GC&DS - 02: Functions, Arguments, and R Scripts

Code

1 Overview

The concepts covered here may be confusing. You may even question why we can't just jump into data manipulation and why all of this matters. In order to code in R so that you can be comfortable using R and with communicating with other users, a very basic understanding of concepts is important. This way, when someone asks you about an object, function, or assignment, you will know what they are taking about. And, well, you can't communicate with R without knowing how they work at a basic level. You cannot just brute force yourself into data science or running models without getting yourself into trouble. So yes, we are here for a reason, though short lived.

2 Simple Overview of Objects

Quite likely, you have been assigned a name. Typically speaking, you cannot leave the hospital without being given a name to be part of the governments records. As part of this process, you are also assigned a SSN. Of course, you may have been born outside of a hospital and someone forgot to process paperwork on your behalf, in which case there may not be any record of a name assigned to you nor may there be a record of you existing. Unlikely is the case, however.

In this example, you are the *object* and a *name* has been *assigned* to you. Names help distinguish you from another person.

Let's use an example of an object called `name`, which can be assigned a set of characters, like *Jim Bob*, which are placed within quotation marks (e.g., single or double, doesn't matter). The quotes let R know the contents are characters or strings. You will see lots of character objects as they represent factor variables (e.g., race,

ethnicity, favorite game, etc.) but you will also see lots of objects that are numeric in some form (e.g., age, rating, cognitive performance, etc.).

You can also think of an *object* as a sort of container that holds something. Containers of different types hold different things and so is true in computer programming. A container for holding water may look different from a container for holding books. In computer speak, one type of container can hold numbers, another can hold characters, another can hold a data frame, etc. The container object is holding whatever you have assigned it to hold.

3 Object Assignment Using `<-`

Before we can create any objects, however, we need to understand a little about *assignment*. In computer programming, an assignment statement sets (or re-sets) an object denoted by a name. Assignment requires using an *assignment operator*, which in R is `<-` . In some computer languages `=` is the assignment operator, so R is a little different if you are familiar with other languages. This practice also helps you distinguish between object assignment and other uses of `=` that don't stand for assignment. Even though you can assign objects with `=` , please use `<-` as this is the practice.

In other words, *assignment* is akin to creating a new word and assigning a meaning to it. You could also think of an assignment statement as **create this thing and set it equal to something** so that the computer understand what that represents. Thus, assignment stores the assigned information as an object of whatever name you decided to call it.

If objects are like containers holding things, we start with the name of the object (e.g., `container`) and then assign `"things"` to it using `<-` .

Silly Example: `container <- "something"`

Hide

```
container <- "things"
```

Back to *Jim Bob*. Of course, there are different people other than Jim Bob who exist in the world but when coding, they do not exist unless you create them. So, let's create an object that holds the name of "Jim Bob".

```
name <- "Jim Bob"
```

Hide

```
name <- "Jim Bob"  # assign string to object named name


n <- "Jim Bob"     # we could have assigned it a different name, say n
```

Whenever you reference the object `name` (or `n`), R will return to you the contents of the object, which in this case will be a character or string object containing a single person's name because that's how we assigned it.

Hide

```
name               # call object to return contents of "name"
```

```
## [1] "Jim Bob"
```

We can use `print()` to do the same thing…

Hide

```
print(name)
```

```
## [1] "Jim Bob"
```

What about numeric information? We can create an object called `year` and assign the current year to it; let's have this object contain the current year in numeric form, not as a string. Remember to use `<-` for assignment.

Hide

```
year <- 2022    # assign a number to year ; notice no quotes
```

In order to know whether this `year` object now contains the year, we can check by typing the name of the object or use `print()` to print the returned value.

Hide

```
year
```

```
## [1] 2022
```

Hide

```
print(year)
```

```
## [1] 2022
```

`name` and `year` are very simple objects. `name` is a simple character/string object we created, which contains only the name of 1 person and `year` only holds the current year. There is something else important about how R treats them that you cannot see on the surface. Both of these objects are also **vectors**. Vectors are *one-dimensional arrays* containing *n* pieces of information. You might also think of a vector so a variable (e.g., IQs of people). Both the `name` and `year` vectors contain only one piece of information, however. If you don't believe me, we can use some functions that will answer this for us.

- `is.vector()` returns a logical (T or F) about whether the object is a vector
- `length()` returns a non-negative numeric integer representing the number of elements contained
- `typeof()` returns the object's type

Let's try them by passing the object `name` inside the function.

