ISOM 3390: Business Programming in R

Instructor: Bingjie Qian (PhD candidate)

Summer 2020, HKUST

Topic 4: Functions

4.1 An Overview of Functions

An example: loan approval decision problem

A bank needs to consider whether or not to offer someone a loan:

Name	Income	Years	Criminal	Decision
Amy	27	4	No	?
Sam	32	1.5	No	?
Jane	55	4	Yes	?

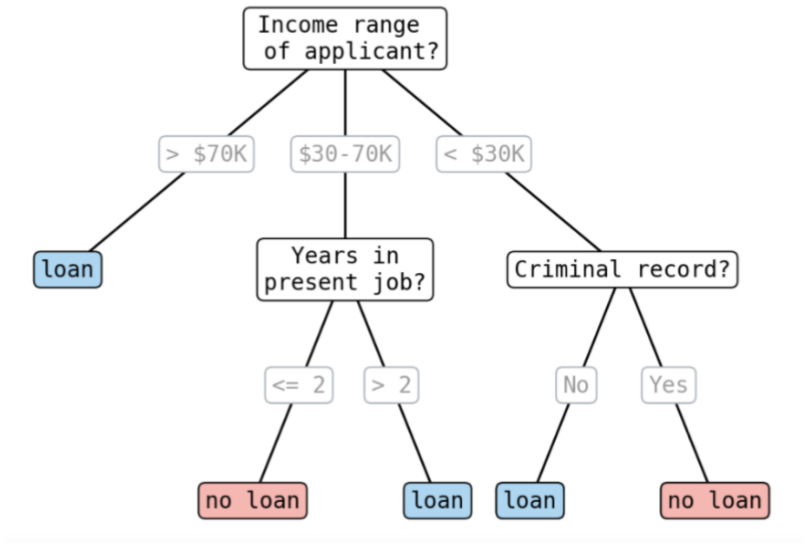

Fig 1. Loan approval example

Decide whether to offer Amy a loan:

```r
Amy <- list(income = 27, years = 4, criminal = F)

if (Amy$income > 70) {
  print("loan")
} else if (Amy$income > 30) {
    if (Amy$years <= 2) {
      print("no loan")
    } else print("loan")
} else if (Amy$criminal) {
    print("no loan")
} else print("loan")

## [1] "loan"
```

If we want to make a decision for other customers, we will have to repeat the above codes over and over.

We should consider writing a **function** whenever we need to copy and paste a block of codes multiple times.

Functions in R

Just as data structures tie related values to an object, functions tie related expressions to an object.

A **function** is a set of expressions that:

- takes input (called **arguments**);
- does something with the input;
- and, in many cases, returns the result (called **return value**).

Functions are needed when we want to automate certain tasks that we have to repeat over and over.

4.2 Creating Functions

Syntax:

```
FunctionName <- function(x1, x2, ... ) {
  operations on x1, x2, ...
  output   # or return(output)
}
```

- We use the keyword `function` to create a function and assign it to `FunctionName`, which is a function object.
- `x1, x2, ...` are function **parameters**, provided as input to the function.
- The function body is a group of expressions contained in `{` and `}`; executed only when the function is called.
  - If the function body is a single-line statement, braces are not necessary. For example, `plus_one <- function(x) x + 1`
- The evaluation result of the output line (without `<-`) implicitly becomes the return value; or use `return()` to specify the return value explicitly.

- The resulting function object can be bound to a name (`FunctionName`) with `<-`, but it is not compulsory (e.g., anonymous functions later).

Now let's create a function to solve the loan approval problem:

```
approve_loan <- function(customer) {
  if (customer$income > 70) {
    return("loan")
  } else if (customer$income > 30) {
      if (customer$years <= 2) {
        return("no loan")
      } else return("loan")
  } else if (customer$criminal) {
      return("no loan")
  } else return("loan")
}

approve_loan
```

```
## function(customer) {
##   if (customer$income > 70) {
##     return("loan")
##   } else if (customer$income > 30) {
##       if (customer$years <= 2) {
##         return("no loan")
##       } else return("loan")
##   } else if (customer$criminal) {
##       return("no loan")
##   } else return("loan")
## }
```

```
body(approve_loan)
```

```
## {
##     if (customer$income > 70) {
##         return("loan")
##     }
##     else if (customer$income > 30) {
##         if (customer$years <= 2) {
##             return("no loan")
##         }
##         else return("loan")
##     }
##     else if (customer$criminal) {
##         return("no loan")
##     }
##     else return("loan")
## }
```

4.3 Calling Functions

We can call or invoke a function with actual inputs (**arguments**) to run the function's body.

