal_weights
```

We can also use functions on vectors, just like we did before.

```r
#Make sure that you are using the correct input type
#sum(animals)
sum(animal_weights)
mean(animal_weights)
```

One key thing to learn in R is how to access individual elements within a vector. Sometimes you might just want to look at the first 5 elements of a vector or you want to return the 8th, 12th, and 20th elements. To access these elements, we will use brackets [] and the c() function.

```r
#Indexing vectors
x <- 100:150
#Using [1] allows you to see the very first element of a vector
x[1]
animals[1]

#You can also look through a sequence of elements.
#fifth through 10th elements
```

```
x[5:10]
animals[2:3]

#Using the c() function allows you to look at the elements in a non-sequential manner.
#40th and 48th elements (non-sequential)
x[c(40, 48)]
animals[c(1, 3)]

#what happens when we call an index beyond the vector
x[1:150]
```

You can also use the indexing features to add or change elements within a vector by placing the vector on the left hand side of the assignment operator.

Adding an element to a vector will increase its size by 1 and will increment the position (if not placed at the end) by 1 of any element that follows.

```
animals[c(4)] <- "lizard"

animals[c(15)] <- "bat"
animals
animals[c(1500)] <- "horse"
animals
animals[c(1, 3)] <- c("anteater", "koala")
```

Data frames

Data frames store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text)

The first step is reading in data from an external source. The first way we will do that is through the read.csv function which allows you to read in a csv file as a data frame. The first argument is the file name and the second option `header =` allows you to specify whether column names are present in the first column of the csv file or not.

```
##load data using read.csv
 gm <- read.csv("Data/gapminder.csv", header = TRUE)

#another option is read_csv in the readr package
```

As with a vector, one of the most important things to do is to inspect your dataframe to ensure that everything was read in in a way that makes sense.

```
#Class shows the type of object
class(gm)

#Head shows the first group of rows and tail the last bunch of rows (the number can be changed in the o
head(gm)
tail(gm, 10)

#Dim provides the number of rows and columns, nrow the number of rows, and ncol the number of columns.
dim(gm)
```

```
nrow(gm)
ncol(gm)

#The names function shows the column/variable names
names(gm)

#The summary function provides a quick descriptive overview of each column. The output is based upon th
summary(gm)

#The str function shows the detailed structure of the dataframe
str(gm)
```

To access a single column in a dataframe, you use the dataset name followed by the `$` symbol and then the name of the dataframe you want to look at.

```
#Using the $ to access variables
gm$pop

#This allows you to provide a function that simply looks at one column (just like we did with a vector)
#The na.rm function removes any missing values from the analysis, since the result of this function wou
mean(gm$lifeExp, na.rm = TRUE)
```

EXERCISE 3

1. What's the standard deviation of the life expectancy (hint: get help on the `sd` function with `?sd`).

```
?sd
sd(gm$lifeExp, na.rm = TRUE)
```

2. What's the mean population size in millions? (hint: divide by `1000000`, or alternatively, `1e6`).

```
mean(gm$pop, na.rm = TRUE) / 1e6
```

3. What's the range of years represented in the data? (hint: `range()`).

```
range(gm$year)
range(gm$pop)
```

4. Run a correlation between life expectancy and GDP per capita (hint: ?cor())

```
cor(gm$lifeExp, gm$gdpPercap, method = "kendall")
```

Let's move on to learning about using R Markdown.