Package envnames helps navigate through user-defined and function execution environments and find objects in nested environments

Daniel Mastropietro (mastropi@uwalumni.com)

2018-07-20

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 pre-defined 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 can be seen by running the following code snippet:

```
env1 <- new.env()
cat("The name of the environment just defined is: ", environmentName(env1), "(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(env1)
```

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

```
## <environment: 0x000000000f3da610>
```

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, including function execution environments.

Why do we care about knowing the name of user-defined 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 the use of `ls()` as a rudimentary tool to identify the human-understandable environment (e.g. `env1`) referred by an environment given by its memory address (e.g. normally `<environment: 0x063dbc90>` in 32-bit systems or `<environment: 0x00000000063dbc90>` in 64-bit systems), as already seen above). For more information and examples, see the `get_env_names()` function 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. For more information and examples, see function `get_obj_value()`.

Apart from this core functionaliy, 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 *recursively* –i.e. in 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.call()`. This functionality is provided by the `get_fun_calling()` and `get_fun_calling_chain()` functions 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).

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

Description of selected functions

This section describes the functionality of 6 selected functions (sorted by relevance):

1) `get_env_names()`, 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), to get the name of user-defined and execution environments.

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

4) `get_fun_calling_chain()`, to get the function calling stack displayed in an easier format than `sys.calls()`.

5) `get_obj_address()` to get the memory address of an object by first looking for it (using `obj_find()`).

6) `address()`, to return the memory address of an object existing in the environment where the function is run.

The following 5 functions are not covered by the vignette at this point:

- `get_fun_calling()`, which returns the last function call in the calling chain returned by `get_fun_calling_chain()`.
- `get_fun_env()`, which returns the execution environment(s) of a function matching a name or a memory address.
- `get_fun_name()`, which returns the name of the function called at a given level of the function calling chain.
- `get_obj_name()`, which returns the name of the object at a specified parent generation leading to a function's parameter.
- `get_obj_value()`, which returns the value of the object at a specified parent generation leading to a function's parameter value.

The title of each sub-section is a sentence stating what can be accomplished with the presented function.

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

The `get_env_names()` function is used to retrieve the address-name lookup table that is used by the other functions in the package to perform their magic. Essentially this magic is obtained by looking up the memory address of an environment in this lookup table and returning the environment's name.

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

Examples

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`.

