The envnames package helps navigate user-defined and function execution environments, and find objects in nested environments

Daniel Mastropietro (mastropi@uwalumni.com)

2020-12-07

Contents

1 Introduction

The main goal of this package is to overcome the limitation of the `environmentName()` function in the base package which does not return the name of an environment unless it is a package, a namespace, or a system environment (e.g. the global environment, the base environment). In fact, the `environmentName()` function returns an *empty string* when the argument is a user-defined environment.

On the other hand, the environment itself is identified solely by its memory address, which makes it difficult to track an environment once we have defined a number of them. These limitations –and the workaround provided by this package– can be seen by running the following code snippet:

```
myenv <- new.env()
cat("The name of the environment just defined is: ", environmentName(myenv), "(empty)\n")
```

```
## The name of the environment just defined is:   (empty)
```

```
cat("Simply referencing the environment just defined yields its memory address,
    which is not so helpful: "); print(myenv)
```

```
## Simply referencing the environment just defined yields its memory address,
##     which is not so helpful:
```

```
## <environment: 0x00000000090a6778>
```

```
cat("Using the environment_name() function of the envnames package gives
    the environment name:", environment_name(myenv))
```

```
## Using the environment_name() function of the envnames package gives
##     the environment name: myenv
```

Clearly the last one is the result we most likely want, and the envnames package makes this possible by creating a lookup table that maps environment names to their memory addresses. The different functions of this package use this lookup table to provide the user with valuable information, such as the name of the environment where an object resides, be it a package environment, a user-defined environment, or *even* a function execution environment.

Why do we care about knowing the name of user-defined environments and function execution environments? That piece of information may be handy for example under the following scenarios:

- working in a package where user-defined environments have been defined in a nested structure:
  this package facilitates the navigation through those environments and their connection between them, eliminating e.g. the use of `ls()` as a rudimentary tool to identify the human-understandable environment (e.g. `myenv`) referred by an environment given by its memory address (e.g. `<environment: 0x063dbc90>` in 32-bit systems or `<environment: 0x00000000063dbc90>` in 64-bit systems), as already seen above.
  For more information and examples, see below the section on function `get_env_names()`, that returns a map of currently defined environments and the way they are connected or nested.

- debugging an application:
  this package makes it easier to retrieve variables in different environments; for instance, retrieve the value of a variable in the parent environment to the environment where the debugger is currently positioned, which could well be a function execution environment.
  For more information and examples, see below the sections on functions `get_obj_name()` and `get_obj_value()`, which can be used to retrieve the name and value of the variable leading to a particular variable in connected environments, and the section on `environment_name()`, which can be used to retrieve the name of an execution environment.

Apart from this core functionality, additional tools were added during the package development process which include:

- an enhancement of the built-in `exists()` function with the capability of searching objects inside *user-defined environments* and *recursively* –i.e. in *nested* environments, defined inside other environments–, as well as searching objects that are the *result of expressions*. This functionality is provided by the `obj_find()` function.

- a simplification of the output obtained when retrieving the calling function name and the stack of calling functions, currently provided by the built-in function `sys.calls()`. This functionality is provided by functions `get_fun_name()`, `get_fun_calling()`, and `get_fun_calling_chain()`, which return simple strings or array of strings with the function names of interest.

- the retrieval of the memory address of an object. This functionality is provided by the `get_obj_address()` function.

Currently the package has 11 functions directly accessible to the user (plus one function that is an alias).

Definition of workspace: despite being a widely used concept, we want to emphasize here that in this document we use the word "workspace" to refer to the memory space where all *visible* objects exist. In practice, this includes all the environments that are reachable via the `search()` path, namely the system environments (global environment, base environment), all loaded packages, and all user environments defined within and, if inside packages, *exported. Note that package namespaces are not part of the workspace.*

Naming convention: Function names are all small caps and the underscore is used to separate keywords (e.g. `environment_name()`, `get_obj_address()`, etc.)

2 Description of the 11 functions in the package

This section describes the functionality of the 11 available functions, which are now briefly described as 7 groups made up of functions with similar functionality and sorted by relevance in terms of historical and practical use:

1) `get_env_names()`, used to retrieve the name of all the environments defined in the workspace together with their memory address. This is an address-name lookup table, the core element of the package that allows the "magic" to happen.

2) `environment_name()` / `get_env_name()` (its alias), used to get the name of user-defined and execution environments (as well as all other named environments).

3) `obj_find()`, used to find an object in the workspace and recursively within environments.

4) `get_fun_name()`, `get_fun_calling()`, `get_fun_calling_chain()`, used to get the function calling name and stack displayed in a format that is easier to manipulate than the one provided by `sys.calls()`.

5) `get_fun_env()`, used to retrieve a function's execution environment.

6) `get_obj_name()`, `get_obj_value()`, used to retrieve the *name* and *value* of the object leading to a given function's parameter.

7) `get_obj_address()`, `address()`, used to get the memory address of an object; `get_obj_address()` first looks for the object (using `obj_find()`), while `address()` assumes it exists in the environment where the function is run.

Each of the above set of functions will be described in the following sub-sections, where each title is a sentence stating the main purpose of the presented function(s).

2.1 get_env_names(): retrieve the address-name lookup table of defined environments

The `get_env_names()` function returns a map of all the environments defined in a given environment. If no environment location is given, the map includes all the environments existing in the whole workspace.

In practice, the map is an address-name lookup table that relates the memory address of each environment (be it a system environmet, a package environment, a user-defined environment, or optionally a function execution environment) to its name.

This address-name lookup table is the basis for the operation of most of the other functions in the package, which rely on it to retrieve the names of environments based on their memory addresses.

The signature of the function is the following:
`get_env_names(envir = NULL, include_functions = FALSE)`.

2.1.1 Examples

2.1.1.1 Let's start with the definition of a few environments We define a couple of environments and nested environments.

```
env1 <- new.env()
env_of_envs <- new.env()
with(env_of_envs, env21 <- new.env())
```

Note that environment `env21` is *nested* in environment `env_of_envs`.

2.1.1.2 Basic operation The following call returns a data frame containing the address-name lookup table, where the two main columns are:
- `address` that contains the memory address of the environment, and
- `name` that contains the name of the environment.

