An R package can be viewed as a set of functions, of which only a part are exposed to the user. In this blog post we shall concentrate of the functions that are not exposed to the user, so called internal functions: what are they, how does one handle them in one’s own package, and how can one explore them?

**Internal functions 101**

**What is an internal function?**

It’s a function that lives in your package, but that isn’t surfaced to the user. You could also call it unexported function or helper function; as opposed to exported functions and user-facing functions.

For instance, in the usethis package there’s a base\_and\_recommended() function that is not exported.

# doesn't work

library("usethis")

base\_and\_recommended()

## Error in base\_and\_recommended(): could not find function "base\_and\_recommended"

usethis::base\_and\_recommended()

## Error: 'base\_and\_recommended' is not an exported object from 'namespace:usethis'

# works

usethis:::base\_and\_recommended()

## [1] "base" "boot" "class" "cluster" "codetools"

## [6] "compiler" "datasets" "foreign" "graphics" "grDevices"

## [11] "grid" "KernSmooth" "lattice" "MASS" "Matrix"

## [16] "methods" "mgcv" "nlme" "nnet" "parallel"

## [21] "rpart" "spatial" "splines" "stats" "stats4"

## [26] "survival" "tcltk" "tools" "utils"

As an user, you shouldn’t use unexported functions of another package in your own code.

**Why not export all functions?**

There are at least these two reasons:

* In a package you want to provide your user an API that is useful and stable. You can vouch for a few functions, that serve the package main goals, are documented enough, and that you’d only change [with great care](https://devguide.ropensci.org/evolution.html) if need be. If your package users rely on an internal function that you decide to ditch when re-factoring code, they won’t be happy, so only export what you want to maintain.
* If all packages exposed all their internal functions, the user environment would be flooded and the namespace conflicts would be out of control.

**Why write internal functions?**

Why write internal functions instead of having everything in one block of code inside each exported functions?

When writing R code in general there are several reasons to write functions and it is the same within R packages: you can re-use a bit of code in several places (e.g. an epoch converter used for the output of several endpoints from a web API), and you can give it a self-explaining name (e.g. convert\_epoch()). Any function defined in your package is usable by other functions of your package (unless it is defined *inside* a function of your package, in which case only that parent function can use it).

Having internal functions also means you can test these bits of code on their own. That said if you test internals too much re-factoring your code will mean breaking tests.

To find blocks of code that could be replaced with a function used several times, you could use the dupree package whose planned enhancements include highlighting or printing the similar blocks.

**Where to put internal functions?**

You could save internal functions used in one function only in the R file defining that function, and internal functions used in several other functions in a single utils.R file or specialized utils-dates.R, utils-encoding.R files. Choose a system that helps you and your collaborators find the internal functions easily, R will never have trouble finding them as long they’re somewhere in the R/ directory. 

Another possible approach to helper functions when used in several packages is to pack them up in a package

**How to document internal functions?**

You should at least add a few comments in their code as usual

#' Compare x to 1

#' @param x an integer

#' @NoRd

is\_one <- function(x) {

x == 1

}

The keyword @keywords internal would mean a manual page is created but not present in the function index. A confusing aspect is that you could use it for an *exported, not internal* function you don’t want to be too visible, e.g. a function returning the default app for OAuth in a package wrapping a web API.

#' A function rather aimed at developers

#' @description A function that does blabla, blabla.

#' @keywords internal

#' @export

does\_thing <- function(){

message("I am an exported function")

}

**Explore internal functions**

You might need to have a look at the guts of a package when wanting to contribute to it, or at the guts of several packages to get some inspiration for your code.

