Topic 4: Functions in R

ISOM3390: Business Programming in R

Functions in R

- A function is a set of R statements that:
  - takes input
  - does something with that input
  - and returns the result
- Functions are needed when we want to automate certain tasks that we have to repeat over and over.
- R has a rich collection of functions for performing calculations necessary for statistical analysis.
- Beyond the core distribution there are thousands of packages in which developers have written new functions for specialized tasks.

Defining a Function

User-defined functions can be created using the `function (argument_1, argument_2, ...)` directive and are stored as objects of type **closure** (later).

Once a function is defined, it can be bound to a name with `<-` (the binding step is not compulsory; Anonymous functions later).

```
fn <- function(x1, x2, ... ) {
               y1 <- code about x1, x2, ...
               y2 <- code about x1, x2, ...
               etc.
               output
}
```

- Here, `x1`, `x2`, etc. are the objects the function takes as input.
- The `output` line (without `<-`) contains the R object we want the function to return; Or use `return()` to explicitly specify the output to be returned.

Function Components

All R functions have three parts:

- the `body()`, the code inside the function.
- the `formals()`, the list of formal arguments which controls how we can call the function.
- the `environment()`, the "map" of the location of the function's variables.

```
f <- function(x) x^2
f
```

```
## function(x) x^2
```

```
formals(f)
```

```
## $x
```

```
body(f)
```

```
## x^2
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

Function Parameters

The **parameters** of a function are special kind of variables that are used to refer to data provided as input to the function.

When a function is defined, an ordered list of parameters are specified within parentheses and separated by comma.

The `formals()` function or `args()` can be used to return a list of all the parameters of a function:

```
formals(sd)
```

```
## $x
##
##
## $na.rm
## [1] FALSE
```

```
args(sd)
```

```
## function (x, na.rm = FALSE)
## NULL
```

Parameters potentially have **default values**.

Calling Functions

Arguments are the actual input expression passed/supplied to a function when a function is called/invoked,

In a function call/invocation, the arguments for that call are evaluated, and the resulting values are assigned/bound to the corresponding parameters.

```
f <- function(a, b = 1, c = NULL) {
  if(is.null(c)) a ^ 2 + b
}

# Calling the function in an interactive context causes the result to be automatically printed.
# Function arguments can be missing for parameters having default values.
f(2)
```

```
## [1] 5
```

```
f <- function(a, b) {
  a ^ 2
}
# Arguments for a call are evaluated lazily; that is, they are evaluated only as needed.
f(2)
```

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

Mapping Arguments to Parameters

When calling a function, arguments can be specified by position or by name.

```
mydata <- rnorm(100)
sd(mydata)
```

```
## [1] 1.056133
```

```
sd(x = mydata)
```

```
## [1] 1.056133
```

```
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 1.056133
```

```
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 1.056133
```

```
# matching by partial name
sd(na = FALSE, x = mydata)
```

```
## [1] 1.056133
```

When positional matching is mixed with matching by name, arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.

More on Default Parameter Values

In addition to just constant values, the default value can be defined in terms of other parameters:

```
normalize <- function (x,  m = mean(x),  s = sd(x)) {
          (x - m) / s
}
z <- c(1, 3, 6, 10, NA)
normalize(z)
```

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

```
normalize <- function (x,  m = mean(x, na.rm = y),  s = sd(x, na.rm = y), y = FALSE) {
          (x - m) / s
}
normalize(z, y = TRUE)
```

```
## [1] -1.0215078 -0.5107539  0.2553770  1.2768848         NA
```

```
# This works, but the syntax is a little clunky.
```

The `...` Construct

`...` (called an ellipsis or three dots) captures any number of arguments that aren't otherwise matched, and can be easily passed on to other functions.

There are essentially two situations when we want to use `...`:

- When the number of arguments passed to the function can't be known in advance:

```
args(c)
```

```
## function (...)
## NULL
```

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL)
## NULL
```

```
#Any arguments that appear after ... on the argument list must be fully named.
paste("a", "b", sep = ":")
```

```
## [1] "a:b"
```

