IT137IU: Data Analysis
Lab#1/Assignment#1: Introduction to R

Introduction

In this guide you will learn the basic fundamentals of the statistical software program R. Because R is not a prerequisite for the class, this guide assumes no background in the language. The objectives of the guide are as follows

1. Download R and RStudio
2. Get familiar with the RStudio interface
3. Understand R data types
4. Understand R data structures, in particular vectors and data frames
5. Understand R functions and map
6. Understand R Piping and the process for submitting assignments

This lab guide follows closely and supplements the material presented in Chapters 2, 4, and 21 in the textbook R for Data Science (RDS).

What is R?

R is a free, open source statistical programming language. It is useful for data cleaning, analysis, and visualization. R is an interpreted language, not a compiled one. This means that you type something into R and it does it. It is both a command line software and a programming environment. It is an extensible, open-source language and computing environment for Windows, Macintosh, UNIX, and Linux platforms, which allows for the user to freely distribute, study, change, and improve the software.

Getting R

R can be downloaded from one of the "CRAN" (Comprehensive R Archive Network) sites. In the US, the main site is at http://cran.us.r-project.org/. Look in the "Download and Install R" area. Click on the appropriate link based on your operating system.

If you already have R on your computer, make sure you have the most updated version of R on your personal computer (4.2.2 "Innocent and Trusting").

Mac OS X

1. On the "R for Mac OS X" page, there are multiple packages that could be downloaded. If you are running High Sierra or higher, click on R-4.2.2.pkg; if you are running an earlier

version of OS X, download the appropriate version listed under "Binary for legacy OS X systems."

2. After the package finishes downloading, locate the installer on your hard drive, double-click on the installer package, and after a few screens, select a destination for the installation of the R framework (the program) and the R.app GUI. Note that you will have to supply the Administrator's password. Close the window when the installation is done.

3. An application will appear in the Applications folder: R.app.

4. Browse to the [XQuartz download page](). Click on the most recent version of XQuartz to download the application.

5. Run the XQuartz installer. XQuartz is needed to create windows to display many types of R graphics: this used to be included in MacOS until version 10.8 but now must be downloaded separately.

Windows

1. On the "R for Windows" page, click on the "base" link, which should take you to the "R-4.2.2 for Windows (32/64 bit)" page

2. On this page, click "Download R 4.2.2 for Windows", and save the exe file to your hard disk when prompted. Saving to the desktop is fine.

3. To begin the installation, double-click on the downloaded file. Don't be alarmed if you get unknown publisher type warnings. Window's User Account Control will also worry about an unidentified program wanting access to your computer. Click on "Run".

4. Select the proposed options in each part of the install dialog. When the "Select Components" screen appears, just accept the standard choices

Note: Depending on the age of your computer and version of Windows, you may be running either a "32-bit" or "64-bit" version of the Windows operating system. If you have the 64-bit version (most likely), R will install the appropriate version (R x64 3.5.2) and will also (for backwards compatibility) install the 32-bit version (R i386 3.5.2). You can run either, but you will probably just want to run the 64-bit version.

What is RStudio?

If you click on the R program you just downloaded, you will find a very basic user interface. For example, below is what I get on a Window
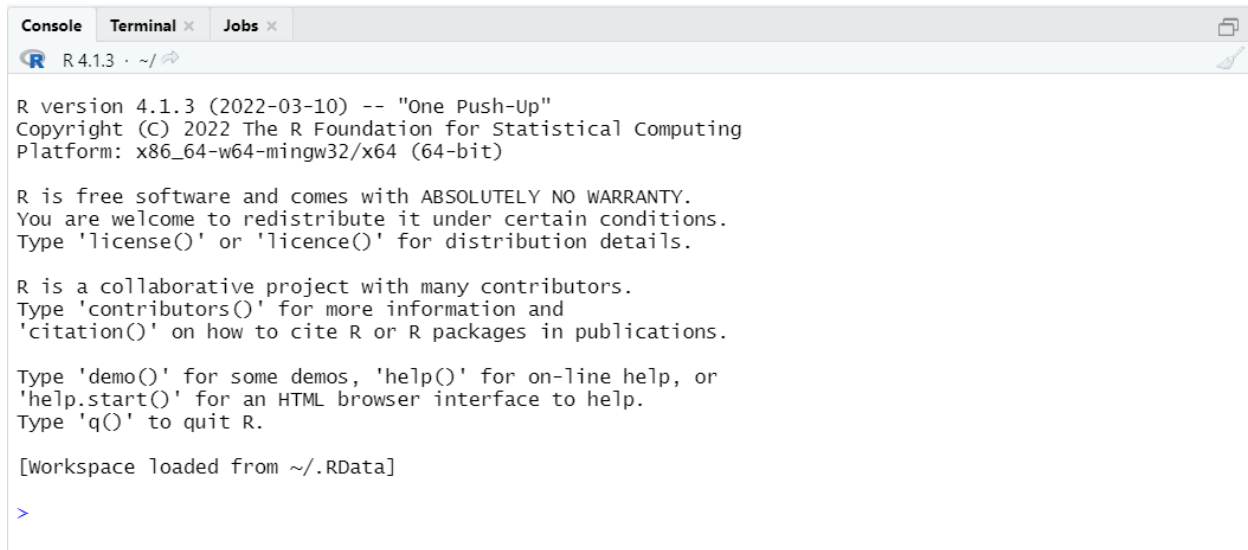
Figure 1: R console

