Package 'envnames'

December 8, 2020

Type Package

Title Keep Track of User-Defined Environment Names

Version 0.4.1

Date 2020-12-05

Author Daniel Mastropietro

Maintainer Daniel Mastropietro <mastropi@uwalumni.com>

Description Set of functions to keep track and find objects in user-defined environments
by identifying environments by name --which cannot be retrieved with the built-
in function environmentName().
The package also provides functionality to obtain simplified information about function call-
ing chains
and to get an object's memory address.

URL https://github.com/mastropi/envnames

BugReports https://github.com/mastropi/envnames/issues

License GPL

Suggests knitr, rmarkdown, testthat

VignetteBuilder knitr

RoxygenNote 7.0.2

NeedsCompilation yes

Repository CRAN

Date/Publication 2020-12-08 06:40:02 UTC

R topics documented:

envnames-package 2
address 5
collapse_root_and_member 5
environment_name 6
get_env_names 8
get_fun_calling 10

get_fun_calling_chain 11
get_fun_env 12
get_fun_name 13
get_obj_address 14
get_obj_name 17
get_obj_value 19
obj_find 22
testenv 24

Index **25**

`envnames-package` *Track user-defined environment names*

Description

The main goal of this package is to overcome the limitation of the built-in `environmentName` function of the base package which cannot retrieve the name of an environment unless it is a package or the global environment. This implies that all user-defined environments don't have a "name assigned" that can be retrieved and refer to the environment.

The envnames package solves this problem by creating a lookup table that maps environment names to their memory addresses. Using this lookup table, it is possible to retrieve the name of any environment where an object resides or, from within a function, to retrieve the calling stack showing the function names and their enclosing environment name, i.e. the environment where the functions have been defined. The latter can be done with just a function call which returns a string that can directly be used inside a `cat()` call to display the function name (as opposed to using the R function `sys.call` which does not return a string, but a more complicated object, namely a `call` object from where the string with the function name is still to be extracted to be used inside `cat()`).

Package conventions: all functions in this package follow the underscore-separated and all-lowercase naming convention (e.g. `environment_name()`, `get_obj_address()`, etc.).

Details

The package currently contains 12 visible functions. Following is an overview on how to use the main functions of the package. Please refer to the vignette for further information.

1) Use `get_obj_address` to retrieve the memory address of any object, including environments.

2) Use `environment_name` to retrieve the name of an environment created with `new.env`. The environment can be given as a string containing its 16-digit memory address.

3) Use `obj_find` to find the environments where a given object is defined.

4) Use `get_fun_calling(n)` from within a function to retrieve the name of the calling function `n` levels up in the calling stack together with their enclosing environment name.

5) Use `get_fun_calling_chain` from within a function to get the calling functions stack.

Author(s)

Daniel Mastropietro

Maintainer: Daniel Mastropietro <mastropi@uwalumni.com>

References

Motivation for developing this package:

- A comment by Quantide's instructor Andrea Spano during his "R for developers" course (http://www.quantide.com/courses-overview/r-for-developers) about the impossibility of retrieving the name of user-defined environments.

- A question posted by Gabor Grothendieck at the R-Help forum (https://stat.ethz.ch/pipermail/r-help/2010-July/245646.html)

See Also

environmentName in the base package for the built-in function that retrieves environment names of packages.

exists and find for alternatives of looking for objects in the workspace.

sys.call for other alternatives for retrieving the function calling stack.

Examples

```
library(envnames)
rm(list=ls())

### Example 1: Retrieve the names of user-defined environments (created with new.env())
# Create new environments
env1 <- new.env()                       # Environment in .GlobalEnv
env2 <- new.env()                       # Environment in .GlobalEnv
env3 <- new.env(parent=baseenv())     # Environment whose enclosure or parent environment
                                        # is the base environment
                                        # (as opposed to the global environment)
env_of_envs <- new.env()          # User-defined environment that contains other environments
with(env_of_envs, env11 <- new.env()) # Environment defined inside environment env_of_envs

# Retrieve the environment name
environment_name(env1)               # named array with value "env1" and name "R_GlobalEnv"
environment_name(env3)               # named array with value "env3" and name "R_GlobalEnv"
environment_name(env9)                      # NULL (env9 does not exist)
environment_name(env_of_envs)             # named array with value "env_of_envs" and name
                                            # "R_GlobalEnv"
# (2018/11/19) THE FOLLOWING IS AN IMPORTANT TEST BECAUSE IT TESTS THE CASE WHERE THE ADDRESS-NAME
# LOOKUP TABLE CONTAINS ONLY ONE ROW (namely the row for the env11 environment present in
# env_of_envs), WHICH CANNOT BE TESTED VIA TESTS USING THE testthat PACKAGE BECAUSE IN THAT CONTEXT
# THE LOOKUP TABLE NEVER HAS ONLY ONE ROW!
# (for more info about this limitation see the test commented out at the beginning of
# test-get_env_names.r.
environment_name(env11, envir=env_of_envs)   # "env11"
environment_name(env11)                      # named array with value "env11" and name
                                             # "R_GlobalEnv$env_of_envs"


### Example 2: Retrieve calling functions and their environments
### Note in particular the complicated use of sys.call() to retrieve the call as a string...
# Define two environments
```

```
env1 <- new.env()
env2 <- new.env()
# Define function g() in environment env2 to be called by function f() below
# Function g() prints the name of the calling function.
with(env2,
     g <- function(x) {
       # Current function name
       fun_current_name = get_fun_name()

       # Get the name of the calling environment and function
 fun_calling_name = get_fun_calling()

       # Show calling environment using and not using the envnames package
       cat("Now inside function", fun_current_name, "\n")
       cat("Calling environment name (using environmentName(parent.frame())):  \"",
            environmentName(parent.frame()), "\"\n", sep="")
       cat("Calling environment name (using sys.call(1) inside
           'as.character( as.list(sys.call(1))[[1]]) )':", " \"",
            as.character( as.list(sys.call(1))[[1]]), "\"\n", sep="")
       cat("Calling environment name (using envnames::get_fun_calling()): \"",
           fun_calling_name, "\"\n", sep="")

       # Start process
       x = x + 2;
       return(invisible(x))
     }
)

# Calling function whose name should be printed when g() is run
with(env1,
     f <- function(x) {
       # Start
       gval = env2$g(x)
       return(invisible(gval))
     }
)

# Run function f to show the difference between using and
# not using the envnames package to retrieve the function calling stack.
env1$f(7)


### Example 3: find the location of an object
# This differs from the R function exists() because it also searches
# in user-defined environments and any environments wihin.
obj_find(f)                   # "env1"
obj_find("f")                 # Same thing: "env1"
obj_find("f", silent=FALSE)   # Same result, but run verbosely

env2$x <- 2
obj_find(x)                   # "env2"

obj_find(nonexistent)         # NULL
```