The other columns are used to give context to the environments, such as:
- `path` which tells us how to reach user-defined environments that are nested within other user-defined environments. The path is relative to the `envir` environment given as parameter, or from the global environment if no `envir` is given. As an example, see the case for environment `env21` nested within `env_of_envs`.
- `location` which indicates, for instance, the package where an environment is defined, or the name of the enclosing environment of a function –i.e. where the function is defined–, if the concerned environment is a function's execution environment.

```
get_env_names()
```

```
##              type         location    locationaddress            address
## 1            user      R_GlobalEnv <000000000447C3F8> <00000000094AF8D0>
## 2            user      R_GlobalEnv <000000000447C3F8> <00000000093B2F28>
## 3            user      R_GlobalEnv <000000000447C3F8> <0000000009227280>
## 4            user      R_GlobalEnv <000000000447C3F8> <00000000090A6778>
## 5            user package:envnames <00000000094D39C0> <0000000006A925E0>
## 6            user package:envnames <00000000094D39C0> <0000000006A8F038>
## 7            user package:envnames <00000000094D39C0> <0000000006A8FA48>
## 8        function            tools <00000000089894F8> <0000000008DD6A68>
## 9        function             base <000000000447C350> <0000000008C73A08>
## 10       function         tryCatch <0000000008C73A08> <0000000008C73340>
## 11       function         tryCatch <0000000008C73A08> <0000000008C72FF8>
## 12       function      tryCatchOne <0000000008C72FF8> <0000000008C72C78>
## 13       function            knitr <0000000008EBD870> <0000000008C725E8>
## 14       function            knitr <0000000008EBD870> <0000000008C81D50>
```

```
## 15       function        rmarkdown <0000000008CD2080> <00000000068C02D0>
## 16       function            knitr <0000000008EBD870> <0000000008EDD810>
## 17       function            knitr <0000000008EBD870> <0000000008F9A2C8>
## 18       function             base <000000000447C350> <00000000090DFF60>
## 19       function            knitr <0000000008EBD870> <00000000090E0468>
## 20       function            knitr <0000000008EBD870> <00000000090E0708>
## 21       function            knitr <0000000008EBD870> <00000000090E07E8>
## 22       function            knitr <0000000008EBD870> <00000000090B4200>
## 23       function            knitr <0000000008EBD870> <0000000009081968>
## 24       function            knitr <0000000008EBD870> <000000000907CF58>
## 25       function         evaluate <0000000008EB5390> <000000000907A938>
## 26       function         evaluate <0000000008EB5390> <0000000008FEBFC0>
## 27       function    evaluate_call <0000000008FEBFC0> <0000000008FCD600>
## 28       function    evaluate_call <0000000008FEBFC0> <0000000008FCD6E0>
## 29       function             base <000000000447C350> <0000000008FCD9F0>
## 30       function             base <000000000447C350> <0000000008FCDFA0>
## 31       function             base <000000000447C350> <0000000008FCE438>
## 32       function             base <000000000447C350> <000000000447C3F8>
## 33 system/package             <NA>               <NA> <000000000447C3F8>
## 34 system/package             <NA>               <NA> <00000000094D39C0>
## 35 system/package             <NA>               <NA> <000000000805EAD0>
## 36 system/package             <NA>               <NA> <0000000007042008>
## 37 system/package             <NA>               <NA> <00000000077B2FD8>
## 38 system/package             <NA>               <NA> <000000000800F158>
## 39 system/package             <NA>               <NA> <000000000764D7D8>
## 40 system/package             <NA>               <NA> <0000000007588170>
## 41 system/package             <NA>               <NA> <00000000067A9618>
## 42 system/package             <NA>               <NA> <0000000004441E80>
## 43      namespace             <NA>               <NA> <000000000953A158>
## 44      namespace             <NA>               <NA> <0000000006D3E760>
## 45      namespace             <NA>               <NA> <00000000074FC128>
## 46      namespace             <NA>               <NA> <0000000007FB3EF0>
## 47      namespace             <NA>               <NA> <00000000076C9150>
## 48      namespace             <NA>               <NA> <0000000007629E38>
## 49      namespace             <NA>               <NA> <0000000006B6F6E8>
## 50      namespace             <NA>               <NA> <000000000447C350>
## 51          empty             <NA>               <NA> <0000000004441EB8>
##                 pathname         path                   name
## 1                   env1                                env1
## 2            env_of_envs                        env_of_envs
## 3      env_of_envs$env21  env_of_envs                 env21
## 4                  myenv                               myenv
## 5                testenv                             testenv
## 6           testenv$env1      testenv                   env1
## 7     testenv$env1$env22 testenv$env1                  env22
## 8  tools::buildVignettes              tools::buildVignettes
## 9               tryCatch                            tryCatch
## 10          tryCatchList                        tryCatchList
## 11           tryCatchOne                         tryCatchOne
## 12            doTryCatch                          doTryCatch
## 13         engine$weave       engine                  weave
## 14      vweave_rmarkdown                    vweave_rmarkdown
## 15     rmarkdown::render                   rmarkdown::render
## 16           knitr::knit                         knitr::knit
```

```
## 17          process_file                       process_file
## 18   withCallingHandlers                withCallingHandlers
## 19         process_group                      process_group
## 20   process_group.block                process_group.block
## 21            call_block                         call_block
## 22            block_exec                         block_exec
## 23                in_dir                             in_dir
## 24              evaluate                           evaluate
## 25    evaluate::evaluate                 evaluate::evaluate
## 26         evaluate_call                      evaluate_call
## 27             timing_fn                          timing_fn
## 28                handle                             handle
## 29   withCallingHandlers                withCallingHandlers
## 30           withVisible                        withVisible
## 31                  eval                               eval
## 32                  eval                               eval
## 33            .GlobalEnv                         .GlobalEnv
## 34      package:envnames                   package:envnames
## 35         package:stats                      package:stats
## 36      package:graphics                   package:graphics
## 37     package:grDevices                  package:grDevices
## 38         package:utils                      package:utils
## 39      package:datasets                   package:datasets
## 40       package:methods                    package:methods
## 41             Autoloads                          Autoloads
## 42          package:base                       package:base
## 43      package:envnames                   package:envnames
## 44         package:stats                      package:stats
## 45      package:graphics                   package:graphics
## 46     package:grDevices                  package:grDevices
## 47         package:utils                      package:utils
## 48      package:datasets                   package:datasets
## 49       package:methods                    package:methods
## 50          package:base                       package:base
## 51            R_EmptyEnv                         R_EmptyEnv
```

For instance, in the above map we can see that the `envnames` package defines an environment called `testenv` which contains two other nested enviroments: `env1` and `env22`.