We will not use R's direct interface to run analyses. Instead, we will use the program RStudio. RStudio gives you a true integrated development environment (IDE), where you can write code in a window, see results in other windows, see locations of files, see objects you've created, and so on. To clarify which is which: R is the name of the programming language itself and RStudio is a convenient interface.

Getting RStudio

To download and install RStudio, follow the directions below

1. Navigate to RStudio's download site
2. Click on the appropriate link based on your OS (Windows, Mac, Linux and many others). Do not download anything from the "Zip/Tarballs" section.
3. Click on the installer that you downloaded. Follow the installation wizard's directions, making sure to keep all defaults intact. After installation, RStudio should pop up in your Applications or Programs folder/menu.

Note that the most recent version of RStudio works only for certain operating systems (OS). If you have an older OS, you will need to download an older version RStudio, which you can find here.

The RStudio Interface

Open up RStudio. You should see the interface shown in the figure below which has three windows.
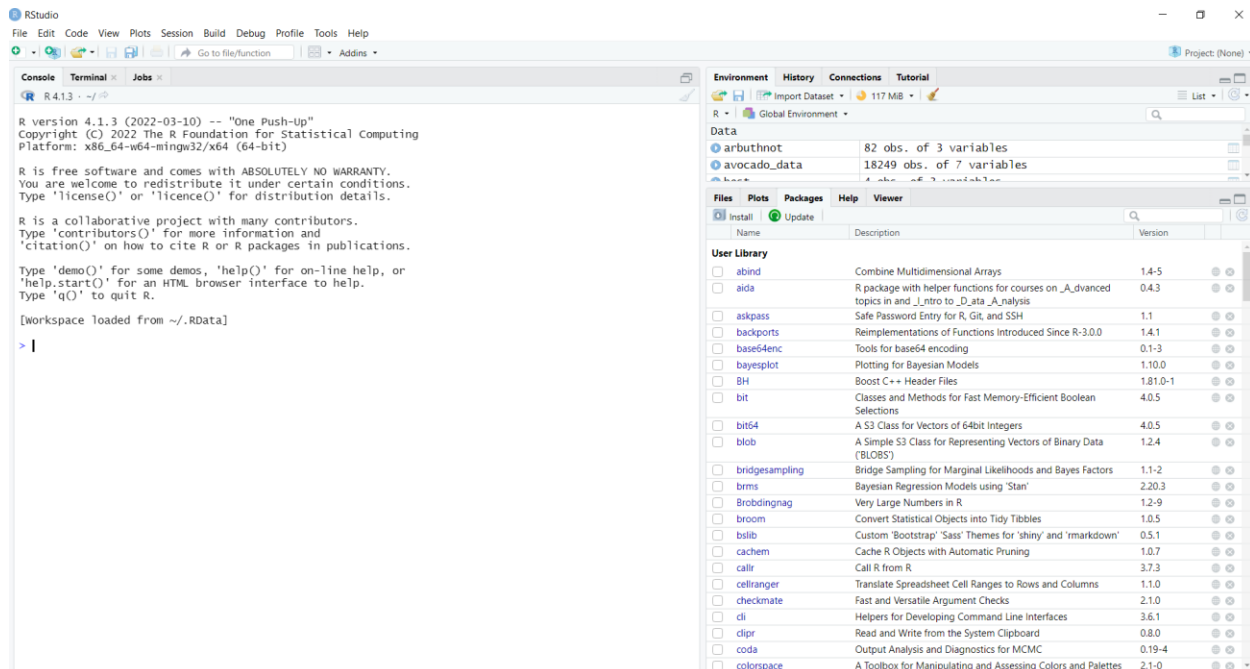
Figure 2. Rstudio Interface

