CT5102: Programming for Data Analytics

Lecture 9: Environments & Functions

Dr. Jim Duggan,

School of Engineering & Informatics

National University of Ireland Galway.

https://github.com/JimDuggan/PDAR

https://twitter.com/_jimduggan

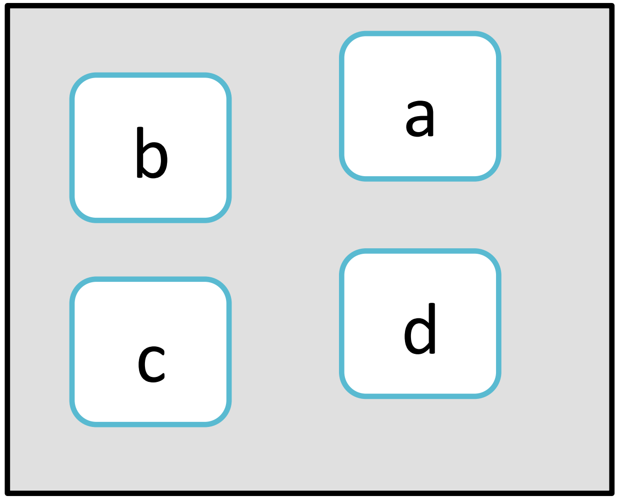
Overview

- Summary of environments
- Superassignment operator
- Function environments
  - Enclosing
  - Binding
  - Execution
  - Calling
- Designing closures

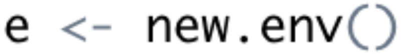
Lectures I-3	**R Fundamentals** *Atomic Vectors – Functions – Lists – Matrices – Data Frames*
Lectures 4-9	**Data Science with R** *ggplot2 – dplyr – tidyr – stringr – lubridate - purrr*
Lectures 10-11	**Advanced Programming with R** ***Environments*** *–* ***Closures*** *– S3 Object System*
Lectures 12	**Machine Learning with R – Case Studies** *Electricity Generation, Health*

Environment Basics

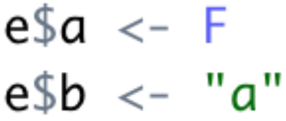
- The job of an environment is to associate a set of names to a set of values (a bag of names) (Wickham 2015)



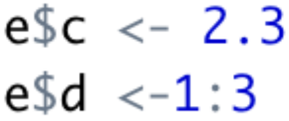
Each name points to an object stored elsewhere in memory



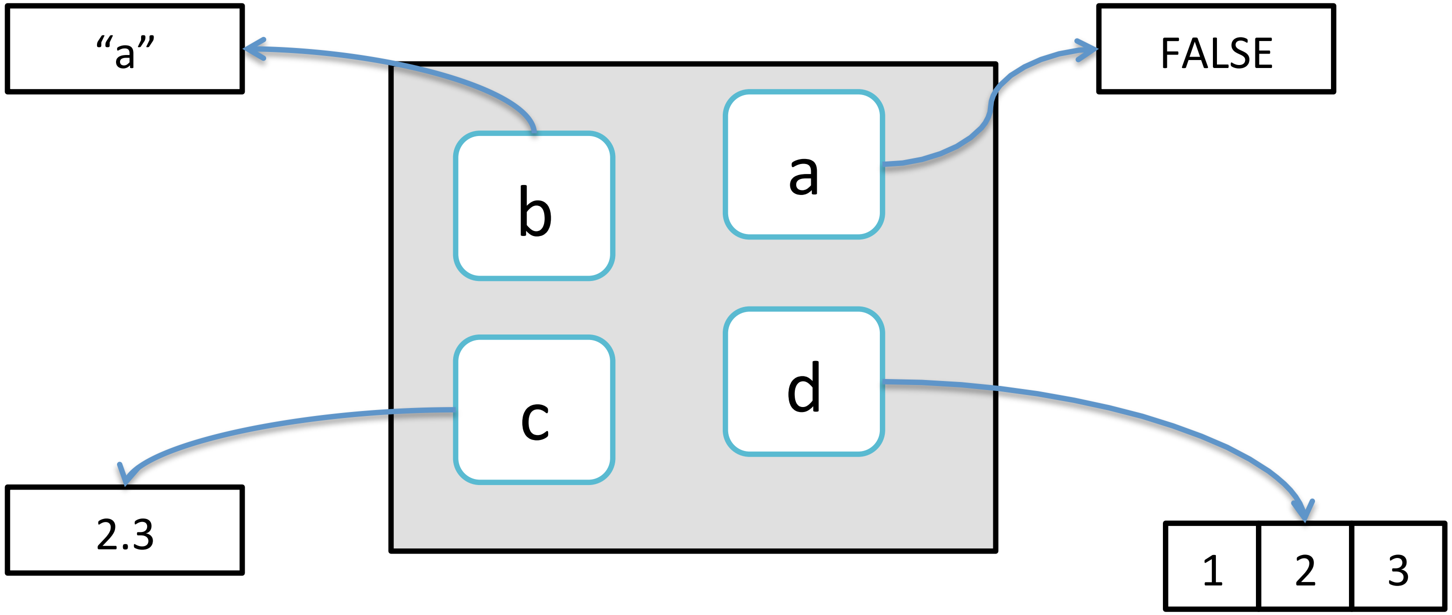
```
e <- new.env()
```

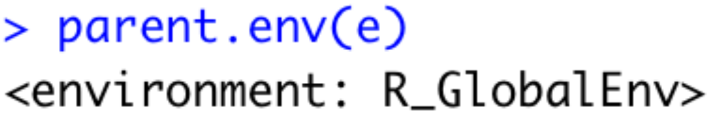


```
e$a <- F
e$b <- "a"
```



```
e$c <- 2.3
e$d <-1:3
```

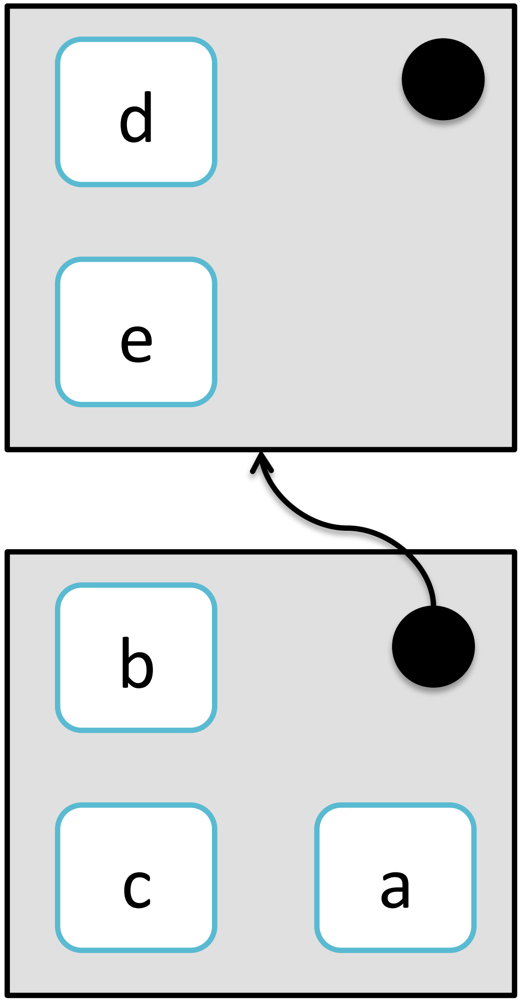




```
> parent.env(e)
<environment: R_GlobalEnv>
```

Hierarchies

- Every environment has a parent, another environment
- The parent is used to implement lexical scoping
- Only one environment does not have a parent – the **empty** environment
- An environment does not have information on its "children"



Properties of an environment

- Generally, an environment is similar to a list, with four exceptions:
  - Every object in an environment has a unique name
  - The objects in an environment are not ordered
  - An environment has a parent
  - Environments have reference semantics: *When you modify a binding in an environment, the environment is not copied; it's modified in place*