We can also restrict the lookup table to the environments defined within another environment. Now the `path` to the environment is relative to the environment on which the search is restricted (indicated as the `envir` parameter of the function).

```
get_env_names(envir=env_of_envs)
```

```
##        type      location    locationaddress            address
## 1      user   env_of_envs <00000000093B2F28> <0000000009227280>
## 2  function         tools <00000000089894F8> <0000000008DD6A68>
## 3  function          base <000000000447C350> <0000000008C73A08>
## 4  function      tryCatch <0000000008C73A08> <0000000008C73340>
## 5  function      tryCatch <0000000008C73A08> <0000000008C72FF8>
## 6  function   tryCatchOne <0000000008C72FF8> <0000000008C72C78>
## 7  function         knitr <0000000008EBD870> <0000000008C725E8>
## 8  function         knitr <0000000008EBD870> <0000000008C81D50>
## 9  function     rmarkdown <0000000008CD2080> <00000000068C02D0>
## 10 function         knitr <0000000008EBD870> <0000000008EDD810>
```

```
## 11 function         knitr <0000000008EBD870> <0000000008F9A2C8>
## 12 function          base <000000000447C350> <000000000E2625D8>
## 13 function         knitr <0000000008EBD870> <000000000E237060>
## 14 function         knitr <0000000008EBD870> <000000000E237338>
## 15 function         knitr <0000000008EBD870> <000000000E237418>
## 16 function         knitr <0000000008EBD870> <000000000E014E30>
## 17 function         knitr <0000000008EBD870> <000000000DC7EB28>
## 18 function         knitr <0000000008EBD870> <000000000DC801E8>
## 19 function      evaluate <0000000008EB5390> <000000000DC6C4D8>
## 20 function      evaluate <0000000008EB5390> <000000000D655850>
## 21 function evaluate_call <000000000D655850> <000000000D562578>
## 22 function evaluate_call <000000000D655850> <000000000D562658>
## 23 function          base <000000000447C350> <000000000D560380>
## 24 function          base <000000000447C350> <000000000D557780>
## 25 function          base <000000000447C350> <000000000D5579B0>
## 26 function          base <000000000447C350> <000000000447C3F8>
##                 pathname   path                  name
## 1                   env21                       env21
## 2  tools::buildVignettes        tools::buildVignettes
## 3                tryCatch                    tryCatch
## 4           tryCatchList                tryCatchList
## 5            tryCatchOne                 tryCatchOne
## 6             doTryCatch                  doTryCatch
## 7           engine$weave engine                weave
## 8       vweave_rmarkdown             vweave_rmarkdown
## 9      rmarkdown::render            rmarkdown::render
## 10           knitr::knit                 knitr::knit
## 11          process_file                process_file
## 12   withCallingHandlers          withCallingHandlers
## 13         process_group               process_group
## 14   process_group.block          process_group.block
## 15            call_block                  call_block
## 16            block_exec                  block_exec
## 17                in_dir                      in_dir
## 18              evaluate                    evaluate
## 19    evaluate::evaluate           evaluate::evaluate
## 20         evaluate_call                evaluate_call
## 21             timing_fn                   timing_fn
## 22                handle                      handle
## 23   withCallingHandlers          withCallingHandlers
## 24           withVisible                 withVisible
## 25                 eval                        eval
## 26                 eval                        eval
```

2.2 `environment_name()`: retrieve name of user-defined and function execution environments

The `environment_name()` function (or its alias `get_env_name()`) extends the functionality of the built-in `environmentName()` function by also retrieving the name of user-defined environments and function execution environments.

Although the name of an environment can be easily retrieved with `deparse(substitute(env1))` where `env1` is a user-defined environment, the most useful scenario is when we have just the *memory address* of the environment where e.g. an object resides (as in e.g. `<environment: 0x0437fb40>` in 32-bit systems or `<environment: 0x000000000437fb40>` in 64-bit systems). In this scenario, `environment_name()` can tell us the *name* of the environment having that memory address.

Note that the address-to-name resolution also works for *function execution environments*, as we shall see in the examples below.

The signature of the function is the following:
`environment_name(env, envir = NULL, envmap = NULL, matchname = FALSE, ignore = NULL, include_functions = FALSE)`.

2.2.1 Examples

2.2.1.1 Basic operation Let's retrieve the names of the environments defined above. This may sound trivial because we are already typing the environment name! However, we receive additional information as follows:

- the output from the first call includes *all* the environments where the environment with the given name (e.g. `env1`) is found.
- the output from the second call contains the *path* to use (starting from the calling environment) in order to reach the environment being searched for (e.g. stating that `env21` is found inside environment `env_of_envs`).

```
cat("Name of environment 'env1':\n")
```

```
## Name of environment 'env1':
```

```
environment_name(env1)
```

```
##              R_GlobalEnv package:envnames$testenv
##                   "env1"                   "env1"
```

```
cat("Name of environment 'env21':\n")
```

```
## Name of environment 'env21':
```

```
environment_name(env21)
```

```
## [1] "env_of_envs$env21"
```

For future reference, let us point out that the first case above is a case of *environments with the same name* existing in *different environments*.

If we already know the environment where the environment of interest is defined, we can specify it in the `envir` parameter so that the search for the environment is restricted to the specified environment:

```
cat("Name of environment 'env1' when we specify its location:\n")
```

```
## Name of environment 'env1' when we specify its location:
```

```
environment_name(env1, envir=globalenv())
```

```
## [1] "env1"
```

```
cat("Name of environment 'env21' when we specify its location:\n")
```

```
## Name of environment 'env21' when we specify its location:
```

```
environment_name(env21, envir=env_of_envs)
```

```
## [1] "env21"
```

Note that *no path information is attached now* to the returned names in either case, because only one environment is found inside the respective specified environments.

We can also retrieve the name of the `testenv` environment:

```
cat("Name of environment 'testenv':\n")
```

```
## Name of environment 'testenv':
```

```
environment_name(testenv)
```

```
## [1] "package:envnames$testenv"
```

where we obtain the information that `testenv` is defined in package `envnames`.

2.2.1.2 More advanced examples

As above we saw a case of environments with the same name existing in different environments, let's now see a case of *different environments* having the *same memory address*.

So, let's define a new environment that points to one of the already defined environments, and let's retrieve its name as above:

```
e_proxy <- env_of_envs$env21
environment_name(e_proxy)
```

```
## R_GlobalEnv env_of_envs
##   "e_proxy"     "env21"
```

What we get is a named array containing the names of *all* the environments (in alphabetical order) that point to the same memory address (in this case `env21` and `e_proxy`). The names attribute of the array contains the environments where these environments are found (in this case `env_of_envs` defined in the global environment, and `R_GlobalEnv`, the global environment).

We can disable the behaviour of matching environments *just* by memory address by setting the `matchname` parameter to `TRUE` so that the returned environments must match both the memory address *and* the given name:

```
environment_name(e_proxy, matchname=TRUE)
```

```
## [1] "e_proxy"
```

Now the result is an *unnamed* array because there is only one environment matched by the search for the `e_proxy` environemnt. Furthermore, the result indicates that the environment is defined in the global environment, as otherwise the location where it were defined would be part of the name (as in e.g. `env1$e_proxy`).