Basic operation

The following call returns a data frame containing the address-name lookup table:

```
get_env_names()
```

```
##              type      location    locationaddress             address
## 1            user   R_GlobalEnv               <NA> <000000000EDA52F0>
## 2            user   R_GlobalEnv               <NA> <000000000D35D060>
## 3            user   R_GlobalEnv               <NA> <000000000D2FD0C8>
## 4        function         tools <000000000B4AB078> <000000000B82B0A8>
## 5        function          base <0000000004520638> <000000000BB43788>
## 6        function      tryCatch <000000000BB43788> <000000000BB42F60>
## 7        function      tryCatch <000000000BB43788> <000000000BB42348>
## 8        function   tryCatchOne <000000000BB42348> <000000000BB42818>
## 9        function         knitr <000000000A7705E0> <000000000BB41DF8>
## 10       function         knitr <000000000A7705E0> <000000000BB360B8>
## 11       function     rmarkdown <000000000BAC7260> <000000000CED32C0>
## 12       function         knitr <000000000A7705E0> <000000000CEF0880>
## 13       function         knitr <000000000A7705E0> <000000000CF94C68>
## 14       function          base <0000000004520638> <000000000D2512B8>
## 15       function         knitr <000000000A7705E0> <000000000D2517F8>
## 16       function         knitr <000000000A7705E0> <000000000D250B38>
## 17       function         knitr <000000000A7705E0> <000000000D250C50>
## 18       function         knitr <000000000A7705E0> <000000000D2226A8>
## 19       function         knitr <000000000A7705E0> <000000000D1D36C0>
## 20       function         knitr <000000000A7705E0> <000000000D1D0FB8>
## 21       function      evaluate <000000000CBEA8B0> <000000000D1CB2E0>
## 22       function      evaluate <000000000CBEA8B0> <000000000D12F3F8>
## 23       function evaluate_call <000000000D12F3F8> <000000000D0E7E40>
## 24       function evaluate_call <000000000D12F3F8> <000000000D0E7F20>
## 25       function          base <0000000004520638> <000000000D0E7458>
## 26       function          base <0000000004520638> <000000000D0E6FE8>
```

```
## 27       function          base <0000000004520638> <000000000D0E7218>
## 28       function          base <0000000004520638> <00000000045206E0>
## 29 system/package          <NA>               <NA> <00000000045206E0>
## 30 system/package          <NA>               <NA> <000000000FB83488>
## 31 system/package          <NA>               <NA> <000000000AD82860>
## 32 system/package          <NA>               <NA> <000000000B8E5888>
## 33 system/package          <NA>               <NA> <000000000B704D20>
## 34 system/package          <NA>               <NA> <000000000B4FE4F8>
## 35 system/package          <NA>               <NA> <000000000B0415B8>
## 36 system/package          <NA>               <NA> <000000000B3C1F08>
## 37 system/package          <NA>               <NA> <000000000A684C18>
## 38 system/package          <NA>               <NA> <00000000044F0718>
## 39     namespace          <NA>               <NA> <000000000FB34C20>
## 40     namespace          <NA>               <NA> <000000000A9311A0>
## 41     namespace          <NA>               <NA> <000000000B78D328>
## 42     namespace          <NA>               <NA> <000000000B58D9F0>
## 43     namespace          <NA>               <NA> <000000000AEB6078>
## 44     namespace          <NA>               <NA> <000000000B0DC420>
## 45     namespace          <NA>               <NA> <000000000AA3A688>
## 46     namespace          <NA>               <NA> <0000000004520638>
##                pathname        path                  name
## 1                  env1                              env1
## 2           env_of_envs                       env_of_envs
## 3     env_of_envs$env21 env_of_envs                 env21
## 4  tools::buildVignettes            tools::buildVignettes
## 5              tryCatch                          tryCatch
## 6          tryCatchList                      tryCatchList
## 7           tryCatchOne                       tryCatchOne
## 8            doTryCatch                        doTryCatch
## 9          engine$weave      engine                 weave
## 10     vweave_rmarkdown                  vweave_rmarkdown
## 11    rmarkdown::render                 rmarkdown::render
## 12          knitr::knit                       knitr::knit
## 13         process_file                      process_file
## 14  withCallingHandlers               withCallingHandlers
## 15        process_group                     process_group
## 16  process_group.block               process_group.block
## 17           call_block                        call_block
## 18           block_exec                        block_exec
## 19               in_dir                            in_dir
## 20            evaluate                          evaluate
## 21   evaluate::evaluate               evaluate::evaluate
## 22        evaluate_call                     evaluate_call
## 23            timing_fn                         timing_fn
## 24               handle                            handle
## 25  withCallingHandlers               withCallingHandlers
## 26          withVisible                       withVisible
## 27                eval                              eval
## 28                eval                              eval
## 29           .GlobalEnv                        .GlobalEnv
## 30     package:envnames                  package:envnames
## 31        package:stats                     package:stats
## 32     package:graphics                  package:graphics
## 33    package:grDevices                 package:grDevices
```

```
## 34         package:utils                    package:utils
## 35       package:datasets                 package:datasets
## 36        package:methods                  package:methods
## 37              Autoloads                        Autoloads
## 38           package:base                     package:base
## 39       package:envnames                 package:envnames
## 40          package:stats                    package:stats
## 41       package:graphics                 package:graphics
## 42      package:grDevices                package:grDevices
## 43          package:utils                    package:utils
## 44       package:datasets                 package:datasets
## 45        package:methods                  package:methods
## 46           package:base                     package:base
```

The main columns defining the lookup are `address` (containing the memory address of the environment) and `name` (containing the name of the environment).