Useful Definition

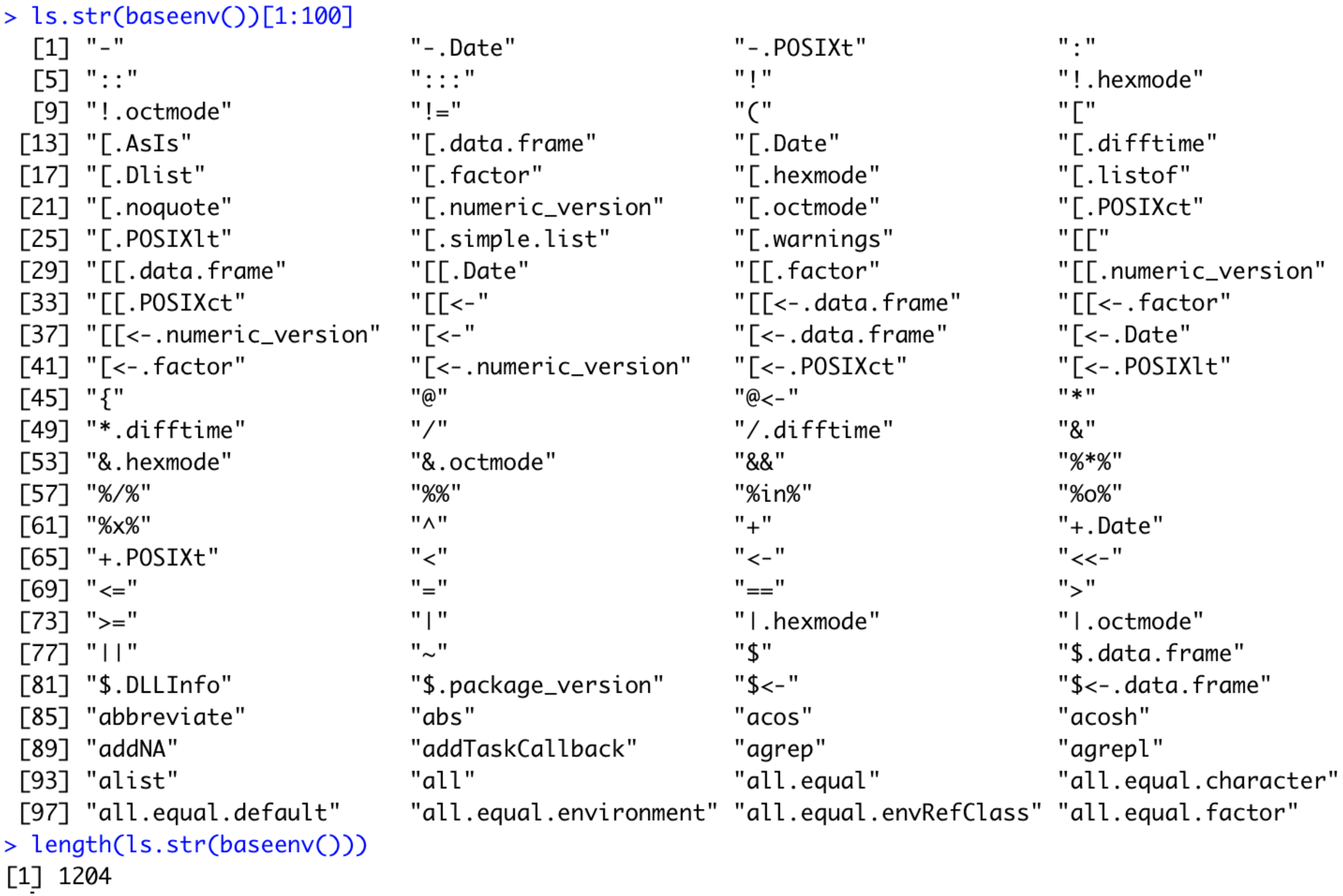
https://www.r-bloggers.com/environments-in-r/

- Environments can be thought of as consisting of two things: a frame, which is a set of symbol-value pairs, and an enclosure, a pointer to an enclosing environment.
- When R looks up the value for a symbol the frame is examined and if a matching symbol is found its value will be returned.
- If not, the enclosing environment is then accessed and the process repeated.
- Environments form a tree structure in which the enclosures play the role of parents. The tree of environments is rooted in an empty environment, available through emptyenv(), which has no parent.

There are 4 special environments

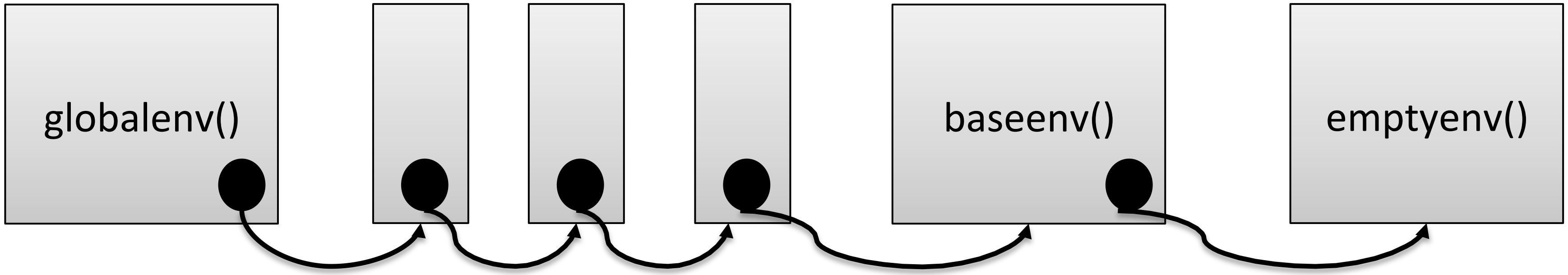
- **globalenv()** is the interactive workspace. The parent of this is the last package attached with library() or require()
- **baseenv()** is the environment of the base package
- **emptyenv()** is the ultimate ancestor of all environments, and the only one without a parent
- **environment()** is the current environment

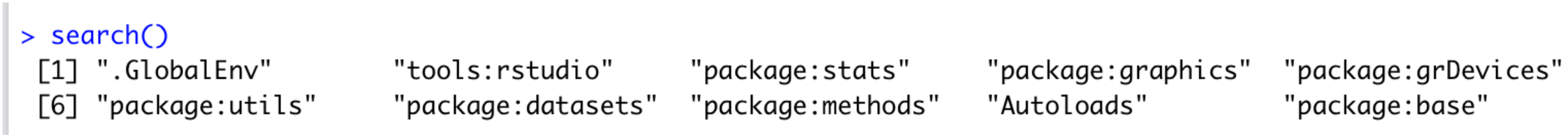
Example: baseenv()



```
> ls.str(baseenv())[1:100]
  [1] "-"                      "-.Date"                 "-.POSIXt"               ":"
  [5] "::"                     ":::"                    "!"                      "!.hexmode"
  [9] "!.octmode"              "!="                     "("                      "["
 [13] "[.AsIs"                 "[.data.frame"           "[.Date"                 "[.difftime"
 [17] "[.Dlist"                "[.factor"               "[.hexmode"              "[.listof"
 [21] "[.noquote"              "[.numeric_version"      "[.octmode"              "[.POSIXct"
 [25] "[.POSIXlt"              "[.simple.list"          "[.warnings"             "[["
 [29] "[[.data.frame"          "[[.Date"                "[[.factor"              "[[.numeric_version"
 [33] "[[.POSIXct"             "[[<-"                   "[[<-.data.frame"        "[[<-.factor"
 [37] "[[<-.numeric_version"   "[<-"                    "[<-.data.frame"         "[<-.Date"
 [41] "[<-.factor"             "[<-.numeric_version"    "[<-.POSIXct"            "[<-.POSIXlt"
 [45] "{"                      "@"                      "@<-"                    "*"
 [49] "*.difftime"             "/"                      "/.difftime"             "&"
 [53] "&.hexmode"              "&.octmode"              "&&"                     "%*%"
 [57] "%/%"                    "%%"                     "%in%"                   "%o%"
 [61] "%x%"                    "^"                      "+"                      "+.Date"
 [65] "+.POSIXt"               "<"                      "<-"                     "<<-"
 [69] "<="                     "="                      "=="                     ">"
 [73] ">="                     "|"                      "|.hexmode"              "|.octmode"
 [77] "||"                     "~"                      "$"                      "$.data.frame"
 [81] "$.DLLInfo"              "$.package_version"      "$<-"                    "$<-.data.frame"
 [85] "abbreviate"             "abs"                    "acos"                   "acosh"
 [89] "addNA"                  "addTaskCallback"        "agrep"                  "agrepl"
 [93] "alist"                  "all"                    "all.equal"              "all.equal.character"
 [97] "all.equal.default"      "all.equal.environment"  "all.equal.envRefClass"  "all.equal.factor"
> length(ls.str(baseenv()))
[1] 1204
```

