Video 14: Environments

Stats 102A

Miles Chen, PhD

Environments

Related Reading

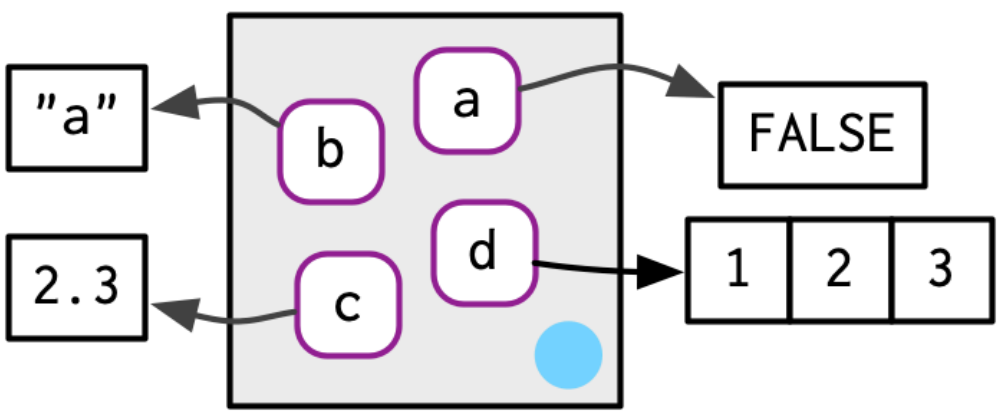
https://adv-r.hadley.nz/environments.html

Environments

The **environment** is the data structure that powers scoping.

The job of an environment is to associate, or **bind**, a set of names to a set of values. You can think of an environment as a bag of names, each name pointing to an object stored elsewhere in memory.

```
1  e <- new.env()
2  e$a <- FALSE
3  e$b <- "a"
4  e$c <- 2.3
5  e$d <- 1:3
```



Environment Objects

As with everything in R, environments are also objects. The syntax to work with environment objects is similar to lists in that the double bracket `[[` and `$` operator can be used to get and set values.

```
1 typeof(e)
```

```
[1] "environment"
```

```
1 e$c
```

```
[1] 2.3
```

```
1 e[["d"]]
```

```
[1] 1 2 3
```

```
1 e[[1]] # Objects in environments are not ordered. Numeric index prc
```

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

Environment Objects

The `ls()` and `ls.str()` functions are useful for looking at the objects inside or the structure of the environment.

```
1 ls(e)
```

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

```
1 ls.str(e)
```

```
a :  logi FALSE
b :  chr "a"
c :  num 2.3
d :  int [1:3] 1 2 3
```

Environment Objects

Unlike lists:

- The single square bracket `[` does not work for environments.
- Setting an object to `NULL` does not remove the object.
- Removing objects can be done using the `rm()` function.

```
e$d <- NULL
ls(e)
```

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

```
rm(d, envir = e)
ls(e)
```

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

Parent Environments

Every environment has a **parent**, another environment. In the diagrams, the pointer to the parent is a small blue circle.

The parent is used to implement **lexical scoping**: If a name is not found in an environment, then R will look in its parent (and so on).

Only one environment does not have a parent: the empty environment.

The Important Environments

There are four special environments:

The Search Path

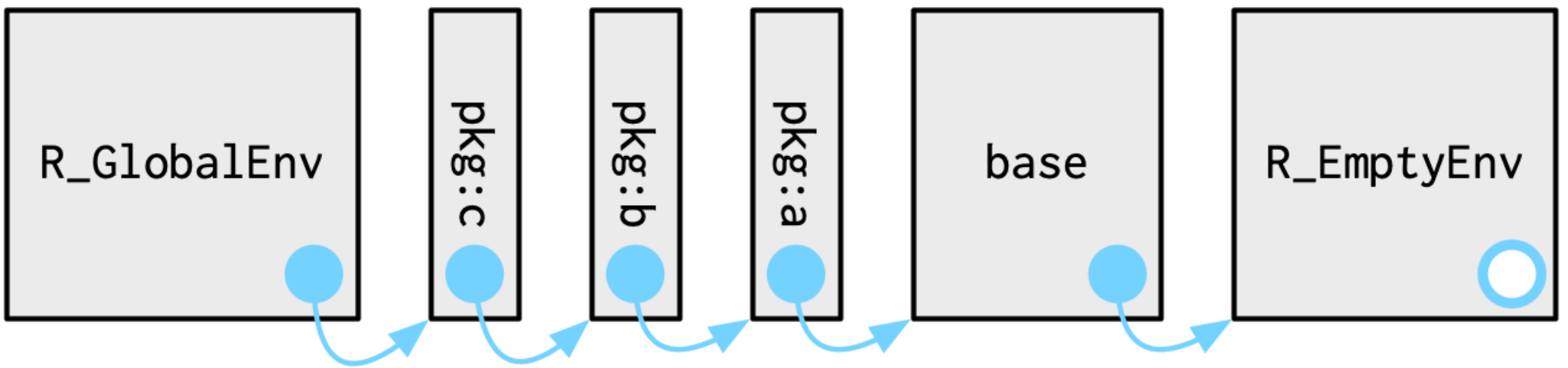
The `search()` function lists all parents of the global environment. This is called the **search path** because objects in these environments can be found from the top-level interactive workspace.

```
1 search()
```

```
 [1] ".GlobalEnv"        "package:knitr"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

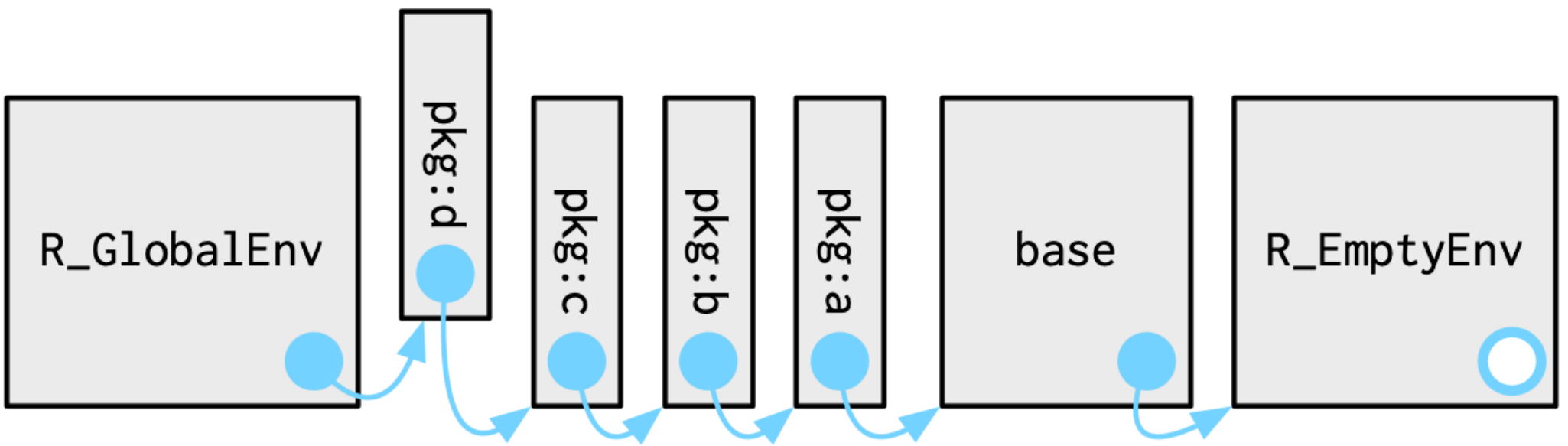
The Search Path Visualized

The `globalenv()`, `baseenv()`, the environments on the search path, and `emptyenv()` are connected as shown below.



The Search Path Visualized

Each time you load a new package with `library()`, the package environment is inserted between the global environment and the package that was previously at the top of the search path.



Note that the parent of the global environment changes.

Function Environments

Most environments are created as a consequence of using functions. This section discusses the four types of environments associated with a function: enclosing, binding, execution, and calling.

The **enclosing** environment is the environment where the function was created. Every function has one and only one enclosing environment. For the three other types of environment, there may be 0, 1, or many environments associated with each function:

- Binding a function to a name with the assignemnt `<-` operator defines a **binding** environment.
- Calling a function creates a temporary **execution** environment that stores variables created during execution.
- Every execution environment is associated with a **calling** environment, which tells you where the function was called.

The Enclosing and Binding environments

When a function is created, it gains a reference to the environment where it was made. This is the enclosing environment and is used for lexical scoping.

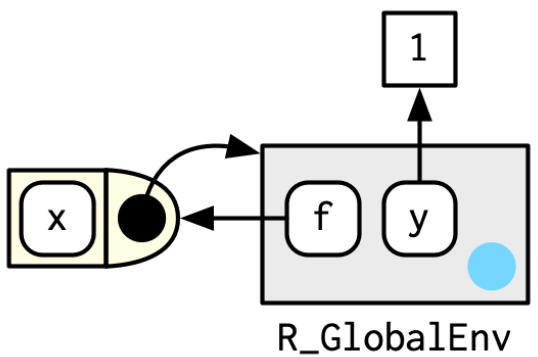
When you name a function, the environment where the name exists, is the binding environment.

In most scenarios, the enclosing environment and binding environment is the same.

Enclosing and Binding Environments

