7_Environments

2022-10-19

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.8
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(rlang)
```

```
##
## Attaching package: 'rlang'

## The following objects are masked from 'package:purrr':
##
##     %@%, as_function, flatten, flatten_chr, flatten_dbl, flatten_int,
##     flatten_lgl, flatten_raw, invoke, splice
```

7 Environments

7.1 Introduction

Quiz

1. List at least three ways that an environment differs from a list.
2. What is the parent of the global environment? What is the only environment that doesn't have a parent?
3. What is the enclosing environment of a function? Why is it important?
4. How do you determine the environment from which a function was called?
5. How are <- and «- different?

7.2 Environment basics

7.2.1 Basics

```
e1 <- env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3,
)
```

```
e1$d <- e1
```

```
e1
```

```
## <environment: 0x0000000024033318>
```

```
env_print(e1)
```

```
## <environment: 0x0000000024033318>
## Parent: <environment: global>
## Bindings:
## * a: <lgl>
## * b: <chr>
## * c: <dbl>
## * d: <env>
```

```
env_names(e1)
```

```
## [1] "a" "b" "c" "d"
```

7.2.2 Important environments

```
identical(global_env(), current_env())
```

```
## [1] TRUE
```

```
global_env() == current_env()
```

```
## Error in global_env() == current_env(): comparison (1) is possible only for atomic and list types
```

7.2.3 Parents

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```

```
env_parent(e2b)
```

```
## <environment: 0x00000000250fd2f0>
```

```
env_parent(e2a)
```

```
## <environment: R_GlobalEnv>
```

```
e2c <- env(empty_env(), d = 4, e = 5)
e2d <- env(e2c, a = 1, b = 2, c = 3)
```

```
env_parents(e2b)
```

```
## [[1]]   <env: 0x00000000250fd2f0>
## [[2]] $ <env: global>
```

```
env_parents(e2d)
```

```
## [[1]]   <env: 0x0000000021247560>
## [[2]] $ <env: empty>
```

```
env_parents(e2b, last = empty_env())
```

```
##  [[1]]   <env: 0x00000000250fd2f0>
##  [[2]] $ <env: global>
##  [[3]] $ <env: package:rlang>
##  [[4]] $ <env: package:forcats>
##  [[5]] $ <env: package:stringr>
##  [[6]] $ <env: package:dplyr>
##  [[7]] $ <env: package:purrr>
##  [[8]] $ <env: package:readr>
##  [[9]] $ <env: package:tidyr>
## [[10]] $ <env: package:tibble>
## [[11]] $ <env: package:ggplot2>
## [[12]] $ <env: package:tidyverse>
## [[13]] $ <env: package:stats>
## [[14]] $ <env: package:graphics>
## [[15]] $ <env: package:grDevices>
## [[16]] $ <env: package:utils>
## [[17]] $ <env: package:datasets>
## [[18]] $ <env: package:methods>
## [[19]] $ <env: Autoloads>
## [[20]] $ <env: package:base>
## ... and 1 more environments
```

Use `parent.env()` to find the parent of an environment. No base function returns all ancestors.

7.2.4 Super assignment, <<-

```
x <- 0
f <- function() {
  x <<- 1
}
x
```

```
## [1] 0
```

```
f()
x
```

```
## [1] 1
```

7.2.5 Getting and setting

```
e3 <- env(x = 1, y = 2)
e3$x
```

```
## [1] 1
```

```
e3$z <- 3
e3[["z"]]
```

```
## [1] 3
```

```
e3[[1]]
```

```
## Error in e3[[1]]: wrong arguments for subsetting an environment
```

```
e3[c("x", "y")]
```

```
## Error in e3[c("x", "y")]: object of type 'environment' is not subsettable
```