Note however that the last call could actually return *more than one environment* in the case where environments sharing the same name (`e_proxy` in the above example) were defined in different environments. We could have this situation if we defined an environment called `"e_proxy"` in environment `env_of_envs`, as shown in the following example:

```
env_of_envs$e_proxy <- new.env()
environment_name(e_proxy, matchname=TRUE)
```

```
## R_GlobalEnv env_of_envs
##   "e_proxy"   "e_proxy"
```

Again a named array is returned with all the matches (by name) to the searched environment.

Finally, if we try to retrieve the environment name of a non-existing environment, we get `NULL`.

```
environment_name(non_existing_env)
```

```
## NULL
```

2.2.1.3 Retrieving the environment name associated with a memory address Now suppose we have a memory address and we would like to know if that memory address represents an environment. We can simply call `environment_name()` with the memory address passed as character argument, as shown in the following example:

```
env1_address = get_obj_address(testenv$env1)
environment_name(env1_address)
```

```
## [1] "package:envnames$testenv$env1"
```

Of course, in practice we would not call the `get_obj_address()` function to get the environment's memory address; we would simply type in the memory address we are after. Note that this memory address depends on the architecture (32-bit or 64-bit) and it can be given in one of the following four ways:

- an 8-digit (32-bit) / 16-digit (64-bit) address, e.g. `"0000000011D7A150"` (64-bit architecture)
- a 10-digit (32-bit) / 18-digit (64-bit) address, e.g. `"0x0000000011D7A150"` (64-bit architecture)
- either of the above addresses enclosed in `< >`, e.g. `"<0000000011D7A150>"` or `"<0x0000000011D7A150>"` (64-bit archiecture)
- a 10-digit (32-bit) / 18-digit (64-bit) address preceeded by the `environment:` keyword and enclosed in `< >`, e.g.: `"<environment: 0x0000000011D7A150>"` (64-bit architecture)

(note: Linux Debian distributions may have a 12-digit memory address representation. The best way to know what the memory address representation is in a particular system is to call e.g. `address("x")`.)

The last format is particularly useful when copying & pasting the result of querying an environment object, for example when typing `testenv$env1` at the `R` command prompt, in which case we get:

```
testenv$env1
```

```
## <environment: 0x0000000006a8f038>
```

If the memory address does not match any of the above formats or does not represent an environment, `environment_name()` returns `NULL`. Ex:

```
x = 2
environment_name(get_obj_address(x))
```

```
## NULL
```

as the address of `x` is not the address of an environment.

2.2.1.4 Retrieving a function execution environment If called from within a function with no arguments, `environment_name()` returns the execution environment of the function, which is identified by the name of the function. This is given with its *full path*, as in e.g. `env1$f`, when `environment_name()` is called from function `f()` defined in environment `env1`.

Since the first argument of `environment_name()` is the environment whose name we want to retrieve, we could also retrieve the execution environment of any calling function by specifying the corresponding `parent.frame`. Once again the name of such parent execution environment would be the name of the function given with its *full path*.

The following example illustrates the above two use cases.

```
with(env_of_envs$env21, {
  f <- function() {
    cat("1) We are inside function:", environment_name(), "\n")
    cat("2) The calling environment is:", environment_name(parent.frame()), "\n")
  }
  g <- function() {
    f()
  }
})
cat("Having defined both f() and g() in environment env_of_envs$env21,
    and having function g() call f()...\n")
```

```
## Having defined both f() and g() in environment env_of_envs$env21,
##     and having function g() call f()...
```

```
cat("...when we call env_of_envs$env21$f() from the global environment,
    we get the output that follows:\n")
```

```
## ...when we call env_of_envs$env21$f() from the global environment,
##     we get the output that follows:
```

```
env_of_envs$env21$f()
```

```
## 1) We are inside function: env_of_envs$env21$f
## 2) The calling environment is: R_GlobalEnv
```

```
cat("\n...and when we call f() from inside function g(),
    we get the output that follows:\n")
```

```
##
## ...and when we call f() from inside function g(),
##     we get the output that follows:
```

```
env_of_envs$env21$g()
```

```
## 1) We are inside function: e_proxy$f
## 2) The calling environment is: env_of_envs$env21$g
```

Note that, in the second case when `f()` is called from `g()` –and not directly from the global environment–, the enviroment showing as path to `f()` is *not* `env_of_envs$env21` (as we would have expected) but `e_proxy`. The reason is that environment `e_proxy` (in the global environment) points to the same memory address as `env_of_envs$env21`. And since environment names are retrieved by their memory address (which in this case is the memory address of `f`'s execution environment), there may be more than one environment matching the same memory address. In such cases, the rule implemented in `environment_name()` is to retrieve the matching environment whose name comes first in alphabetical order (which in this case is `e_proxy` –coming before both `env_of_envs$e_proxy` and `env_of_envs$env21` in alphabetical order, all environments that match the memory address of the environment where `f()` is defined).

But if we call `env_of_envs$env21$f()` (instead of calling `f()` as above) from a function `h()` defined in the `env_of_envs$env21` environment, we get:

```
with(env_of_envs$env21, {
  f <- function() {
    cat("1) We are inside function", environment_name(), "\n")
    cat("2) The calling environment is:", environment_name(parent.frame()), "\n")
  }
  h <- function() {
```

```
    env_of_envs$env21$f()
  }
  }
)
env_of_envs$env21$h()
```

```
## 1) We are inside function env_of_envs$env21$f
## 2) The calling environment is: env_of_envs$env21$h
```

i.e., when making explicit the location of function `f()`, such location is shown as part of the name of the execution environment (as opposed to seeing a "supposedly strange" location `e_proxy` as above).

2.3 `obj_find()`: find the environments where (visible) objects exist

With the `obj_find()` function we can check if an object exists in the whole workspace and retrieve all the environments where it has been found. In the case of packages, only *exported* objects are searched for.

All environments –including system environments, packages, user-defined environments, and optionally function execution environments– are crawled and searched for the object. This includes any environments that are defined *within* other environments (*nested*).

It therefore represents an enhancement to the built-in `exists()` function, which does *not* search for an object inside user-defined and nested environments, nor tells use *where* the object is defined.

The function returns a character array with all the environments where the object has been found.

Objects to search for can be specified either as a symbol or as a string. Ex: `obj_find(x)` and `obj_find("x")` both look for an object called "x". They can also be the result of an expression as in `v[1]`.

The function returns `NULL` if the object is not found or if the expression is invalid. For instance `obj_find(unquote(quote(x)))` returns `NULL` because the `unquote()` function does not exist in R.

