Day_2_session_01

Module VI:

Writing functions in R

- Function in R
- Modify default behavior of functions using arguments in R
- Demonstrate how to create user-defined functions in R

Basic/Built-in functions

What are functions?

A key feature of R is functions. Functions are “self contained” modules of code that accomplish a specific task. Functions usually take in some sort of data structure (value, vector, dataframe etc.), process it, and return a result. The general usage for a function is the name of the function followed by parentheses:

function_name(input)

The input(s) are called arguments, which can include:

```
sqrt(81)
round(3.14159)
round(3.14159, 2)
```

The basic (built-in) functions are available as part of R’s built in capabilities, and we will explore a few more of these base functions below.

```
sqrt(81)
my_vec<-c(88, 95, 92, 97, 96, 97, 94, 86, 91, 95, 97, 88, 85, 76, 68)
mean(my_vec)
median(my_vec)
sd(my_vec)
hist(my_vec)
range(my_vec)
sum(my_vec)
summary(my_vec)

p_value<- .034/15476587634566
p_value
```

Useful Built-in Functions in R

There are plenty of helpful built-in functions in R used for various purposes. Some of the most popular ones are:

- min(), max(), mean(), median() – return the minimum / maximum / mean / median value of a numeric vector, correspondingly
- sum() – returns the sum of a numeric vector
- range() – returns the minimum and maximum values of a numeric vector
- abs() – returns the absulute value of a number
- str() – shows the structure of an R object
- print() – displays an R object on the console
- ncol() – returns the number of columns of a matrix or a dataframe
- length() – returns the number of items in an R object (a vector, a list, etc.)
- nchar() – returns the number of characters in a character object
- sort() – sorts a vector in ascending or descending (decreasing=TRUE) order
- exists() – returns TRUE or FALSE depending on whether or not a variable is defined in the R environment

User-defined Functions

One of the best ways to improve your reach as a data scientist is to write functions. Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting. Writing a function has three big advantages over using copy-and-paste:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another).

Writing good functions is a lifetime journey. Even after using R for many years, you still learn new techniques and better ways of approaching old problems. The most important reasons are:

- DRY principle: Don't repeat yourself. Functions let you reuse the same computational building block in different parts of a program or script.
- Procedural programming: Procedural programming is a programming paradigm, a way to write well-structured (modular) code. Using functions allows one to add structure to the code which helps to increase readability and maintainability (find errors/bugs). The opposite would be to just write one line after another, an endless series of commands without a clear structure. Such code is also called spaghetti code and should be avoided whenever possible (always).
- Testing/Debugging: When writing functions, one can split a larger project in separate smaller computational building blocks (a bit like Lego). One big advantage is that this allows us to test individual smaller blocks to ensure that they work properly. This makes it much easier to find potential bugs and errors (called debugging).

Writing functions is the key feature to evolve from a '**basic user**' to a '**developer**'.

Illustrative example: we have three vectors called x1, x2, and x3

```
# "Manually" calculate the standard deviation
sqrt(sum((x - mean(x))^2) / (length(x) - 1))
```

We now have to calculate the standard deviation for three different vectors called x1, x2, and x3;

```
# Define the three vectors (random numbers)
set.seed(3) # Pseudo-randomization
x1 <- rnorm(1000, 0, 1.0)
x2 <- rnorm(1000, 0, 1.5)
x3 <- rnorm(1000, 0, 5.0)

# Calculate standard deviation once ...
sd1 <- sqrt(sum((x1 - mean(x1))^2) / (length(x1) - 1))
# ... and again, ...
sd2 <- sqrt(sum((x2 - mean(x2))^2) / (length(x2) - 1))
# ... and again ...
sd3 <- sqrt(sum((x3 - mean(x1))^2) / (length(x3) - 1))
c(sd1 = sd1, sd2 = sd2, sd3 = sd3)
```

Rather than doing this spaghetti-style coding, we now use functions.

```
# User-defined standard deviation function
sdfun <- function(x) {
  res <- sqrt(sum((x - mean(x))^2) / (length(x) - 1))
  return(res)
}
```

```
# Define some values
set.seed(3) # Pseudo-randomization
x1 <- rnorm(1000, 0, 1)
x2 <- rnorm(1000, 0, 1.5)
x3 <- rnorm(1000, 0, 5)

# Calculate standard deviation
sd1 <- sdfun(x1)
sd2 <- sdfun(x2)
sd3 <- sdfun(x3)
c(sd1 = sd1, sd2 = sd2, sd3 = sd3)
```

Function Components

The different parts of a function are –

Function Name − This is the actual name of the function. It is stored in R environment as an object with this name.

Arguments − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.

Function Body − The function body contains a collection of statements that defines what the function does.

Return Value − The return value of a function is the last expression in the function body to be evaluated.

The structure of a function is given below:

```
myfun <- function() {
    # No instructions
}
```

```
name_of_function <-      function(argument1,argument2){
        statements or code that does something
return(something)
}
```

- The keyword function(...) creates a new function.
- (...) is used to specify the arguments (in the example above there are no arguments, thus () is empty).
- Everything inside the curly brackets {...} is executed when calling the function (instructions; here empty/just a comment).
- The new function() { } is assigned to a new object called myfun.