address *Call the C function address() that retrieves the memory address of an R object*

Description

Call the C function address() that retrieves the memory address of an R object

Usage

```
address(x)
```

Arguments

x	object whose memory address is of interest.

Value

the memory address of object x or NULL if the object does not exist in the R workspace.

collapse_root_and_member *Put together a root name with a member name*

Description

This is the opposite operation of extract_root_and_last_member(): the root and its supposed member are put together using the $ separator, as in env_of_envs$env1$x, where the root and the member could be either env_of_envs$env1 and x or env_of_envs and env1$x.

Usage

```
collapse_root_and_member(root, member)
```

Arguments

root	String containing the root name to concatenate. It may be NULL or empty.
member	String containing the member name to concatenate. It may be NULL or empty.

Value

A string concatenating the root and the member names with the $ symbol. If any of them is empty or NULL, the other name is returned or "" if the other name is also empty or NULL.

See Also

`extract_root_and_last_member()`

`environment_name` *Retrieve the name of an environment*

Description

Retrieve the name of an environment as the `environmentName` function of the base package does, but extending its functionality to retrieving the names of user-defined environments and function execution environments.

Usage

```
environment_name(
  env = parent.frame(),
  envir = NULL,
  envmap = NULL,
  matchname = FALSE,
  ignore = NULL,
  include_functions = FALSE
)
```

Arguments

`env`	environment whose name is of interest. It can be given as an object of class environment, as a string with the name of the environment, or as a string with the memory address of the environment. The latter is useful to find out if a given memory address is the reference of an environment object. Note that the variable passed may or may *not* exist in the calling environment, as the purpose of this function is also to search for it (and return its name if it is an environment). It defaults to parent.frame(), meaning that the name of the environment that calls this function is retrieved.
`envir`	environment where `env` should be searched for. When `NULL`, `env` is searched in the whole workspace, including packages and user-defined environments, recursively.
`envmap`	data frame containing a lookup table with name-address pairs of environment names and addresses to be used when searching for environment `env`. It defaults to `NULL` which means that the lookup table is constructed on the fly with the environments defined in the `envir` environment –if not `NULL`–, or in the whole workspace if `envir=NULL`. See the details section for more information on its structure.
`matchname`	flag indicating whether the match for `env` is based on its name or on its memory address. In the latter case all environments sharing the same memory address of the given environment are returned. Such scenario happens when, for instance, different environment objects have been defined equal to another environment (as in `env1 <-env`). It defaults to `FALSE`.

`ignore` one or more environment names to ignore if found during the search. These environments are removed from the output. It should be given as a character array if more than one environments should be ignored. See the details section for more information.

`include_functions` flag indicating whether to look for user-defined environments inside function execution environments. This should be used with care because in a complicated function chain, some function execution environments may contain environments that point to other environments (e.g. the 'envclos' environment in the `eval()` function when running tests using the `test_that` package).

Details

If `env` is an environment it is searched for in the `envir` environment using its memory address. If `env` is a string containing a valid 16-digit memory address (enclosed in < >), the memory address itself is searched for among the defined environments in the `envir` environment. In both cases, if `envir=NULL` the search is carried out in the whole workspace.

It may happen that more than one environment exist with the same memory address (for instance if an environment is a copy of another environment). In such case, if `matchname=FALSE`, the names of ALL the environments matching `env`'s memory address are returned. Otherwise, only the environments matching the given name are returned.

If `envmap` is passed it should be a data frame providing an address-name pair lookup table of environments and should contain at least the following columns:

- `location` for user-defined environments, the name of the environment where the environment is located; otherwise `NA`.
- `pathname` the full *environment path* to reach the environment separated by `$` (e.g. `"env1$env$envx"`)
- `address` an 8-digit (32-bit architectures) thru 16-digit (64-bit architectures) memory address of the environment given in `pathname` enclosed in < > (e.g. `"<0000000007DCFB38>"` (64-bit architectures)) Be ware that Linux Debian distributions may have a 12-digit memory address representation. So the best way to know is to check a memory address by calling e.g. 'address("x")'.