```
e3$xyz
```

```
## NULL
```

```
env_get(e3, "xyz")
```

```
## Error in `env_get()`:
## ! Can't find `xyz` in environment.
```

```
env_get(e3, "xyz", default = NA)
```

```
## [1] NA
```

```
env_poke(e3, "a", 100)
e3$a
```

```
## [1] 100
```

```
env_bind(e3, a = 10, b = 20)
env_names(e3)
```

```
## [1] "x" "y" "z" "a" "b"
```

```
env_has(e3, "a")
```

```
##    a
## TRUE
```

```
e3$a <- NULL
env_has(e3, "a")
```

```
##    a
## TRUE
```

```
env_unbind(e3, "a")
env_has(e3, "a")
```

```
##     a
## FALSE
```

7.2.6 Advanced bindings

```
env_bind_lazy(current_env(), b = {Sys.sleep(1); 1})

system.time(print(b))
```

```
## [1] 1
```

```
##    user  system elapsed
##    0.00    0.00    1.02
```

```
system.time(print(b))
```

```
## [1] 1
```

```
##    user  system elapsed
##       0       0       0
```

```
env_bind_active(current_env(), z1 = function(val) runif(1))
z1
```

```
## [1] 0.7419056
```

```
z1
```

```
## [1] 0.03174555
```

7.2.7 Exercises

1. List three ways in which an environment differs from a list.

There's no order to the objects, names must be unique, they have parents, are not copied when modified

2. Create an environment as illustrated by this picture.

```
e5 <- env()
e5$loop <- e5
env_print(e5)
```

```
## <environment: 0x0000000023522698>
## Parent: <environment: global>
## Bindings:
## * loop: <env>
```

3. Create a pair of environments as illustrated by this picture.

```
ea <- env()
eb <- env()
ea$loop <- eb
eb$dedoop <- ea

env_print(ea)
```

```
## <environment: 0x00000000228b0a48>
## Parent: <environment: global>
## Bindings:
## * loop: <env>
```

```
env_print(eb)
```

```
## <environment: 0x00000000226d43f0>
## Parent: <environment: global>
## Bindings:
## * dedoop: <env>
```

4. Explain why `e[[1]]` and `e[c("a", "b")]` don't make sense when e is an environment.

for `e[[1]]` Environments have no set order so you can't index by number. `e[c("a", "b")]` would try to return two objects, but they objects can be anything which makes returning a vector of them both hard.

5. Create a version of `env_poke()` that will only bind new names, never re-bind old names. Some programming languages only do this, and are known as single assignment languages.

```
env_pokey <- function(e, name, value){
  if(!env_has(e, name)){
    env_poke(e, name, value)
  } else cat("name already exists \n")
}

aaa <- env()
aaa$a <- 21
env_print(aaa)
```

```
## <environment: 0x0000000013933b50>
## Parent: <environment: global>
## Bindings:
## * a: <dbl>
```

```
env_pokey(aaa, "a", "abcde")
```

```
## name already exists
```

```
env_print(aaa)
```

```
## <environment: 0x0000000013933b50>
## Parent: <environment: global>
## Bindings:
## * a: <dbl>
```

```
env_pokey(aaa, "b", "abcde")
env_print(aaa)
```

```
## <environment: 0x0000000013933b50>
## Parent: <environment: global>
## Bindings:
## * a: <dbl>
## * b: <chr>
```

6. What does this function do? How does it differ from «- and why might you prefer it?

```
rebind <- function(name, value, env = caller_env()) {
  if (identical(env, empty_env())) {
    stop("Can't find `", name, "`", call. = FALSE)
  } else if (env_has(env, name)) {
    env_poke(env, name, value)
  } else {
    rebind(name, value, env_parent(env))
  }
}
rebind("a", 10)
```

```
## Error: Can't find `a`
```

```
a <- 5
rebind("a", 10)
a
```

```
## [1] 10
```

It recursively looks for an object by name. It starts in the current environment. If it finds the object it rebinds it to the new value. If it does not, it then calls itself again using the parent of current iteration's environment. If it goes all the way to the empty environment then it prints an error saying the object can't be found.

7.3 Recursing over environments