When defining the function you will want to provide the list of arguments required (inputs and/or options to modify behavior of the function), and wrapped between curly brackets place the tasks that are being executed on/using those arguments. The argument(s) can be any type of object (like a scalar, a matrix, a dataframe, a vector, a logical, etc), and it's not necessary to define what it is in any way.

Finally, you can "return" the value of the object from the function, meaning pass the value of it into the global environment. The important idea behind functions is that objects that are created within the function are local to the environment of the function – they don't exist outside of the function.

Calling functions

A function call consists of the name of the function and a (possibly empty) argument list in round brackets (function_name(...)).

The NULL value in R

The NULL value in R is what the NoneType (‘None’) is in Python or ‘NONE’ in SQL to mention two examples outside R. NULL is an empty object (NULL basically means ‘nothing’).

```
x <- NULL
length(x)
## [1] 0
class(x)
## [1] "NULL"
typeof(x)
## [1] "NULL"
is.null(x)
## [1] TRUE
```

Functions cat() and paste()

- paste(): Allows to combine different elements into a character string, e.g., different words to build a long character string with a full sentence. Always returns a character vector.
- cat(): Can also be used to combine different elements, but will immediately show the result on the console. Used to displays some information to us as users. Always returns NULL.

The function cat() is used to concatenate one or multiple elements and immediately show the result to us as users. To get nice line-by-line output we can add a "\n" which is interpreted as a line break by our computer. A few examples:

```
# Output simple character string
cat("What a great day!\n")
## What a great day!
# Combine a variable `x` with a character string + \n
x <- "What a great day"
cat(x, "today!\n")
```

```
# Concatenate character strings and elements of named numeric vector
housecat <- c("height" = 46, "weight" = 4.5)
cat("The average house cat is", housecat["height"],
    "cm tall\nand weighs", housecat["weight"], "kilograms.\n")
```

Concatenate strings

The other function we will use is paste() which works similarly but is used for a different purpose. As cat(), paste() can take up a series of elements and combines them to a longer character string.

```r
res <- paste("The average house cat is", housecat["height"],
             "cm tall\nand weighs", housecat["weight"], "kilograms.\n")
length(res)
## [1] 1
class(res)
## [1] "character"
cat(res)
```

Let's try creating a simple example function. This function will take in a numeric value as input, and return the squared value.

```r
square_it<-function(x){
square=x*x
return(square)
}

square_it(5)
```

Drill: A

```r
say_hello <- function() {
    cat("Hello World!\n")    # Hello World + line break
}
# Call function
say_hello()
```

- **Name**: say_hello.
- **Arguments**: None.
- **Instructions**: Outputs "Hello world!" on the console.
- **Return value**: No explicit return (output).

As the function has no input arguments, nothing has to be declared between the round brackets (`()`) when calling the function. We are not even allowed to do so (try `say_hello("test")`).

```r
test <- say_hello()
## Hello World!
print(test)
## NULL
class(test)
```

Drill: B

- **Name**: say_hello (same name; redeclare function).
- **Arguments:** One argument x.
- **Instructions:** Paste and output the result using cat().
- **Return value:** No explicit return (output).
  - `# Re-declare the function (overwrites the previous one!)`

- `say_hello <- function(x) {`
- `    cat("Good morning", x, "\n")`
- `}`
- `# Call function`
- `say_hello("Jochen")`

We now have one input argument to control the behavior of the function.

```
say_hello()
say_hello("Brenda")
```

Drill: C

- **Name**: hello (new name).
- **Arguments:** One argument x.
- **Instructions:** Combine "Hi" and argument x and store the resulting character string on res. Do not print/show the result.
- **Return value:** Explicit return of the result.

```
# Declare the function
hello <- function(x) {
 res <- paste("Hi", x)
return(res)
}
result <- hello("Peter")
```

```
print(result)
## [1] "Hi Peter"
class(result)
## [1] "character"
```

Quick detour: return() and invisible()

Whenever writing a function, we shall always have an explicit return at the end of the function.

- return(): Returns one object. The return value is printed unless assigned to an object (implicit printing).
- invisible(): Also returns one object. Will not be printed, but can still be assigned to an object.

```
# Using return()
hello_return <- function(x) {
    res <- paste("Hi", x)
    return(res)
}

# Using invisible()
hello_invisible <- function(x) {
    res <- paste("Hello", x)
    invisible(res)
}

# The difference can be seen when we call the two functions.

hello_return("Maria")
## [1] "Hi Maria"
hello_invisible("Maria")


#When calling hello_return("Maria") we can immediately see the result ("Hi
#Maria"), while we do not get any output when calling hello_invisible("Maria")

result1 <- hello_return("Maria")
result2 <- hello_invisible("Maria")
print(result1)
## [1] "Hi Maria"
print(result2)
```

Invisible returns are used frequently in *R* by a wide range of functions. One example is the function boxplot()

```
set.seed(6020)          # Random seed
x   <- rnorm(200, 20, 3) # Draw 200 random values
res <- boxplot(x, main = "Demo Boxplot")
```