- **Console** (left) - The way R works is you write a line of code to execute some kind of task on a data object. The R Console allows you to run code interactively. The screen prompt > is an invitation from R to enter its world. This is where you type code in, press enter to execute the code, and see the results.
- **Environment, History, and Connections tabs** (upper-right)
  - **Environment** - shows all the R objects that are currently open in your workspace. This is the place, for example, where you will see any data you've loaded into R. When you exit RStudio, R will clear all objects in this window. You can also click on to clear out all the objects loaded and created in your current session.
  - **History** - shows a list of executed commands in the current session.
  - **Connections** - you can connect to a variety of data sources, and explore the objects and data inside the connection. I typically don't use this window, but you can.
- **Files, Plots, Packages, Help and Viewer tabs** (lower-right)
  - **Files** - shows all the files and folders in your current working directory (more on what this means later).
  - **Plots** - shows any charts, graphs, maps and plots you've successfully executed (we'll be using this window starting in Lab 5).
  - **Packages** - tells you all the R packages that you have access to (more on this in Lab 2).
  - **Help** - shows help documentation for R commands that you've called up.
  - **Viewer** - allows you to view local web content (won't be using this much).

Setting RStudio Defaults

While not required, I strongly suggest that you change preferences in RStudio to never save the workspace so you always open with a clean environment. See Ch. 8.1 of R4DS for some more background

1. From the Tools menu on RStudio, open the Tools menu and then select Global Options.
2. If not already highlighted, click on the General button from the left panel.
3. Uncheck the following Restore boxes

- Restore most recently opened project at startup
- Restore previously open source documents at startup
- Restore .RData into workspace at startup

4. Set Save Workspace to .RData on exit to *Never*
5. Click OK at the bottom to save the changes and close the preferences window. You may need to restart RStudio.
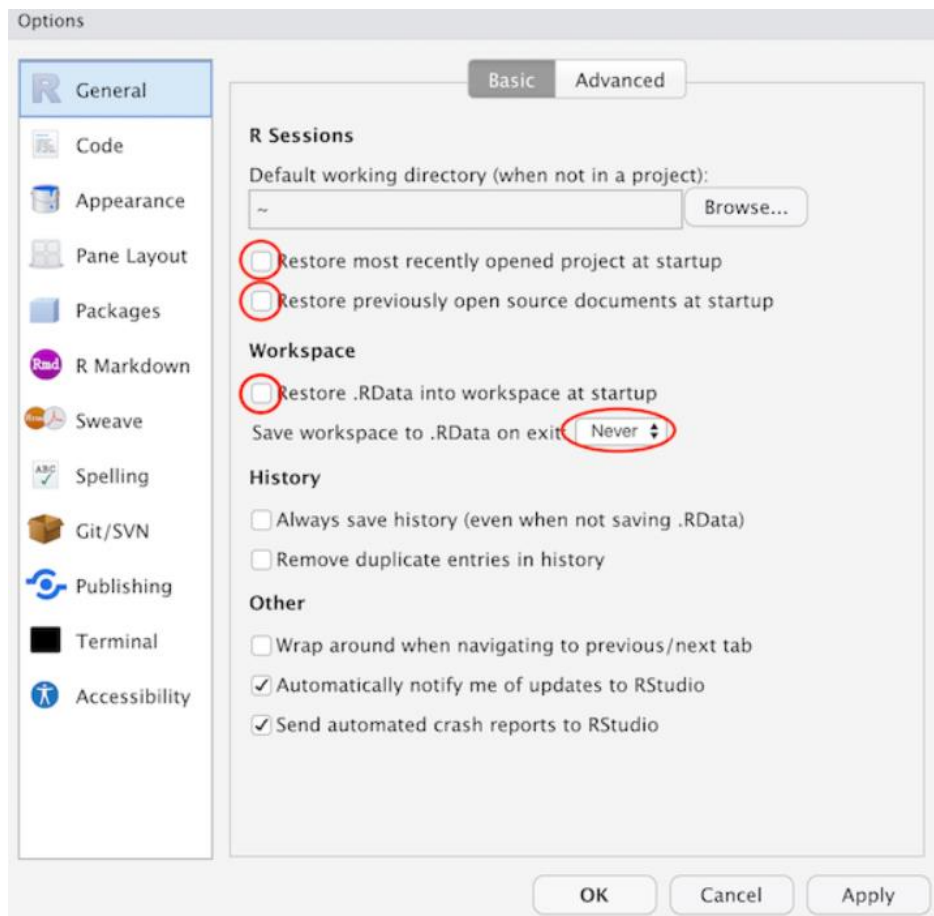
Figure 3. Rstudio default

