R-Ladies NL Book-Club

Advanced R: Functions (Chapter 6)

Josephine Daub

2020-06-09

Welcome R-Ladies Netherlands Book-Club!

- R-Ladies is a global organization to promote gender diversity in the R community via meetups, mentorship in a safe and inclusive environment.

- **R-Ladies Netherlands Book-Club** is a collaborative effort between RLadies-NL chapters in Nijmengen, Rotterdam, Den Bosch, Amsterdam, Utrecht.

- We meet every **2 weeks** to go through one of the chapters of Hadley Wickam *Advanced R*, and run through exercises to put the concepts into practice.

Today's Session!

- Starts with a 30-45 min presentation
- Breakout session - we **split** into breakout rooms to practice exercises.
- Please use the **HackMD** (shared in email and in chat) to present yourself, ask overarching questions, and to find your break out room.
- Use the **chat** to participate in the discussion during the presentation and your breakout session.
- The Bookclub github repository has also been made available.
- Any questions?

Resources

- Solutions to the exercises from *Advanced R* can be found in the Advanced R Solutions Book
- The R4DS book club repo has a Q&A section : github.com/r4ds/bookclub-Advanced_R
- We are always looking for new speakers! If you are interested, please sign up to present a chapter at https://rladiesnl.github.io/book_club/

Functions

Outline

The outline for today is:

- Function fundamentals
- Function composition
- Lexical scoping
- Lazy evaluation
- ... (dot-dot-dot)
- Exiting a function
- Function forms

Let's get to it!

Function fundamentals

In R, a **function** has three parts:

1. `formals()`: list of arguments that control how you call the function
2. `body()`: the code inside the function
3. `environment()`: data structure that determines how the function finds the values associated with the names

The formals and body are specified **explicitly** when creating a function
The environment is specified **implicitly**, based on *where* you define a function

Example

```r
f02 <- function(x, y) {
  # A comment
  x + y
}

formals(f02)
```

```
## $x
##
##
## $y
```

```r
body(f02)
```

```
## {
##     x + y
## }
```

```r
environment(f02)
```

Function fundamentals

Functions can have additional `attributes()`, such as `srcref` (source reference):

```
attr(f02, "srcref")
```

```
## function(x, y) {
##   # A comment
##   x + y
## }
```

Exception: **primitive functions** call C code directly:

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

```
`[`
```

Function fundamentals

Functions are **objects**

Create a function object (with `function`) and bind it to a name with `<-`:

```
f01 <- function(x) {
  sin(1 / x ^ 2)
}
```

Or: you can choose not to give a name to get a **anonymous function**:

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

Or: put functions in a **list**:

Function composition

How to compose **multiple** function calls?

Nest the function calls

```
sqrt(mean(x))
```

Save **intermediate results**

```
y<-mean(x)
sqrt(y))
```

Use **magrittr** package to use pipe (`%>%`)

```
library(magrittr)

x %>%
  mean() %>%
  sqrt()
```

Lexical scoping

Scoping: finding the value associated with a name

What will `g01()` return?

```r
x <- 10
g01 <- function() {
  x <- 20
  x
}

g01()
```

R uses lexical scoping: look up values of names based on how a function is defined, not how it is called

Four rules:

1. Name masking
2. Functions versus variables
3. A fresh start

Lexical scoping

Name masking: names defined inside a function mask names defined outside a function

```r
x <- 10
y <- 20
g02 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g02()
```

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

If a name is not defined inside a function, R looks **one level up**:

```r
x <- 2
g03 <- function() {
  y <- 1
  c(x, y)
}
```

Lexical scoping

Name masking: names defined inside a function mask names defined outside a function

The same rules apply if a **function is defined inside another function**:

```
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
```

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

Lexical scoping

Functions vs variables

Scoping rules also apply to functions:

```r
g07 <- function(x) x + 1
g08 <- function() {
  g07 <- function(x) x + 100
  g07(10)
}
g08()
```

```
## [1] 110
```

But what happens when a **function** and a **non-function** share the same name?

```r
g09 <- function(x) x + 100
g10 <- function() {
  g09 <- 10
  g09(g09)
```

Lexical scoping

A fresh start

What happens to values between invocations of a function?

```r
g11 <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

g11()
```

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

What happens if you call `g11()` again?

```r
g11()
```

Lexical scoping

Dynamic lookup

R looks for values when the function is **run**, not when the function is **created**

```
g12 <- function() x + 1
x <- 15
g12()
```

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

```
x <- 20
g12()
```

```
## [1] 21
```

Use `codetools::findGlobals()` to list all unbound symbols within a

Lazy evaluation

Arguments are **lazily evaluated**: they're only evaluated if accessed

```
h01 <- function(x) {
  10
}
h01(stop("This is an error!"))
```

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

Lazy evaluation is powered by a data structure called a **promise** (see Chapter 20)

A promise has **three components**:

1. An **expression**
2. An **environment** where the expression should be evaluated
3. A **value**, computed and cached the first time when a promise is

Lazy evaluation

Example:

```
y <- 10
h02 <- function(x) {
  y <- 100
  x + 1
}

h02(y)
```

```
## [1] 11
```

When doing **assignment inside a function call**, the variable is bound **outside** of the function, not inside of it:

```
h02(y <- 1000)
```

```
## [1] 1001
```

```
y
```

Lazy evaluation

Default arguments can be defined in terms of other arguments or variables defined later in the function:

```r
h04 <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100

  c(x, y, z)
}

h04()
```

```
## [1]   1   2 110
```

Not recommended!