Let us check what the function returns (invisibly):

```
res
```

Drill: D

- **Name**: hello (redeclare the function).
- **Arguments:** One argument x.
- **Instructions:** Use paste() to create the character string and show it on the console using cat(). Then, calculate (count) the number of characters in this string.
- **Return value:** Explicit return; number of characters of the newly created string.

```
# Re-declare the function
hello <- function(x) {
    # First, create the new string using paste, and immediately print it
    # As we no longer need 'x' later on, we simply overwrite it here.
    x <- paste("Hello", x)
    cat(x, "\n")
    # Count the number of characters in 'x'
    res <- nchar(x)
    # Return the object 'res'
    return(res)
}
result <- hello("Max")
class(result)
## [1] "integer"
result
```

```
my_love<- function(myLove = "Disney") {
  paste("I like to visit ", myLove)
}

my_love("Niagara Falls")
my_love("Statue of Liberty")
my_love() # will get the default value, which is Disney
my_love("The Grand Canyon")
my_love("Chicago")
```

```
new.function_1<-function(a){
for(i in1:a){
   b <- i^2
print(b)
}
}

# Call the function new.function supplying 6 as an argument.
new.function_1(6)
```

Multiple arguments

```
# Re-declare the function once more
# Shows the message using cat() and invisible returns the same
# character string to be used later if needed.
hello <- function(x, greeting) {
    res <- paste(greeting, x)
    cat(res, "\n")
    invisible(res)
}
hello("Jordan", "Hi")
## Hi Jordan
hello("Reto", "Good afternoon")
```

```r
# Declare a second 'hello2' function
hello2 <- function(x, name) {
    res <- paste(x, name)
    cat(res, "\n")
    invisible(res)
}
# Calling 'hello2()' and 'hello()'
hello2("Hi", "Eva!")
hello("Eva!", "Hi")
```

Missing arguments

```r
hello("Rob")

# Additional (unused!) argument 'prefix'
hello <- function(x, greeting, prefix) {
    # Combine and return
    res <- paste(greeting, x)
    cat(res, "\n")
    invisible(res)
}
hello("Rob", "Hello")
```

Also,

```r
# Create a function with arguments.
new.function_2<-function(a,b,c){
  result <- a * b + c
print(result)
}

# Call the function by position of arguments.
new.function_2(5,3,11)

# Call the function by names of the arguments.
new.function_2(a =11, b =5, c =3)
```

Let’s try more:

```r
fahr_to_cel<- function(temp_F) {
temp_C<- (temp_F - 32) * 5 / 9
  return(temp_C)
}

fahr_to_cel(32)
fahr_to_cel(78)
```

```
mean_median<- function(vector){
    mean <- mean(vector)
    median <- median(vector)
    return(c(mean, median))
}
print(mean_median(c(1, 1, 1, 2, 3)))
```

```
add_or_subtract<- function(first_num, second_num, type = "add") {

  if (type == "add") {
first_num + second_num
  } else if (type == "subtract") {
first_num - second_num
  } else {
    stop("Please choose `add` or `subtract` as the type.")
  }

}

add_or_subtract(10, 6, type = "add")

add_or_subtract(5, 6, type = "subtract")

add_or_subtract(5, 6, type = "multiply")
```

Alternative function syntax

Version 1: The preferred one.

```
add2 <- function(x) {
  return(x + 2)
}
```

Version 2: One-liner.

```
add2 <- function(x) { return(x + 2) }
```

Version 3: Without brackets.

```
add2 <- function(x) return(x + 2)
```

Version 4: Without brackets, without explicit return.

```
add2 <- function(x) x + 2
```

Version 5: Without explicit return (but brackets).

```
add2 <- function(x) {
    x + 2
}
```

Argument specification

When calling functions, the arguments to the function can be named or unnamed. Named arguments are always matched first, the remaining ones are matched by position.

Let us use this function again:

```
Re-declare the function
hello <- function(x, greeting) {
    res <- paste(greeting, x)
    cat(res, "\n")
    invisible(res)
}
```

All unnamed

```
hello("Rob", "Hello")
```

All named: Alternatively, we can always name all arguments.

```
hello(x = "Rob", greeting = "Hello")
hello(greeting = "Hello", x = "Rob")
hello(greeting = "Hello", "Rob")
hello("Rob", greeting = "Hello")
hello(x = "Rob", "Hello")
hello("Rob", greeting = "Hello")
hello("Rob", gr = "Welcome")
```

Ref:

- Hadley Wickham’s Advanced R book
- Roger Peng’s R Programming for Data Science book
- DataCamp’s Intermediate R course
- Coursera’s R Programming course
- Data Carpentry (http://datacarpentry.org/), data camp, data quest, Kaggle
- Harvard Chan Bioinformatics Core (HBC) under the open access terms of the Creative Commons Attribution license (CC BY 4.0),
- The Book of R: A First Course in Programming and Statistics by Tilman M. Davies
- UC Business Analytics R Programming Guide and R_bootcamp