

Writing Your Own Functions

DSC205 Introduction to Advanced Data Science

Marcus Birkenkrahe

March 8, 2023

README

- Creating `function` objects
- Using the `return` function
- Lazy argument evaluation
- Setting argument defaults
- Checking for missing arguments
- Dealing with ellipses (`...`)

The function command

- Template:

```
function_name <- function (arg_1, arg_2, arg_3, ...) {  
  do any code in here when function_name is called  
  return (return_object)  
}
```

- `function_name` can be any valid R object name
- You can use any number of arguments or none: `function()` like: `ls()`, `options()`, `search()`, `par()`, which can be subset and searched.

Example: hello, world!

- The function arguments are not workspace objects. Check that:

1. define a **function** named `hello_world`
2. no arguments
3. **return** the string "hello world"
4. call the function

```
hello_world <- function() {  
  return ("hello world")  
}  
hello_world()
```

```
[1] "hello world"
```

- Solution:

```
hello_world <- function() {  
  return ("hello, world!")  
}
```

- Modify `hello_world` - create a new function `hello` that takes a name as an argument and prints it to the screen:

1. define a **function** named `hello`
2. `hello` should have one argument, `name`
3. **return** the `name` together with "Hello," using **paste**
4. call the function with your name as the (string) argument
5. check if `name` is in the list of user-defined objects using **any**

```
hello <- function(name) {  
  return (paste("Hello,", name))  
}  
hello("Marcus")  
any(ls()=="name")
```

```
[1] "Hello, Marcus"  
[1] FALSE
```

- Solution:

```
hello <- function(name) {
  return (paste("Hello,", name)) # 'name' is local
}
hello("Marcus")
any(ls() == "name")

[1] "Hello, Marcus"
[1] FALSE
```

- Ask ChatGPT to create the function `hello` for you - very nice: it only returns the code not the detailed explanation.

```
any(ls()=="ask_chatgpt")
args(ask_chatgpt)

[1] TRUE
function (prompt)
NULL

any(ls() == "ask_chatgpt")
ask_chatgpt("Write a 'hello' function in R that takes
            my 'name' as input and returns
            the message 'hello [name]'".)

[1] TRUE
Error in body_config(body, match.arg(encode)) :
  object 'promptPPPPPPPPPPPP' not found

hello <- function(name) {
  message <- paste("Hello ", name, "!", sep="")
  return(message)
}
hello("Marcus")

[1] "Hello Marcus!"

# ask_chatgpt("what's the command in R to define a function?\n")
```

Example: Fibonacci sequence generator

- Remember the Fibonacci sequence generator (cut off at 150)?
- Pseudocode

```
INITIALIZE SEQUENCE/COUNTER
REPEAT
  INCREASE COUNTER
  COMPUTE NEW VALUE
  APPEND TO SEQUENCE
  CHECK IF VALUE > 150
```

- R code block (named "fibonacci")

```
fib <- rep(NA,10); fib[2] <- fib[1] <- 1; i = 2 # initialize
repeat { i <- i + 1 # counter
  append(fib,fib[i] <- fib[i-2] + fib[i-1]) # build sequence
  if (fib[i] > 150) break # break for values > 150
}
fib
```

```
[1] 1 1 2 3 5 8 13 21 34 55 89 144 233
```

- Turn the Fibonacci sequence generator into a function `myfib`:
 1. Use `function` to create the function `myfib`
 2. Use « and » to include the named code block above.
 3. Check the package environment with `search`
 4. Run the function `myfib`

```
...
```

```
Error: '...' used in an incorrect context
```

- Solution:

```

myfib <- function() {
  fib <- rep(NA,10); fib[2] <- fib[1] <- 1; i = 2 # initialize
  repeat { i <- i + 1 # counter
    append(fib,fib[i] <- fib[i-2] + fib[i-1]) # build sequence
    if (fib[i] > 150) break # break for values > 150
  }
  fib
}
search()
myfib()

[1] ".GlobalEnv"      "ESSR"              "package:stats"
[4] "package:graphics" "package:grDevices" "package:utils"
[7] "package:datasets" "package:stringr"    "package:httr"
[10] "package:methods"  "Autoloads"          "package:base"
[1] 1 1 2 3 5 8 13 21 34 55 89 144 233

```

Adding arguments

- Modify `myfib` to take a single argument, `threshold`, to break off the generator (e.g. `threshold=150`):
 1. modify the code block "fib_threshold" below accordingly.
 2. create a code block for `myfib2` that takes the `threshold` argument
 3. `return` the result `fib`
 4. search the list of user-defined objects for "myfib2"
 5. Run `myfib2` for `threshold = 150, 250, 100000, 1000000`
 6. Run the function individually first, then in a loop

```

fib <- rep(NA,10); fib[2] <- fib[1] <- 1; i = 2 # initialize
repeat { i <- i + 1 # count up
  append(fib,fib[i] <- fib[i-2] + fib[i-1])
  if (fib[i] > ...) break # break for values > threshold
}

```

```
Error in fib[i] > ... : '...' used in an incorrect context
```

```
## your solution here
```

- Solution I: initialize

```
fib <- rep(NA,10); fib[2] <- fib[1] <- 1; i = 2 # initialize
```

- Solution II: function body

```
fib <- rep(NA,10); fib[2] <- fib[1] <- 1; i = 2 # initialize
repeat { i <- i + 1 # count up
  append(fib,fib[i] <- fib[i-2] + fib[i-1])
  if (fib[i] > threshold) break # break for values > threshold
}
```

```
Error in if (fib[i] > threshold) break : the condition has length > 1
```