We can also restrict the lookup table to the environments defined within another environment:

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

```
##        type      location    locationaddress            address
## 1      user   env_of_envs               <NA> <000000000D2FD0C8>
## 2  function         tools <000000000B4AB078> <000000000B82B0A8>
## 3  function          base <0000000004520638> <000000000BB43788>
## 4  function      tryCatch <000000000BB43788> <000000000BB42F60>
## 5  function      tryCatch <000000000BB43788> <000000000BB42348>
## 6  function   tryCatchOne <000000000BB42348> <000000000BB42818>
## 7  function         knitr <000000000A7705E0> <000000000BB41DF8>
## 8  function         knitr <000000000A7705E0> <000000000BB360B8>
## 9  function     rmarkdown <000000000BAC7260> <000000000CED32C0>
## 10 function         knitr <000000000A7705E0> <000000000CEF0880>
## 11 function         knitr <000000000A7705E0> <000000000CF94C68>
## 12 function          base <0000000004520638> <000000000F666718>
## 13 function         knitr <000000000A7705E0> <000000000F666C58>
## 14 function         knitr <000000000A7705E0> <000000000F665F98>
## 15 function         knitr <000000000A7705E0> <000000000F6660B0>
## 16 function         knitr <000000000A7705E0> <000000000D175D88>
## 17 function         knitr <000000000A7705E0> <000000000CB7C428>
## 18 function         knitr <000000000A7705E0> <000000000CB356E8>
## 19 function      evaluate <000000000CBEA8B0> <000000000CB1DF18>
## 20 function      evaluate <000000000CBEA8B0> <000000000BE977E0>
## 21 function evaluate_call <000000000BE977E0> <000000000BB5FAE0>
## 22 function evaluate_call <000000000BE977E0> <000000000BB5FBC0>
## 23 function          base <0000000004520638> <000000000BB5F018>
## 24 function          base <0000000004520638> <000000000BB5D8B8>
## 25 function          base <0000000004520638> <000000000BB5DB20>
## 26 function          base <0000000004520638> <00000000045206E0>
##                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) `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 for the use of `environment_name()` is when some function tells us that an object is part of a user-defined environment, but this environment is only given as a *memory address* (as in e.g. normally `<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.

The address-to-name conversion also works for *function execution environments*.

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

Examples

Basic operation

Let's retrieve the names of the environments just defined using the `environment_name()` function. This may sound trivial because we are already typing the environment name! However, look at the output from the second call: it contains the *path* to the environment being searched for stating that `env21` is found inside environment `env_of_envs`.

```
environment_name(env1)
```

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

```
environment_name(env21)
```

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

If we already know in which environment the environment we are after is defined, we can specify it in the `envir` parameter and get their name without any path information:

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

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

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

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

More advanced examples

Suppose now that we define another environment that points to one of the already defined environments.

```
e <- env_of_envs$env21
```

Now, let's retrieve its name:

```
environment_name(e)
```

```
## R_GlobalEnv env_of_envs
##         "e"     "env21"
```

What we get is the names of *all* the environments (in alphabetical order) that point to the same memory address. We can disable such behaviour of matching environments 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, matchname=TRUE)
```

```
## [1] "e"
```

Note that the result of this last call could actually return *more than one environment* when environments sharing the same name (`e` in the above example) are defined in different environments *all pointing to the same environment*. In the above example we could have this situation if we defined an environment called `"e"` in environment `env_of_envs` pointing to environment `e` defined in the global environment (with the command `env_of_envs$e <- e`).

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

```
environment_name(non_existing_env)
```

```
## NULL
```

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 as argument, as shown in the following example:

```
env1_obj_address = get_obj_address(env1)
environment_name(env1_obj_address)
```

```
## [1] "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 latter format is particularly useful when copying & pasting the result of querying an environment object, for example when typing `env1` at the `R` command prompt, assuming that `env1` is a user-defined environment.

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
```

Retrieving the execution environment of a function

If called inside a function, `environment_name()` returns the name of the function and the environment where the function is defined. The following example shows this use of `environment_name()`.

```
with(env1,
  f <- function() {
    cat("1) We are inside function", environment_name(), "\n")
  }
)
env1$f()
```

```
## 1) We are inside function env1$f
```

3) `obj_find()`: check the existence of an object

With the `obj_find()` function we can check if an object exists in any existing environment. All environments –including system environments, packages, user-defined environments, and optionally function execution environments– are crawled to search for the object. This includes any environments that are defined *within* other environments (*nested*), and it represents an enhancement to the built-in `exists()` function. 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)`

Examples

Let's start with a few object definitions

We define a couple of objects in the environments previously defined:

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

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"                 "env_of_envs$env21"
```

(if your are seeing more environments than you expected in the above output, recall that environment `e` points 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"                 "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()`:

```
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"
##
## $y
## [1] "e"                 "env_of_envs$env21"
##
## $z
## NULL
```

Note how calling `obj_find()` from within the `env1` environment still searches for the objects everywhere. This is because parameter `globalsearch` is set to `TRUE`. If we set it to `FALSE` and we add `envir=env1` as searching environment, we would get:

```
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 are running `sapply()` inside environment `env1`, it's important to add parameter `envir=env1` to the call to `obj_find()`; if we don't add it, the objects are *not* found because the parent environmnt of the call to `obj_find()` 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:"); print(environments_where_obj_x_is_found)
```

```
## Object 'x' found
## in the following environments:
## [1] "R_GlobalEnv" "env1"
```

Finally, we can also search for objects defined in packages:

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

```
## Object 'aov' found
## in the following environments:
## [1] "package:stats"
```

4) `get_fun_calling_chain()`: retrieve the function calling chain (stack)

Following is an example that shows the difference between using the built-in `sys.call()` function and the package function `get_fun_calling_chain()` to retrieve the calling stack.

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.

The signature of the function is the following:
`get_fun_calling_chain(n = NULL, showParameters = FALSE, silent = TRUE)`

Examples

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` does something different 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))
  }
)
```

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 -> $doTryCatch -> eng
##
## When h(x) is called by env11$f(x=0) the output is: 1
```

- Output from `env12$f()`:

```
## Function calling chain:
##  tools$tools::buildVignettes -> base$tryCatch -> $tryCatchList -> $tryCatchOne -> $doTryCatch -> eng
##
## When h(x) is called by env12$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.

5 & 6) `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)`

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                 env1
## "<000000001112D330>" "<00000000134102D8>"
```

```
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                 env1
## "<000000001112D330>" "<00000000134102D8>"
```

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 address 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)
```

```
##                 env1
## "<00000000134102D8>"
```

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

```
## [1] "<00000000134102D8>"
```

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

```
## [1] "<00000000134102D8>"
```

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)
```

```
## [1] "<00000000134102D8>"
```

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

```
## $x
##          R_GlobalEnv                 env1
## "<000000001112D330>" "<00000000134102D8>"
##
## $y
##                     e    env_of_envs$env21
## "<0000000013410278>" "<0000000013410278>"
##
## $nonexistent
## NULL
```

(if your are seeing more environments than you expected in the above output, recall that environment `e` 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] "<00000000134102D8>"
```

```
address(e$y)
```

```
## [1] "<0000000013410278>"
```

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 environmnt 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] "<00000000044F0788>"
```

```
address(NULL)
```

```
## [1] "<00000000044F0788>"
```

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

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

```
## NULL
```

Summing up

We have described the following 6 functions of the `envnames` package and shown examples of using them:

1) `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) `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) `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
   - searching for the object recursively (i.e. in environments defined inside other environments)

4) `get_fun_calling_chain()` to get the stack of calling functions. This function returns 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.

5) `get_obj_address()` to retrieve the memory address of an object. This function gives 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()` in this package.

6) `address()` to retrieve the memory address of an existing object, but without looking for it first.

We have *not* described the other 5 functions in the package:

- `get_fun_calling()`, which returns the last function call in the calling chain returned by `get_fun_calling_chain()`.
- `get_fun_env()`, which returns the execution environment(s) of a function matching a name or a memory address.
- `get_fun_name()`, which returns the name of the function called at a given level of the function calling chain.
- `get_obj_name()`, which returns the name of the object at a specified parent generation leading to a function's parameter.
- `get_obj_value()`, which returns the value of the object at a specified parent generation leading to a function's parameter value.

This vignette was generated under the following platform:

```
##          SystemInfo
## sysname     Windows
## release      10 x64
## version build 17134
## machine      x86-64
```

```
##                 _
## platform        x86_64-w64-mingw32
## arch            x86_64
```

```
## os             mingw32
## system         x86_64, mingw32
## status
## major          3
## minor          4.1
## year           2017
## month          06
## day            30
## svn rev        72865
## language       R
## version.string R version 3.4.1 (2017-06-30)
## nickname       Single Candle
```