```
where <- function(name, env = caller_env()) {
  if (identical(env, empty_env())) {
    # Base case
    stop("Can't find ", name, call. = FALSE)
  } else if (env_has(env, name)) {
    # Success case
    env
  } else {
    # Recursive case
    where(name, env_parent(env))
  }
}
```

```
where("yyy")
```

```
## Error: Can't find yyy
```

```
x <- 5
where("x")
```

```
## <environment: R_GlobalEnv>
```

```
where("mean")
```

```
## <environment: base>
```

```
e4a <- env(empty_env(), a = 1, b = 2)
e4b <- env(e4a, x = 10, a = 11)
```

```
where("a", e4b)
```

```
## <environment: 0x0000000023550238>
```

```
where("b", e4b)
```

```
## <environment: 0x0000000023af7a48>
```

```
where("c", e4b)
```

```
## Error: Can't find c
```

```
f <- function(..., env = caller_env()) {
  if (identical(env, empty_env())) {
    # base case
  } else if (success) {
    # success case
  } else {
    # recursive case
    f(..., env = env_parent(env))
  }
}
```

```
f2 <- function(..., env = caller_env()) {
  while (!identical(env, empty_env())) {
    if (success) {
      # success case
      return()
    }
    # inspect parent
    env <- env_parent(env)
  }

  # base case
}
```

7.3.1 Exercises

1. Modify where() to return all environments that contain a binding for name. Carefully think through what type of object the function will need to return.

```
where <- function(name, env = caller_env(), hits = list()) { # Need to carry list over each iteration
  if (identical(env, empty_env()) & length(hits) == 0) {
    # Base case
    stop("Can't find ", name, call. = FALSE)
  } else if (env_has(env, name)) {
    # Success case
    hits <- c(hits, env)
    if(!identical(env_parent(env), empty_env())){
      where(name, env_parent(env), hits)
      } else {
          hits
        }
  } else {
    # Recursive case
```

```
    where(name, env_parent(env), hits)
  }
}

where("a", e4b)
```

```
## [[1]]
## <environment: 0x0000000023550238>
##
## [[2]]
## <environment: 0x0000000023af7a48>
```

```
where("a", e4a)
```

```
## [[1]]
## <environment: 0x0000000023af7a48>
```

2. Write a function called fget() that finds only function objects. It should have two arguments, name and env, and should obey the regular scoping rules for functions: if there's an object with a matching name that's not a function, look in the parent. For an added challenge, also add an inherits argument which controls whether the function recurses up the parents or only looks in one environment.

```
fget <- function(name, env, inherits = T) {
  if (identical(env, empty_env())) {
    # Base case
    stop(paste0("Can't find ", name, " in any environment"), call. = FALSE)
  } else if (env_has(env, name) && is.function(env_get(env, name))) {
    env
  } else if (inherits) {
    fget(name, env_parent(env))
  } else {
    stop(paste0("Can't find ", name, " in ", deparse(substitute(env))), call. = FALSE)
  }
}

fget(name = "mean", env = caller_env())
```

```
## <environment: namespace:base>
```

```
fget(name = "sdaafasf", env = caller_env(), inherits = T)
```

```
## Error: Can't find sdaafasf in any environment
```

```
fget(name = "fget", env = caller_env())
```

```
## <environment: R_GlobalEnv>
```

```
fget(name = "mean", env = caller_env(), inherits = F)
```

```
## Error: Can't find mean in caller_env()
```

7.4 Special environments

7.4.1 Package environments and the search path

The immediate parent of the global environment is the last package you attached43, the parent of that package is the second to last package you attached, ...