The reason for making these changes is that it is preferable for reproducibility to start each R session with a clean environment. You can restore a previous environment either by rerunning code or by manually loading a previously saved session.

The R Studio environment is modified when you execute code from files or from the console. If you always start fresh, you do not need to be concerned about things not working because of something you typed in the console, but did not save in a file.

You only need to set these preferences once.

R Data Types

Let's now explore what R can do. R is really just a big fancy calculator. For example, type in the following mathematical expression next to the > in the R console (left window)

```
1+1
```

Note that spacing does not matter: 1+1 will generate the same answer as 1   +   1. Can you say hello to the world?

```
hello world
```

```
## Error: <text>:1:7: unexpected symbol
## 1: hello world
##           ^
```

Nope. What is the problem here? We need to put quotes around it.

```
"hello world"
```

```
## [1] "hello world"
```

"hello world" is a character and R recognizes characters only if there are quotes around it. This brings us to the topic of basic data types in R. There are four basic data types in R: character, logical, numeric, and factors (there are two others - complex and raw - but we won't cover them because they are rarely used).

Characters

Characters are used to represent words or letters in R. We saw this above with "hello world". Character values are also known as strings. You might think that the value "1" is a number. Well,

with quotes around, it isn't! Anything with quotes will be interpreted as a character. No ifs, ands or buts about it.

Logicals

A logical takes on two values: FALSE or TRUE. Logicals are usually constructed with comparison operators, which we'll go through more carefully in Lab 2. Think of a logical as the answer to a question like "Is this value greater than (lower than/equal to) this other value?" The answer will be either TRUE or FALSE. TRUE and FALSE are logical values in R. For example, typing in the following

```
3>2
```

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

gives us a true. What about the following?

```
"prof visser" == "prof cannon"
```

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

Numeric

Numerics are separated into two types: integer and double. The distinction between integers and doubles is usually not important. R treats numerics as doubles by default because it is a less restrictive data type. You can do any mathematical operation on numeric values. We added one and one above. We can also multiply using the * operator

```
2*3
```

```
## [1] 6
```

Divide

```
4/2
```

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

And even take the logarithm!

```
log(1)
```

```
## [1] 0
```

```
log(0)
```

```
## [1] -Inf
```

Uh oh. What is -Inf? Well, you can't take the logarithm of 0, so R is telling you that you're getting a non numeric value in return. The value -Inf is another type of value type that you can get in R.

Factors

Think of a factor as a categorical variable. It is sort of like a character, but not really. It is actually a numeric code with character-valued levels. Think of a character as a true string and a factor as a set of categories represented as characters. We won't use factors too much in this course.

R Data Structures

You learned that R has four basic data types. Now, let's go through how we can store data in R. That is, you type in the character "hello world" or the number 3, and you want to store these values. You do this by using R's various data structures.

Vectors

A vector is the most common and basic R data structure and is pretty much the workhorse of the language. A vector is simply a sequence of values which can be of any data type but all of the same type. There are a number of ways to create a vector depending on the data type, but the most common is to insert the data you want to save in a vector into the command c(). For example, to save the values 4, 16 and 9 in a vector type in

```r
c(4, 16, 9)
```

```
## [1]  4 16  9
```

You can also have a vector of character values

```r
c("martin", "anne", "clare")
```

```
## [1] "martin" "anne"   "clare"
```

The above code does not actually "save" the values 4, 16, and 9 - it just presents it on the screen in a vector. If you want to use these values again without having to type out c(4, 16, 9), you can save it in a data object. At the heart of almost everything you will do (or ever likely to do) in R is the concept that everything in R is an object. These objects can be almost anything, from a single number or character string (like a word) to highly complex structures like the output of a plot, a map, or a summary of your statistical analysis.