Passing an `envmap` lookup table is useful for speedup purposes, in case several calls to this function will be performed in the context of an unchanged set of defined environments. Such `envmap` data frame can be created by calling get_env_names. Use this parameter with care, as the matrix passed may not correspond to the actual mapping of existing environments to their addresses and in that case results may be different from those expected.

The following example illustrates the use of the `ignore` parameter:

```
for (e in c(globalenv(),baseenv())) { print(environment_name(e,ignore="e")) }
```

That is, we iterate on a set of environments and we don't want the loop variable (an environment itself) to show up as part of the output generated by the call to `environment_name()`.

Value

If `matchname=FALSE` (the default), an array containing the names of all the environments (defined in the `envir` environment if `envir` is not `NULL`) having the same memory address as the `env` environment.

If matchname=TRUE, the environment name contained in env is used in addition to the memory address to check the matched environments (potentially many if they have the same memory address) so that only the environments having the same name and address as the env environment are returned. Note that several environments may be found if environments with the same name are defined in different environments. WARNING: in this case, the name is matched exactly as the expression given in env. So for instance, if env=globalenv()$env1 the name "globalenv()$env1" is checked and this will not return any environments since no environment can be called like that. For such scenario call the function with parameter env=env1 instead, or optionally with env=env1 and envir=globalenv() if the env1 environment should be searched for just in the global environment.

If env is not found or it is not an environment, NULL is returned.

Examples

```
# Retrieve name of a user-defined environment
env1 <- new.env()
environment_name(env1)                    # "env1"

# Retrieve the name of an environment given as a memory address
env1_address = get_obj_address(globalenv()$env1)
environment_name(env1_address)            # "env1"

# Create a copy of the above environment
env1_copy <- env1
environment_name(env1)                    # "env1" "env1_copy"
# Retrieve just the env1 environment name
environment_name(env1, matchname=TRUE)    # "env1"

# Retrieve the name of an environment defined within another environment
with(env1, envx <- new.env())
environment_name(env1$envx)               # "env1$envx" "env1_copy$envx"
environment_name(env1$envx, matchname=TRUE)
  ## NULL, because the environment name is "envx", NOT "env1$envx"

# Get a function's execution environment name
with(env1, f <- function() { cat("We are inside function", environment_name()) })
    ## "We are inside function env1$f"
```

---

get_env_names *Create a lookup table with address-name pairs of environments*

Description

Return a data frame containing the address-name pairs of system, package, namespace, user-defined, and function execution environments in the whole workspace or within a given environment.

Usage

```
get_env_names(envir = NULL, include_functions = FALSE)
```

Arguments

envir	environment where environments are searched for to construct the lookup table. It defaults to NULL which means that all environments in the whole workspace should be searched for and all packages in the search() path should be returned including their namespace environments.
include_functions	flag indicating whether to include in the returned data frame user-defined environments defined inside function execution environments.

Details

The table includes the empty environment as well when the address-name pairs map is constructed on the whole workspace.

The search for environments is recursive, meaning that a search is carried out for environments defined within other user-defined environments and, when include_functions=TRUE within function execution environments.

The search within packages is always on *exported objects* only.

If envir=NULL the lookup table includes all system, package, and namespace environments in the search() path, as well as all user-defined found in *any* of those environments (with recursive search), and all function execution environments.

If envir is not NULL the lookup table includes just the user-defined and function execution environments found inside the given environment (with recursive search).

Value

A data frame containing the following seven columns:

- type type of environment ("user" for user-defined environments, "function" for function execution environments, "system/package" for system or package environments, "namespace" for namespace environments, and "empty" for empty environments such as emptyenv()).
- location location of the environment, which is only non-NA for user-defined and function execution environments:
  - for a user-defined environment, the location is the system environment or package where the environment resides (note that this may be different from the parent environment if the parent environment was set during creation with the parent= option of the new.env() function or using the parent.env() function)
  - for a function execution environment, the location is the function's enclosing environment, i.e. the environment where the function is defined.
- locationaddress the memory address of the location environment.
- address memory address of the environment. This is the key piece of information used by the package to retrieve the environment name with the environment_name() function. For functions, this is the address of the function's execution environment.
- pathname path to the environment and its name. This is the combination of columns path and name whose values are put together separated by $.
- path path to the environment (i.e. all environments that need to be traversed in order to reach the environment).

- name name of the environment.

The type column is used to distinguish between user-defined environments, function execution environments, package or system environments, namespace environments, and empty environments.

The data frame is empty if no environments are found in the given envir environment.

NULL is returned when an error occurs.

Examples

```
# Create example of chained environments
env1 <- new.env()
with(env1, env11 <- new.env())
with(env1$env11, envx <- new.env())

# Address-name pairs of all environments defined in the workspace,
# including environments in the search path
get_env_names()  # returns a data frame with at least the following user environments:
                 # "env1", "env1$env11", "env1$env11$envx"

# Address-name pairs of the environments defined in a given user-defined environment
get_env_names(env1)  # returns a data frame with the following user environments:
                     # "env11", "env11$envx"

# Address-name pairs of the environments defined in a given package
get_env_names(as.environment("package:stats")) # should return an empty data frame
                                               # (since the stats package does not
                                               # have any environments defined)
```

---

get_fun_calling *Return the name of a calling function with its context or path*

Description

This is a wrapper for get_fun_calling_chain(n) and returns the name of the calling function including the environment where it is defined n levels up. The two pieces of information are separated by the $ sign.