- When other functions are called within a function and these functions can have a variable number of arguments:

```
normalize <- function (x, m = mean(x, ...), s = sd(x, ...), ...) {
          (x - m) / s
}
normalize(z, na.rm = TRUE)
```

```
## [1] -1.0215078 -0.5107539  0.2553770  1.2768848         NA
```

```
commas <- function(...) stringr::str_c(..., collapse = ", ")
commas(letters[1:10])
```

```
## [1] "a, b, c, d, e, f, g, h, i, j"
```

Explicit `return` Statements

The value returned by the function is usually the last statement it evaluates. But we can choose to return early by using `return()`:

```
f <- function(x) {
  if (!x) {
    return(something_short)
  }
  # Do something that takes many lines to express
}
```

Everything that Happens is a Function Call

This includes

- infix operators like +:

```
x <- 8:10; y <- 3:5
x + y
```

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

```
# ` allows us to refer to functions or variables that have otherwise reserved or illegal names
`+`(x, y)
```

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

- flow control operators like `for`, `if`, and `while`:

```
for (i in 1:2) print(i)
```

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

```
`for`(i, 1:2, print(i))
```

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

```
if (i == 1) print("yes!") else print("no.")
```

```
## [1] "no."
```

```
`if`(i == 1, print("yes!"), print("no."))
```

```
## [1] "no."
```

- subsetting operators like `[` and `$`:

```
x[3]
```

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

```
`[`(x,3)
```

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

- and even the curly brace `{`.

```
{ print(1); print(2); print(3) }
```

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

```
`{`(print(1), print(2), print(3))
```

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

What Should be a Function?

Things we're going to re-run, especially if it will be re-run with changes.

Chunks of code we keep highlighting and hitting return on.

Chunks of code which are small parts of bigger analyses.

Chunks which are very similar to other chunks.

Organizing R Code

Functions tie related commands into one object to create organization. Well organised code is:

- easier to understand,
- easier to work with,
- easier to build into larger things

Another means is putting the source code for support functions into their own script files and loading them when needed using the `source` command:

In the `functions.R` file:

```
add <- function(a, b) {
   a + b
}
```

In the `script.R` file:

```
source('functions.R')
add(2, 3)
```

How to Bind a Value to a Name?

```
a <- 1
b <- 2
f <- function(x) {
  a * x + b
}

g <- function (x) {
  a <- 2
  b <- 1
  f(x)
}
```

What is the return value of calling `g(2)`?

The result will be 4. Using global variable a and b.

Symbol-Value Bindings

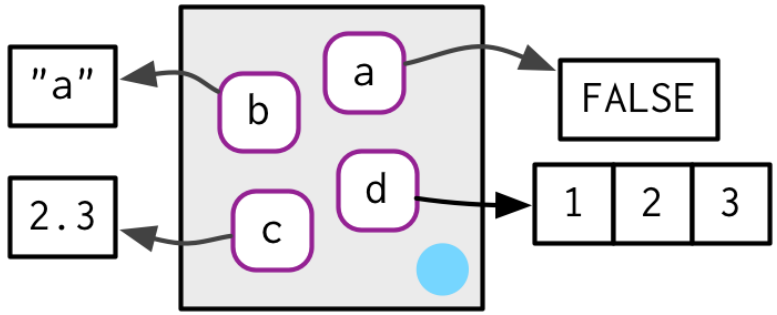
- The symbols which occur in the body of a function can be divided into three classes:
  - **Formal parameters**: those occurring in the argument list of the function.Their values are determined by the process of binding the actual arguments to the formal parameters.
  - **Local variables**: those whose values are determined by the evaluation of expressions in the body of the functions.
  - **Free variables**: not formal parameters or local variables
- How does R determine values for free variables?

The **scoping rules** of a language determine how a value is associated with a free variable in a function.

Environments: A Data Structure Powering Scoping

The job of an environment is to associate, or bind, a set of names to a set of values.

We can think of an environment as a bag of names each pointing to an object stored somewhere in memory.