You assign data to an object using the arrow sign <-. This will create an object in R's memory that can be called back into the command window at any time. For example, you can save "hello world" to a vector called *b* by typing in

```r
b <- "hello world"
b
```

```
## [1] "hello world"
```

You can pronounce the above as "b becomes 'hello world'".

Similarly, you can save the numbers 4, 16 and 9 into a vector called *v1*

```
v1 <- c(4, 16, 9)
v1
```

```
## [1]  4 16  9
```

You should see the objects *b* and *v1* pop up in the Environment tab on the top right window of your RStudio interface.
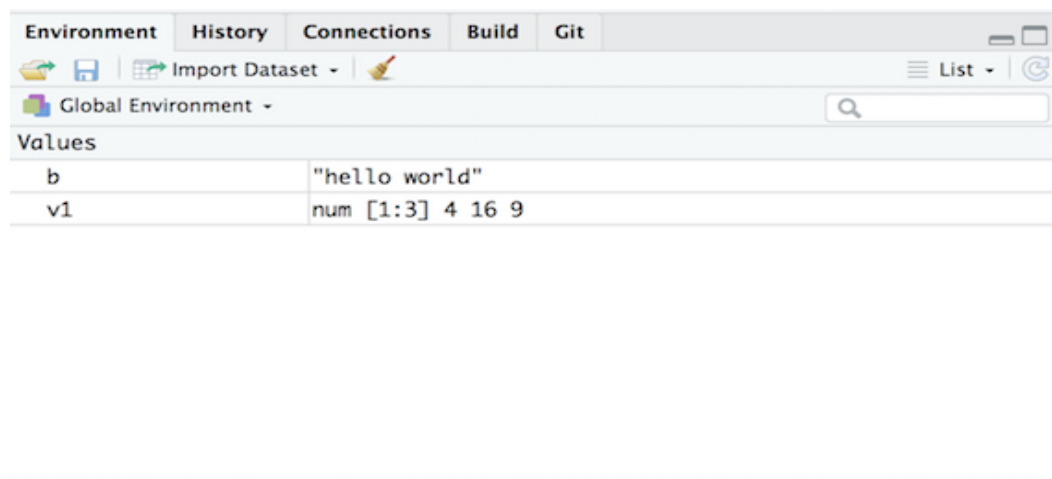
Figure 4. Environment Window

Note that the name *v1* is nothing special here. You could have named the object *x* or *crd150* or your pet's name (mine was *charlie*). You can't, however, name objects using special characters (e.g. !, @, $) or only numbers (although you can combine numbers and letters, but a number cannot be at the beginning e.g. *2d2*). For example, you'll get an error if you save the vector *c(4,16,9)* to an object with the following names

```
123 <- c(4, 16, 9)
!!! <- c(4, 16, 9)
```

```
## Error: <text>:2:5: unexpected assignment
```

```
## 1: 123 <- c(4, 16, 9)

## 2: !!! <-

##        ^
```

Also note that to distinguish a character value from a variable name, it needs to be quoted. "v1" is a character value whereas *v1* is a variable. One of the most common mistakes for beginners is to forget the quotes.

```
brazil
```

```
## Error in eval(expr, envir, enclos): object 'brazil' not found
```

The error occurs because R tries to print the value of the object *brazil*, but there is no such object. So remember that any time you get the error message `object 'something' not found`, the most likely reason is that you forgot to quote a character value. If not, it probably means that you have misspelled, or not yet created, the object that you are referring to.

Every vector has two key properties: *type* and *length*. The type property indicates the data type that the vector is holding. Use the command `typeof()` to determine the type

```
typeof(b)
```

```
## [1] "character"
```

```
typeof(v1)
```

```
## [1] "double"
```

Note that a vector cannot hold values of different types. If different data types exist, R will coerce the values into the highest type based on its internal hierarchy: logical < integer < double < character. Type in `test <- c("r", 6, TRUE)` in your R console. What is the vector type of `test`?