```
1 y <- 1
2 f <- function(x) {
3   x + y
4 }
5 environment(f)
```

```
<environment: R_GlobalEnv>
```



The function `f` is created in the `globalenv()`, so that is the enclosing environment. The name `f` exists in the `globalenv()` so it is the binding environment.

Package Environments

In the search path, we saw that the parent environment of a package varies based on what other packages have been loaded. Does this mean that the package will find different functions if packages are loaded in a different order?

For example, consider the `sd()` function:

```
1 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: 0x000002515a122340>
<environment: namespace:stats>
```

The `sd()` function is defined in terms of the `var()` function. How does R make sure that the `sd()` function is not affected by any function called `var()` in the global environment or in one of the attached packages in the search path?

Namespaces

The distinction between enclosing and binding environments is particularly important for functions inside packages.

To ensure that every package works the same way regardless of what packages are attached by the user, packages require an internal environment called the **namespace** in which the package functions are defined.

Every function in a package is associated with a pair of environments: the package environment and the namespace environment.

Package Environments and the `::` Operator

The **package environment** is the external interface to the package. This is the environment which the R user uses to find a function in an attached package. Its parent is determined by the search path, i.e. the order in which packages have been attached.

The **double colon** `::` operator can be used to access a function directly from the package environment (regardless of the search path).

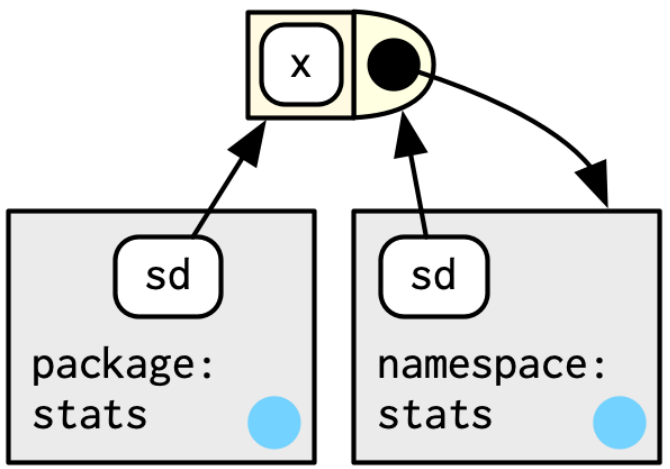
For example, `base::mean()` refers to the `mean()` function from the `base` package, while `mosaic::mean()` refers to the `mean()` function from the `mosaic` package.

Double colon notation allows you to access objects that may be masked by other objects from packages higher in the search path.

Namespace Environments

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.



The namespace environment of a package often contains internal or non-exported variables (bindings) that allows

Execution Environments

Each time a function is called, a new environment is created to host execution. The parent of the execution environment is the enclosing environment of the function. Once the function has completed, this environment is thrown away.

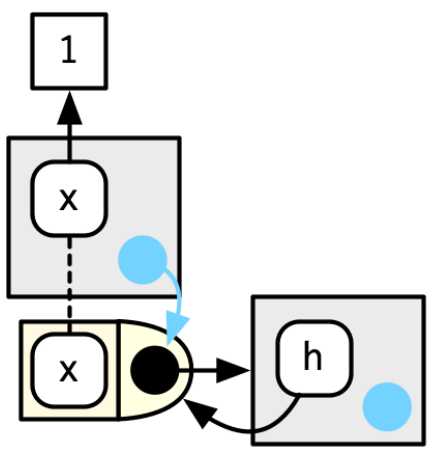
Let's look at a simple function.

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

Execution Environments

