

CALLING FUNCTIONS - SCOPING

DSC 205 - Advanced introduction to data science

Marcus Birkenkrahe

January 15, 2023

README

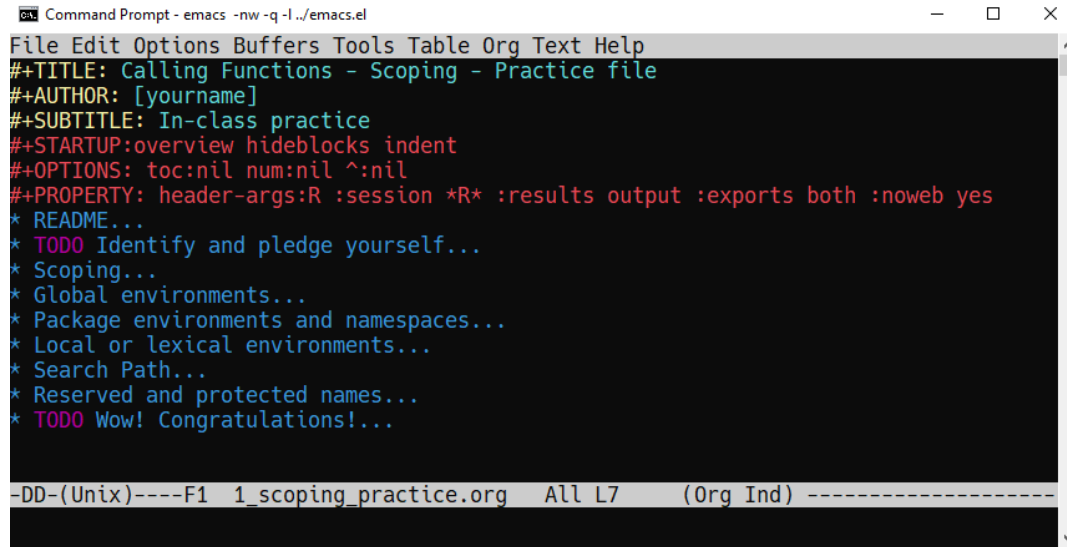


You will learn:

- ☐ How variable names are compartmentalized in R
- ☐ What the rules for naming arguments and objects are
- ☐ How R searches for arguments and variables
- ☐ How you can specify arguments when calling a function

Download the practice file from here: tinyurl.com/bp4edm7w, save it as `1_scoping_practice_1.org` and open it in Emacs to code alongside me.

To test your Emacs mettle, open it on the CMD line with the command `emacs -nw` (no graphics - not needed for this exercise):



```
Command Prompt - emacs -nw -q -l ./emacs.el
File Edit Options Buffers Tools Table Org Text Help
#+TITLE: Calling Functions - Scoping - Practice file
#+AUTHOR: [yourname]
#+SUBTITLE: In-class practice
#+STARTUP:overview hideblocks indent
#+OPTIONS: toc:nil num:nil ^:nil
#+PROPERTY: header-args:R :session *R* :results output :exports both :noweb yes
* README...
* TODO Identify and pledge yourself...
* Scoping...
* Global environments...
* Package environments and namespaces...
* Local or lexical environments...
* Search Path...
* Reserved and protected names...
* TODO Wow! Congratulations!...

--DD-(Unix)----F1 1_scoping_practice.org All L7 (Org Ind) -----
```

Scoping

- Scoping rules determine how R stores and retrieves objects
- Applied e.g. when handling duplicate object names
- Example: `data` as an argument, and as a function -

1. create a row-wise 3x3 matrix of numbers `{1..9}`
2. list all datasets in `ToothGrowth`

```
## create row-wise 2x2 matrices of 1...9
matrix(data=1:9, nrow=3, byrow=TRUE)
## list all datasets in ToothGrowth
data(ToothGrowth)
```

```
      [,1] [,2] [,3]
[1,]     1     2     3
```

```
[2,]    4    5    6
[3,]    7    8    9
```

Environments

- R enforces scoping rules with virtual *environment*
- An environment is a separate compartment for data structures (like vectors) and functions (like **data**).
- Environments are *dynamic* - they can be created, manipulated and removed.
- Technically, an environment is a pointer to the memory location where the R objects are stored.
- There are three types of environments:
 1. **Global** environments
 2. **Package** environments and namespaces
 3. **Local** or lexical environments