The command `length()` determines the number of data values that the vector is storing

```
length(b)
```

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

```
length(v1)
```

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

You can also directly determine if a vector is of a specific data type by using the command `is.X()` where you replace X with the data type. For example, to find out if *v1* is numeric, type in

```
is.numeric(b)
```

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

```
is.numeric(v1)
```

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

There is also `is.logical()`, `is.character()`, and `is.factor()`. You can also coerce a vector of one data type to another. For example, save the value "1" and "2" (both in quotes) into a vector named *x1*

```
x1 <- c("1", "2")
```

```
typeof(x1)
```

```
## [1] "character"
```

To convert *x1* into a numeric, use the command `as.numeric()`

```
x2 <- as.numeric(x1)
typeof(x2)
```

```
## [1] "double"
```

There is also `as.logical()`, `as.character()`, and `as.factor()`.

An important practice you should adopt early is to keep only necessary objects in your current R Environment. For example, we will not be using *x2* any longer in this guide. To remove this object from R forever, use the command `rm()`

```
rm(x2)
```

The data frame object *x2* should have disappeared from the Environment tab.

Also note that when you close down R Studio, the objects you created above will disappear for good. Unless you save them onto your hard drive, all data objects you create in your current R session will go bye bye when you exit the program.

Data Frames

We learned that data values can be stored in data structures known as vectors. The next step is to learn how to store vectors into an even higher level data structure. The data frame can do this. Data frames store vectors of the same length. Create a vector called *v2* storing the values 5, 12, and 25

```
v2 <- c(5,12,25)
```

We can create a data frame using the command data.frame() storing the vectors *v1* and *v2* as columns

```
data.frame(v1, v2)
```

```
##    v1 v2
## 1   4  5
## 2 16 12
## 3  9 25
```

Store this data frame in an object called *df1*

```
df1<-data.frame(v1, v2)
```

df1 should pop up in your Environment window. You'll notice a ● next to *df1*. This tells you that *df1* possesses or holds more than one object. Click on ● and you'll see the two vectors we saved into *df1*. Another neat thing you can do is directly click on *df1* from the Environment window to bring up an Excel style worksheet on the top left window of your RStudio interface. You can also type in

```
View(df1)
```

to bring the worksheet up. You can't edit this worksheet directly, but it allows you to see the values that a higher level R data object contains.

We can store different types of vectors in a data frame. For example, we can save the numeric vector *v1* with a character vector *v3*.

```
v3 <- c("martin", "anne", "clare")
df2 <- data.frame(v1, v3)
df2
```

For higher level data structures like a data frame, use the function `class()` to figure out what kind of object you're working with.

```
class(df2)
```

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

We can't use `length()` on a data frame because it has more than one vector. Instead, it has *dimensions* - the number of rows and columns. You can find the number of rows and columns that a data frame has by using the command `dim()`

```
dim(df1)
```

```
## [1] 3 2
```

Here, the data frame *df1* has 3 rows and 2 columns. Data frames also have column names, which are characters.

```
colnames(df1)
```

```
## [1] "v1" "v2"
```

In this case, the data frame used the vector names for the column names.

We can extract columns from data frames by referring to their names using the `$` sign.

```
df1$v1
```

```
## [1]   4 16   9
```

We can also extract data from data frames using brackets `[ , ]`

```
df1[,1]
```

```
## [1]   4 16   9
```

The value before the comma indicates the row, which you leave empty if you are not selecting by row. The value after the comma indicates the column, which you leave empty if you are not selecting by column. The above line of code selected the first column. Let's select the 2nd row.

```
df1[2,]
```

```
##   v1 v2
## 2 16 12
```

What is the value in the 2nd row *and* 1st column?

```
df1[2,1]
```

```
## [1] 16
```

Functions

Let's take a step back and talk about functions (also known as commands). An R function is a packaged recipe that converts one or more inputs (called arguments) into a single output. You execute most of your tasks in R using functions. We have already used a couple of functions above including `typeof()` and `colnames()`. Every function in R will have the following basic format

```
functionName(arg1 = val1, arg2 = val2, ...)
```

In R, you type in the function's name and set a number of options or parameters within parentheses that are separated by commas. Some options **need** to be set by the user - i.e. the function will spit out an error because a required option is blank - whereas others can be set but are not required because there is a default value established.

Let's use the function `seq()` which makes regular sequences of numbers. You can find out what a function does and its options by calling up its help documentation by typing `?` and the function name

```
? seq
```

The help documentation should have popped up in the bottom right window of your RStudio interface. The documentation should also provide some examples of the function at the bottom of the page. Type the arguments `from = 1, to = 10` inside the parentheses of `seq()`

```
seq(from = 1, to = 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

You should get the same result if you type in

```
seq(1, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

The code above demonstrates something about how R resolves function arguments. When you use a function, you can always specify all the arguments in `arg = value` form. But if you do not, R attempts to resolve by position. So in the code `seq(1, 10)`, it is assumed that we want a sequence `from = 1` that goes `to = 10` because we typed 1 before 10. Type in 10 before 1 and see what happens.