Lazy evaluation

Missing values

You can use `missing()` to determine if an argument's value comes from the **user** or from a **default**:

```r
h06 <- function(x = 10) {
  list(missing(x), x)
}

str(h06())
```

```
## List of 2
##  $ : logi TRUE
##  $ : num 10
```

```r
str(h06(10))
```

```
## List of 2
##  $ : logi FALSE
##  $ : num 10
```

Lazy evaluation

Missing values

You can use `missing()` to determine if an argument's value comes from the **user** or from a **default**:

Recommendation: use `missing()` sparingly, instead use `NULL` to indicate that an argument is not required but can be supplied

```r
args(sample)
```

```
## function (x, size, replace = FALSE, prob = NULL)
## NULL
```

```r
sample <- function(x, size = NULL, replace = FALSE, prob = NULL)
  if (is.null(size)) {
    size <- length(x)
  }
```

... (dot-dot-dot)

Functions can have a special argument: `...`

You can use `...` to pass additional arguments on to another function:

```r
i01 <- function(y, z) {
  list(y = y, z = z)
}

i02 <- function(x, ...) {
  i01(...)
}

str(i02(x = 1, y = 2, z = 3))
```

```
## List of 2
##  $ y: num 2
##  $ z: num 3
```

... (dot-dot-dot)

Functions can have a special argument: `...`

You can use `...` to pass additional arguments on to another function:

```r
args(lapply)
```

```
## function (X, FUN, ...) 
## NULL
```

```r
x <- list(c(1, 3, NA), c(4, NA, 6))
str(lapply(x, mean, na.rm = TRUE))
```

```
## List of 2
##  $ : num 2
##  $ : num 5
```

... (dot-dot-dot)

Functions can have a special argument: `...`

You can use `...` to pass additional arguments on to another function

Downsides:

When you use it to pass arguments to another function, **carefully explain** where those arguments go

A misspelled argument will not raise an error

```r
sum(1, 2, NA, na_rm = TRUE)
```

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

Exiting a function

Two ways that a function can return a value:

Implicitly, the last evaluated expression is the return value:

```
j01 <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
j01(5)
```

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

Explicitly, by calling `return()`:

```
j02 <- function(x) {
  if (x < 10) {
    return(0)
  } else {
```

Exiting a function

Most functions **return visibly**: calling the function prints the result:

```r
j03 <- function() 1
j03()
```

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

Apply `invisible()` to the last value to prevent automatic printing:

```r
j04 <- function() invisible(1)
j04()
```

Exiting a function

Print or **wrap** function call in parentheses to make return value visible, or use `withVisible()`:

```
print(j04())
```

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

```
(j04())
```

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

```
str(withVisible(j04()))
```

```
## List of 2
##  $ value  : num 1
##  $ visible: logi FALSE
```

Exiting a function

Errors

If a function cannot complete its assigned task, it should **throw an error** with `stop()`, which immediately terminates the execution of the function.

```r
j05 <- function() {
  stop("I'm an error")
  return(10)
}
j05()
#> Error in j05(): I'm an error
```

Learn more about error handling in **Chapter 8**

Exiting a function

Exit handlers

Sometimes a function needs to make **temporary changes** to the **global state**.

Use `on.exit()` to set up an **exit handler** with clean-up code that restores global state, even when functions exits with an error

```r
cleanup <- function(dir, code) {
  old_dir <- setwd(dir)
  on.exit(setwd(old_dir), add = TRUE)

  old_opt <- options(stringsAsFactors = FALSE)
  on.exit(options(old_opt), add = TRUE)
}
```

Note: Always set `add = TRUE` to avoid overwriting previous exit

Function forms

Function calls come in four varieties:

- **prefix**: function name before arguments `foofy(a,b,c)`
- **infix**: function name comes in between arguments: `x + y`
- **replacement**: function replaces values by assignment: `names(df)<-c("a","b","c")`
- **special**: functions like `[[`, `if` and `for`

You can always **rewrite** a function in **prefix form**:

```
x + y
`+`(x, y)

names(df) <- c("x", "y", "z")
`names<-`(df, c("x", "y", "z"))
```

Function forms

Useful example with `lapply`:

```
add <- function(x, y) x + y
lapply(list(1:3, 4:5), add, 3)
```

```
## [[1]]
## [1] 4 5 6
##
## [[2]]
## [1] 7 8
```

You can get the same result with:

```
lapply(list(1:3, 4:5), `+`, 3)
```

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

Function forms

Infix functions

Built-in examples: `:`, `::`, `$`, `@`, `^`, `*`, `/`, `+`, etc.

You can **create your own infix function**: bind two arguments to a name that starts and ends with `%`:

```r
`%+%` <- function(a, b) paste0(a, b)
"new " %+% "string"
```

```
## [1] "new string"
```

Function forms

Replacement functions

Replacement functions act like they **modify their arguments in place**, and have the special name `xxx<-`

They must have arguments named `x` and `value`, and must return the modified object:

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
```

Replacement functions are used by placing the function call on the left side of `<-`:

```
x <- 1:10
second(x) <- 5L
x
```

Function forms

Replacement functions

Replacement functions **act like** they modify their arguments in place, and have the special name `xxx<-`

```
x <- 1:10
tracemem(x)
```

```
## [1] "<0x7fe1f1fc74e0>"
```

```
second(x) <- 6L
```

```
## tracemem[0x7fe1f1fc74e0 -> 0x7fe1f5aae7f8]: eval eval withVisible withC
## tracemem[0x7fe1f5aae7f8 -> 0x7fe1f5d0fd08]: second<- eval eval withVisi
```

Function forms

Replacement functions

If your replacement function needs **additional arguments**, place them between `x` and `value`, and call the replacement function with additional arguments on the left:

```r
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- 10
x
```

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

Thank you

Questions? Break for 10 min, and meet in your breakout rooms

Check hackMD for your breakout room assignment