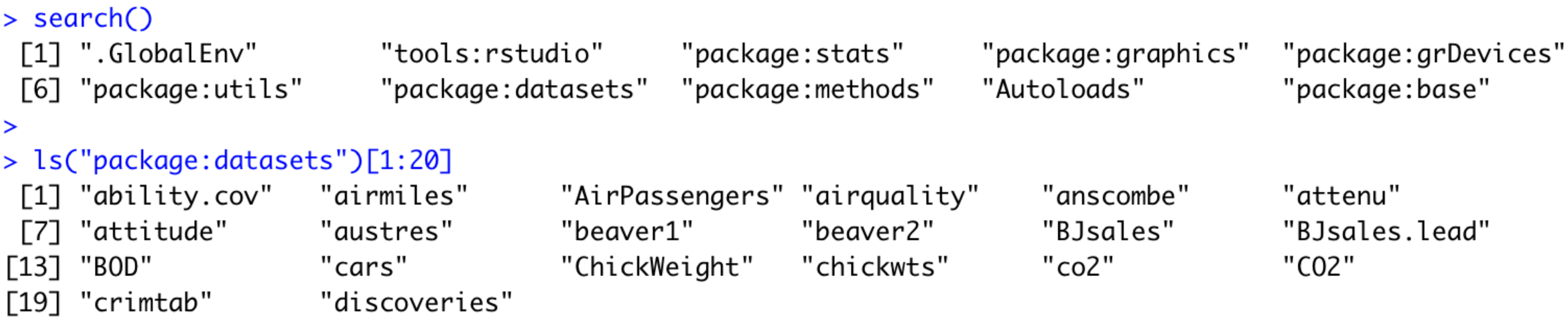
The search path



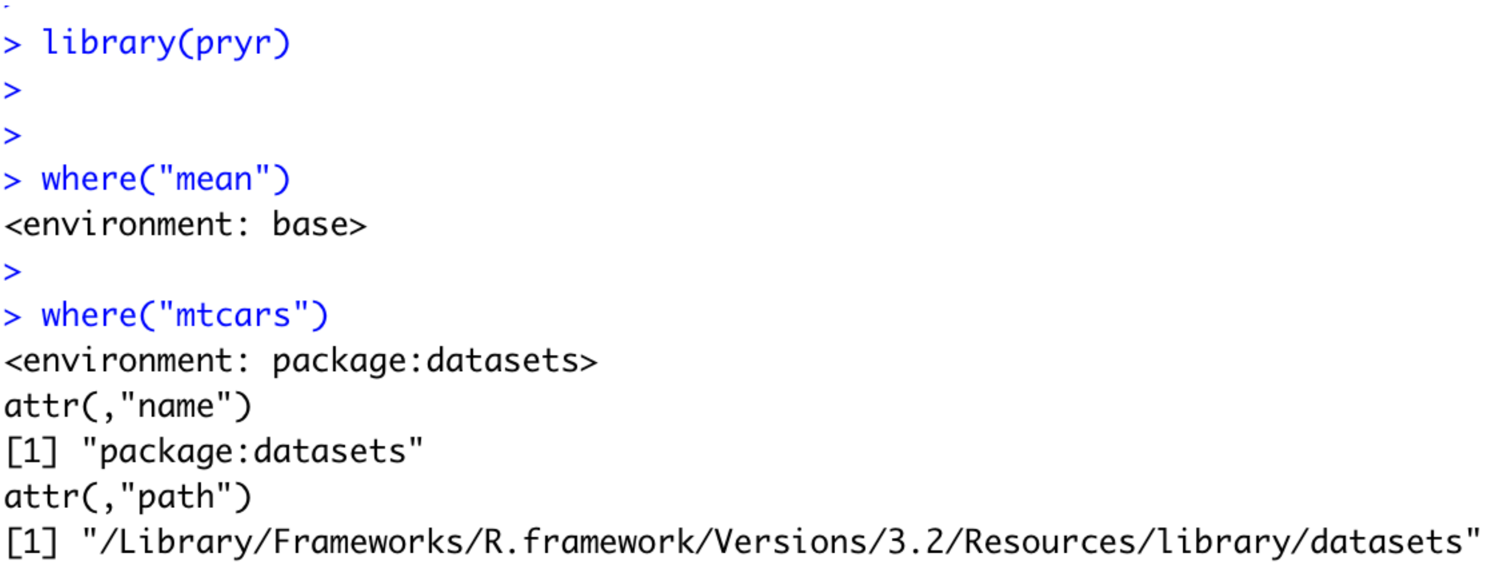


```
> search()
 [1] ".GlobalEnv"        "tools:rstudio"     "package:stats"     "package:graphics"  "package:grDevices"
 [6] "package:utils"     "package:datasets"  "package:methods"   "Autoloads"         "package:base"
```

Searching Environments

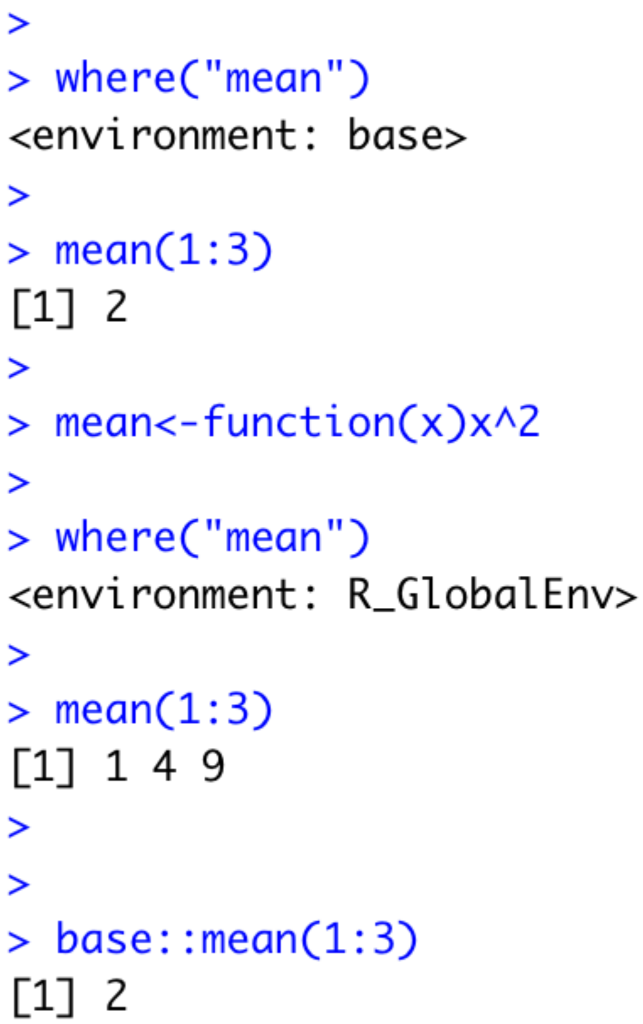


```
> search()
 [1] ".GlobalEnv"        "tools:rstudio"     "package:stats"     "package:graphics"  "package:grDevices"
 [6] "package:utils"     "package:datasets"  "package:methods"   "Autoloads"         "package:base"
>
> ls("package:datasets")[1:20]
 [1] "ability.cov"   "airmiles"      "AirPassengers" "airquality"    "anscombe"      "attenu"
 [7] "attitude"      "austres"       "beaver1"       "beaver2"       "BJsales"       "BJsales.lead"
[13] "BOD"           "cars"          "ChickWeight"   "chickwts"      "co2"           "CO2"
[19] "crimtab"       "discoveries"
```



```
> library(pryr)
>
>
> where("mean")
<environment: base>
>
> where("mtcars")
<environment: package:datasets>
attr(,"name")
[1] "package:datasets"
attr(,"path")
[1] "/Library/Frameworks/R.framework/Versions/3.2/Resources/library/datasets"
```

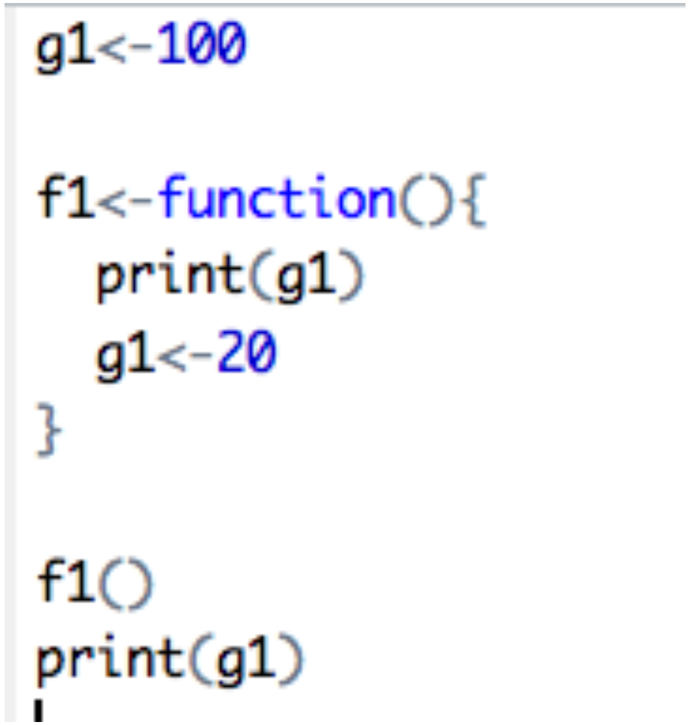
Functions with same names?