Hide

```
is.vector(name)   # is it a vector?
```

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

Hide

```
#?length
length(name)      # how many elements?
```

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

Hide

```
typeof(name)      # what is it's type?
```

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

If `name` contained more than one object, it would still be a vector having a different length. But in order to create such vectors, each element of the vector needs to be separated by a comma and each elements needs to be wrapped by quotes.

If you do not separate strings by a comma…

Hide

```
name <- "Jim Bob Kendra"


name                                # return object; also print(name)
```

```
## [1] "Jim Bob Kendra"
```

Hide

```
is.character(name)                  # is it a character?
```

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

Hide

```
length(name)                        # what is its length?
```

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

If you do use quotes for each element and separate each by a comma, you need to use a function to combine them, which is `c()`.

Hide

```
name <- c("Jim Bob", "Kendra")  # two names, combine with c()



is.character(name)
```

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

Hide

```
length(name)                  # vector with length 2
```

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

As a side note, the pieces/values of a vector are referred to as **elements**. You can reference elements by number representing their position in the vector.

Hide

```
name[1]   # first element
```

```
## [1] "Jim Bob"
```

Hide

```
name[2]   # second element
```

```
## [1] "Kendra"
```

Hide

```
name[3]   # a third element? No. It only has length 3
```

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

Objects in `R`, however, can take on many forms other than strings or numbers just illustrated. Objects can be strings/characters, numeric values, character strings, functions, data frames, vectors, lists, matrices, plots, etc. If you use `typeof()` on a data frame object, the function will return `"list"` because a data frame is also a list. More on this later.

4 Function Characteristics

Given our reliance on functions in this course, there is some terminology to understand how to work with functions.

Here are 5 terms/concepts to know:

- *name* (created by assignment operator `<-`)
- *definition* (code statements or instructions for its usage)
- *function arguments* (optional variables that specify the function's operation)
- *function call* (e.g., execution of a function)
- *returned object* (value returned from the executed function)

5 Functions are Objects with Names and Definitions

In computer language, *functions* are also objects and like all objects are *assigned* names. These names help distinguish one function from another because each function will serve a different purpose. If you are thinking that some people can have the same name even though they are different objects, yes, some functions can have the same name even though they refer to different objects.

The object examples above for `name` and `year` illustrated creation of a string object or a numeric value. These objects didn't perform any operation, mathematical or otherwise. Functions are special types of objects that carry out certain operations for you, like calculate a mean, a standard deviation, or run all the math for a linear model used for regression or ANOVA. These function objects are essentially containers that perform computations for you.

Think of functions like words with definitions. The definition is what the function does/means/stands for and the word is what the function is named. For example, a function to calculate the *mean* of a numeric vector is called `mean` and its definition is the formula for calculating an arithmetic mean.

`R` has built-in functions, functions as part of external `libraries` (or packages), and functions that you define yourself. The term function means exactly what you might expect - code that executes some type of *function*. In R, functions are easily *called* by placing parentheses, `()` , at the end of the function name (e.g., `mean()`). Within the parentheses are any arguments you will need to specify corresponding to the functions parameters. More on arguments later, so just hold on.

6 Objects and Function Objects

Now that you have a basic idea of objects and assignment, there are different types of objects. Unlike the example of assigning a value to an object, functions like `mean()` , `plot()` , `data.frame()` and `mutate()` are objects that contain statements for carrying out operations. As you might imagine, calculating the mean of a set of values using the `mean()` function would involve statements to carry out operations like summation and division.

- `mean()`
- `plot()`
- `data.frame()`
- `dplyr::mutate()`

7 Understanding Functions by Writing Them

The *function* function: `function()`

One way to understand how functions work is to create some yourself. Because functions are objects, you would assign (e.g., using `<-`) code to the function object. Let's create some functions using the function named `function()`, which is required in order to create function. This function is aptly named for its purpose. When using `function()`, you are telling R that the statements that follow are part of a function. The statements you will put between `{` and `}`.

Let's create *2* functions by assigning their contents to objects named `func_a` and `func_b`. In order for functions to `return` the result of their operation(s), we will also need to use the `return()` function as part of the functions statements. Though not a requirement of all functions, if you ever have to write functions, including `return()` eliminates ambiguity of what the function call returns.

Define function definition and *assign* to function *name*.

Hide

```
func_a <- function() {
  return(2022)         # This silly function contains a simple statement; to return a specifi
c numeric value
}


func_b <- function() {
  return("2022")       # This silly function simple returns a string representing the current
year
}
```

8 Function Versus the Function Object

Functions are called by appending `()` to their assigned name. Objects that are not functions don't operate the same way. If you wish to call a function, you need to append parentheses to the name. Without them, the R interpreter will provide you with the functions contents and details about the function itself.