The signature of the function is the following:
`obj_find(obj, envir = NULL, envmap = NULL, globalsearch = TRUE, n = 0, return_address = FALSE, include_functions = FALSE, silent = TRUE)`

2.3.1 Examples

2.3.1.1 Let's start with a few object definitions We define a couple of objects in the environments already defined above:

```
x <- 5
env1$x <- 3
with(env_of_envs, env21$y <- 5)
with(env1, {
  vars_as_string <- c("x", "y", "z")
})
```

2.3.1.2 Basic operation Now let's look for these objects:

```
environments_where_obj_x_is_found = obj_find(x)
cat("Object 'x' found
in the following environments:"); print(environments_where_obj_x_is_found)
```

```
## Object 'x' found
## in the following environments:
```

```
## [1] "R_GlobalEnv" "env1"
```

```
environments_where_obj_y_is_found = obj_find(y)
cat("Object 'y' found
in the following environments:"); print(environments_where_obj_y_is_found)
```

```
## Object 'y' found
## in the following environments:
```

```
## [1] "e_proxy"           "env_of_envs$env21"
```

(if we are seeing more environments than expected in the above output, let us recall that two `e_proxy` environments point to the same environment as `env_of_envs$env21`)

```
environments_where_obj_is_found = obj_find(vars_as_string)
cat("Object 'vars_as_string' found
in the following environments:"); print(environments_where_obj_is_found)
```

```
## Object 'vars_as_string' found
## in the following environments:
```

```
## [1] "env1"
```

Let's also look for the objects defined in `vars_as_string` and `vars_quoted`.

```
environments_where_obj_1_is_found = obj_find(env1$vars_as_string[1])
  ## Here we are looking for the object 'x'
cat(paste("Object '", env1$vars_as_string[1], "' found
in the following environments:")); print(environments_where_obj_1_is_found)
```

```
## Object ' x ' found
## in the following environments:
```

```
## [1] "R_GlobalEnv" "env1"
```

```
environments_where_obj_2_is_found = obj_find(env1$vars_as_string[2])
  ## Here we are looking for the object 'y'
cat(paste("Object '", env1$vars_as_string[2], "' found
in the following environments:")); print(environments_where_obj_2_is_found)
```

```
## Object ' y ' found
## in the following environments:
```

```
## [1] "e_proxy"           "env_of_envs$env21"
```

```
environments_where_obj_3_is_found = obj_find(env1$vars_as_string[3])
  ## Here we are looking for the object 'z' which does not exist
cat(paste("Object '", env1$vars_as_string[3], "' found
in the following environments:")); print(environments_where_obj_3_is_found)
```

```
## Object ' z ' found
## in the following environments:
```

```
## NULL
```

or using `sapply()` to look for all the objects whose names are stored in `env1$vars_as_strings` at once:

```
environments_where_objs_are_found = with(env1, sapply(vars_as_string, obj_find) )
cat("The objects defined in the 'env1$vars_as_string' array are found
    in the following environments:\n");
```

```
## The objects defined in the 'env1$vars_as_string' array are found
##     in the following environments:
```

```
print(environments_where_objs_are_found)
```

```
## $x
## [1] "R_GlobalEnv" "env1"
##
## $y
## [1] "e_proxy"           "env_of_envs$env21"
##
## $z
## NULL
```

Note how calling `obj_find()` from within the `env1` environment (which we do in order to resolve the `vars_as_string` variable –the argument of `obj_find()`) still searches for the objects everywhere. This is because parameter `globalsearch` is set to `TRUE` (by default). If we set it to `FALSE` and we add `envir=env1` as searched environment, we would get a non `NULL` value only for the objects defined in the `env1` environment, as shown below:

```
environments_where_objs_are_found = with(env1,
                      sapply(vars_as_string, obj_find, globalsearch=FALSE, envir=env1) )
cat("The objects defined in the 'env1$vars_as_string' array are found
    in the following environments (no globalsearch):\n");
```

```
## The objects defined in the 'env1$vars_as_string' array are found
##     in the following environments (no globalsearch):
```

```
print(environments_where_objs_are_found)
```

```
## $x
## [1] "env1"
##
## $y
## NULL
##
## $z
## NULL
```

NOTE: Even if we run `sapply()` inside environment `env1`, it is important to add parameter `envir=env1` to the call to `obj_find()`; if we don't add it, *no object is found* because the calling environment for `obj_find()` (i.e. its parent environment) is *not* `env1` but the `sapply()` execution environment, where the objects do not exist.

We can also search for objects given as a symbol:

```
environments_where_obj_x_is_found = obj_find(as.name("x"))
cat("Object 'x' found in the following environments:\n")
```

```
## Object 'x' found in the following environments:
```

```
print(environments_where_obj_x_is_found)
```

```
## [1] "R_GlobalEnv" "env1"
```

Finally, we can also search for visible (exported) objects defined in packages:

```
environments_where_obj_is_found = obj_find(aov)
cat("Object 'aov' found in the following environments:\n")
```

```
## Object 'aov' found in the following environments:
```

```
print(environments_where_obj_is_found)
```

```
## [1] "package:stats"
```

2.4 `get_fun_name()`, `get_fun_calling()`, `get_fun_calling_chain()`: retrieve functions in the function calling chain (stack)

Functions `get_fun_name()`, `get_fun_calling()`, and `get_fun_calling_chain()` can be used to retrieve information about calling functions. The first two retrieve information about *one* function while the latter retrieves information about the functions in the calling chain or stack, in the same spirit as `sys.calls()`.

However, the `get_fun_calling_chain()` function was designed to give an output that is easier to handle than the output from `sys.calls()` in the practical scenario of making a decision based on the *name* of the calling function. The following section shows such an example.

The signatures of the three aforementioned functions are:

```
get_fun_name(n = 0)
get_fun_calling(n = 1, showParameters = FALSE)
get_fun_calling_chain(n = NULL, showParameters = FALSE, silent = TRUE)
```

2.4.1 Examples

The example of this section shows the practical impact of using the `get_fun_calling_chain()` function instead of the built-in `sys.calls()` function to retrieve the calling stack and make decisions based on the calling function names.

In particular note:
- How easy it is to check what the calling function is (just do a string comparison as in e.g. `get_fun_calling() == "env1$f"`). On the contrary, when using `sys.call()` we first need to parse the output before making such a comparison. See **this link** for more details.
- We get a data frame containing the chain of calling functions, from the most recent call to least recent, including function parameters if desired.