Global environments

- Every object you've created or overwritten resides in the global environment of your R session.
- A call to **ls()** lists all objects, variables, and user-defined functions in the global environment
- **Example:** create three new objects and confirm their existence in the global environment:
 1. a **numeric** variable **foo**
 2. a **character** variable **bar**
 3. An anonymous (non-argument) function **hello**
 4. check the contents of the global environment with **ls**
 5. run **hello**

```
foo <- 4 + 5
bar <- "stringtastic"
hello <- function() print("hello")
ls()
hello()
```

```
[1] "bar"          "foo"          "hello"        "ToothGrowth"
[1] "hello"
```

IN PROGRESS Package environments and namespaces

- Package environments are items made available by each package in R.
- You can use `ls` to list the items in a package environment: for example, to list the content of built-in `datasets` (no functions)

```
ls("package:datasets")
```

[1] "ability.cov"	"airmiles"	"AirPassengers"
[4] "airquality"	"anscombe"	"attenu"
[7] "attitude"	"austres"	"beaver1"
[10] "beaver2"	"BJsales"	"BJsales.lead"
[13] "BOD"	"cars"	"ChickWeight"
[16] "chickwts"	"co2"	"CO2"
[19] "crimtab"	"discoveries"	"DNase"
[22] "esoph"	"euro"	"euro.cross"
[25] "eurodist"	"EuStockMarkets"	"faithful"
[28] "fdeaths"	"Formaldehyde"	"freeny"
[31] "freeny.x"	"freeny.y"	"HairEyeColor"
[34] "Harman23.cor"	"Harman74.cor"	"Indometh"
[37] "infert"	"InsectSprays"	"iris"
[40] "iris3"	"islands"	"JohnsonJohnson"
[43] "LakeHuron"	"ldeaths"	"lh"
[46] "LifeCycleSavings"	"Loblolly"	"longley"
[49] "lynx"	"mdeaths"	"morley"
[52] "mtcars"	"nhtemp"	"Nile"
[55] "nottem"	"npk"	"occupationalStatus"
[58] "Orange"	"OrchardSprays"	"PlantGrowth"

[61]	"precip"	"presidents"	"pressure"
[64]	"Puromycin"	"quakes"	"randu"
[67]	"rivers"	"rock"	"Seatbelts"
[70]	"sleep"	"stack.loss"	"stack.x"
[73]	"stackloss"	"state.abb"	"state.area"
[76]	"state.center"	"state.division"	"state.name"
[79]	"state.region"	"state.x77"	"sunspot.month"
[82]	"sunspot.year"	"sunspots"	"swiss"
[85]	"Theoph"	"Titanic"	"ToothGrowth"
[88]	"treering"	"trees"	"UCBAdmissions"
[91]	"UKDriverDeaths"	"UKgas"	"USAccDeaths"
[94]	"USArrests"	"UScitiesD"	"USJudgeRatings"
[97]	"USPersonalExpenditure"	"uspop"	"VADeaths"
[100]	"volcano"	"warpbreaks"	"women"
[103]	"WorldPhones"	"WWWusage"	

Or to list the visible objects of the `graphics` package:

```
ls("package:graphics")
```

[1]	"abline"	"arrows"	"assocplot"	"axis"
[5]	"Axis"	"axis.Date"	"axis.POSIXct"	"axTicks"
[9]	"barplot"	"barplot.default"	"box"	"boxplot"
[13]	"boxplot.default"	"boxplot.matrix"	"bxp"	"cdplot"
[17]	"clip"	"close.screen"	"co.intervals"	"contour"
[21]	"contour.default"	"coplot"	"curve"	"dotchart"
[25]	"erase.screen"	"filled.contour"	"fourfoldplot"	"frame"
[29]	"grconvertX"	"grconvertY"	"grid"	"hist"
[33]	"hist.default"	"identify"	"image"	"image.default"
[37]	"layout"	"layout.show"	"lcm"	"legend"
[41]	"lines"	"lines.default"	"locator"	"matlines"
[45]	"matplot"	"matpoints"	"mosaicplot"	"mtext"
[49]	"pairs"	"pairs.default"	"panel.smooth"	"par"
[53]	"persp"	"pie"	"plot"	"plot.default"
[57]	"plot.design"	"plot.function"	"plot.new"	"plot.window"
[61]	"plot.xy"	"points"	"points.default"	"polygon"
[65]	"polypath"	"rasterImage"	"rect"	"rug"
[69]	"screen"	"segments"	"smoothScatter"	"spineplot"
[73]	"split.screen"	"stars"	"stem"	"strheight"
[77]	"stripchart"	"strwidth"	"sunflowerplot"	"symbols"

```
[81] "text"          "text.default"  "title"          "xinch"
[85] "xspline"       "xyinch"        "yinch"
```