Hide

```
func_a      # If you don't use the (), you'll just see the contents of the function object
```

```
## function() {
##   return(2022)         # This silly function contains a simple statement; to return a spec
ific numeric value
## }
```

Hide

```
func_b
```

```
## function() {
##   return("2022")       # This silly function simple returns a string representing the curr
ent year
## }
```

Add `()` to call the function and see what R returns to you.

Hide

```
func_a()    # Call the function and see it return something
```

```
## [1] 2022
```

Hide

```
func_b()
```

```
## [1] "2022"
```

9 Performing Operations on Objects

Because `func_a()` returns a numeric object, you can perform mathematical operations on it just as you would any numeric object. You cannot perform mathematical operations on character or string objects.

Hide

```
func_a() + 2    # the year plus 2
```

```
## [1] 2024
```

Hide

```
func_a() * 2    # twice the year
```

```
## [1] 4044
```

Hide

```
#func_b() + 2    # the interpreter does not understand this and reports an error.



#func_b() * 2
```

10 Function Parameters and Arguments

Notice that you didn't need to include any of your own instructions for these functions to perform their operations. Although some functions in R operate this way, many will require some additional values or variables. Functions that have `parameters` as part of their code statements will require you to pass `arguments` to be used in those statements.

For example, `mean()` will require you to pass a vector or numeric elements in order to calculate the *mean* of

those elements. If you don't pass an argument, R will not know know what to calculate the mean for and it will throw an error. If you check `?mean`, you will see that `x` is the parameter requiring some argument of values.

Hide

```
#mean()               # nothing passed to function; error


mean(x = c(1,2,5) )   # vector passed as argument to parameter x; mean of vector returned
```

```
## [1] 2.666667
```

Hide

```
mean( c(1,2,5) )      # x is not needed because it's the first parameter
```

```
## [1] 2.666667
```

Back to parameters. Let's create another function to demonstrate the usage. We will add a single parameter within the parentheses. This parameter has no default value so in order for the function to work, you'd need to pass an argument.

```
func_c <- function(parameter) {  code statements to do something
```

```
`return something`
```

```
}
```

Specify the parameter as `a` or whatever you wish.

Hide

```
func_c <- function(a) {
  return(a + 2)  # add 2 to what is passed to parameter a
}
```

Call the function and see what the function call returns…

Hide

```
#func_c()       # call the function but forgot to give parameter a an argument value
```

Hide

```
func_c(a = 1)    # call the function by setting parameter = 1
```

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

Redefine a function. Note, this will overwrite the previous definition of `func_c()`.

Hide

```
func_c <- function(a) {
  b <- a + 2    # add 2 to what is passed to parameter a and assign to object b

  return(b)     # then return object b
}
```

Call the function and see what the function call returns…

Hide

```
func_c         # see what the function is doing
```

```
## function(a) {
##   b <- a + 2    # add 2 to what is passed to parameter a and assign to object b
##
##   return(b)     # then return object b
## }
```

Hide

```
func_c(a = 1)  # call func_c() setting the argument for parameter a = 1
```

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

11 Name Functions in Useful Ways

Of course, naming the function `func_c` does not really help you understand what tasks the function is executing. Rather, you may wish to name the function **add2** or something useful and also change the name of the parameter to something meaningful, like `value` .

Hide

```
add2 <- function(value) {
  return(value + 2)  # add 2 to the value argument
}
```

Call the function, passing `value = 7` …

Hide

```
add2(value = 7)
```

```
## [1] 9
```

Unlike the other functions, because `add2()` contains a *parameter* for which you pass an *argument*. Whereas the parameter is part of the function definition, you will need to specify the information to pass to it so that the function knows how to carry out the instructions.

Hide

```
#add2()       # No argument passed, Hmm, error. Note: remove # to test.
```

12 Passing Objects as Function Arguments

Because you created an object named `year` earlier, R knows this exists. You could pass this object as an argument to a function assuming it is of the type the function needs. For example, if the function is performing addition, the object passed needs to take a numeric form. Also, because this silly function contains only one argument, we don't need to specify the argument by name. We can simply drop out the argument and pass the object.

Hide

```
func_c(a = year)   # passing an argument that is an object
```

```
## [1] 2024
```

Hide

```
func_c(year)       # dropping out of the argument by name
```

```
## [1] 2024
```

You could also pass some other value as an argument or even a mathematical operation.

Hide

```
func_c(4)      # pass a number as an argument
```

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

Hide

```
func_c(10 * 1) # a mathematical operation, though odd perhaps
```

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

Hide

```
#func_d("two")  # but not a string. Note: remove the # to test.
```

13 Functions with Default Arguments

Just for clarity, if a parameter has a default argument value, you don't need to set an argument but if you do, it will be used in lieu of the default.