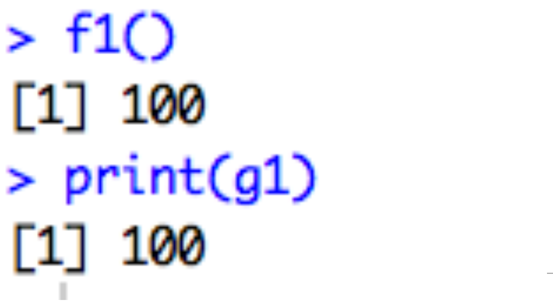
```
>
> where("mean")
<environment: base>
>
> mean(1:3)
[1] 2
>
> mean<-function(x)x^2
>
> where("mean")
<environment: R_GlobalEnv>
>
> mean(1:3)
[1] 1 4 9
>
>
> base::mean(1:3)
[1] 2
```

Double arrow assignment operator

- Code that exists at a certain level of the environment has at least read access to all the variables the level above it
- However, direct write access to variables at higher levels via the standard <- operator is not possible



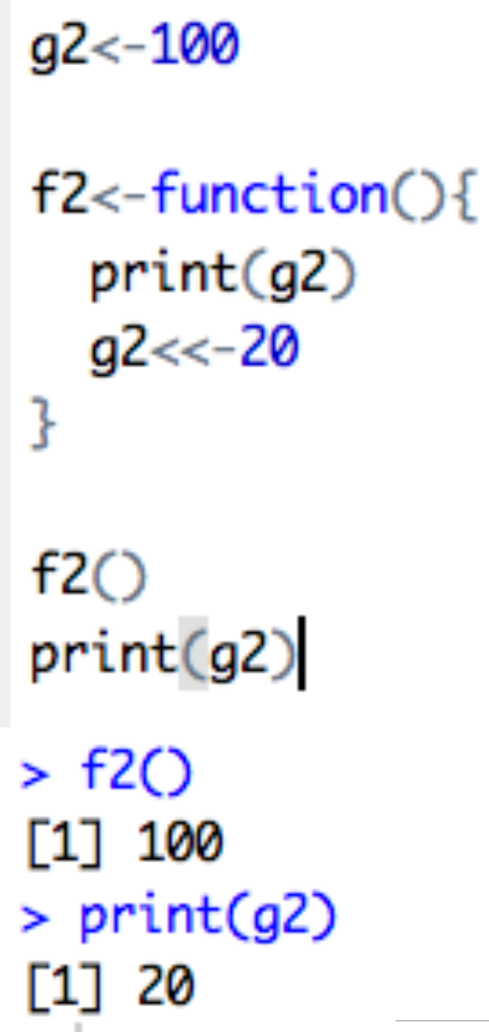
```
g1<-100

f1<-function(){
  print(g1)
  g1<-20
}

f1()
print(g1)
```



```
> f1()
[1] 100
> print(g1)
[1] 100
```

Solution
Change a global value

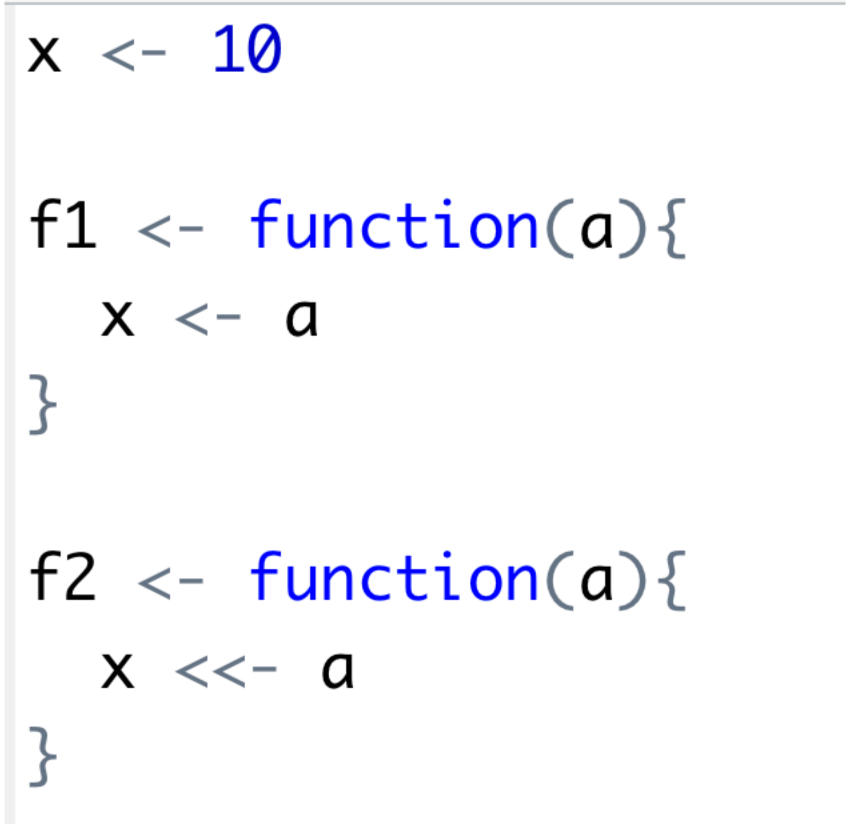
- Double arrow assignment operator (aka superassignment operator) changes the global value
- Typically used to write to a top level variable
- However:
  - The operator will search up the environment hierarchy, stopping at the first level the name is encountered
  - If no name is found, the variable is assigned at the global level.



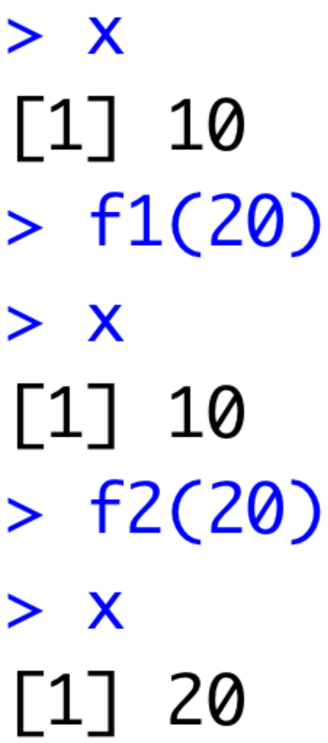
```
g2<-100

f2<-function(){
  print(g2)
  g2<<-20
}

f2()
print(g2)
```

```
> f2()
[1] 100
> print(g2)
[1] 20
```

Example 1

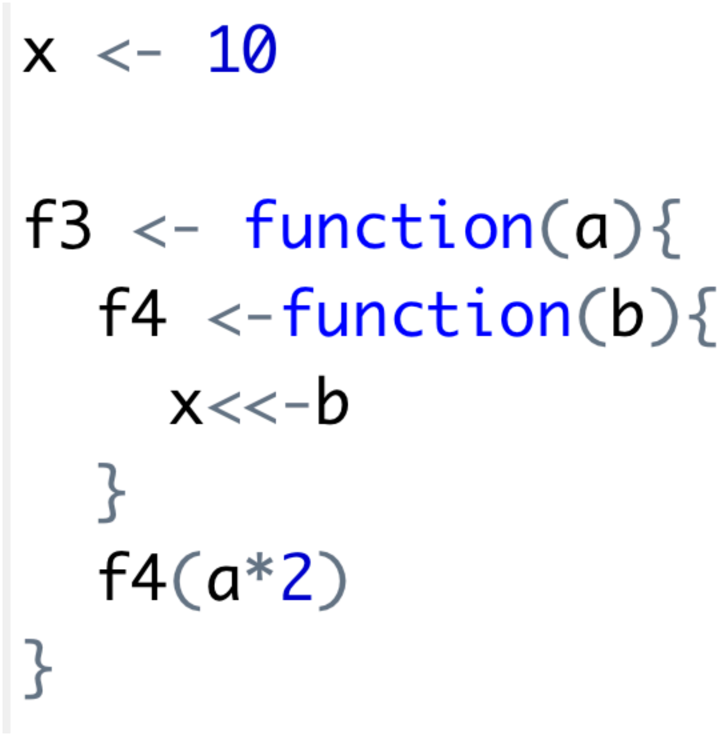


```r
x <- 10

f1 <- function(a){
  x <- a
}

f2 <- function(a){
  x <<- a
}
```