```
search()
```

```
##  [1] ".GlobalEnv"        "package:rlang"     "package:forcats"
##  [4] "package:stringr"   "package:dplyr"     "package:purrr"
##  [7] "package:readr"     "package:tidyr"     "package:tibble"
## [10] "package:ggplot2"   "package:tidyverse" "package:stats"
## [13] "package:graphics"  "package:grDevices" "package:utils"
## [16] "package:datasets"  "package:methods"   "Autoloads"
## [19] "org:r-lib"         "package:base"
```

```
search_envs()
```

```
##  [[1]] $ <env: global>
##  [[2]] $ <env: package:rlang>
##  [[3]] $ <env: package:forcats>
##  [[4]] $ <env: package:stringr>
##  [[5]] $ <env: package:dplyr>
##  [[6]] $ <env: package:purrr>
##  [[7]] $ <env: package:readr>
##  [[8]] $ <env: package:tidyr>
##  [[9]] $ <env: package:tibble>
## [[10]] $ <env: package:ggplot2>
## [[11]] $ <env: package:tidyverse>
## [[12]] $ <env: package:stats>
## [[13]] $ <env: package:graphics>
## [[14]] $ <env: package:grDevices>
## [[15]] $ <env: package:utils>
## [[16]] $ <env: package:datasets>
## [[17]] $ <env: package:methods>
## [[18]] $ <env: Autoloads>
## [[19]] $ <env: org:r-lib>
## [[20]] $ <env: package:base>
```

7.4.2 The function environment

```
y <- 1
f <- function(x) x + y
fn_env(f)
```

```
## <environment: R_GlobalEnv>
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

```
e <- env()
e$g <- function() 1
```

7.4.3 Namespaces

```
sd
```

```
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x000000001354e9a8>
## <environment: namespace:stats>
```

- The package environment is the external interface to the package. It's how you, the R user, find a function in an attached package or with `::`. Its parent is determined by search path, i.e. the order in which packages have been attached.
- The namespace environment is the internal interface to the package. The package environment controls how we find the function; the namespace controls how the function finds its variables.
- Each namespace has an imports environment that contains bindings to all the functions used by the package. The imports environment is controlled by the package developer with the NAMESPACE file.
- Explicitly importing every base function would be tiresome, so the parent of the imports environment is the base namespace. The base namespace contains the same bindings as the base environment, but it has a different parent.
- The parent of the base namespace is the global environment. This means that if a binding isn't defined in the imports environment the package will look for it in the usual way. This is usually a bad idea (because it makes code depend on other loaded packages), so R CMD check automatically warns about such code. It is needed primarily for historical reasons, particularly due to how S3 method dispatch works.

7.4.4 Execution environments

```
g <- function(x) {
  if (!env_has(current_env(), "a")) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
g(10)
```

```
## Defining a
```

```
## [1] 1
```

```
g(10)
```

```
## Defining a

## [1] 1
```

```
h <- function(x) {
  # 1.
  a <- 2 # 2.
  x + a
}
y <- h(1) #3
y
```

```
## [1] 3
```

```
h2 <- function(x) {
  a <- x * 2
  current_env()
}

e <- h2(x = 10)
env_print(e)
```

```
## <environment: 0x0000000022b0c3c8>
## Parent: <environment: global>
## Bindings:
## * a: <dbl>
## * x: <dbl>
```

```
fn_env(h2)
```

```
## <environment: R_GlobalEnv>
```

```
plus <- function(x) {
  function(y) x + y
}

plus_one <- plus(1)
plus_one
```

```
## function(y) x + y
## <environment: 0x0000000023c913d0>
```

```
plus_one(5) # 6
```

```
## [1] 6
```