Hide

```
func_c <- function(x = 2021) {  # assign x to 2021 as default
  return(x + 2)                 # add 2 to x
}
```

Call the function…

Hide

```
func_c()        # not specifying the argument will assume the default value (e.g., 2021)
```

```
## [1] 2023
```

Hide

```
func_c(10)      # specifying a value will override the default
```

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

14 Functions with Multiple Parameters/Arguments

Many functions have multiple parameters and can thus take multiple arguments.

Hide

```
sum_xy <- function(x = NULL,
                   y = NULL) {  # two arguments, each set to NULL as default
  return(x + y)
}
```

Call the function…

Hide

```
sum_xy()      # summing two NULL values (the default) is nothing
```

```
## integer(0)
```

Hide

```
sum_xy(2021, 2)
```

```
## [1] 2023
```

Hide

```
sum_xy(year, 2)
```

```
## [1] 2024
```

15 Sourcing code with `source()`

`source()` is a function that will read and execute R code. One benefit of this function relates to having files of code. Let's say you create an R script file with extension `.R` (see **RStudio -> File -> New File -> R Script**) and that file contains R code to read a data file, or clean up variables in a data frame, or create plot objects. All of the code in that `.R` file can be executed from the R console, from within another `.R` file, or from a `.Rmd` file. As result, `source()` can be used to compartmentalize code serving different purposes and to keep each file from being too busy or complicated. Let's try.

Create an `.R` script file from `RStudio`. In it, type `message("My first script file.")` and save it as `my_first_script` (the `.R` should be automatic) in your `"GCDS/r"` directory.

Because the `.Rmd` file you are working with is already saved in `"GCDS/r"` (if you saved it there correctly), you can source the file by name.

Hide

```
source("my_first_script.R")   # runs code saved in "my_first_script.R" file; print message
```

```
## My first script file.
```

External code functions used for this course will be sourced from pastebin.com (pastebin.com) or from https://github.com/ (https://github.com/). This site for example, https://pastebin.com/raw/97NNTTzu (https://pastebin.com/raw/97NNTTzu) contains R code that is called and executed using `source()` in order to limit work for you.

16 Using Functions from Libraries

Functions in base `R` are easy to operate on data out of the box; they don't require installation. For example, `head()` will look a the first few rows (or cases) of data frame. A data frame contains n rows and m columns. is a n x m, which is a in a data file. In order to demonstrate this, we will use the `USArrests` data set that is part of base `R`. Then, we will use `mean()` to calculate the mean of a vector of data. A column of a data frame is a vector of some type (e.g., numbers or characters).

- `head()` returns the top of the data frame; compare to `tail()`
- `View()` (uppercase V): returns a table of the data frame; built into R, looks crappy, steals focus
- `view()` (lowercase v): returns an filterable html table of the data frame; my alternative to `View()`

Hide

```
head(USArrests)   # 6 rows by default
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

If you query the function in the console by typing `?head`, you will see in the that the seconds argument for `head()` is `n`, the number of rows for the function to return. We can change that.

And if you set the arguments in their order, you do not need to reference them.

But if you change their order, you will need to reference the arguments. You cannot call `head(3, USArrests)` but you can call `head(n = 3, x = USArrests)`. You normally would not wish to change the order of arguments for `head()` but for more complicated functions, you might wish to for different reasons.

Using the viewing options…

Hide

```
#View(USArrests)


view(USArrests, rows = T, show = 5)   # but same as view(head(USArrests), rows = T)
```

Show 5 **entries** **Search:**

	Murder	Assault	UrbanPop	Rape
	All	All	All	All
Alabama	13.2	236	58	21.2
Alaska	10	263	48	44.5
Arizona	8.1	294	80	31
Arkansas	8.8	190	50	19.5
California	9	276	91	40.6

Showing 1 to 5 of 50 entries Previous 1 2 3 4 5 … 10 Next

You can also check whether the `USArrests` data file is a data frame using `is.data.frame()`, which will return `TRUE` indicating that it is indeed a data frame.

- `is.data.frame()` returns logical (T or F) about data frame as two-dimensional array

That seems tedious, however. You can learn a lot more about the data frame by examining its structure using

`str()`. The `USArrests` object is a data frame, contains 50 observations (e.g., rows) and 4 variables (columns). All column variables appear to contains numbers, with two of them being numeric, abbreviated `num` and two are integers, abbreviated `int`.

- `str()` returns the structure of a data frame

And you can check the names of the columns using `names()`. What is actually returned to you is a character vector, or a vector whose elements are of character type. You can test whether the column names is a vector by *wrapping* `names()` with the `is.vector()` function. Similarly, wrapping `names()` in `typeof()` will tell you the type is `character`.

- `names()` returns names of data frame
- `is.vector()` returns logical if/if not a vector (see other `is.` functions)
- `typeof()` returns the type of the object

Hide