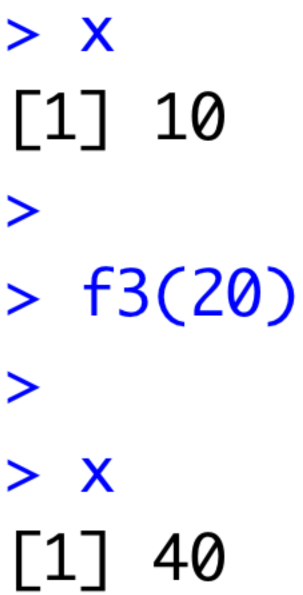


```
> x
[1] 10
> f1(20)
> x
[1] 10
> f2(20)
> x
[1] 20
```

Example 2

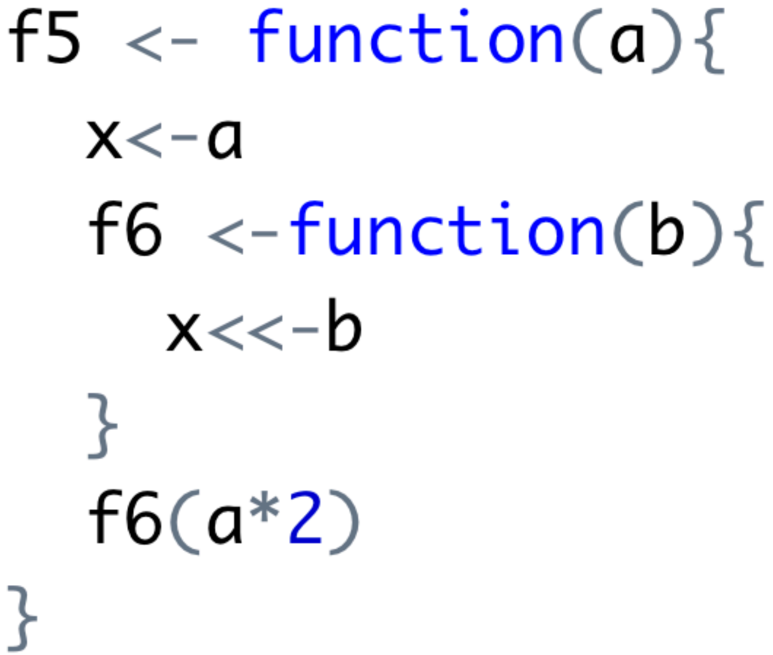


```r
x <- 10

f3 <- function(a){
  f4 <-function(b){
    x<<-b
  }
  f4(a*2)
}
```

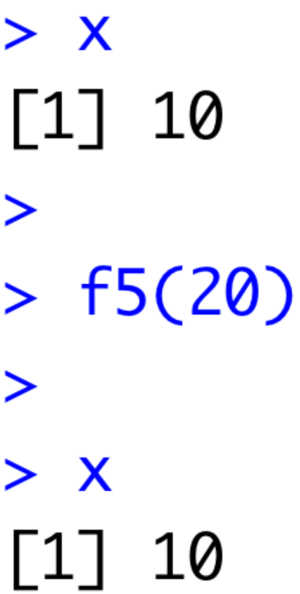


```
> x
[1] 10
>
> f3(20)
>
> x
[1] 40
```

Example 3

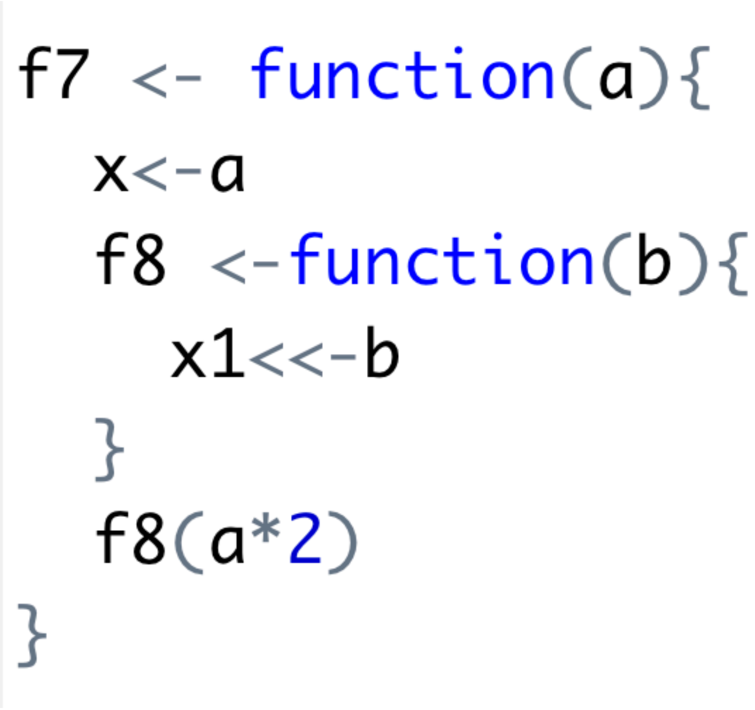


```
f5 <- function(a){
  x<-a
  f6 <-function(b){
    x<<-b
  }
  f6(a*2)
}
```

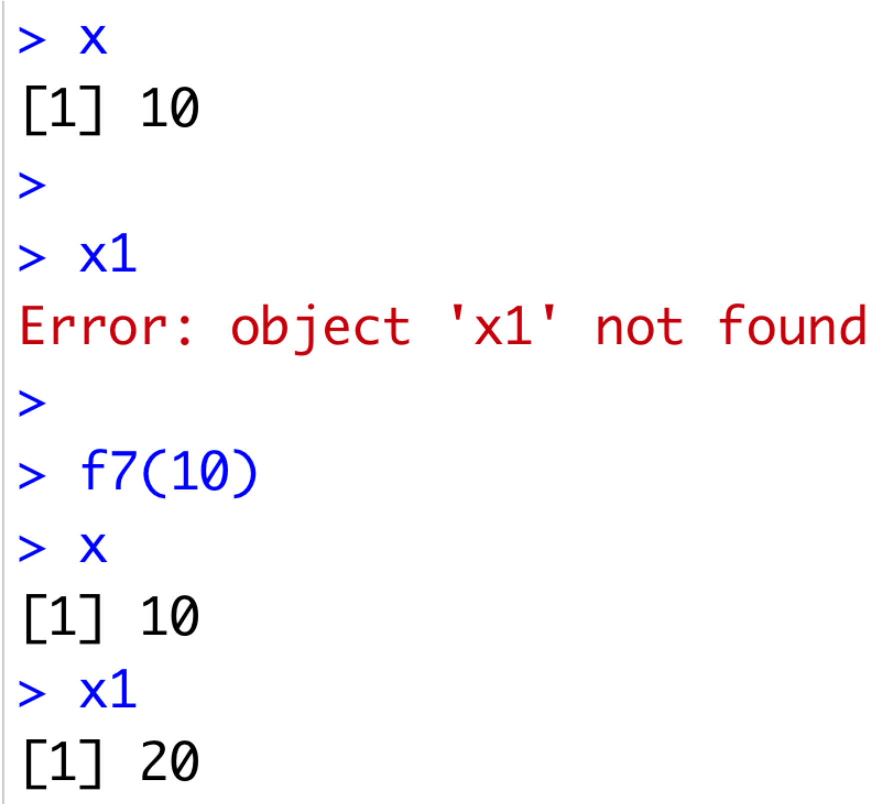


```
> x
[1] 10
>
> f5(20)
>
> x
[1] 10
```

Example 4



```
f7 <- function(a){
  x<-a
  f8 <-function(b){
    x1<<-b
  }
  f8(a*2)
}
```



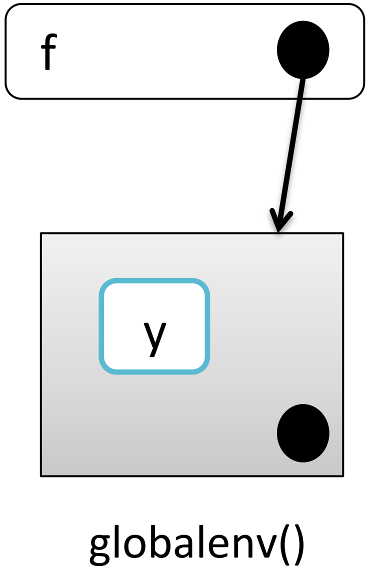
```
> x
[1] 10
>
> x1
Error: object 'x1' not found
>
> f7(10)
> x
[1] 10
> x1
[1] 20
```

