CALLING FUNCTIONS - SCOPING

DSC 205 - Advanced introduction to data science

Marcus Birkenkrahe

January 12, 2025

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 files from GitHub, save them as `1_scoping_practice_1.org` and `1_scoping_practice_2.org`.

The first practice file is code along while I lecture, the second practice file is an independent exercise. Solutions in the pdf repo.

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 built-in datasets

```
## create row-wise 2x2 matrics of 1...9
matrix(data=1:9, nrow=3, byrow=TRUE)

## list all datasets in the MASS package (if installed)
data() -> datasets
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

- The R object `datasets` hold the `data()` information now. You can save this object externally in `.RData` format:

```
save(datasets, file="datasets.RData")
system("file -b datasets.RData")
```

```
gzip compressed data, from Unix, original size modulo 2^32 11881
```

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"      "datasets" "foo"      "hello"
[1] "hello"
```

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 (built-in) `graphics` package:

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

```
 [1] "abline"          "arrows"          "assocplot"       "axis"            "Axis
 [6] "axis.Date"       "axis.POSIXct"    "axTicks"         "barplot"         "barp
[11] "box"             "boxplot"         "boxplot.default" "boxplot.matrix"  "bxp"
[16] "cdplot"          "clip"            "close.screen"    "co.intervals"    "cont
[21] "contour.default" "coplot"          "curve"           "dotchart"        "eras
[26] "filled.contour"  "fourfoldplot"    "frame"           "grconvertX"      "grco
[31] "grid"            "hist"            "hist.default"    "identify"        "imag
[36] "image.default"   "layout"          "layout.show"     "lcm"             "lege
[41] "lines"           "lines.default"   "locator"         "matlines"        "matp
[46] "matpoints"       "mosaicplot"      "mtext"           "pairs"           "pair
[51] "panel.smooth"    "par"             "persp"           "pie"             "plot
[56] "plot.default"    "plot.design"     "plot.function"   "plot.new"        "plot
[61] "plot.xy"         "points"          "points.default"  "polygon"         "poly
[66] "rasterImage"     "rect"            "rug"             "screen"          "segm
[71] "smoothScatter"   "spineplot"       "split.screen"    "stars"           "sten
[76] "strheight"       "stripchart"      "strwidth"        "sunflowerplot"   "symb
[81] "text"            "text.default"    "title"           "xinch"           "xspl
[86] "xyinch"          "yinch"
```

- Though you may have used it often already, check out the `help` page for `ls`. If you're on Linux, you'll get a proper man page.
- 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, ..., .by = NULL, .preserve = FALSE)
{
    check_by_typo(...)
    by <- enquo(.by)
    if (!quo_is_null(by) && !is_false(.preserve)) {
abort("Can't supply both `.by` and `.preserve`.")
    }
    UseMethod("filter")
}
<bytecode: 0x5c0ee10219b0>
<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` named `nerdspeak`, and pass in the argument `data`: "IDK", "LOL", "BRB", "AFK":

```
nerdspeak <- matrix(data = c("IDK", "LOL", "BRB", "AFK"),
    nrow=2, ncol=2)
nerdspeak
```

```
     [,1]  [,2]
[1,] "IDK" "BRB"
[2,] "LOL" "AFK"
```

- 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`.

Namespaces and Environments in R and Python

Both R and Python manage **scoping** and **namespaces** through **environments**, but they do so differently.

In Python

- Functions also use **lexical (local) scoping**.
- **Namespaces** exist at the module, class, and function levels.
- Python uses **dictionaries** internally to manage namespaces.
- Python has built-in functions to explore namespaces:
  - `globals()` returns the global namespace as a dictionary.

    ```
    ## define global objects
    I_AM_A_VARIABLE = 100
    import pandas as pd
    ## print global objects
    print("Global objects:")
    [print(_) for _ in globals()]
    ```

  - `locals()` returns the local namespace as a dictionary.
  - `dir()` lists the names defined in a namespace.

    ```
    ## print names in pandas
    print("\nNames in ‘pandas‘ package:")
    [print(_) for _ in dir(pd)]
    ```

Comparison of Namespace and Environment Handling

Feature	R	Python
Scoping	Lexical (local)	Lexical (local)
Environment	Explicit environment object	Implicit via dictionaries
Namespace Levels	Package, function	Module, class, function
Built-in Functions	`new.env()`, `parent.env()`	`globals()`, `locals()`, `dir()`

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:Rcpp"      "package:dplyr"     "ESSR"
[5] "package:stats"     "package:graphics"  "package:grDevices" "package:utils"
[9] "package:datasets"  "package:methods"   "Autoloads"         "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. Let's load `Rcpp`.

```
library('Rcpp')
search()
```

```
[1] ".GlobalEnv"        "package:Rcpp"      "package:dplyr"     "ESSR"
[5] "package:stats"     "package:graphics"  "package:grDevices" "package:utils"
[9] "package:datasets"  "package:methods"   "Autoloads"         "package:base"
```

- Do you remember how to list the contents of `Rcpp`?

```
ls('package:Rcpp')
```

```
 [1] "compileAttributes"      "cpp_object_dummy"       "cpp_object_initializer"
 [4] "cppFunction"            "demangle"               "evalCpp"
 [7] "exposeClass"            "formals<-"              "getRcppVersion"
[10] "initialize"             "LdFlags"                "loadModule"
[13] "loadRcppClass"          "loadRcppModules"        "Module"
[16] "populate"               "prompt"                 "Rcpp.package.skeleton"
[19] "Rcpp.plugin.maker"      "RcppLdFlags"            "registerPlugin"
[22] "setRcppClass"           "show"                   "sizeof"
[25] "sourceCpp"
```

- 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"       "datasets"  "F"         "False"     "foo"       "hel
 [8] "inf"       "nan"       "nerdspeak" "Null"      "T"
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 `base::` for the `base` package
`ls()`	List global environment
`ls(package:base)`	List functions in the `base` package
Local environment	Objects created when function is called
Search path	List of environments searched, `search()`
`matrix`	Create matrix
`seq`	Create numerical sequence vector
`base::data`	List or load dataset
`NaN`	Not a number
`Inf`	Infinite numerical value
`NA`	Missing value
`NULL`	Null object - returned when value undefined
`paste`	Paste arguments together as string
`rm`	Remove R objectts, e.g. `rm(list=ls())`

References

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