```
names(USArrests)          # what are the names of the columns?
```

```
## [1] "Murder"   "Assault"  "UrbanPop" "Rape"
```

Hide

```
is.vector(names(USArrests))
```

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

Hide

```
typeof(names(USArrests))   # what is type of structure are the names
```

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

17 Using `dplyr` and `magrittr`

One library you will use a lot of is `dplyr` for data manipulation. Here is an example of easy to read code that takes the `USArrests` data frame, then `mutate()` new variables that represent the mean of all states and the z-score of each state and then filters based on those values.

Assign the final objects to data frame objects named `a` and `b`. The objects `a` and `b` simply represent two different filters of the data from on the `filter()` line.

Hide

```
library(magrittr)
library(dplyr)
```

```
##
## 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
```

Hide

```
a = USArrests %>%                                  # take this data frame
  dplyr::mutate(mean_murder = mean(Murder)) %>%    # mutate a new variable
  dplyr::filter(Murder > mean_murder)              # filter


b = USArrests %>%
  dplyr::mutate(z_murder = scale(Murder)) %>%
  dplyr::filter(z_murder > 0)
```

The astute reader would notice that states for which `Murder` rate is higher than the average will be the same as those that are greater than a z-score of 0 (the mean) so the two data-frame objects are logically equivalent except for the new variable columns. We can check using the logical operator `==` . *Note:* Do not confuse the assignment operator `=` with the logical operator `==` . If you code `a = b` , you will assign the object `b` to object `a` and overwrite `a` 's previous assignment in the process.

Hide

```
head(a)
```

```
##            Murder Assault UrbanPop Rape mean_murder
## Alabama      13.2     236       58 21.2       7.788
## Alaska       10.0     263       48 44.5       7.788
## Arizona       8.1     294       80 31.0       7.788
## Arkansas      8.8     190       50 19.5       7.788
## California    9.0     276       91 40.6       7.788
## Colorado      7.9     204       78 38.7       7.788
```

Hide

```
head(b)
```

```
##            Murder Assault UrbanPop Rape   z_murder
## Alabama      13.2     236       58 21.2 1.24256408
## Alaska       10.0     263       48 44.5 0.50786248
## Arizona       8.1     294       80 31.0 0.07163341
## Arkansas      8.8     190       50 19.5 0.23234938
## California    9.0     276       91 40.6 0.27826823
## Colorado      7.9     204       78 38.7 0.02571456
```

Are the data-frame objects equivalent?

Hide

```
a == b    # logical test
```

```
##                Murder Assault UrbanPop Rape mean_murder
## Alabama          TRUE    TRUE     TRUE TRUE       FALSE
## Alaska           TRUE    TRUE     TRUE TRUE       FALSE
## Arizona          TRUE    TRUE     TRUE TRUE       FALSE
## Arkansas         TRUE    TRUE     TRUE TRUE       FALSE
## California       TRUE    TRUE     TRUE TRUE       FALSE
## Colorado         TRUE    TRUE     TRUE TRUE       FALSE
## Florida          TRUE    TRUE     TRUE TRUE       FALSE
## Georgia          TRUE    TRUE     TRUE TRUE       FALSE
## Illinois         TRUE    TRUE     TRUE TRUE       FALSE
## Kentucky         TRUE    TRUE     TRUE TRUE       FALSE
## Louisiana        TRUE    TRUE     TRUE TRUE       FALSE
## Maryland         TRUE    TRUE     TRUE TRUE       FALSE
## Michigan         TRUE    TRUE     TRUE TRUE       FALSE
## Mississippi      TRUE    TRUE     TRUE TRUE       FALSE
## Missouri         TRUE    TRUE     TRUE TRUE       FALSE
## Nevada           TRUE    TRUE     TRUE TRUE       FALSE
## New Mexico       TRUE    TRUE     TRUE TRUE       FALSE
## New York         TRUE    TRUE     TRUE TRUE       FALSE
## North Carolina   TRUE    TRUE     TRUE TRUE       FALSE
## South Carolina   TRUE    TRUE     TRUE TRUE       FALSE
## Tennessee        TRUE    TRUE     TRUE TRUE       FALSE
## Texas            TRUE    TRUE     TRUE TRUE       FALSE
## Virginia         TRUE    TRUE     TRUE TRUE       FALSE
```