The arguments are bound to matched parameters so as to be operated by expressions to generate the return value.

```
Amy <- list(income = 27, years = 4, criminal = F)
approve_loan(Amy)

## [1] "loan"

Sam <- list(income = 32, years = 1.5, criminal = F)
approve_loan(Sam)

## [1] "no loan"
```

Argument Matching

When calling a function, arguments match parameters **by position** or **by name**.

Take the function `sd()` as an example.

```
str(sd)

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

- The function has two parameters `x` and `na.rm`, `na.rm` has a defualt value `FALSE`.
- It computes the standard deviation of the values in `x`. If `na.rm` is `TRUE` then missing values are removed before computation proceeds.

Call the function `sd()`:

```
set.seed(0)
mydata <- rnorm(20)
mydata

##  [1]  1.262954285 -0.326233361  1.329799263  1.272429321  0.414641434
##  [6] -1.539950042 -0.928567035 -0.294720447 -0.005767173  2.404653389
## [11]  0.763593461 -0.799009249 -1.147657009 -0.289461574 -0.299215118
## [16] -0.411510833  0.252223448 -0.891921127  0.435683299 -1.237538422

sd(mydata)       # by position

## [1] 1.021491

sd(x = mydata)   # by name

## [1] 1.021491

sd(x = mydata, na.rm = FALSE)    # by name

## [1] 1.021491
```

```r
sd(na.rm = FALSE, x = mydata)    # can switch the order
```

```
## [1] 1.021491
```

```r
sd(na = FALSE, x = mydata)       # by partial name (not recommended)
```

```
## [1] 1.021491
```

```r
sd(na.rm = FALSE, mydata)        # mixed
```

```
## [1] 1.021491
```

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

Default Parameter Values

In `sd()`, the parameter `na.rm` has a default value `FALSE`.

In addition to constant values, a parameter's default value can be defined in terms of other parameters.

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

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

In the above example, the default values of `m` and `s` are defined in terms of `x`.

What if `x` contains `NA`s?

```r
normalize(x = c(1, 3, 6, 10, NA))
```

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

The return value is a vector of `NA`s, because `m` and `s` are `NA`s:

```r
mean(c(1, 3, 6, 10, NA))
```

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

```r
sd(c(1, 3, 6, 10, NA))
```

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

```r
mean(c(1, 3, 6, 10, NA), na.rm=TRUE)  # the default value of na.rm is FALSE,
set it to TRUE
```

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

```r
sd(c(1, 3, 6, 10, NA), na.rm=TRUE)
```

```
## [1] 3.91578
```

Let's revise the function `normalize()` so that `NA`s in `x` are removed.

We can include another parameter `y` in `normalize()` to control the parameter `na.rm` in `mean()` and `sd()`.

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

normalize(c(1, 3, 6, 10, NA))

## [1] NA NA NA NA NA

normalize(c(1, 3, 6, 10, NA), y=TRUE)

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

Alternatively, we can use `...` (**ellipsis**, or **three dots**) to solve the problem.

`...` captures any number of arguments that aren't matched, and can easily pass these arguments to *inner function* calls. Inner functions are the functions that are called within a function (e.g., `mean()` and `sd()` are the inner functions of `normalize()`).

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

normalize(c(1, 3, 6, 10, NA))

## [1] NA NA NA NA NA

normalize(c(1, 3, 6, 10, NA), na.rm = TRUE)

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

In the above example, `na.rm` is not in the argument list of the function `normalize()`, it is passed to the inner functions `mean()` and `sd()`.

`...` can also be used when the number of arguments passed to a function can't be known in advance.

```
str(paste)

## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
```

- The function `paste()` converts its arguments to character strings and concatenates them (separating them by `sep`).
- `...` represents one or more R objects, to be converted to character vectors.

```
paste("ISOM", "3390")      # 2 arguments in ...
```

```
## [1] "ISOM 3390"

paste("ISOM", "3390", "Business Programming")   # 3 arguments in ...