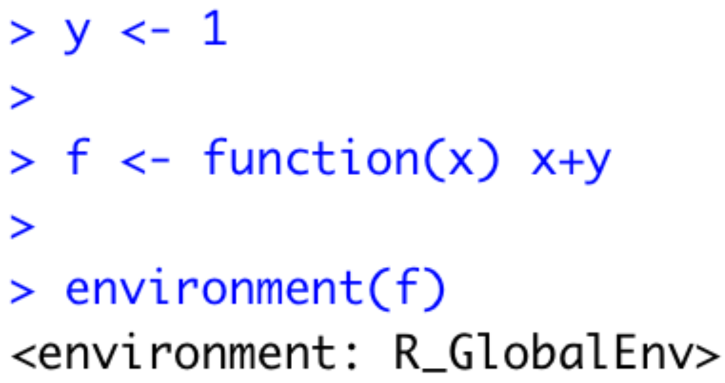
Function Environments

- Most environments are created as a consequence of using functions
- The are four types of environments associated with a function:
  - Enclosing environment
  - Execution environment
  - Binding environment
  - Calling environment

(1) Enclosing Environment

- When a function is created, it gains a reference to the environment where it was made
- The enclosing environment determines how the function finds values.

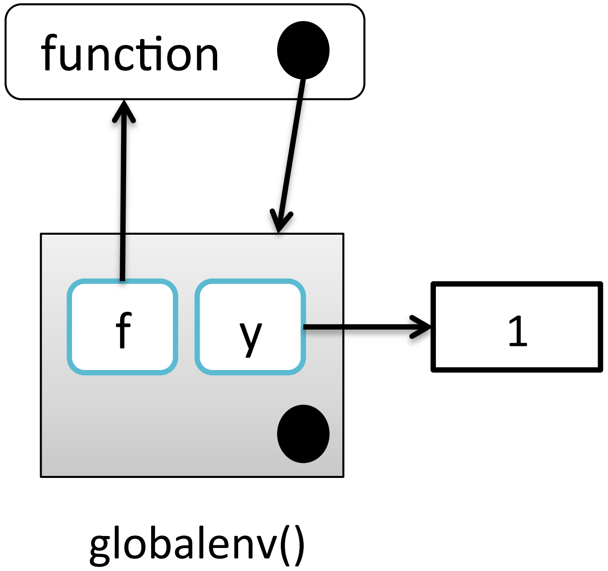


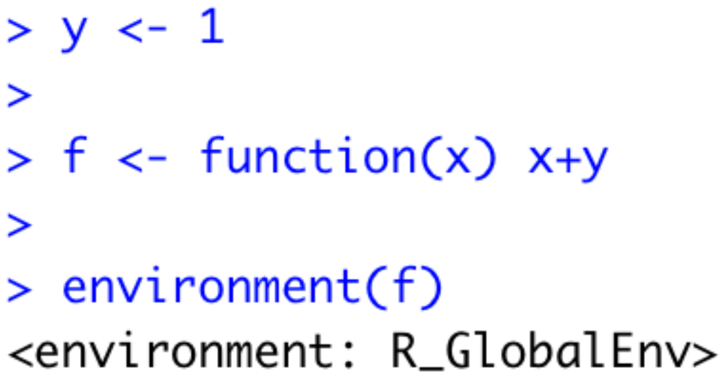


```
> y <- 1
>
> f <- function(x) x+y
>
> environment(f)
<environment: R_GlobalEnv>
```

(2) Binding Environment

- Previous diagram too simple because functions don’t have names.
- The binding environments of a function are all the environments which have a binding to it.
- The binding environment determines how we find the function

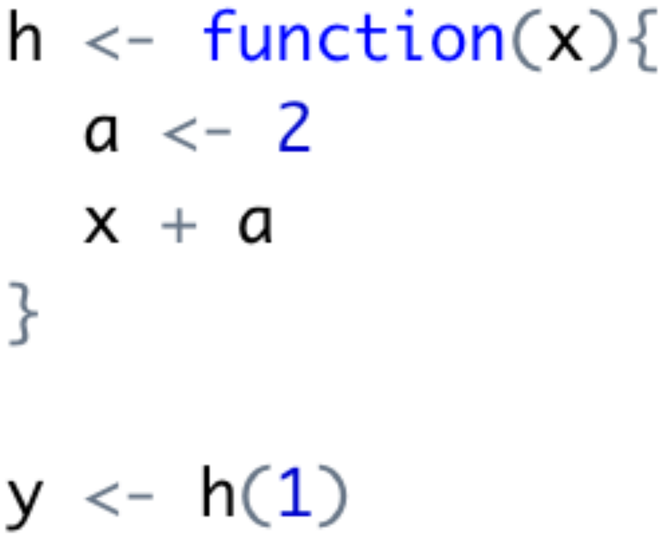




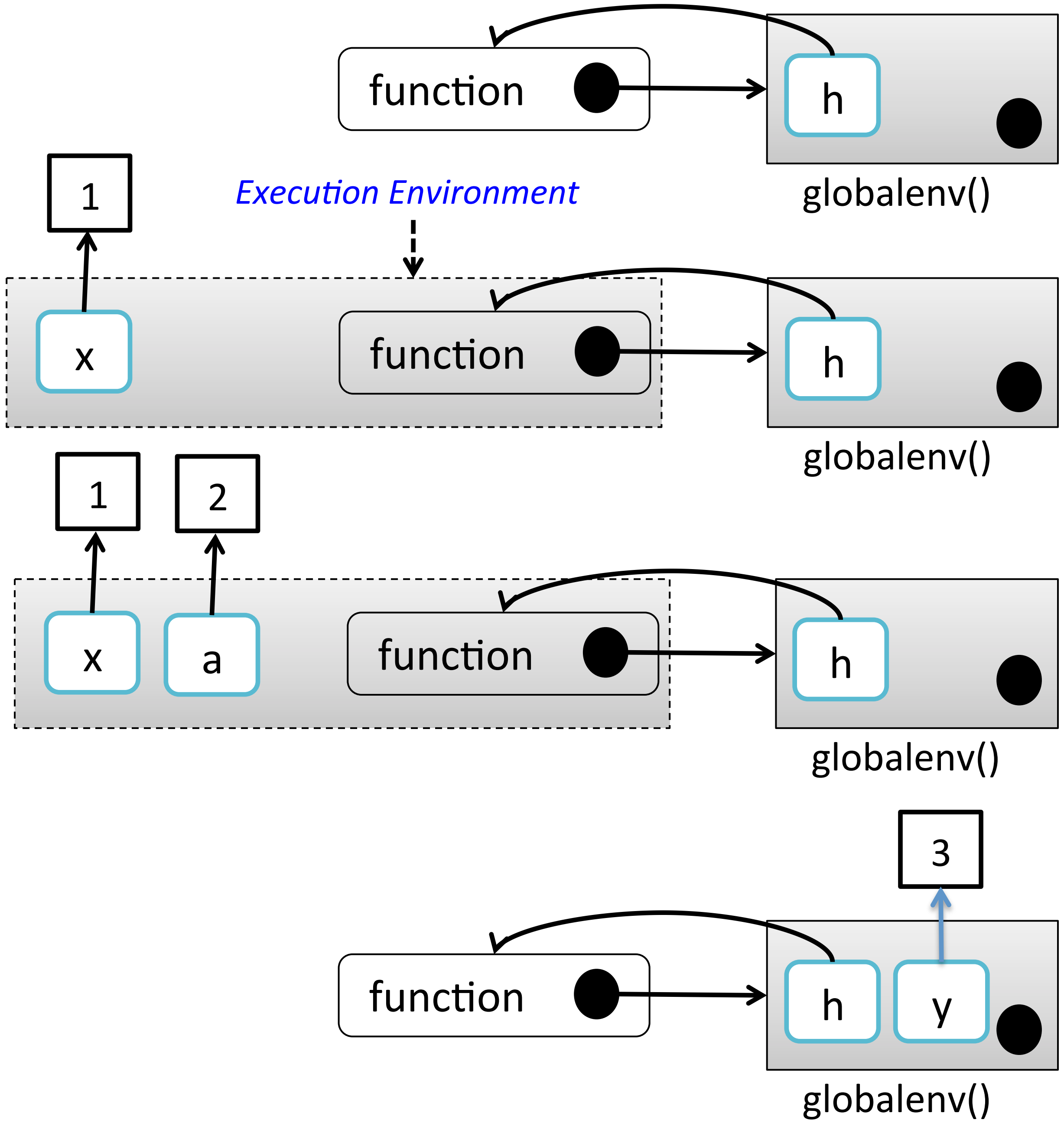
```
> y <- 1
>
> f <- function(x) x+y
>
> environment(f)
<environment: R_GlobalEnv>
```

