CALLING FUNCTIONS - SCOPING

DSC 205 - Advanced introduction to data science

Marcus Birkenkrahe

January 30, 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

There is a code along file and a practice file:

1. Code along: tinyurl.com/scoping-codealong-org
2. Practice: tinyurl.com/scoping-practice-org

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 11316
```

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"        "df"              "foo"
[5] "hello"           "i"               "j"               "op"
[9] "package_methods"
[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"
 [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"
```

- 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


Attaching package: ‘dplyr’

The following objects are masked from ‘package:stats’:

    filter, lag

The following objects are masked from ‘package:base’:

    intersect, setdiff, setequal, union
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: 0x5bcee4f65a60>
<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.
  - `locals()` returns the local namespace as a dictionary.

– `dir()` lists the names defined in a namespace.

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

```
Python 3.10.12 (main, Jan 17 2025, 14:35:34) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Global objects:
__name__
__doc__
__package__
__loader__
__spec__
__annotations__
__builtins__
__org_babel_python_fname
__org_babel_python_fh
I_AM_A_VARIABLE
pd
python.el: native completion setup loaded
```

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

```
Names in ‘pandas‘ package:
ArrowDtype
BooleanDtype
Categorical
CategoricalDtype
CategoricalIndex
DataFrame
DateOffset
DatetimeIndex
DatetimeTZDtype
ExcelFile
ExcelWriter
```

```
Flags
Float32Dtype
Float64Dtype
Grouper
HDFStore
Index
IndexSlice
Int16Dtype
Int32Dtype
Int64Dtype
Int8Dtype
Interval
IntervalDtype
IntervalIndex
MultiIndex
NA
NaT
NamedAgg
Period
PeriodDtype
PeriodIndex
RangeIndex
Series
SparseDtype
StringDtype
Timedelta
TimedeltaIndex
Timestamp
UInt16Dtype
UInt32Dtype
UInt64Dtype
UInt8Dtype
__all__
__builtins__
__cached__
__doc__
__docformat__
__file__
__git_version__
__loader__
```

```
__name__
__package__
__path__
__spec__
__version__
_built_with_meson
_config
_is_numpy_dev
_libs
_pandas_datetime_CAPI
_pandas_parser_CAPI
_testing
_typing
_version_meson
annotations
api
array
arrays
bdate_range
compat
concat
core
crosstab
cut
date_range
describe_option
errors
eval
factorize
from_dummies
get_dummies
get_option
infer_freq
interval_range
io
isna
isnull
json_normalize
lreshape
melt
```

```
merge
merge_asof
merge_ordered
notna
notnull
offsets
option_context
options
pandas
period_range
pivot
pivot_table
plotting
qcut
read_clipboard
read_csv
read_excel
read_feather
read_fwf
read_gbq
read_hdf
read_html
read_json
read_orc
read_parquet
read_pickle
read_sas
read_spss
read_sql
read_sql_query
read_sql_table
read_stata
read_table
read_xml
reset_option
set_eng_float_format
set_option
show_versions
test
testing
```

```
timedelta_range
to_datetime
to_numeric
to_pickle
to_timedelta
tseries
unique
util
value_counts
wide_to_long
```

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()`

Demo for R's environment creation functions:

```
my_env <- new.env() # create a new environment
my_env$a <- 1:10 # define an R object (numeric vector) there
ls(my_env) # list the environment members
class(my_env)  # what object class is the environment
parent.env(my_env) # what's the parent of this environment
```

```
[1] "a"
[1] "environment"
<environment: R_GlobalEnv>
```

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:dplyr"     "ESSR"              "package:stats"
[5] "package:graphics"  "package:grDevices" "package:utils"     "package:datasets"
[9] "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)
library(dplyr)      # once I load this, ‘filter‘ will be masked
environment(filter)
```

```
<environment: namespace:base>
<environment: namespace:graphics>
<environment: namespace:dplyr>
<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')  # must install.packages("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` or `MASS`?

```
ls('package:base') |> length()
```

```
[1] 1252
```

- 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 three 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"  # this is not FALSE
nan <- "this" # this is not NaN
inf <- "is" # this is not Inf
Null <- "very"  # this is not NULL
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"        "df"
 [5] "F"               "False"           "foo"             "hello"
 [9] "i"               "inf"             "j"               "my_env"
[13] "nan"             "nerdspeak"       "Null"            "op"
[17] "package_methods" "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.