## [1] "ISOM 3390 Business Programming"

paste("ISOM", "3390", sep = "-")     # the arguments after ... in the argument
list must be named

## [1] "ISOM-3390"

paste("ISOM", "3390", "-")          # "-" is taken as one of the arguments in
...

## [1] "ISOM 3390 -"
```

[Task 1: Writing a Data Summarization Function]

In Task 5 of the 3rd in-class exercise, we used 3 different ways to generate a summarization of a data frame `pros.dat`. Specifically, we caclualate the groupwise sum of each column. The result is a matrix (or data frame) with 10 rows and 9 columns.

```
setwd("/Users/qbj/Desktop/Summer_2020/04_Functions")
pros.dat <- read.table("pros.dat")
set.seed(0)
grouping <- sample(1:10, size = 97, replace = T)

groupwise.sum <- rowsum(pros.dat, grouping)
groupwise.sum

##       lcavol  lweight age       lbph svi        lcp gleason pgg45     lpsa
## 1  11.818350 29.18716 500 -4.6744465   4  1.6724007      53   195 18.61620
## 2   8.233756 25.71542 432  1.1945128   0 -5.9904884      44   110 15.37785
## 3  11.344319 25.88829 444  3.1961628   2  1.6974904      50   249 17.88977
## 4   4.070979 21.74769 384 -1.9519282   0 -7.1421928      38    75 11.52228
## 5  14.689329 32.04309 555 -3.3360443   3  3.2147904      61   200 21.91695
## 6  16.077460 46.44447 850  3.3291378   1 -4.4274876      88   346 33.13438
## 7  19.085069 50.51550 904 -7.1246377   3  0.2620661      99   380 36.28621
## 8  11.935786 29.21921 490 11.2940771   3 -1.1274941      53   160 25.33853
## 9  11.788775 40.30110 700  7.4707160   1 -6.4797309      74   260 21.77831
## 10 21.907106 50.94550 936  0.3369441   4  0.9221852      95   390 38.54303
```

(a) Rewrite the code as a function with the name `groupwise.sum.fun` and the parameter list `(data, columns, grouping)`. It should allow users to change the data frame, the columns and the grouping rule. The parameter `columns` supports indexing with integer and character vectors.

If you run the following code:

```
grouping <- sample(2:5, size = 97, replace = T)
groupwise.sum.fun(data = pros.dat, columns = c(2, 8, 3, 6), grouping =
grouping)
```

The expected result should be:

.	lweight	pgg45	age	lcp
2	74.94312	364	1275	-10.944977
3	108.50418	975	1937	-1.612451
4	71.12394	350	1266	-8.492804
5	97.43620	676	1717	3.651771

(b) Define default values for the `columns` and `grouping` parameters for the `groupwise.sum.fun` function.

- The default value of `columns` should be all columns in the data frame.
- The default value of `grouping` should group all observations into a single group.

Tips: Try `rep.int(1, times = 3)` to see how it works.

If you run the following code:

```
groupwise.sum.fun(pros.dat)
```

The expected result should be:

.	lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45	lpsa
1	130.9509	352.0074	6195	9.734494	21	-17.39846	655	2365	240.4035

(c) Use `...` to revise the `groupwise.fun` function in (b) to allow us to use the `na.rm` option of the function `rowsum()` to handle missing values when needed.

Use the following code to create a new data frame with NAs to test your new function.

```
test <- pros.dat
test[2, 1:3] <- NA
head(test)
```

[End of Task 1]

4.4 Environments and Scoping Rules

Read the following code:

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

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

Question: what is the return value of calling `g(4)`?

```r
g(4)