(3) 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 is completed, this execution environment is discarded



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

y <- h(1)
```



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

y <- h(1)
```

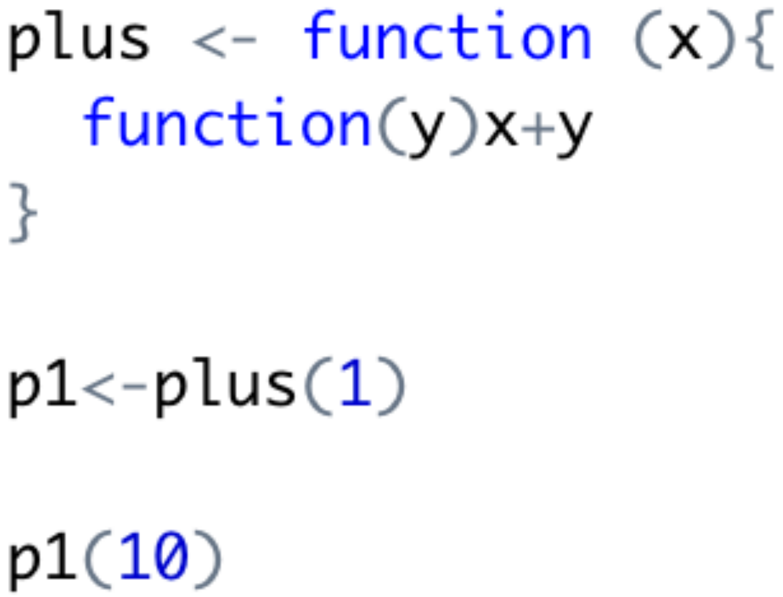
(1) Function called with x =1

(2) a assigned value 2

(3) Function completes returning value 3, and the execution environment is discarded

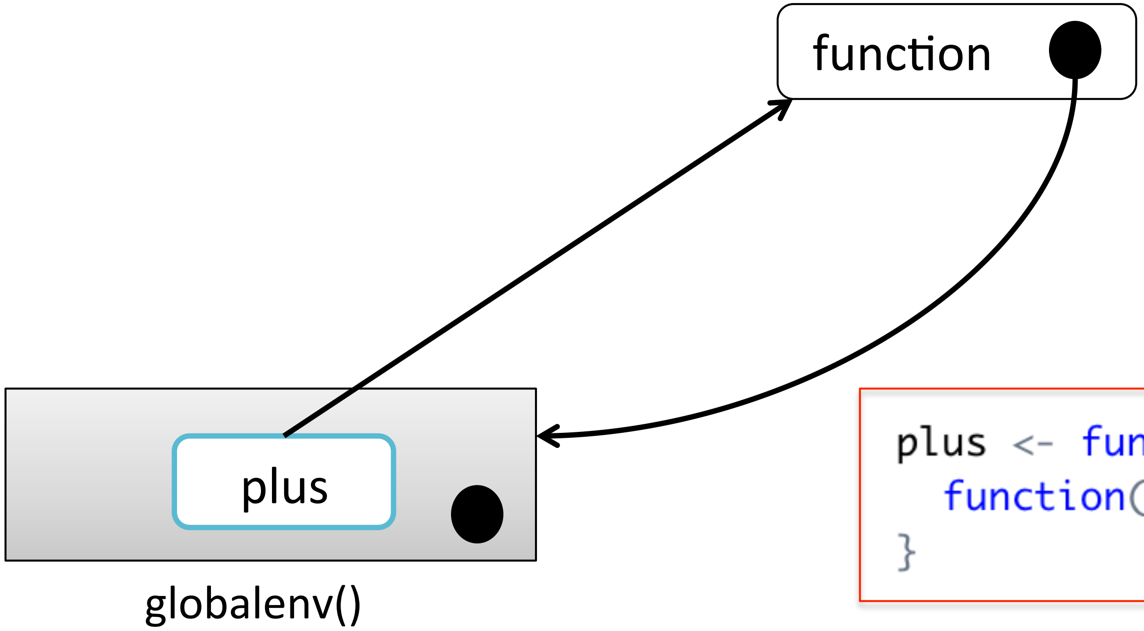
Key Point

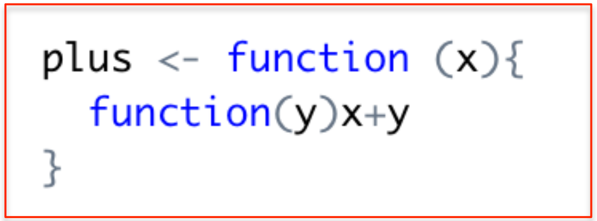
- When you create a function inside another function ***the enclosing environment of the child function is the execution environment of the parent***
- Therefore, the execution environment is no longer ephemeral
- ***What will p1(10) return?***



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

p1<-plus(1)

p1(10)
```

Define the function plus

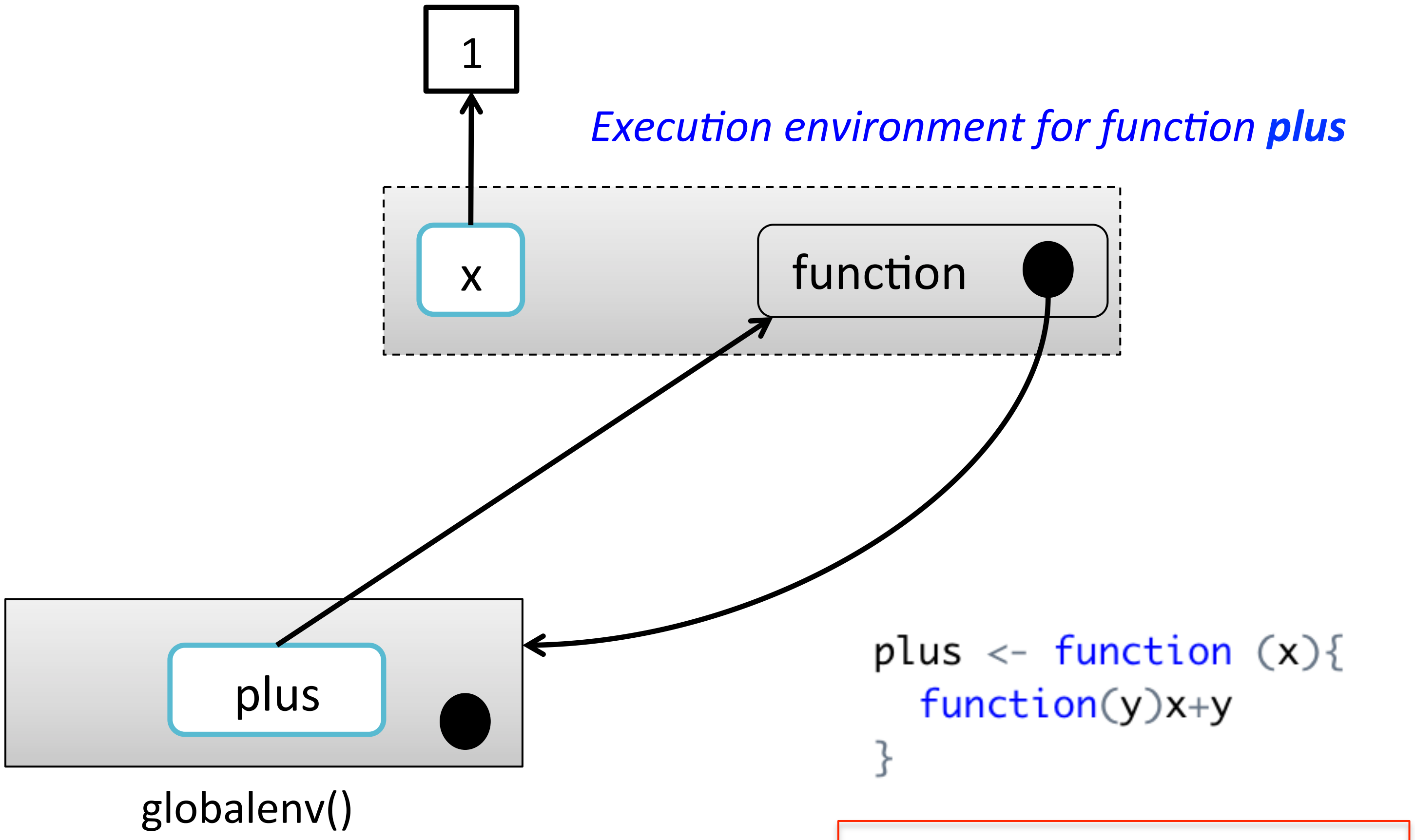




```
plus <- function (x){
  function(y)x+y
}
```

```
p1<-plus(1)