- Solution III: function definition

```
myfib2 <- function(threshold) {
  fib <- rep(NA,10); fib[2] <- fib[1] <- 1; i = 2 # initialize
  repeat { i <- i + 1 # count up
    append(fib,fib[i] <- fib[i-2] + fib[i-1])
    if (fib[i] > threshold) break # break for values > threshold
  }
  return (fib)
}
```

- Solution IV: execution

```
ls()[which(ls()=="myfib2")] # print function name if it's loaded
## define vector of arguments
threshold <- c(150, 250, 1e5, 1e6);
## loop over threshold
for (i in threshold) {
  print(myfib2(i))
}
```

```
[1] "myfib2"
[1] 1 1 2 3 5 8 13 21 34 55 89 144 233
[1] 1 1 2 3 5 8 13 21 34 55 89 144 233 377
[1] 1 1 2 3 5 8 13 21 34 55
[11] 89 144 233 377 610 987 1597 2584 4181 6765
```

```

[21] 10946 17711 28657 46368 75025 121393
[1]      1      1      2      3      5      8     13     21     34
[10]      55      89     144     233     377     610     987    1597    2584
[19]     4181     6765    10946    17711    28657    46368    75025    121393    196418
[28]   317811   514229   832040  1346269

```

- Print only those results of `myfib2(1e6)` that are greater than 150 and smaller than 500,000:

1. Save `myfib2(threshold=1e6)` in an object `foo`
2. Subset `foo` so that only the range (150,500000) is printed

```

foo <- myfib2(1e6)
subset(foo,foo>150 & foo <5e5)

```

```

[1]      233      377      610      987     1597     2584     4181     6765     10946     17711
[11]    28657    46368    75025   121393   196418   317811

```

Using return

- If there is no `return` statement inside a function, the function will end when the last line in the body has been run and return the most recently assigned or created object.
- If nothing is created, the function returns `NULL` (the empty object).
- Enter two dummy functions with some `dummy_code` then check `ls()`:

```

aa <- 2.5
bb <- "string me along"
cc <- "string 'em up"
dd <- 4:8

dummy1 <- function() {
  aa <- 2.5
  bb <- "string me along"
  cc <- "string 'em up"
  dd <- 4:8
}
dummy2 <- function() {

```

```

    aa <- 2.5
    bb <- "string me along"
    cc <- "string 'em up"
    dd <- 4:8
    return(dd)
}
ls()

```

```

[1] "aa"          "api_key"      "ask_chatgpt" "bar"          "bb"
[6] "cc"          "dd"           "dummy1"       "dummy2"       "dummy3"
[11] "dummy4"      "f"            "fib"          "foo"          "hello"
[16] "hello_world" "i"            "myfac"        "myfac1"       "myfib"
[21] "myfib2"      "mynum"        "mynum.fac"    "pipe"         "return"
[26] "threshold"   "v1"           "v2"

```

- `dummy1` assigns four objects in its lexical (not global) environment.
- `dummy2` returns the value of `dd` to global but not the variable.
- Assign `dummy1` and `dummy2` to `foo` and `bar`, respectively:

```

foo <- dummy1()
foo
bar <- dummy2()
bar

```

```

[1] 4 5 6 7 8
[1] 4 5 6 7 8

```

- Create a third function `dummy3` that returns `aa` and `bb` in two separate calls, then run the function:

```

dummy3 <- function() {
  aa <- 2.5
  bb <- "string me along"
  return (aa)
  cc <- "string 'em up"
  dd <- 4:8
  return (bb)
}
dummy3()

```



```
[1] 2.5
```

- Only `aa` is returned because the function exits at that point. The last three lines will never be executed.
- Which code would return all four values?

```
dummy4 <- function() {  
  aa <- 2.5  
  bb <- "string me along"  
  cc <- "string 'em up"  
  dd <- 4:8  
  ...  
}
```

- Solution:

```
dummy4 <- function() {  
  aa <- 2.5  
  bb <- "string me along"  
  cc <- "string 'em up"  
  dd <- 4:8  
  return (c(aa,bb,cc,dd))  
}  
return <- dummy4()  
return # all values are returned to global  
names(return) # the element names are lost to global  
  
[1] "2.5"          "string me along" "string 'em up"   "4"  
[5] "5"           "6"              "7"              "8"  
NULL
```

Exercise: write a factorial function

- Preparation: accept `int` as an argument to a function `myfac`, set its default value to 1, then print `int` in the body of the function:
- Solution:

```
myfac <- function(int=1) print(int)  
myfac()  
myfac(5)
```

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

- Expand `myfac` to include the computation of `int!` and test it for:

1. $1! = 1$
2. $5! = 120$
3. $12! = 479,001,600$
4. $1! = 1$

- The pseudocode for the function body is as follows:

```
INITIALIZE fac as 1
WHILE int GREATER 1
  fac * int -> fac    ## so int! = int * int-1 * int-2 * ...
  int - 1
```

- Solution:

```
myfac <- function(int=1) {
  fac <- 1
  while (int > 1) {
    fac * int -> fac
    int - 1    -> int
  } # end of while
  return (fac)
} # end of function
## test the function
myfac()
myfac(1)
myfac(5)
myfac(12)
myfac(0)
```

```
[1] 1
[1] 1
[1] 120
[1] 479001600
[1] 1
```

- What happens if you remove the default and feed the function with a negative or non-integer value?
- **Extension:** write another version of your factorial function, naming it `myfac2`. This time, assume `int` will be supplied as an integer, but not that it will be non-negative. If negative, the function should return `NaN`. Test it on the values 1, 5, 12, 0, and -6.
- R has a `base::factorial` function, defined via the Gamma function:

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt$$

```
factorial(1)
factorial(5)
factorial(12)
factorial(0)
factorial(-6)

[1] 1
[1] 120
[1] 479001600
[1] 1
[1] NaN
Warning message:
In gamma(x + 1) : NaNs produced
```