Usage

```
get_fun_calling(n = 1, showParameters = FALSE)
```

Arguments

n	non-negative integer indicating the number of levels to go up from the calling function to retrieve the function in the calling chain. It defaults to 1, which means "return the last function in the calling chain".
showParameters	flag indicating whether the parameters of the function call should also be shown in the output.

See Also

get_fun_name to retrieve *just* the name of the function, without its context (e.g. "f").

Examples

```
# Prepare environments
env1 <- new.env()
env2 <- new.env()
with(env2, env21 <- new.env())

# Function that shows the names of calling functions in the chain and their environments
f <- function(x) {
 cat("Now in function:", get_fun_calling(0), "\n")
 cat("\tName of the calling function:", get_fun_calling(), "\n")
 cat("\tName of the calling function two levels up:", get_fun_calling(2), "\n")
 cat("\tName of the calling function three levels up:", get_fun_calling(3), "\n")
 cat("\tName of the calling function four levels up:", get_fun_calling(4), "\n")
}

# Prepare a calling chain
with(env1, g <- function() { f(3) })
with(env2, h <- function() { env1$g() })
with(env2$env21, hh <- function() { env2$h() })

# Run the different functions defined to show the different calling chains
env1$g()
env2$h()
env2$env21$hh()
```

---

get_fun_calling_chain *Return the chain of calling functions*

Description

Return a data frame with the stack or chain of function calls, or optionally the information on one particular function in this chain.

Usage

```
get_fun_calling_chain(n = NULL, showParameters = FALSE, silent = TRUE)
```

Arguments

n	non-negative integer specifying the level of interest in the function calling chain, where 0 means the function calling get_fun_calling_chain. It defaults to NULL, in which case the full chain is returned.
showParameters	flag indicating whether the parameters of the function call should also be shown in the output.
silent	whether to run in silent mode. If FALSE, the calling chain is shown in an intuitive way. It defaults to TRUE.

Value

If n=NULL (the default) a data frame with the function calling chain information, with the following columns:

- fun: the function name (including parameters if showParameters=TRUE)
- env: the function's enclosing enviroment, i.e. the environment where the function is defined as returned by environment(<function>)
- envfun: the environment where the function is defined together with the function name (and its parameters if showParameters=TRUE) separated by a $ sign. Ex: env1$f()

The rownames of the data frame are the stack level of the function calls in the chain, from 0 up to the number of functions in the chain, where 0 indicates the current function (i.e. the function that called get_fun_calling_chain).

The functions in the data frame are sorted from most recent to least recent call, much like the common way of displaying the function stack in debug mode.

If the function is NOT called from within a function, NULL is returned.

If n is not NULL and is non-negative, the environment and the function name (including parameters if showParameters=TRUE) separated by a $ sign are returned (ex: env1$f(x = 3,n = 1)).

if n < 0 or if n is larger than the function calling chain length, NULL is returned.

---

get_fun_env *Return the execution environment of a function*

Description

Return the execution environment of a function by going over the execution environments of all functions in the calling chain.

Usage

```
get_fun_env(fun_name_or_address)
```

Arguments

fun_name_or_address
: string containing either the name of the function of interest or the memory address of the execution environment to retrieve (N.B. this sould not be the memory address of the *function itself*, but the memory address of its *execution environment*). When the function name is given, it should be given with its full path, i.e. including the environment where it is defined (e.g. "env1$f") and with no arguments.

Details

This function is expected to be called from within a function. Otherwise, the function calling chain is empty and the function returns NULL.

Value

When the input parameter is a memory address, the execution environment of the function whose memory address (of the execution environment) equals the given memory address.

When the input parameter is a function name, a list of ALL the execution environments belonging to a function whose name coincides with the given name (including any given path). Note that these may be many environments as the same function may be called several times in the function calling chain.

Examples

```
# Define the function that is called to show the behaviour of get_fun_env()
h <- function(x) {
  # Get the value of parameter 'x' in the execution environment of function 'env1$g'
 # If function 'env1$g' is not found, 'x' is evaluated in the current environment or function
  xval = evalq(x, get_fun_env("env1$g")[[1]])
  return(xval)
}
# Define the function that calls h() in a user-defined environment
env1 <- new.env()
with(env1,
  g <- function(y) {
    x = 2
    return( h(y) )
  }
)
# Call env1$g()
cat("The value of variable 'x' inside env1$g is", env1$g(3), "\n")
  ## Prints '2', because the value of x inside env1$g() is 2
  ## ('3' is the value of variable 'y' in env1$g(), not of variable 'x')

# When get_fun_env() is called from outside a function, it returns NULL
get_fun_env("env1$g")  # NULL, even if function 'g' exists,
                       # but we are not calling get_fun_env() from a function
```

`get_fun_name` *Return the name of the current function or a calling function in the chain*

Description

Return the name of the function that has been called `n` levels up from a given function's body. This function is intended to be called only within a function.

Usage

```
get_fun_name(n = 0)
```

Arguments

n	number of levels to go up in the calling chain in search of the calling function name. Defaults to n=0, meaning that the name returned is the name of the function that calls get_fun_name.