Each argument requires a certain type of data type. For example, you'll get an error when you use a character in `seq()`

```
seq("p", "w")
```

```
## Error in seq.default("p", "w"): 'from' must be a finite number
```

Exercise 1:

Create a function called bigger_100 which takes two numbers as input and outputs 0 if their product is less than or equal to 100, and 1 otherwise. (Hint: remember that you can cast a Boolean value to an integer with as.integer).

Anonymous functions

Notice that we can feed functions as parameters to other functions. This is an important ingredient of a functional-style of programming, and something that we will rely on heavily in this course. When supplying a function as an argument to another function, we might not want to name the function that is passed. Here's a (stupid, but hopefully illustrating) example.

We first define the named function new_applier_function which takes two arguments as input: an input vector, which is locally called input in the scope of the function's body, and a function, which is locally called function_to_apply. Our new function new_applier_function first checks whether the input vector has more than one element, throws an error if not, and otherwise applies the argument function function_to_apply to the vector input.

```
# define a function that takes a vector and a function as an argument
new_applier_function <- function(input, function_to_apply) {
  # check if input vector has at least 2 elements
  if (length(input) <= 1) {
    # terminate and show informative error message
    stop("Error in 'new_applier_function': input vector has length <= 1.")
  }
  # otherwise apply the function to the input vector
  return(function_to_apply(input))
}
```

We use this new function to show the difference between named and unnamed functions, in particular why the latter can be very handy and elegant. First, we consider a case where we use new_applier_function in connection with the named built-in function sum:

```
# sum vector with built-in & named function
new_applier_function(
  input = 1:3,              # input vector
```

```
  function_to_apply = sum    # built-in & named function to apply
)   # returns 6
```

If instead of an existing named function, we want to use a new function to supply to new_applier_function, we could define that function first and give it a name, but if we only need it "in situ" for calling new_applier_function once, we can also write this:

```
# Sum vector with anonymous function
new_applier_function(
  input = 1:3,               # input vector
  function_to_apply = function(in_vec) {
    return(in_vec[1] + in_vec[2])
  }
)   # returns 3 (as it only sums the first two arguments)
```

Exercise 2:

How many arguments should you pass to a function that…

        a. …tells if the sum of two numbers is even?
        b. …applies two different operations on a variable and sums the results? Operations are not fixed in the function.

Call the function new_applier_function with input = 1:3 and an anonymous function that returns just the first two elements of the input vector in reverse order (as a vector).

Loops and maps

For-loops

For iteratively performing computation steps, R has a special syntax for for loops. Here is an example of an (again, stupid, but illustrative) example of a for loop in R:

```
# fix a vector to transform
input_vector <- 1:6

# create output vector for memory allocation
output_vector <- integer(length(input_vector))

# iterate over length of input
for (i in 1:length(input_vector)) {
  # multiply by 10 if even
  if (input_vector[i] %% 2 == 0) {
    output_vector[i] <- input_vector[i] * 10
  }
  # otherwise leave unchanged
  else {
    output_vector[i] <- input_vector[i]
```

```
  }
}

output_vector
```

```
## [1]  1 20  3 40  5 60
```

Exercise 3:

Let's practice for-loops and if/else statements! Create a vector a with 10 random integers from range (1:50). Create a second vector b that has the same length as vector a. Then fill vector b such that the ith entry in b is the mean of a[(i-1):(i+1)]. Do that using a for-loop. Note that missing values are equal to 0 (see example below). Print out the result as a tibble whose columns are a and b.

Example: If a has the values [25, 39, 12, 33, 47, 3, 48, 14, 45, 8], then vector b should contain the values [21, 25, 28, 31, 28, 33, 22, 36, 22, 18] when rounded to whole integers. The value in the fourth position of b (value 31), is obtained with (a[3] + a[4] + a[5])/3. The value in the first position of b (value 21) is obtained with (0 + a[1] + a[2])/3 and similarly the last value with (a[9] + a[10] + 0)/3. (Hint: use conditional statements if, if else and else to deal specifically with the edge cases (first and last entry in the vectors).)