2.4.1.1 Let's start with a few object definitions

1) First we define a couple of new environments:

```
env11 <- new.env()
env12 <- new.env()
```

2) Now we define an example function `h` to be called by two different functions `f` defined in two different user-environments. This function `h` sums `+1` or `+2` to the input parameter `x` depending on which function `f` was responsible for calling it.

```
with(globalenv(),
h <- function(x, silent=TRUE) {
  fun_calling_chain = get_fun_calling_chain(silent=silent)

  # Do a different operation on input parameter x depending on the calling function
  fun_calling = get_fun_calling(showParameters=FALSE)
  if (fun_calling == "env11$f") { x = x + 1 }
  else if (fun_calling == "env12$f") { x = x + 2 }

  return(x)
}
)
```

3) Finally we define the two functions `f` that call `h`, respectively in environments `env11` and `env12`:

```
with(env11,
  f <- function(x, silent=TRUE) {
    fun_calling_chain = get_fun_calling_chain()
```

```
    return(h(x, silent=silent))
  }
)

with(env12,
  f <- function(x, silent=TRUE) {
    fun_calling_chain = get_fun_calling_chain()
    return(h(x, silent=silent))
  }
)
```

2.4.2 Basic operation

We now run these functions `f` and take note of their output.

- Output from `env11$f()`:

```
## Function calling chain:
##  tools$tools::buildVignettes -> base$tryCatch -> tryCatchList -> tryCatchOne -> doTryCatch -> engine$
##
## When h(x) is called by env11$f(x=0) the output is: 1
```

- Output from `env12$f()`:

```
## Function calling chain:
##  tools$tools::buildVignettes -> base$tryCatch -> tryCatchList -> tryCatchOne -> doTryCatch -> engine$
##
## When h(x) is called by env12$f(x=0) the output is: 2
```

Note how easy it was (by using just a string comparison) to decide what action to take based on the `f()` function calling `h()` and perform a different operation.

Note also that, in order to decide between the two possible calling functions `env11$f()` or `env12$f()` we used `get_fun_calling()`, as opposed to `get_fun_name()`, because the latter returns *just* the function name, devoided of any environment name.

2.5 get_fun_env(): retrieve a function's execution environment

The get_fun_env() function can be used to retrieve the execution environment of a function by simply giving the function's name.

This removes the need of knowing the *position* of the function in the calling chain, which is a piece of information that is required by the usual way of retrieving a function's execution environment, namely with parent.frame().

The following example illustrates.

2.5.1 Basic operation

Let's start defining a couple of functions that make up a function calling chain. The called function h() retrieves and displays the value of variable x both inside h() and inside the calling function env1$g(), whose execution environment is retrieved by get_fun_env("env1$g").

```
h <- function(x) {
  # Get the value of parameter 'x' in the execution environment of function 'env1$g'
  # The returned value is a list because there may exist different instances of the
  # same function.
  xval_h = x
  xval_g = evalq(x, get_fun_env("env1$g")[[1]])
  cat("The value of variable 'x' in function", get_fun_name(), "is", xval_h, "\n")
  cat("The value of variable 'x' inside function env1$g is", xval_g, "\n")
}
env1 <- new.env()
with(env1,
  g <- function() {
    x = 2
    return( h(3) )
  }
)
env1$g()
```

```
## The value of variable 'x' in function h is 3
## The value of variable 'x' inside function env1$g is 2
```

When get_fun_env() is called from outside a function, it returns NULL, even when the function exists.

```
cat("The execution environment of a function that is not in the calling chain is:\n")
```

```
## The execution environment of a function that is not in the calling chain is:
```

```
print(get_fun_env("env1$g"))
```

```
## NULL
```

2.5.2 Advanced example that puts together get_fun_calling() and get_fun_env()

In this example the parent frame of function h() (i.e. the execution environment of the calling function) is retrieved with get_fun_env(get_fun_calling()).

```
h <- function(x) {
  parent_function_name = get_fun_calling(n=1)
  cat("Using get_fun_calling() and environment_name() functions:
      The parent frame of function", get_fun_name(), "is", get_fun_calling(n=2), "\n")
  # Get the value of parameter 'x' in the execution environment of function 'env1$g'
  # The returned value is a list because there may exist different instances of the
```

```
  # same function.
  xval_h = x
  xval_g = evalq(x, get_fun_env(parent_function_name)[[1]])
  cat("Using get_fun_name():
      The value of variable 'x' in function", get_fun_name(), "is", xval_h, "\n")
  cat("Using get_fun_env() and evalq() functions:
      The value of variable 'x' inside function", parent_function_name, "is", xval_g,"\n")
}
env1 <- new.env()
with(env1,
  g <- function() {
    x = 2
    return( h(3) )
  }
)
env1$g()
```

```
## Using get_fun_calling() and environment_name() functions:
##       The parent frame of function h is env1$g
## Using get_fun_name():
##       The value of variable 'x' in function h is 3
## Using get_fun_env() and evalq() functions:
##       The value of variable 'x' inside function env1$g is 2
```

Clearly in the above examples we *already know* the position of function `env1$g()` in the calling chain, so using `parent.frame()` would have sufficed. However, using `get_fun_env()` could help in case the function calling chain from withing `h()` changes in the future, in which case we would not need to update the number of the parent frame in order to refer to the execution environment of function `env1$g()`.

2.6 `get_obj_name()`, `get_obj_value()`: retrieve the name/value of an object at a specified parent generation

The `get_obj_name()` and `get_obj_value()` functions are intended to help track objects and their values as they are passed through different environments. The most useful of the two is `get_obj_name()`, because the *values* of linked objects are the same as they traverse the different environments, making `get_obj_value()` be almost the same as calling `eval()` or `evalq()` at any environment (except for some special cases described in the function's documentation). However, `get_obj_value()` provides some kind of shortcut to the required `eval()` or `evalq()` expressions that do the same thing.

When called from within a function `get_obj_name()` can be used to know the name of the object that *leads* to a particular parameter a few generations back following the function calling chain. In other words, it helps us know the object in a given parent generation that is "responsible" for a function's parameter value.

After learning a little more about `get_obj_name()`, one may have the impression that it gives the same result as the one provided by `deparse(substitute())`. However, this is **not** the case as is shown in the examples that follow.