Value

A string containing the name of the function that has been called n levels up from the function calling get_env_name. The function name is returned without context, that is the enclosing environment of the function is not part of the returned value. (e.g. if the function is env1$f or env1$env2$f only "f" will be returned).

See Also

get_fun_calling to retrieve the name of the function with its context (e.g. "env1$f").

Examples

```
# Show the name of the active function
f <- function() { cat("We are in function:", get_fun_name(), "\n") }
f()

# Show the name of the calling function
f <- function(x) { cat("Calling function name is:", get_fun_name(1), "\n") }
env1 <- new.env()
with(env1, g <- function() { f(3) })
env1$g()
```

---

get_obj_address *Return the memory address of an object*

Description

Return the memory address of an object after recursively searching for the object in all the environments defined in a specified environment or in all the environments defined in the whole workspace.

Usage

```
get_obj_address(
  obj,
  envir = NULL,
  envmap = NULL,
  n = 0,
  include_functions = FALSE
)
```

Arguments

obj	object whose memory address is requested. It can be given as a variable name or an expression. Strings representing object names are not interpreted and return NULL.
envir	environment where the object should be searched for. All parent environments of envir are searched as well. It defaults to NULL which means that it should be searched in the whole workspace (including packages, namespaces, and user-defined environments).
envmap	data frame containing a lookup table with name-address pairs of environment names and addresses to be used when searching for environment env. It defaults to NULL which means that the lookup table is constructed on the fly with the environments defined in the envir environment –if not NULL–, or in the whole workspace if envir=sNULL. See the details section for more information on its structure.
n	number of levels to go up from the calling function environment to resolve the name of obj. It defaults to 0 which implies the calling environment.
include_functions	whether to include funtion execution environments as environments where the object is searched for. Set this flag to TRUE with caution because there may be several functions where the same object is defined, for instance functions that are called as part of the object searching process!

Details

The object is first searched recursively in all environments defined in the specified environment (if any), by calling obj_find. If no environment is specified, the object is searched recursively in the whole workspace.

The memory address is then retrieved for every object found in those environments having the same name as the given object obj.

Strings return NULL but strings can be the result of an expression passed as argument to this function. In that case, the string is interpreted as an object and its memory address is returned as long as the object exists.

If envmap is passed it should be a data frame providing an address-name pair lookup table of environments and should contain at least the following columns:

- location for user-defined environments, the name of the environment where the environment is located; otherwise NA.
- pathname the full *environment path* to reach the environment separated by $ (e.g. "env1envenvx")
- address an 8-digit (32-bit architectures) thru 16-digit (64-bit architectures) memory address of the environment given in pathname enclosed in < > (e.g. "<0000000007DCFB38>" (64-bit architectures)) Be ware that Linux Debian distributions may have a 12-digit memory address representation. So the best way to know is to check a memory address by calling e.g. 'address("x")'.

Passing an envmap lookup table is useful for speedup purposes, in case several calls to this function will be performed in the context of an unchanged set of defined environments. Such envmap data

frame can be created by calling get_env_names. Use this parameter with care, as the matrix passed may not correspond to the actual mapping of existing environments to their addresses and in that case results may be different from those expected.

Value

The 8-digit (32-bit architectures) thru 16-digit (64-bit architectures) memory address of the input object given as a string enclosed in <> (e.g. "<0000000005E90988>") (note that Ubuntu Debian may use 12-digit memory addresses), or NULL under any of the following situations:

- the object is NULL, NA, or a string, or any other object whose memory address changes every time the object is referred to (for instance for alist[1] –as opposed to alist[[1]]– where alist is a list.
- the object is a constant (e.g. TRUE, 3, etc.)
- the object does not exist in the given environment.
- the object is an expression that cannot be evaluated in the given environment.

Note that for the last case, although constants have a memory address, this address is meaningless as it changes with every invocation of the function. For instance, running address(3) several times will show a different memory address each time, and that is why get_obj_address returns NULL in those cases.

When envir=NULL (the default) or when an object exists in several environments, the memory address is returned for all of the environments where the object is found. In that case, the addresses are stored in an array whose names attribute shows the environments where the object is found.

Examples

```
env1 = new.env()
env1$x = 3                        # x defined in environment 'env1'
x = 4                             # x defined in the Global Environment
get_obj_address(env1$x)           # returns the memory address of the object 'x'
                                  # defined in the 'env1' environment
get_obj_address(x, envir=env1)    # same as above
get_obj_address(x)              # Searches for object 'x' everywhere in the workspace and
                                # returns a named array with the memory address of all its
                                  # occurrences, where the names are the names of the
                                  # environments where x was found.

# Memory addresses of objects whose names are stored in an array and retrieved using sapply()
env1$y <- 2;
objects <- c("x", "y")
sapply(objects, FUN=get_obj_address, envir=env1) # Note that the address of object "x"
                                                 # is the same as the one returned above
                                                 # by get_obj_address(x, envir=env1)

# Memory address of elements of a list
alist <- list("x")
get_obj_address(alist[[1]])       # memory address of object 'x'
get_obj_address(alist[1])         # NULL because alist[1] has a memory address
                                  # that changes every time alist[1] is referred to.
```

get_obj_name *Return the name of an object at a given parent generation from an environment*

Description

A practical use of this function is to retrieve the name of the object leading to a function's parameter in the function calling chain, at any parent generation.