7.4.5 Exercises

1. How is `search_envs()` different from `env_parents(global_env())`?

```
search_envs()
```

```
##  [[1]] $ <env: global>
##  [[2]] $ <env: package:rlang>
##  [[3]] $ <env: package:forcats>
##  [[4]] $ <env: package:stringr>
##  [[5]] $ <env: package:dplyr>
##  [[6]] $ <env: package:purrr>
##  [[7]] $ <env: package:readr>
##  [[8]] $ <env: package:tidyr>
##  [[9]] $ <env: package:tibble>
## [[10]] $ <env: package:ggplot2>
## [[11]] $ <env: package:tidyverse>
## [[12]] $ <env: package:stats>
## [[13]] $ <env: package:graphics>
## [[14]] $ <env: package:grDevices>
## [[15]] $ <env: package:utils>
## [[16]] $ <env: package:datasets>
## [[17]] $ <env: package:methods>
## [[18]] $ <env: Autoloads>
## [[19]] $ <env: org:r-lib>
## [[20]] $ <env: package:base>
```

```
env_parents(global_env())
```

```
##  [[1]] $ <env: package:rlang>
##  [[2]] $ <env: package:forcats>
##  [[3]] $ <env: package:stringr>
##  [[4]] $ <env: package:dplyr>
##  [[5]] $ <env: package:purrr>
##  [[6]] $ <env: package:readr>
##  [[7]] $ <env: package:tidyr>
##  [[8]] $ <env: package:tibble>
##  [[9]] $ <env: package:ggplot2>
## [[10]] $ <env: package:tidyverse>
## [[11]] $ <env: package:stats>
## [[12]] $ <env: package:graphics>
## [[13]] $ <env: package:grDevices>
## [[14]] $ <env: package:utils>
## [[15]] $ <env: package:datasets>
## [[16]] $ <env: package:methods>
## [[17]] $ <env: Autoloads>
## [[18]] $ <env: org:r-lib>
## [[19]] $ <env: package:base>
## [[20]] $ <env: empty>
```

One lists all the paths available including global but not empty. The other lists all the parents from GlobalEnv going all the way to empty, excluding global itself.

2. Draw a diagram that shows the enclosing environments of this function

```
f1 <- function(x1) {
  f2 <- function(x2) {
    f3 <- function(x3) {
      x1 + x2 + x3
    }
    f3(3)
  }
  f2(2)
}
f1(1)
```

```
## [1] 6
```