**Explore internal functions within a package**

Say you’ve started working on a new-to-you package (or resumed work on a long forgotten package of yours ). How to know how it all hangs together? **You can use the same methods as for debugging code, exploring code is like debugging it and vice versa!**

Another useful tool is the in development pkgapi package. Let’s look at the cranlogs source code.

map <- pkgapi::map\_package("/home/maelle/Documents/R-hub/cranlogs")

We can see all defined functions, exported or not.

str(map$defs)

## 'data.frame': 8 obs. of 7 variables:

## $ name : chr "check\_date" "cran\_downloads" "cran\_top\_downloads" "cranlogs\_badge" ...

## $ file : chr "R/utils.R" "R/cranlogs.R" "R/cranlogs.R" "R/badge.R" ...

## $ line1 : int 1 61 184 16 137 105 117 126

## $ col1 : int 1 1 1 1 1 1 1 1

## $ line2 : int 6 103 208 33 153 115 124 135

## $ col2 : int 1 1 1 1 1 1 1 1

## $ exported: logi FALSE TRUE TRUE TRUE FALSE FALSE ...

We can see all calls inside the package code, to functions from the package and other packages.

str(map$calls)

## 'data.frame': 84 obs. of 9 variables:

## $ file : chr "R/badge.R" "R/badge.R" "R/badge.R" "R/badge.R" ...

## $ from : chr "cranlogs\_badge" "cranlogs\_badge" "cranlogs\_badge" "cranlogs\_badge" ...

## $ to : chr "base::c" "base::match.arg" "base::paste0" "base::paste0" ...

## $ type : chr "call" "call" "call" "call" ...

## $ line1: int 17 21 23 25 30 7 8 62 65 66 ...

## $ line2: int 17 21 23 25 30 7 8 62 65 66 ...

## $ col1 : int 38 14 14 16 3 14 14 35 8 17 ...

## $ col2 : int 38 22 19 21 8 19 19 35 14 25 ...

## $ str : chr "c" "match.arg" "paste0" "paste0" ...

We can filter that data.frame to only keep calls between functions defined in the package.

library("magrittr")

internal\_calls <- map$calls[map$calls$to %in% glue::glue("{map$name}::{map$defs$name}"),]

internal\_calls %>%

dplyr::arrange(to)

## file from to type line1 line2 col1

## 1 R/cranlogs.R cran\_downloads cranlogs::check\_date call 69 69 7

## 2 R/cranlogs.R cran\_downloads cranlogs::check\_date call 73 73 7

## 3 R/cranlogs.R to\_df\_1 cranlogs::fill\_in\_dates call 123 123 3

## 4 R/cranlogs.R cran\_downloads cranlogs::to\_df call 101 101 3

## 5 R/cranlogs.R to\_df cranlogs::to\_df\_1 call 109 109 5

## 6 R/cranlogs.R to\_df cranlogs::to\_df\_r call 107 107 5

## col2 str

## 1 16 check\_date

## 2 16 check\_date

## 3 15 fill\_in\_dates

## 4 7 to\_df

## 5 11 to\_df\_1

## 6 11 to\_df\_r

That table can help understand how a package works. One could combine that with a network visualization.

library("visNetwork")

internal\_calls <- internal\_calls %>%

dplyr::mutate(to = gsub("cranlogs\\:\\:", "", to))

nodes <- tibble::tibble(id = map$defs$name,

title = map$defs$file,

label = map$defs$name,

shape = dplyr::if\_else(map$defs$exported,

"triangle",

"square"))

edges <- internal\_calls[, c("from", "to")]

visNetwork(nodes, edges, height = "500px") %>%

visLayout(randomSeed = 42) %>%

visNodes(size = 10)

In this interactive visualization one sees three exported functions (triangles), with only one that calls internal functions. Such a network visualization might not be that useful for bigger packages, and in our workflow is limited to pkgapi’s capabilities (e.g. not memoised functions)… but it’s at least quite pretty.

**Explore internal functions across packages**

Looking at helpers in other packages can help you write your own, e.g. looking at a package elegantly wrapping a web API could help you wrap another one elegantly too.

**Conclusion**

In this post we explained what internal functions are, and gave a few tips as to how to explore them within a package and across packages. We hope the post can help clear up a few doubts. Feel free to comment about further ideas or questions you may have.