Usage

```
get_obj_name(obj, n = 0, eval = FALSE, silent = TRUE)
```

Arguments

obj	object whose name at a given parent generation is of interest.
n	number of parent generations to go back from the calling environment to retrieve the name of the object that leads to obj in the function calling chain. See details for more information.
eval	whether to evaluate obj in the n-th parent generation before getting the object's name in that environment. See details for more information.
silent	when FALSE, the names of the environments and objects in those environments are printed as those environments are traversed by this function.

Details

In particular, it provides a handy way of retrieving the name of a function's parameter and use it in e.g. messages to the user describing the arguments received by the function. In this context, it is a shortcut to calling `as.list(environment())`, which returns a list of parameter names and parameter values. See the Examples section for an illustration.

This function goes back to each parent generation from the calling function's environment and at each of those parent generations it retrieves the name of the object that is part of the parameter chain leading to the calling function's parameter.

To illustrate: suppose we call a function `f <-function(x)` by running the piece of code `f(z)`, and that `f` calls another function `g <-function(y)` by running the piece of code `g(x)`.

That is, we have the parameter chain: `z -> x -> y`

If, inside function `g()`, we call `get_obj_name()` as follows, we obtain respectively: `get_obj_name(y,n=1)` yields `"x"` `get_obj_name(y,n=2)` yields `"z"`

because these calls are telling "give me the name of object `y` as it was called `n` levels up from the calling environment –i.e. from the environment of `g()`.

Note that the results of these two calls are different from making the following two `deparse(substitute())` calls: `deparse(substitute(y,parent.frame(n=1)))` `deparse(substitute(y,parent.frame(n=2)))` because these calls simply `substitute` or evaluate `y` at the `n`-th parent generation. If `y` is not defined at those parent generations, the `substitute()` calls return simply `"y"`.

On the contrary, the previous two calls to `get_obj_name()` return the name of the object in the parameter chain (`z -> x -> y`) *leading* to `y`, which is a quite different piece of information.

When eval=TRUE, the result is the same as the result of `deparse()` except for the following three cases:

- if the object passed to `get_obj_name()` evaluates to a name, it returns that name, without any added quotes. For example, if `v = "x"` then `get_obj_name(v,eval=TRUE)` returns `"x"` while `deparse(v)` returns `"\"x\""`.
- the result of `NULL` is `NULL` instead of `"NULL"` which is the case with `deparse()`.
- the result of a non-existent object is `NULL`, while `deparse()` returns an error stating that the object does not exist.

When `get_obj_name` operates on non-existent objects it works at follows:

- when `eval=FALSE` it returns the name of the non-existent object enclosed in quotes (e.g. `get_obj_name(nonexistent)` returns `"nonexistent"`, assuming `nonexistent` does not exist).
- when `eval=TRUE` it returns NULL.

Finally `get_obj_name(NULL)` returns `NULL`, while `as.character(NULL)` returns `as.character(0)`.

Value

The name of the object in the n-th parent generation environment.

See Also

get_obj_value

Examples

```
# Example 1:
# This example shows the difference between using get_obj_name() and deparse(substitute())
g <- function(y) { return(list(obj_name=get_obj_name(y, n=2, silent=FALSE),
                               substitute=deparse(substitute(y, parent.frame(n=2))) )) }
f <- function(x) { g(x) }
z = 3;
f(z)           # After showing the names of objects as they
               # are traversed in the parameter chain (silent=FALSE),
               # this function returns a list where
               # the first element (result of get_obj_name()) is "z"
               # and the second element (result of deparse(substitute())) is "y".
               # Note that 'z' is the object leading to object 'y'
               # inside function g() if we follow the parameter names
               # leading to 'y' in the function calling chain.

# Example 2:
# When eval=TRUE, get_obj_name() behaves the same way as deparse()
# (except for the cases noted in the Details section)
# because the values of all objects linked by the parameter chain
# are ALL the same.
```

```
g <- function(y) { return(list(obj_name=get_obj_name(y, n=2, eval=TRUE),
                               deparse=deparse(y))) }
f <- function(x) { g(x) }
z = 3
f(z)           # Returns a list where both elements are equal to "3"
               # because the output of get_obj_name() with eval=TRUE
               # and deparse() are the same.

# Example 3:
# This example shows how we can use get_obj_name() to get the parameter names
# of non '...' parameters, which are then used in messages to the user.
# The advantage of using get_obj_name() as opposed to the hard-coded parameter name
# is that an error is raised if the parameter does not exist.
# An example is also shown that uses as.list(environment()), which clearly is more
# general... get_obj_name() should be used when referring to a couple of specific
# parameters.
f <- function(x, y, ...) {
  cat("Arguments received by the function (using get_obj_name()) (explicit listing):\n")
  cat(get_obj_name(x), ":", x, "\n")
  cat(get_obj_name(y), ":", y, "\n")
 cat("Arguments received by the function (using as.list(environment())) (automatic listing):\n")
  paramsList = as.list(environment())
  paramsNames = names(paramsList)
  sapply(paramsNames, get_obj_name)
  for (p in paramsNames) {
    cat(p, ":", paramsList[[p]], "\n")
  }
}
z = 5
extra_param = "a '...' parameter"
  ## Note: this exra parameter is NOT shown neither by get_obj_name()
  ## nor by as.list(environment())
f("test", z, extra_param)
```

`get_obj_value` *Return the value of the object at a given parent generation leading to the specified object*

Description

This function is mostly useful in debugging contexts to query the value of a variable in specific environments of the calling stack.