- A package *namespace* allows the package writer to hide functions and data that are only for internal use, and stops functions from breaking when a user or another package writer uses a duplicate name.
- As an example, load (after installation) the `dplyr` package (don't print the content - it has 300 functions!) and run `dplyr::filter`.

```
library(dplyr)
dplyr::filter

function (.data, ..., .preserve = FALSE)
{
  UseMethod("filter")
}
<bytecode: 0x0000016a25a7d808>
<environment: namespace:dplyr>
```

- If you look at the output (the definition of `filter` in this package, you notice an internal (**base**) function, `UseMethod`, which is not listed in the visible content of `dplyr`, and the name of the `namespace` environment.
- When loading `dplyr`, you were informed that `dplyr::filter` masks another function, `stats::filter`. This means that using `filter` without the namespace reverts to `dplyr::filter`. If you want to use the function of the same name in `stats`, you need to call `stats::filter`.

Local or lexical environments

- Each time a function is called, a new environment called *local* or *lexical* is created.
- It contains all objects and variables created in and visible to the function, including any arguments you've supplied during execution.
- Example: create a 2x2 `matrix` and pass in the argument `data`: "OMG", "LOL", "WTF", "YOLO":

```

youthspeak <- matrix(data = c("OMG", "LOL", "WTF", "YOLO"),
  nrow=2, ncol=2)
youthspeak

      [,1] [,2]
[1,] "OMG" "WTF"
[2,] "LOL" "YOLO"

```

- Calling `matrix` like this creates a local environment containing the `data` vector
- When you execute the function, it begins by looking for `data` in this local environment. It is not confused by other objects named `data`, such as `utils::data`.
- If a required item is not found in the local environment, R does begin to widen its search.
- Once the function has completed, the local environment is automatically removed. The same goes for `nrow` and `ncol`.

Search Path

- To access data structures and functions other than the immediate global environment (of user-created objects), R follows a *search path*.
- You can view the search path with `search()`:

```

search()

[1] ".GlobalEnv"      "package:car"      "package:carData"
[4] "package:dplyr"    "ESSR"             "package:stats"
[7] "package:graphics" "package:grDevices" "package:utils"
[10] "package:datasets" "package:methods"   "Autoloads"
[13] "package:base"

```

- The path always begins at `.GlobalEnv` and ends after `base`. It stops if an object is found in any environment along the path.
- If it does not find what it wanted, the *empty environment* is reached.

- Example: let's see what happens when we create a vector with `seq`:

1. create a vector of 5 elements with `seq`
2. the values should lay between the (included) values 0 and 3

```
baz <- seq(from=0, to=3, length.out=5)
baz
```

```
[1] 0.00 0.75 1.50 2.25 3.00
```

- R searches `.GlobalEnv` for `seq`, goes through the list and finds it in `base`. `seq` is executed and `baz` is created in the global environment.
- In the subsequent call to `baz`, R finds it immediately in `.GlobalEnv`.
- You can look up the environment of any function using `environment`:

```
environment(seq)
environment(abline)
environment(filter)
```

```
<environment: namespace:base>
<environment: namespace:graphics>
<environment: namespace:dplyr>
```

- When a package is loaded with `library`, it is inserted in the search path right after the global environment, along with all its dependencies:

```
library('car')
search()
```

```
[1] ".GlobalEnv"      "package:car"      "package:carData"
[4] "package:dplyr"    "ESSR"             "package:stats"
[7] "package:graphics" "package:grDevices" "package:utils"
[10] "package:datasets" "package:methods"  "Autoloads"
[13] "package:base"
```