Environments can be created with the `new.env()` function.

```
(e1 <- new.env())
```

```
## <environment: 0x7fd31f707f90>
```

```
# Printing an evironment just displays its memory address.
```

We can get and set elements of an environment with `$` and `[[` in the same way as with a list or using `get()` and `assign()`:

```
e1[["a"]] <- FALSE
e1$b <- "a"
e1[["c"]] <- 2.3
e1$d <- c(1, 2, 3)
e1[["d"]]
```

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

```
assign("f", c("business programming", "business analytics"), envir = e1)
get("f", envir = e1)
```

```
## [1] "business programming" "business analytics"
```

Most of the syntax for lists also works for environments.

We can coerce a list to be an environment and vice versa:

```
(a_list <- as.list(e1))
```

```
## $a
## [1] FALSE
##
## $b
## [1] "a"
##
## $c
## [1] 2.3
##
## $d
## [1] 1 2 3
##
## $f
## [1] "business programming" "business analytics"
```

```
#...and back again.
as.environment(a_list)
```

```
## <environment: 0x7fd31f48c408>
```

The `ls()` and `ls.str()` functions take an environment argument (defaults to the current environment), allowing us to list its content.

```
ls(envir = e1)
```

```
## [1] "a" "b" "c" "d" "f"
```

```
ls.str(envir = e1)
```

```
## a :  logi FALSE
## b :  chr "a"
## c :  num 2.3
## d :  num [1:3] 1 2 3
## f :  chr [1:2] "business programming" "business analytics"
```

We can test to see if a variable exists in an environment using the `exists()` function:

```
exists("a", e1)
```

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

```
exists("non-existing", e1)
```

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

`rm()` can be used to remove objects in an environment.

```
rm(a, b, envir = e1)
rm(list = c("c", "d"), envir = e1)
as.list(e1)
```

```
## $f
## [1] "business programming" "business analytics"
```

The following code can be used to delete all objects in the workspace:

```
rm(list = ls())
```

Special Environments

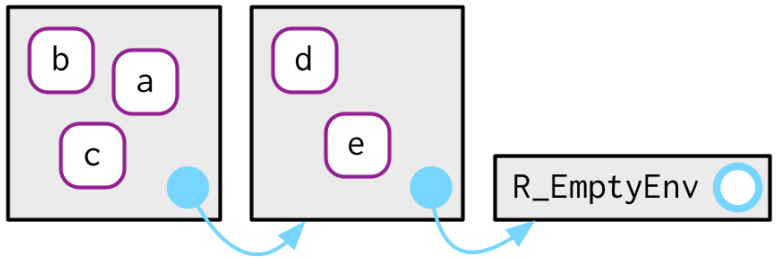
- The **current environment**, accessed with `environment()` (with no arguments provided), is the environment in which code is currently executing.
- The **global environment**, printed as `R_GlobalEnv` or `.GlobalEnv`, is also called the **"workspace"**. It's where all interactive (i.e. outside of a function) computation takes place.
- The **empty environment**, printed as `R_EmptyEnv`, is the ultimate ancestor of all environments and the only environment without a parent.
- The **base environment** is the environment of the base package. Its parent is the empty environment.

Parents

Each environment has a parent except the **empty environment**.

Environments are chained/nested. And the ancestors of an environment include all parent environments up to the **empty environment**.

```
(e2a <- new.env(parent = emptyenv()))
## <environment: 0x7fd31ec539f0>
(e2b <- new.env(parent = e2a))
## <environment: 0x7fd31eca9298>
parent.env(e2b)
## <environment: 0x7fd31ec539f0>
parent.env(e2a)
## <environment: R_EmptyEnv>
```



Function Environments

Most environments are not created with `new.env()` but are created as a consequence of using functions.