## [1] 5
```

Find a Value for a Name

The names that occur in a function can be divided into three classes:

- **Parameters**: those occurring in the parameter list of the function. Their values are determined when calling the function and assigning the actual arguments to the parameters.
- **Local variables**: those whose values are determined by the evaluation of expressions in the body of the function. Local variables exist only within the function and are released when the function call ends.
- **Free variables**: those that are not parameters or local variables.

`a` in the function `f` is a free variable.

How does R determine the value of a *free variable*?

- It depends the **scoping rule** of R. The scoping rule of a programming language tells us how to find a value for a name.

Environments

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

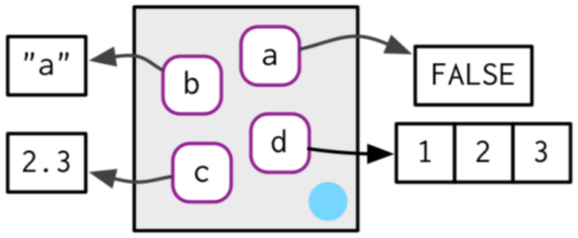

Fig 2. Environment

Environments are **chained/nested**:

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

- The **ancestors** of an environment include all parent environments up to the empty environment.

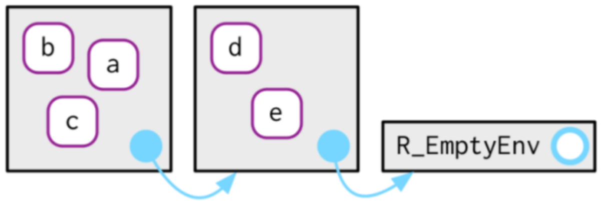

Fig 3. Parent Environment

Some special environments:

- The **empty environment** is the ultimate ancestor of all environments and the only environment without a parent. It can be accessed with `emptyenv()` and printed as `R_EmptyEnv`.

```
emptyenv()

## <environment: R_EmptyEnv>
```

- The **global environment** is where all interactive computations (i.e. outside of a function) take place and is also called the "workspace". It can be accessed with `globalenv()` and printed as `R_GlobalEnv` or `.GlobalEnv`. (Check the top right panel in RStudio for the global environment.)

```
globalenv()

## <environment: R_GlobalEnv>
```

- The **current environment** is where code is currently executing. It can be accessed with `environment()` (with no arguments provided).

```
environment()

## <environment: R_GlobalEnv>
```

Operations on Environments

- Create an environment with the `new.env()` function.

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

## <environment: 0x7fb801bd9b00>
```

- Get and set elements of an environment with `$` and `[[]]` in the same way as with a *list*.

```
e1$a <- FALSE
e1[["b"]] <- 3.5
e1[["c"]] <- c(1, 2, 3)
e1$a

## [1] FALSE
```

```
e1$b

## [1] 3.5

e1$c

## [1] 1 2 3
```

- Coerce an environment to a list and vice versa.

```
list1 <- as.list(e1)
list1

## $a
## [1] FALSE
##
## $b
## [1] 3.5
##
## $c
## [1] 1 2 3

as.environment(list1)

## <environment: 0x7fb802238738>
```

- List the objects in an environment with `ls()` and `ls.str()`.

```
ls(e1)        # list objects in e1

## [1] "a" "b" "c"

ls.str(e1)    # list objects and their structure in e1

## a :  logi FALSE
## b :  num 3.5
## c :  num [1:3] 1 2 3

ls()          # list objects in the current environment

##  [1] "a"             "Amy"           "approve_loan"  "e1"
##  [5] "f"             "g"             "grouping"      "groupwise.sum"
##  [9] "list1"         "mydata"        "normalize"     "pros.dat"
## [13] "Sam"

ls.str()

## a :  num 1
## Amy : List of 3
##  $ income  : num 27
##  $ years   : num 4
##  $ criminal: logi FALSE
## approve_loan : function (customer)
## e1 : <environment: 0x7fb801bd9b00>
## f : function (x)
## g : function (x)
## grouping :  int [1:97] 9 4 7 1 2 7 2 3 1 5 ...
## groupwise.sum : 'data.frame':    10 obs. of  9 variables:
##  $ lcavol : num  11.82 8.23 11.34 4.07 14.69 ...
##  $ lweight: num  29.2 25.7 25.9 21.7 32 ...
##  $ age    : int  500 432 444 384 555 850 904 490 700 936
##  $ lbph   : num  -4.67 1.19 3.2 -1.95 -3.34 ...
##  $ svi    : int  4 0 2 0 3 1 3 3 1 4
##  $ lcp    : num  1.67 -5.99 1.7 -7.14 3.21 ...
##  $ gleason: int  53 44 50 38 61 88 99 53 74 95
##  $ pgg45  : int  195 110 249 75 200 346 380 160 260 390
##  $ lpsa   : num  18.6 15.4 17.9 11.5 21.9 ...
## list1 : List of 3
##  $ a: logi FALSE
##  $ b: num 3.5
##  $ c: num [1:3] 1 2 3
## mydata :  num [1:20] 1.263 -0.326 1.33 1.272 0.415 ...
## normalize : function (x, m = mean(x, ...), s = sd(x, ...), ...)
## pros.dat : 'data.frame': 97 obs. of  9 variables:
##  $ lcavol : num  -0.58 -0.994 -0.511 -1.204 0.751 ...
##  $ lweight: num  2.77 3.32 2.69 3.28 3.43 ...
##  $ age    : int  50 58 74 58 62 50 64 58 47 63 ...
##  $ lbph   : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
##  $ svi    : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ lcp    : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
##  $ gleason: int  6 6 7 6 6 6 6 6 6 6 ...
##  $ pgg45  : int  0 0 20 0 0 0 0 0 0 0 ...
##  $ lpsa   : num  -0.431 -0.163 -0.163 -0.163 0.372 ...
## Sam : List of 3
##  $ income  : num 32
##  $ years   : num 1.5
##  $ criminal: logi FALSE
```