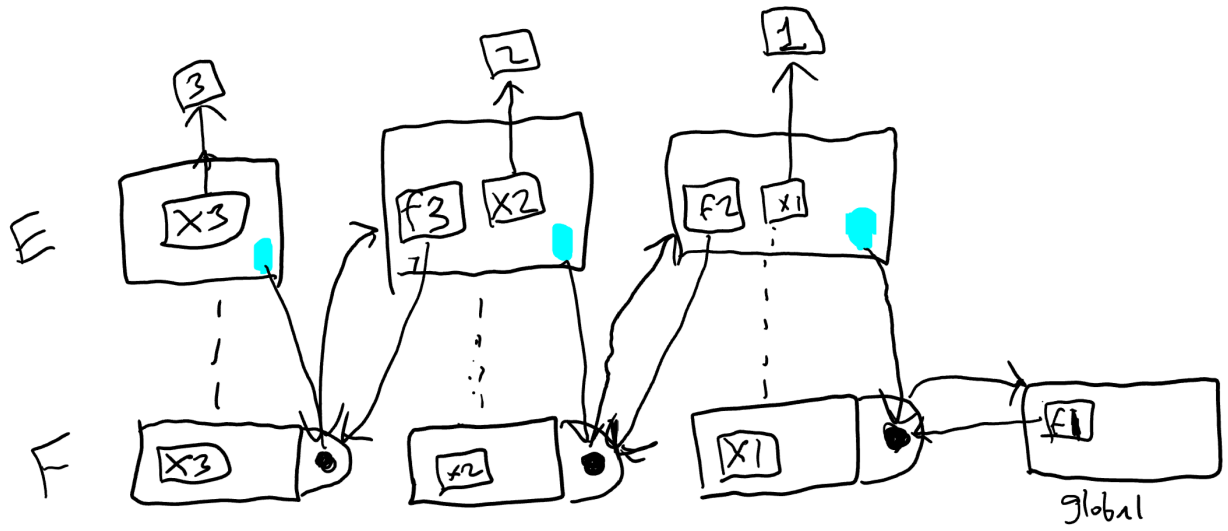

Figure 1: Functions Network

3. Write an enhanced version of str() that provides more information about functions. Show where the function was found and what environment it was defined in.

```
fget <- function(name, env, inherits = T) {
  if (identical(env, empty_env())) {
    # Base case
    stop(paste0("Can't find ", name, " in any environment"), call. = FALSE)
  } else if (env_has(env, name) && is.function(env_get(env, name))) {
    list(func = env_get(env, name), env = env)
  } else if (inherits) {
    fget(name, env_parent(env))
  } else {
    stop(paste0("Can't find ", name, " in ", deparse(substitute(env))), call. = FALSE)
  }
}

fget("mean", current_env())
```

```
## $func
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000016a629b8>
## <environment: namespace:base>
##
## $env
## <environment: base>
```

```
fget("env_pokey", current_env())
```

```
## $func
## function(e, name, value){
##   if(!env_has(e, name)){
##     env_poke(e, name, value)
##   } else cat("name already exists \n")
## }
## <bytecode: 0x0000000012ad8888>
##
## $env
## <environment: R_GlobalEnv>
```

7.5 Call stacks

7.5.1 Simple call stacks

```
f <- function(x) {
  g(x = 2)
}
g <- function(x) {
  h(x = 3)
}
h <- function(x) {
  stop()
}
```

```
f(x = 1)
```

```
## Error in h(x = 3):
```

```
traceback()
```

```
## No traceback available
```

```
h <- function(x) {
  lobstr::cst()
}
f(x = 1)
```

```
##     x
##  1. \-global f(x = 1)
##  2.   \-global g(x = 2)
##  3.     \-global h(x = 3)
##  4.       \-lobstr::cst()
```

7.5.2 Lazy evaluation

```
a <- function(x) b(x)
b <- function(x) c(x)
c <- function(x) x

a(f())
```

```
##     x
##  1. +-global a(f())
##  2. | \-global b(x)
##  3. |   \-global c(x)
##  4. \-global f()
##  5.   \-global g(x = 2)
##  6.     \-global h(x = 3)
##  7.       \-lobstr::cst()
```

7.5.3 Frames

7.5.4 Dynamic scope

7.5.5 Exercises

1. Write a function that lists all the variables defined in the environment in which it was called. It should return the same results as ls()

```
lser <- function(env = caller_env()){
  env_names(env)
}

aaa <- env(
  a = 22,
  b = 12,
  c = "asdad"
)
ls(aaa)
```

```
## [1] "a" "b" "c"
```

```
lser(aaa)
```

```
## [1] "a" "b" "c"
```

```
ls(all.names = T)
```

```
##  [1] ".Random.seed" "a"            "aaa"          "b"            "c"
##  [6] "e"            "e1"           "e2a"          "e2b"          "e2c"
## [11] "e2d"          "e3"           "e4a"          "e4b"          "e5"
## [16] "ea"           "eb"           "env_pokey"    "f"            "f1"
## [21] "f2"           "fget"         "g"            "h"            "h2"
## [26] "lser"         "plus"         "plus_one"     "rebind"       "where"
## [31] "x"            "y"            "z1"
```

```
lser()
```

```
##  [1] "f"            "g"            "where"        "e3"           "h"
##  [6] "e5"           "env_pokey"    "plus"         "ea"           "h2"
## [11] "eb"           "f1"           "fget"         "f2"           "rebind"
## [16] "x"            "y"            "lser"         ".Random.seed" "e4a"
## [21] "e4b"          "e2a"          "a"            "aaa"          "z1"
## [26] "e2b"          "b"            "c"            "e2c"          "e2d"
## [31] "plus_one"     "e"            "e1"
```

7.6 As data structures

```
my_env <- new.env(parent = emptyenv())
my_env$a <- 1

get_a <- function() {
  my_env$a
}
set_a <- function(value) {
  old <- my_env$a
  my_env$a <- value
  invisible(old)
}
```