There're 4 types of environments associated with a function:

- Binding a function to a name with `<-` defines a **binding environment**. It determines how we find the function.
- The **enclosing environment** is the environment where the function is defined. It determines how the function finds values as we will see.

```
environment()

## <environment: R_GlobalEnv>

e1$g <- function(x) x + 10
environment(e1$g) # The enclosing environment is the current environment

## <environment: R_GlobalEnv>

exists("g", envir = environment()) # But the current environment doesn't contain g.

## [1] FALSE
```

Enclosing: Global Env
Binding: Linked variable (Where can find this var)

- Calling a function creates an **execution environment** that holds the the execution and stores variables created during execution.

```
simple_function <- function () {
  print(environment()) # Returns the current environment without arguments being specified
}
simple_function()
environment(simple_function) # Returns the enclosing environment of the function when it takes a function
```

- Every execution environment is associated with a **calling environment** from which is the function was called.

```
# define g inside f
y <- 3
f <- function (x) {
  y <- 1
  g <- function (x) {
    (x + y) / 2
  }
  g(x)
}
```

This will have result of 3

```
# define g outside f
y <- 3
f <- function (x) {
  y <- 1
  g(x)
}
g <- function (x) {
  (x + y) / 2
}
```

This will have result of 4

What would be result of executing `f(5)` in either of the two cases?

Lexical (Static) Scoping

How to resolve the free viariable bindings (i.e., find a value for a symbol) in R?

- R walks up the **chain of environments** and uses the first binding for a symbol it finds.

Then the question goes to how R determines the order to search the environments for bindings.

- The parent of the **execution environment** is the **enclosing environment** (as opposed to **calling environment**) of the function.
- The values of free variables encountered during execution are searched for in the environment in which the function is defined.
- The scoping rule is called **lexical scoping** or **static scoping**.
    - It's the code's static structure, rather than the dynamic relationship manifested through the code execution, determines where to search for bindings.

A Revisit to the Motivating Example

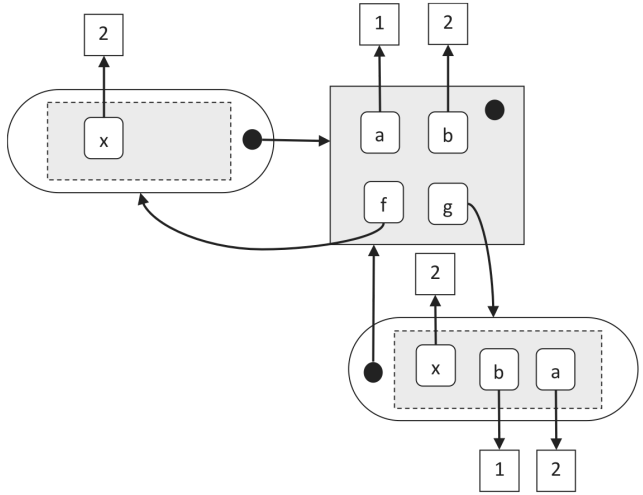
```
a <- 1
b <- 2

f <- function(x) {
  a * x + b
}

g <- function (x) {
  a <- 2
  b <- 1
  f(x)
}

g(2)
```

The value of a = 1 & b = 2 will be used



"First-Class Functions"

In R, functions are objects in their own right.

By "**first-class**", we mean that functions can be computed, passed, stored, etc. wherever other objects can be computed, passed, stored, etc.

For example, a function is able to return another function as its value.

```
a <- 1
b <- 2
f <- function(a, b){
  function(x) a * x + b  # An anonymous function
}
g <- f(2, 1)
```

The result will return 5. The function(x) //Anonymous function has scope inside function(a,b), so the value of a & b will be used from f(2,1) instead of global variables 1 & 2

What is the object referenced by `g`? What returns by calling `g(2)`?

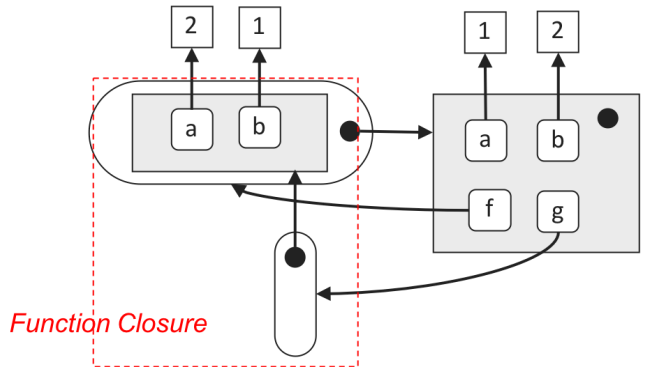
Function Closures

A **function closure** or **closure** is a function together with an environment that provides value bindings for free varialbes.