The signatures of these two functions are:
`get_obj_name(obj, n = 0, eval = FALSE, silent = TRUE)` `get_obj_value(obj, n = 0, silent = TRUE)`

2.6.1 Examples

```
getObjNameAndCompareWithSubstitute <- function(y, eval=FALSE) {
  parent_generation = 2
  get_obj_name_result = get_obj_name(y, n=parent_generation, eval=eval)
  deparse_result = deparse(y)
  substitute_result = substitute(y, parent.frame(n=parent_generation))
  deparse_substitute_result = deparse(substitute(y, parent.frame(n=parent_generation)))
  eval_result = evalq(y, envir=parent.frame(n=parent_generation))
  if (!eval) {
    cat("Result of get_obj_name(y, n=", parent_generation, "): ", get_obj_name_result,
        "\n\tConceptually this is the name of the object at parent generation ",
        parent_generation,
        "\n\tLEADING to *parameter* 'y'.\n", sep="")
    cat("Result of deparse(substitute(y, parent.frame(n=", parent_generation, "))): ",
        deparse_substitute_result,
        "\n\tConceptually this is the substitution of *variable* 'y'
        at parent generation ", parent_generation,
        "\n\tconverted to a string.\n", sep="")
  } else {
    cat("Result of get_obj_name(y, n=", parent_generation, ", eval=", eval, "): ",
        get_obj_name_result,
        "\n\tConceptually this is the object LEADING to *parameter* 'y' evaluated
        at parent generation ", parent_generation, ".\n", sep="")
    cat("Result of deparse(y): ", deparse_result,
        "\n\tConceptually this is the value of *parameter* 'y' converted to a character
        string.\n", sep="")
    cat("Result of substitute(y, parent.frame(n=", parent_generation, ")): ",
        substitute_result,
        "\n\tConceptually this is the substitution of *variable* 'y' at parent generation ",
        parent_generation,
        ".\n", sep="")
    cat("Result of evalq(y, envir=parent.frame(n=", parent_generation, ")): ",
```

```
        eval_result,
        "\n\tConceptually this is the evaluation of *variable* 'y' at parent generation ",
        parent_generation,
        ".\n", sep="")
  }
}

callGetObjNameAndCompareWithSubstitute <- function(x, eval=FALSE) {
  getObjNameAndCompareWithSubstitute(x, eval=eval)
}
```

2.6.1.1 Let's start with a few function definitions

2.6.1.2 Basic operation Let's compare the result of calling `get_obj_name()` with the result of `deparse(substitute())`:

```
y <- -9   # Global variable with the same name as the parameter of testing function
z <- 3
callGetObjNameAndCompareWithSubstitute(z)
```

```
## Result of get_obj_name(y, n=2): z
##  Conceptually this is the name of the object at parent generation 2
##  LEADING to *parameter* 'y'.
## Result of deparse(substitute(y, parent.frame(n=2))): y
##  Conceptually this is the substitution of *variable* 'y'
##         at parent generation 2
##  converted to a string.
```

Note the conceptual difference: `deparse(substitute(y, parent.frame(n=2)))` retrieves the object assigned to `y` at parent generation 2 (`substitution`) and returns it as a string (`deparsing`), while `get_obj_name(y, n=2)` **first** *traces back* the object names in parent generations leading to parameter `y`, and **then** returns the name of the object at the specified parent generation.

When `eval=TRUE`, `get_obj_name()` behaves the same way as `deparse()`, because the values of the objects leading to parameter `y` in parent generations is always the same and equal to the parameter's value. This result is the same as the one obtained by calling `get_obj_value()`. The following example illustrates:

```
y <- -9   # Global variable with the same name as the parameter of testing function
z <- 3
callGetObjNameAndCompareWithSubstitute(z, eval=TRUE)
```

```
## Result of get_obj_name(y, n=2, eval=TRUE): 3
##  Conceptually this is the object LEADING to *parameter* 'y' evaluated
##         at parent generation 2.
## Result of deparse(y): 3
##  Conceptually this is the value of *parameter* 'y' converted to a character
##         string.
## Result of substitute(y, parent.frame(n=2)): y
##  Conceptually this is the substitution of *variable* 'y' at parent generation 2.
## Result of evalq(y, envir=parent.frame(n=2)): -9
##  Conceptually this is the evaluation of *variable* 'y' at parent generation 2.
```

That is, calling `get_obj_name(y, n=n, eval=TRUE)` (or its equivalent `get_obj_value()`) retrieves the value of parameter `y` in parent generation `n`, which is the same in all parent generations and equal to the value of parameter `y` inside the calling function. Therefore, this is the same as the result of `deparse(y)`. On the other hand substituting or evaluating variable `y` in parent generation 2 concerns directly variable `y` in *that* parent generation.

2.6.1.3 Finding the parameter path leading to a given function's parameter

The `get_obj_name()` function can also be used to find the set of variables in the different parent environments leading to a specified variable in the current environment. A particular case of this is the parameter path in a function calling chain leading to a function's parameter, which is illustrated below.

Let's define a set of simple functions that create a calling chain, `f1() -> f2() -> f3()` each of them having a parameter with a different name (`x`, `y`, and `z`):

```
f1 <- function(x) {
  cat("f1(x) is calling f2(y=x)...\n")
  f2(x)
}
f2 <- function(y) {
  cat("f2(y) is calling f3(z=y)...\n")
  f3(y)
}
f3 <- function(z) {
  cat("f3(z) is retrieving the parameter path from three parent environments
      leading to function parameter z...\n\n")
  cat("Output from get_obj_name(z, n=3, silent=FALSE):\n")
  variable_leading_to_z_3levels_back = get_obj_name(z, n=3, silent=FALSE)
}
w = 1.3
f1(w)
```

```
## f1(x) is calling f2(y=x)...
## f2(y) is calling f3(z=y)...
## f3(z) is retrieving the parameter path from three parent environments
##       leading to function parameter z...
##
## Output from get_obj_name(z, n=3, silent=FALSE):
## Start at environment f3, object name is 'z'
## Level 1 back: environment = f2, object name is 'y'
## Level 2 back: environment = f1, object name is 'x'
## Level 3 back: environment = R_GlobalEnv, object name is 'w'
```

So, we clearly see the environments and variables leading to parameter `z` from `R_GlobalEnv$w`:
`R_GlobalEnv$w -> f1$x -> f2$y -> f3$z`

2.6.2 Use of `get_obj_value()`

The result of calling `get_obj_value()` is the same as calling `get_obj_name()` with `eval=TRUE`. It may come as a handy function (by reducing writing) to use in debugger contexts to find out the value of variables in different environments.