Functional Iteration

Base R provides functional iterators (e.g., `apply`), but we will use the functional iterators from the `purrr` package. The main functional operator from `purrr` is `map` which takes a vector and a function, applies the function to each element in the vector and returns a list with the outcome. There are also versions of `map`, written as `map_dbl` (double), `map_lgl` (logical) or `map_df` (data frame), which return a vector of doubles, Booleans or a data frame. The following code repeats the previous example which used a for-loop but now within a functional style using the functional iterator `map_dbl`:

```
input_vector <- 1:6
map_dbl(
  input_vector,
  function(i) {
    if (input_vector[i] %% 2 == 0) {
      return(input_vector[i] * 10)
    }
    else {
      return (input_vector[i])
    }
  }
)
```

```
## [1]  1 20  3 40  5 60
```

We can write this even shorter, using `purrr`'s short-hand notation for functions

```
input_vector <- 1:6
map_dbl(
  input_vector,
  ~ ifelse(.x %% 2 == 0, .x * 10, .x)
)
```

```
## [1]  1 20  3 40  5 60
```

The trailing ~ indicates that we define an anonymous function. It, therefore, replaces the usual function(...) call which indicates which arguments the anonymous function expects. To make up for this, after the ~ we can use .x for the first (and only) argument of our anonymous function.
To apply a function to more than one input vector, element per element, we can use pmap and its derivatives, like pmap_dbl etc. pmap takes a list of vectors and a function. In short-hand notation, we can define an anonymous function with ~ and integers like ..1, ..2 etc, for the first, second … argument. For example:

```
x <- 1:3
y <- 4:6
z <- 7:9

pmap_dbl(
  list(x, y, z),
  ~ ..1 - ..2 + ..3
)
```

```
## [1] 4 5 6
```

Exercise 4: Use map_dbl and an anonymous function to take the following input vector and return a vector whose i^{th} element is the cumulative product of input up to the ith position divided by the cumulative sum of input up to that position. (Hint: the cumulative product up to position i is produced by prod(input[1:i]); notice that you need to "loop over", so to speak, the index i, not the elements of the vector input.)

```
input <- c(12, 6, 18)
```

Piping

When we use a functional style of programming, piping is your best friend. Consider the standard example of applying functions in what linguists would call "center-embedding". We start with the input (written inside the inner-most bracketing), then apply the first function `round`, then the second `mean`, writing each next function call "around" the previous.

```
# define input
input_vector <- c(0.4, 0.5, 0.6)

# first round, then take mean
mean(round(input_vector))
```

```
## [1] 0.3333333
```

Things quickly get out of hand when more commands are nested. A common practice is to store intermediate results of computations in new variables which are only used to pass the result into the next step.

```
# define input
input_vector <- c(0.4, 0.5, 0.6)

# rounded input
rounded_input <- round(input_vector)

# mean of rounded input
mean(rounded_input)
```

```
## [1] 0.3333333
```

Piping lets you pass the result of a previous function call into the next. The `magrittr` package supplies a special infix operator %>% for piping. The pipe %>% essentially takes what results from evaluating the expression on its left-hand side and inputs it as the first argument in the function on its right-hand side. So x %>% f is equivalent to f(x). Or, to continue the example from above, we can now write:

```
input_vector %>% round %>% mean
```

```
## [1] 0.3333333
```

The functions defined as part of the tidyverse are all constructed in such a way that the first argument is the most likely input you would like to pipe into them. But if you want to pipe the

left-hand side into another argument slot than the first, you can do that by using the `.` notation to mark the slot where the left-hand side should be piped into: `y %>% f(x, .)` is equivalent to `f(x, y)`.

Exercise 5:

A friendly colleague has sent reaction time data in a weird format:

```
weird_RTs <- c("RT = 323", "RT = 345", "RT = 421", "RT = 50")
```
Starting with that vector, use a chain of pipes to: extract the numeric information from the string, cast the information into a vector of type numeric, take the log, take the mean, round to 2 significant digits. (Hint: to get the numeric information use stringr::str_sub, which works in this case because the numeric information starts after the exact same number of characters.)

What to submit:

Your submission should include the following:

1. Lab report answers to the five exercises above and source code.
2. Please create a folder called "yourname_MSSV" that includes all the required files and generate a zip file called "yourname_MSSV.zip".
3. Please submit your work (.zip) to Blackboard.