A function call uses both parts of the closure because the code of the function is evaluated using variables encapsulated in the environment part.

Almost all functions in R are closures as they remember the environment where they are created.

```
a <- 1
b <- 2
f <- function(a,b){
  function(x) {
    a * x + b
  }
}
g <- f(2, 1)
```



When creating a function inside another function, the enclosing environment of the child function is the execution environment of the parent, which is no longer ephemeral.

Example: Generating a Family of Power Functions

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
```

Use `power()` to make two new functions:

```
square <- power(2)
```

```
## <environment: 0x7fd31f5796d0>
```

```
square
```

```
## function(x) x ^ exponent
## <environment: 0x7fd31f5796d0>
```

```
cube <- power(3)
```

```
## <environment: 0x7fd31f267520>
```

```
cube
```

```
## function(x) x ^ exponent
## <bytecode: 0x7fd31ead4df8>
## <environment: 0x7fd31f267520>
```

The two closures differ in their enclosing environments, which bind distinct values to the symbol `exponent`.

We can convert an environment to a list to see its content:

```
as.list(environment(square))
```

```
## $exponent
## [1] 2
```

```
as.list(environment(cube))
```

```
## $exponent
## [1] 3
```

A function that makes new functions is called a **function factory**.

Super/Deep Assignment: <<–

The change made by the regular assignment, <–, within the function are local and temporary.

```
a <- 1
f <- function(x) {a <- a + x; a}
f(1)
```

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

What change has been made to a? No changes has been made to a. The a preserve the value of a.

Instead, the super/deep assignment arrow, `<<-`, modifies an existing variable found by walking up the chains of environments.

```
a <- 1
g <- function(x) {a <<- a + x; a}
g(1)
```

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

```
a
```

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

If `<<-` doesn't find an existing variable, it creates one in the global environment.

Package Environments

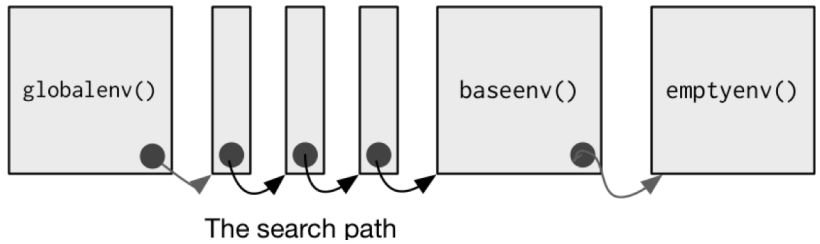
In R, the use of packages gives another way to group R code and create organization on it.

```
search() # It gives the packages that are currently loaded
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

Each entry in the list represents a package environment that contains every publicly accessible function defined in the corresponding package.

They make up the search path (starting with the global environment) that R walks up to find the first binding for a symbol.



We can access any environment on the search list using `as.environment()`:

```
as.environment("package:stats")
```

```
## <environment: package:stats>
## attr(,"name")
## [1] "package:stats"
## attr(,"path")
## [1] "/Library/Frameworks/R.framework/Versions/3.4/Resources/library/stats"
```

The following code prints all publicly accessible functions in the `package:stats` environment:

```
ls(envir = as.environment("package:stats"))
```

Each time a new package is loaded with `library()`, it is inserted between the **global environment** and the package that was previously at the top of the search path.

```
library("tidyr")
z <- list(mean = "fluffernutter", median = 3.4)
attach(z)
mean
```

```
## [1] "fluffernutter"
```

```
search()[2:3]
```

```
## [1] "z"             "package:tidyr"
```

`attach()` creates an environment, slots it into the list right after the global environment and populates it with the objects we're attaching.

```
mean <- function(x) x + 1
# The mean() function in the package:base environment is masked by the user-defined function
mean(1:6)
```

```
## [1] 2 3 4 5 6 7
```

```
# The :: operator lets us specify which mean() function we want
base::mean(1:6)
```

```
## [1] 3.5
```