```
1 h <- function(x) {
2   a <- 2
3   x + a
4 }
5 y <- h(1)
```

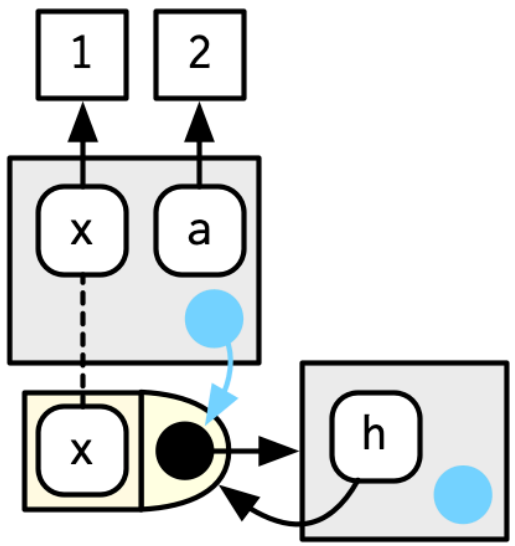
1. Function called with x = 1



Execution Environments

```
1 h <- function(x) {
2   a <- 2
3   x + a
4 }
5 y <- h(1)
```

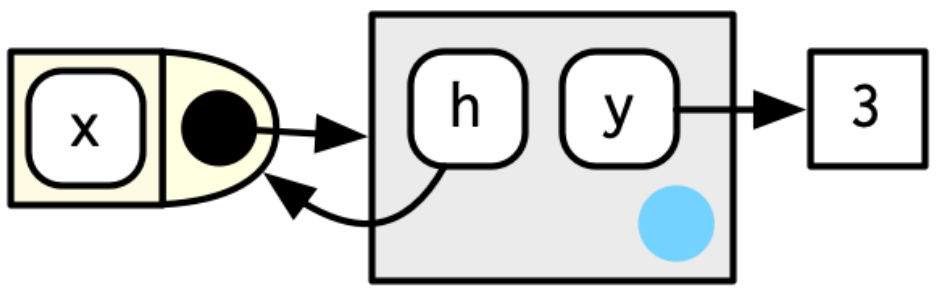
2. a bound to value 2



Execution Environments

```
1 h <- function(x) {
2   a <- 2
3   x + a
4 }
5 y <- h(1)
```

3. Function completes returning value 3.
Execution environment goes away.



Calling Environment

When you call a function, it creates an execution environment. The execution environment has two parent environments: the enclosing environment (where the function is created) and calling environment (where the function is called).

If you created a function in the `globalenv()` and call the function in the `globalenv()`, then these are the same. But in some cases, you might call a function from within another function. In this case, the environments are different.

R's normal scoping rules will use the enclosing environment, so if the function is looking for values, it looks in the enclosing environment.

However, R also supports **dynamic scoping** where you can look for values in the calling environment. This is achieved with the function `get()`

For example, if you want to get the value of `y` from the calling environment of the function, rather than the parent environment of the function, use:

```
y <- get("y", envir = parent.frame())
```