Usage

```
get_obj_value(obj, n = 0, silent = TRUE)
```

Arguments

obj	object whose value should be returned. The object can be passed either as a variable name or as a string representing the object whose value is of interest.
n	number of parent generations to go back to retrieve the value of the object that leads to obj in the function calling chain. See details for more information.
silent	when FALSE, the names of the environments and objects in those environments are printed, as those environments are traversed by this function.

Details

The result of this function is similar to using `eval()` or `evalq()` but not quite the same. Refer to the Details and Examples sections for explantion and illustration of the differences.

The purpose of this function is to get the value of object `obj` in a given parent environment.

Note that conceptually this is NOT the same as calling `evalq(obj,parent.frame(n))`, because of the following:

- `evalq()` evaluates the object named `obj` in the environment that is at the `n`-th parent generation. (Note the use of `evalq()` and not `eval()` because the latter evaluates the object at the calling environment first, before passing it for evaluation to the given parent environment.)
- `get_obj_value()` first looks for the object in the `n`-th parent generation that *led* to the `obj` object in the calling environment (i.e. the environment that calls `get_obj_value()` and only *then* evaluates it at the `n`-th parent generation.

The job performed by `get_obj_value()` is done as follows: at each parent generation, there is a pair of "object name" <-> "object value". The task of this function is to retrieve the object name at a given parent generation and then its value based on the "path" (of variable names) that leads to the variable in the function that calls `get_obj_value()`.

In practice though the result of `get_obj_value()` is the same as the value of the queried object at the calling function, since the value of the variables leading to that object are all the same through the calling stack. But using `get_obj_value()` can provide additional information if we set parameter `silent=FALSE`: in such case the function shows the name of the different variables that lead to the queried object in the calling function. An example is given in the Examples section.

The function can also be used to query the value of any object in a particular environment, i.e. not necessarily the value of an object *leading* to an object existing in the calling environment. This can be done somewhat with less writing than using `evalq()`.

If the `obj` is given as a string, it also evaluates to the object value when an object with that name exists in the given parent generation. However, the object should be passed with no explicit reference to the environment where it is defined. For instance we should use `with(env1,get_obj_value("z"))` and *not* `get_obj_value("env1$z")`, which returns simply `"env1$z"`.

Value

The value of the object in the `n`-th parent generation from the calling environment, as described in the Details section.

See Also

get_obj_name() which returns the *name* of the object in the calling stack leading to the queried object in the calling environment.

Examples

```
# Example of using get_obj_value() from within a function
# The value returned by get_obj_value() is compared to the values returned by eval() and evalq()
compareResultsOfDiferentEvaluations <- function(x) {
  cat("Looking at the path of variables leading to parameter 'x':\n")
  xval = get_obj_value(x, n=1, silent=FALSE)
  cat("Value of 'x' at parent generation 1 using get_obj_value():", xval, "\n")
  cat("Value of 'x' at parent generation 1 using eval():", eval(x, parent.frame(1)), "\n")
  cat("Value of 'x' at parent generation 1 using evalq():", evalq(x, parent.frame(1)), "\n")
}
g <- function(y) {
  x = 2
  compareResultsOfDiferentEvaluations(y)
}
z = 3
g(z)
   ## Note how the result of get_obj_value() is the same as eval() (=3)
   ## but not the same as evalq() (=2) because the queried object (x)
   ## exists in the queried parent generation (g()) with value 2.
   ## The results of eval() and get_obj_value() are the same but
   ## obtained in two different ways:
   ## - eval() returns the value of 'x' in the calling function (even though
   ## the evaluation environment is parent.frame(1), because eval() first
   ## evaluates the object in the calling environment)
   ## - get_obj_value() returns the value of 'y' in the parent generation
   ## of the calling function (which is the execution environment of g())
   ## since 'y' is the variable leading to variable 'x' in the calling function.
   ##
   ## NOTE however, that using get_obj_value() does NOT provide any new
   ## information to the result of eval(), since the variable values are
   ## transmitted UNTOUCHED through the different generations in the
   ## function calling chain.
   ## FURTHERMORE, the same value is returned by simply referencing 'x'
   ## so we don't need neither the use of get_obj_value() nor eval().
   ## The only interesting result would be provided by the evalq() call
   ## which looks for variable 'x' at the parent generation and evaluates it.

# Example of calling get_obj_value() from outside a function
x = 3
v = c(4, 2)
get_obj_value(x)         # 3
get_obj_value("x")       # 3
get_obj_value(3)         # 3
get_obj_value(v[1])      # 4
```

obj_find *Find an object in the workspace including user-defined environments*

Description

Look for an object in the whole workspace including all environments defined within it (possibly recursively) and return ALL the environment(s) where the object is found. User-defined environments are also searched. Note that both the "recursive search" and the "user-defined environments search" makes this function quite different from functions find and exists of the base package. Optionally, the search can be limited to a specified environment, as opposed to carrying it out in the whole workspace. Still, all user-defined environments defined inside the specified environment are searched.