The following example illustrates the use of the function from within a function and shows the difference with the result of using `evalq()`. Let's start defining two functions:

```
getObjValueAndCompareWithEval <- function(y) {
  parent_generation = 2
  get_obj_value_result = get_obj_value(y, n=parent_generation)
  eval_result = evalq(y, envir=parent.frame(n=parent_generation))
  cat("Result of get_obj_value(y, n=", parent_generation, "): ", get_obj_value_result,
      "\n\tConceptually this is the object LEADING to *parameter* 'y'
      \tevaluated at parent generation ",
      parent_generation, ".\n", sep="")
  cat("Result of evalq(y, envir=parent.frame(n=", parent_generation, ")): ", eval_result,
```

```
        "\n\tConceptually this is the evaluation of *variable* 'y' at parent generation ",
        parent_generation, ".\n", sep="")
}

callGetObjValueAndCompareWithEval <- function(x) { getObjValueAndCompareWithEval(x) }
```

Now let's see the results of calling this function which explains the differences between `get_obj_value()` and `evalq()`.

```
y <- -9   # Global variable with the same name as the parameter of testing function
z <- 3
callGetObjValueAndCompareWithEval(z)
```

```
## Result of get_obj_value(y, n=2): 3
##  Conceptually this is the object LEADING to *parameter* 'y'
##          evaluated at parent generation 2.
## Result of evalq(y, envir=parent.frame(n=2)): -9
##  Conceptually this is the evaluation of *variable* 'y' at parent generation 2.
```

2.7 `get_obj_address()` and `address()`: retrieve the memory address of an object

Following are examples of using the `get_obj_address()` function to retrieve the memory address of an object, which is then checked by the `address()` function that calls the direct method (via a C function call) to retrieve an object's memory address. The differences between these two functions are also explained.

In the `get_obj_address()` function, the object can be given either as a symbol or as an expression. If given as an expression, the memory address of the *result* of the expression is returned. If the result is yet *another* expression, the process stops, i.e. the memory address of that final expression is returned.

Internally this funcion first calls `obj_find()` to look for the object (using `globalsearch=TRUE`) and then retrieves the object's memory address, showing the name of all the environments where the object was found, or `NULL` if the object is not found.

The signature of the function is the following:
`get_obj_address(obj, envir = NULL, envmap = NULL, n = 0, include_functions = FALSE)`

2.7.1 Examples

The following two calls return the same result:

```
obj_address1 = get_obj_address(x)
cat("Output of 'get_obj_address(x)':\n"); print(obj_address1)
```

```
## Output of 'get_obj_address(x)':
```

```
##          R_GlobalEnv
## "<000000000FB1A3E8>"
```

```
obj_address2 = with(env1, get_obj_address(x))
cat("Output of 'with(env1, get_obj_address(x))':\n"); print(obj_address2)
```

```
## Output of 'with(env1, get_obj_address(x))':
```

```
##          R_GlobalEnv
## "<000000000FB1A3E8>"
```

Note especially the last case, where calling `get_obj_address()` from within the `env1` environment still searches for the object everywhere.

We can restrict the memory addresses returned by making the environment where the object is located explicit –by either using the `$` notation or the `envir` parameter of `get_obj_address()`. In this case only the address of the specified object is returned, even if other objects with the same name exist within the specified environment. A few examples follow:

```
get_obj_address(env1$x)
```

```
## NULL
```

```
get_obj_address(x, envir=env1)
```

```
## NULL
```

```
with(env1, get_obj_address(x, envir=env1))
```

```
## NULL
```

Note there is a slight difference between calling `get_obj_address()` using the `$` notation and calling it with the `envir=` parameter: in the latter case, the result is an *unnamed* array.

Suppose now the object is an expression referencing three potential existing objects as strings, more specifically an array:

```
vars = c("x", "y", "nonexistent")
get_obj_address(vars[1], envir=env1)
```

```
## NULL
```

```
sapply(vars, get_obj_address)
```

```
## $x
##          R_GlobalEnv
## "<000000000FB1A3E8>"
##
## $y
##          R_GlobalEnv               e_proxy    env_of_envs$env21
## "<000000000FDC5378>" "<000000000C1B0DB0>" "<000000000C1B0DB0>"
##
## $nonexistent
## NULL
```

(if we are seeing more environments than expected in the above output, let us recall that environment `e_proxy` points to the same environment as `env_of_envs$env21`)

We can check that the memory address is correct by running the internal function `address()` which calls a C function that retrieves the memory address of an object:

```
address(env1$x)
```

```
## [1] "<00000000044441EF0>"
```

```
address(e_proxy$y)
```

```
## [1] "<000000000C1B0DB0>"
```

Finally: why would we use `get_obj_address()` instead of `address()` to retrieve the memory address of an object? For two main reasons:
- `get_obj_address()` first searches for the object in all user-defined environments, while `address()` needs to be called from within the environment where the object is defined.
- `get_obj_address()` returns `NULL` if the object does not exist, while `address()` returns the *memory address* of the `NULL` object, which may be misleading.

To prove the second statement, we simply run the following two commands which yield the same result:

```
address(env1$nonexistent)
```

```
## [1] "<00000000044441EF0>"
```

```
address(NULL)
```

```
## [1] "<00000000044441EF0>"
```

while running `get_obj_address()` on the non-existent object yields `NULL`:

```
get_obj_address(env1$nonexistent)
```

```
## NULL
```

3 Summing up

We have described all the 11 visible functions defined in the `envnames` package and shown examples of using them, as follows:

1) We have used `get_env_names()` to retrieve all the environments defined in the workspace in the form of a lookup table where the environment name can be looked up from its memory address.

2) We have used `environment_name()` / `get_env_name()` (its alias) to retrieve the name of an environment. This function extends the functionality of the built-in `environmentName()` function by retrieving:

   - the name of a user-defined environment
   - the name and path to environments defined inside other environments
   - the name and path to the function associated to an execution environment
   - the name of the environment associated to a memory address

3) We have used `obj_find()` to find an object in the workspace. This function extends the functionality of the built-in `exists()` function by:

   - searching for the object in user-defined environments and in function execution environments
   - searching for the object recursively (i.e. in environments defined inside other environments)
   - showing the environment where the object is defined

4) We have used `get_fun_name()`, `get_fun_calling()`, `get_fun_calling_chain()` to get the stack of calling functions. These functions return the stack information in a manner that is much simpler than the built-in `sys.calls()` function, making it easier to check the *names* of the calling functions and make decisions that depend on them.

5) We have used `get_fun_env()` to get the execution environment of a function in the calling chain by *simply passing the function's name*, so that we can retrieve the value of objects that exist within.

6) We have used `get_obj_name()`, `get_obj_value()` to retrieve the name and value of the object leading to a given function's parameter.

7) We have used `get_obj_address()`, `address()` to retrieve the memory address of an object. These functions provide a functionality that is not available in base R. Note that the `data.table` package also provides a function called `address()` to retrieve the memory address of an object; however the object is *not searched for in the whole workspace* as is the case with the `get_obj_address()` function in this package.

This vignette was generated under the following platform:

```
##          SystemInfo
## sysname     Windows
## release      10 x64
## version build 18363
## machine      x86-64
```

```
##                _
## platform       x86_64-w64-mingw32
## arch           x86_64
## os             mingw32
## system         x86_64, mingw32
## status
## major          3
## minor          6.3
## year           2020
## month          02
## day            29
## svn rev        77875
## language       R
## version.string R version 3.6.3 (2020-02-29)
## nickname       Holding the Windsock
```