- In the example, loading `car` lead to the inclusion of the function package and its accompanying dataset package: do you remember how to list the contents of `carData`?


```
ls('package:carData')
```

```

[1] "Adler"           "AMSSurvey"      "Angell"         "Anscombe"
[5] "Arrests"        "Baumann"        "BEPS"           "Bfox"
[9] "Blackmore"      "Burt"           "CanPop"         "CES11"
[13] "Chile"          "Chirot"         "Cowles"         "Davis"
[17] "DavisThin"      "Depredations"   "Duncan"         "Ericksen"
[21] "Florida"        "Freedman"       "Friendly"       "Ginzberg"
[25] "Greene"         "GSSvocab"       "Guyer"          "Hartnagel"
[29] "Highway1"       "KosteckiDillon" "Leinhardt"      "LoBD"
[33] "Mandel"         "Migration"      "Moore"          "MplsDemo"
[37] "MplsStops"      "Mroz"           "OBrienKaiser"   "OBrienKaiserLong"
[41] "Ornstein"       "Pottery"        "Prestige"       "Quartet"
[45] "Robey"          "Rossi"          "Sahlins"        "Salaries"
[49] "SLID"           "Soils"          "States"         "TitanicSurvival"
[53] "Transact"       "UN"             "UN98"           "USPop"
[57] "Vocab"          "WeightLoss"     "Wells"          "Womenlf"
[61] "Wong"           "Wool"           "WVS"

```

- An error is thrown if you request a function or object
 - that you haven't **defined**,
 - that doesn't **exist**,
 - that is in a contributed package that you've forgotten to **load**

```

neither.here() # undefined function
nor.there      # undefined object

```

```

Error in neither.here() : could not find function "neither.here"
Error: object 'nor.there' not found

```

- Read Gupta (2012) for more details on R environments. (This would also make an excellent term project topic.)

Reserved and protected names

- Key terms that are forbidden from being used as R object names:
 - `if` and `else`

- `for`, `while`, and `in`
 - `repeat`, `break`, and `next`
 - `TRUE`, and `FALSE`
 - `Inf` and `-Inf`
 - `NA`, `NaN`, and `NULL`
- The first four line items are the core tools for programming in R, followed by Boolean values and special values.
 - What happens when you assign a value to an `NaN`?

```
NaN <- 5
```

```
Error in NaN <- 5 : invalid (do_set) left-hand side to assignment
```

- Since R is case-sensitive, you can assign values to case variants of these keywords, causing much confusion:

```
False <- "confusing"
nan <- "this"
inf <- "is"
Null <- "very"
paste(nan,inf,Null,False)

[1] "this is very confusing"
```

- `T` and `F` can also be overwritten - don't do it since they are the abbreviations for `TRUE` and `FALSE`:

```
T <- FALSE
F <- TRUE
paste(T,"is",F)
paste("2+2=5 is", (2+2==5) == T)
(2+2==5) == TRUE

[1] "FALSE is TRUE"
[1] "2+2=5 is TRUE"
[1] FALSE
```

- With all these confusing changes, clear the global environment now!

```
ls()
rm(list=ls()) ## remove the list of user-defined R objects
ls()

[1] "bar"          "baz"          "F"            "False"        "foo"
[6] "hello"        "inf"          "nan"          "Null"         "T"
[11] "ToothGrowth" "youthspeak"
character(0)
```

Glossary

TERM	MEANING
Scoping	Rules of storing/retrieving objects
Environment	Virtual compartment for data and functions
Global environment	All user-created objects
Package environments	Objects contained in packages
Namespace	Defines visibility of package functions E.g. in <code>base::</code> for the <code>base</code> package
<code>ls()</code>	List global environment
<code>ls(package=base)</code>	List functions in the <code>base</code> package
Local environment	Objects created when function is called
Search path	List of environments searched, <code>search()</code>
<code>matrix</code>	Create matrix
<code>seq</code>	Create numerical sequence vector
<code>base::data</code>	List or load dataset
<code>NaN</code>	Not a number
<code>Inf</code>	Infinite numerical value
<code>NA</code>	Missing value
<code>NULL</code>	Null object - returned when value undefined
<code>paste</code>	Paste arguments together as string
<code>rm</code>	Remove R objects, e.g. <code>rm(list=ls())</code>

References

- Gupta, S. (Mar 29, 2012). How R Searches and Finds Stuff. URL: blog.thatbuthow.com.