Usage

```
obj_find(
  obj,
  envir = NULL,
  envmap = NULL,
  globalsearch = TRUE,
  n = 0,
  return_address = FALSE,
  include_functions = FALSE,
  silent = TRUE
)
```

Arguments

obj	object to be searched given as the object itself or as a character string. If given as an object, expressions are accepted (see details on how expressions are handled).
envir	environment where the search for obj should be carried out. It defaults to NULL which means obj is searched in the calling environment (i.e. in the environment calling this function), unless globalsearch=TRUE in which case it is searched in the whole workspace.
envmap	data frame containing a lookup table with name-address pairs of environment names and addresses to be used when searching for environment env. It defaults to NULL which means that the lookup table is constructed on the fly with the environments defined in the envir environment –if not NULL–, or in the whole workspace if envir=NULL. See the details section for more information on its structure.
globalsearch	when envir=NULL it specifies whether the search for obj should be done globally, i.e. in the whole workspace, or just within the calling environment.
n	non-negative integer indicating the number of levels to go up from the calling function environment to evaluate obj. It defaults to 0 which implies that obj is evaluated in the environment of the calling function (i.e. the function that calls obj_find()).

`return_address` whether to return the address of the environments where the object is found in addition to their names.

`include_functions`
whether to include funtion execution environments as environments where the object is searched for. Set this flag to `TRUE` with caution because there may be several functions where the same object is defined, for instance functions that are called as part of the object searching process!

`silent` run in silent mode? If not, the search history is shown, listing all the environments that are searched for object `obj`. It defaults to TRUE.

Details

An object is found in an environment if it is reachable from within that environment. An object is considered reachable in an environment if either one of the following occurs:

- it exists in the given environment
- it exists in a user-defined environment defined inside the given environment or in any environment recursively defined inside them

Note that `obj_find` differs from base functions `find` and `exists` in that `obj_find` searches for the object inside user-defined environments within any given environment in a *recursive* way.

In particular, compared to:

- `find`: `obj_find` searches for objects inside user-defined environments while `find` is not able to do so (see examples).
- `exists`: `obj_find` *never* searches for objects in the parent environment of `envir` when `envir` is not `NULL`, as is the case with the `exists` function when its `inherits` parameter is set to `TRUE` (the default). If it is wished to search for objects in parent environments, simply set `envir=NULL` and `globalsearch=TRUE`, in which case the object will be searched in the whole workspace and the environments where it is found will be returned.

When the object is found, an array containing the names of all the environments where the object is found is returned.

When `envir` is not `NULL` attached packages are not included in the search for `obj`, unless of course `envir` is itself a package environment.

When given as an object, `obj` can be an expression. If the expression is an attribute of a list or an array element, the object contained therein is searched for. Ex: if `alist$var = "x"` then object `x` is searched.

If `envmap` is passed it should be a data frame providing an address-name pair lookup table of environments and should contain at least the following columns:

- `location` for user-defined environments, the name of the environment where the environment is located; otherwise `NA`.
- `pathname` the full *environment path* to reach the environment separated by `$` (e.g. `"env1$env$envx"`)
- `address` the 8-digit (32-bit architectures) thru 16-digit (64-bit architectures) memory address of the environment given in `pathname` enclosed in < > (e.g. `"<0000000007DCFB38>"` (64-bit architectures)) Be ware that Linux Debian distributions may have a 12-digit memory address representation. So the best way to know is to check a memory address by calling e.g. 'address("x")'.

Passing an envmap lookup table is useful for speedup purposes, in case several calls to this function will be performed in the context of an unchanged set of defined environments. Such envmap data frame can be created by calling get_env_names. Use this parameter with care, as the matrix passed may not correspond to the actual mapping of existing environments to their addresses and in that case results may be different from those expected.

Value

The return value depends on the value of parameter return_address: when FALSE (the default) it returns an array containing the names of the environments where the object obj is found; when TRUE it returns a list with two attributes: "env_full_names" and "env_addresses" with respectively the environment names and addresses where the object is found.

Examples

```
# Define a variable in the global environment
x = 4
# Create new environments, some nested
env1 = new.env()
with(env1, envx <- new.env())
env1$x = 3
env1$envx$x = 2
env1$y = 5

# Look for objects (crawling environments recursively)
obj_find(x)                  # "env1" "env1$envx" "R_GlobalEnv"
obj_find("x")                # "env1" "env1$envx" "R_GlobalEnv"
obj_find("x", envir=env1)   # "env1" "envx" (as the search is limited to the env1 environment)
obj_find("y")                # "env1"
obj_find(nonexistent)      # NULL (note that NO error is raised even if the object does not exist)
```

---

testenv *Environment used in testing the package*

Description

Environment used in testing the package

Usage

```
testenv
```

Format

An object of class environment of length 1.

Index

∗ **datasets**
testenv, 24
∗ **environment**
envnames-package, 2
∗ **package**
envnames-package, 2

address, 5

collapse_root_and_member, 5

environment_name, *2*, 6
environmentName, *2, 3*, *6*
envnames *(*envnames-package*)*, 2
envnames-package, 2
exists, *3*, *22*

find, *3*, *22*

get_env_name *(*environment_name*)*, 6
get_env_names, *7*, 8, *16*, *24*
get_fun_calling, *2*, 10, *14*
get_fun_calling_chain, *2*, 11
get_fun_env, 12
get_fun_name, *11*, 13
get_obj_address, *2*, 14
get_obj_name, 17
get_obj_value, *18*, 19

new.env, *2*

obj_find, *2*, 22

sys.call, *2, 3*

testenv, 24