- Remove objects from a specified environment with `rm()`.

```
rm(a, b, envir = e1)   # alternatively, rm(list = c("a", "b"), envir = e1)
ls(e1)

## [1] "c"

rm(list = ls(e1), envir = e1)   # remove all objects in e1
ls(e1)

## character(0)

rm(list = ls())    # remove all objects in the current environment
ls()

## character(0)
```

Function Environments

When we define and call functions, new environments are created.

There're several important environments associated with a function:

- The **enclosing environment** is the environment where the function is *defined*. It determines how the function finds values.

```
simple_function <- function(x) x
environment(simple_function)      # enclosing environment of simple_function

## <environment: R_GlobalEnv>
```

When a funciton is passed into `environment()`, it returns the enclosing environment of the function. When nothing is passed into `environment()`, it returns the current environment.

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

```
simple_function <- function(x) {
  a <- 1
  print(environment())   # execution environment of the function
  ls.str(environment())
}

simple_function(2)       # call the function

## <environment: 0x7fb802063640>

## a :  num 1
## x :  num 2

environment(simple_function)  # enclosing environment of the function

## <environment: R_GlobalEnv>
```

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

Revisit the example:

```
rm(list = ls())
a <- 1
f <- function(x) {
  x + a
}
print(environment())    # enclosing environment of f

## <environment: R_GlobalEnv>

print(ls.str())

## a :  num 1
## f : function (x)
```

```
g <- function(x) {
  a <- 2
  print(environment())  # calling environment of f (execution environment of
g)
  print(ls.str())
  f(x)
}

g(4)

## <environment: 0x7fb801d72230>
## a :  num 2
## x :  num 4

## [1] 5
```

R cannot find the value of `a` in the execution environment of `f`, but it can find two different values of `a` in the enclosing environment of `f` (1), and the calling environment of `f` (2).

Which value will R use? How does R find the value of a free variable of a function?

Scoping Rule of R

- R walks up the **chain of environments** and uses the first value for a name it finds. If a name's value is not found in an environment, R will look in its parent, if still not found, then go to its parent's parent, so on and so forth.
- The **parent** of a function's **execution environment** is its **enclosing environment** (as opposed to its **calling environment**).
- Because the search for name-value bindings depends on the code's **static structure**, rather than the dynamic relationship manifested through the code execution, R's scoping rule is called **lexical scoping** or **static scoping**.

Therefore, for the above example, R will use `1` (in the enclosing environment) as the value of the free variable `a`.

4.5 Functions as First-Class Objects

First-Class Functions

In R, functions are objects in their own right. Moreover, they are first-class objects.

First-class functions are functions that can be computed, passed and stored wherever other objects can be computed, passed and stored.

We can pass functions as arguments to other functions, return them as the values from other functions, and assign them to variables or store them in data structures.

The names of functions do not have any special status; they are treated like ordinary variables with a function type.

```
a <- 1
f <- function(a) {
  print(environment())
  function(x) x + a     # return a function
}

g <- f(2)

## <environment: 0x7fb80190b678>
```

Question: What is the object referenced by g? What returns by calling `g(4)`?

```
g        # a function and its enclosing environment

## function(x) x + a
## <environment: 0x7fb80190b678>

g(4)

## [1] 6
```

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

Closures get their name because they **enclose** the environment of the parent function and can access all its variables. In the above example, `f` is the parent function of g, because g is created by `f` (g is defined within the body of `f`).

This is useful because it allows us to have two levels of parameters: a parent level that controls operation and a child level that does the work.

When you print a closure, you don't see anything terribly useful. That's because the function itself doesn't change. The difference is the enclosing environment, `environment(g)`.

Check the contents of the enclosing environment of g:

```
ls.str(environment(g))

## a :  num 2
```

[Task 2: Free Variables]

Compare the following two code snippets and predict what would be the result of calling `f(5)` in both cases. Explain why.

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

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

[End of Task 2]

Function Factories

Generate a family of power functions:

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

# use power() to make two new functions:
square <- power(2)
square

## function(x) x ^ exponent
## <environment: 0x7fb801f3a798>

square(10)

## [1] 100

cube <- power(3)
cube

## function(x) x ^ exponent
## <bytecode: 0x7fb80228fb10>
## <environment: 0x7fb8026a88e8>

cube(10)

## [1] 1000
```

The two closures differ in their enclosing environments, which bind different values to the name `exponent`. A function that makes new functions like `power()` is called a **function factory**.

Super/Deep Assignment

The change made by the regular assignment <- within a function are local and temporary.

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

f(1)

## [1] 3
```

Question: what change has been made to a?

```
a

## [1] 2
```

The value of a does not change.

Different from the regular assignment operator <-, the **super/deep assignment** operator <<- modifies an existing variable found by walking up the chain of environments.

```
a <- 2
g <- function(x) {
  a <<- a + x    # super/deep assignment
  a
}

g(1)

## [1] 3

a

## [1] 3
```

[Task 3: Deep Assignment]

We define a function factory for creating counters that record the number of times a function has been called:

```
new_counter <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
```

Then we create two counters by using this function factory as follows:

```r
counter_1 <- new_counter()
counter_2 <- new_counter()
```

Run the following R code

```r
counter_1()

## [1] 1

counter_1()

## [1] 2

counter_2()

## [1] 1
```

Please explain how these function counters work.

[End of Task 3]

4.6 Package Environment

Display all the packages that are currently loaded with `search()`:

```r
search()

## [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 name defined in that package.

```r
# ls(envir = as.environment("package:stats"))   # list all the names defined
in the package `stats`
```

They make up the **search path** (starting with the global environment) that R walks up to find the first binding for a name, e.g., `sd`, `mean`, `sum`, etc.

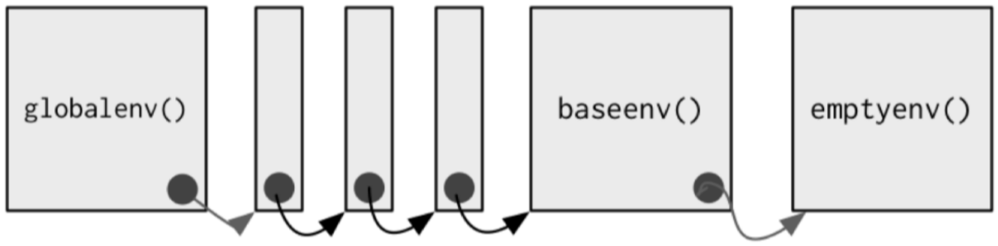

Fig 4. Search Path

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")
search()
```

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

`attach()` inserts an object into the search path, populating the resulting environment with names it contains.

```
z <- list(mean = "mean value", median = 3.4)
attach(z)
search()
```

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

Now when R search for the name `mean`, the first binding it finds is the `mean` in `z`:

```
mean
```

```
## [1] "mean value"
```

After detaching `z`, the first binding for the name `mean` will be the function defined in base R.

```
detach(z)
search()
```

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

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7fb802fdabf8>
## <environment: namespace:base>
```

Name Masking

In R, a function masks a function, and a non-function masks a non-function.

```
# function `mean()` from base R:
mean(1:20)
```

```
## [1] 10.5
```

```
# define a new function `mean()`:
mean <- function(x) x + 10
mean(1:20)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

We can use the `::` (double colon) operator to specify which function we want.

```
base::mean(1:20)
```

```
## [1] 10.5
```

Remove the new function to addres name conflicts:

```
rm(mean)
mean(1:20)
```

```
## [1] 10.5
```