p1(10)
```

Call the function plus (1) – step 1

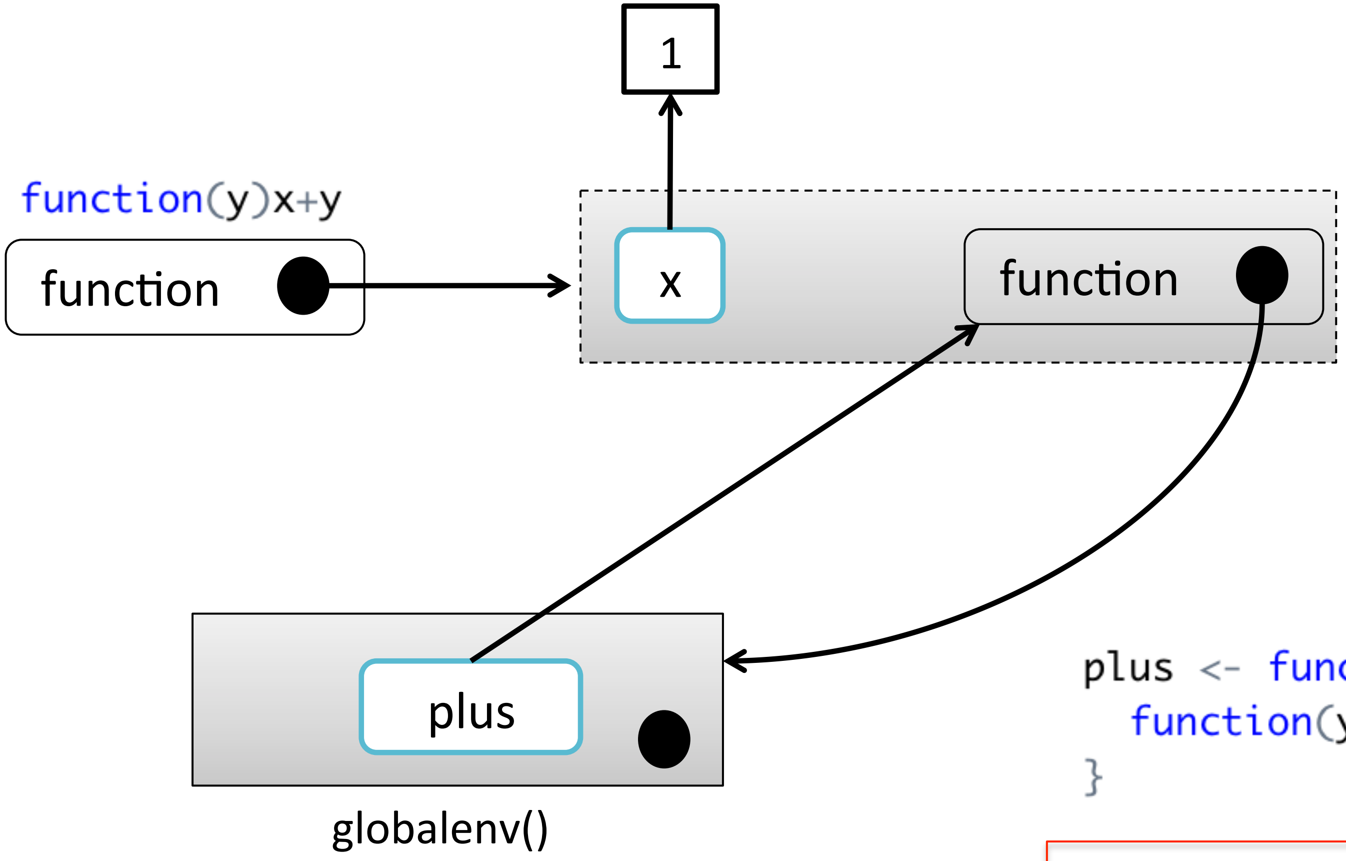


```
plus <- function (x){
  function(y)x+y
}
```

```
p1<-plus(1)
```

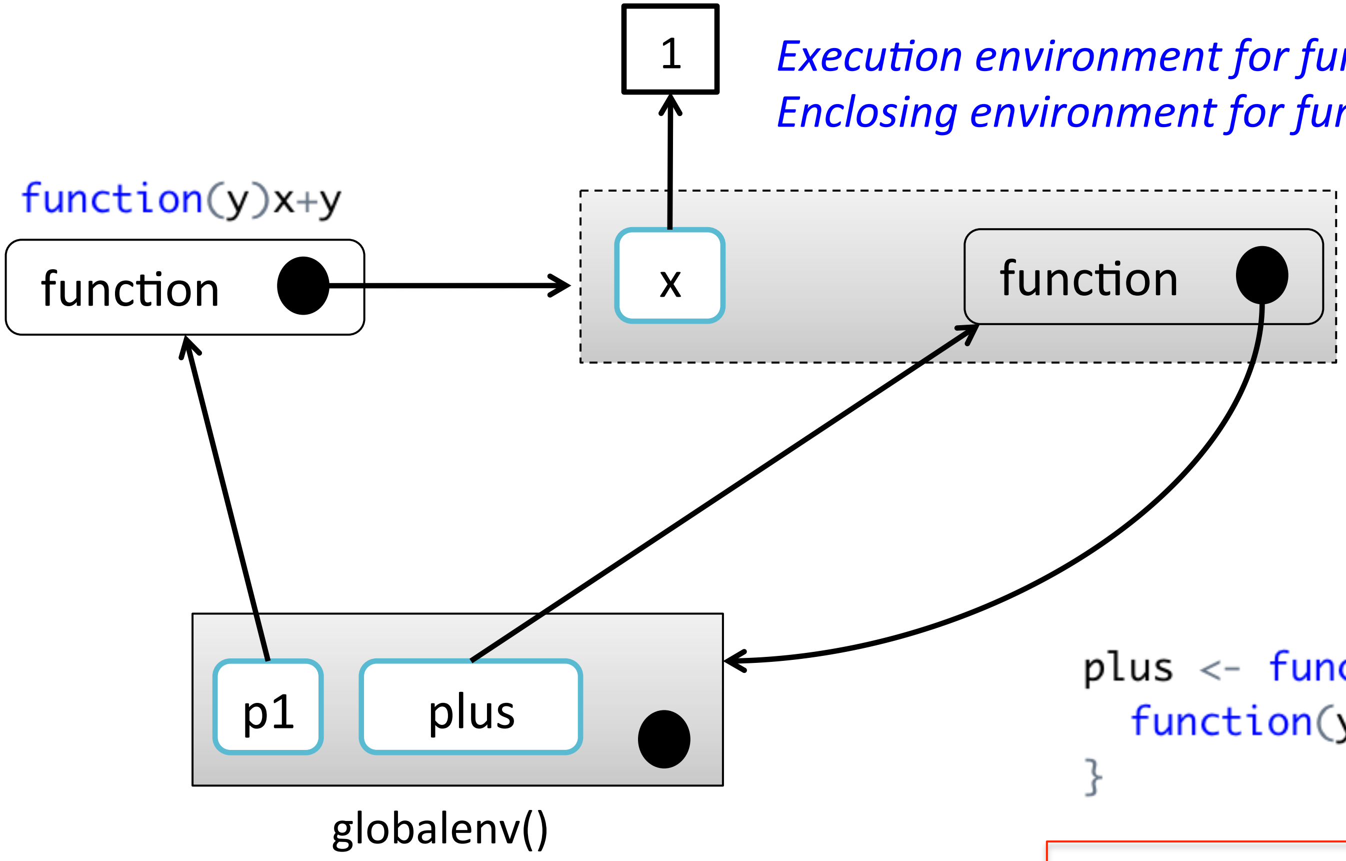
```
p1(10)
```

Call the function plus (1) – step 2



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

p1<-plus(1)

p1(10)
```

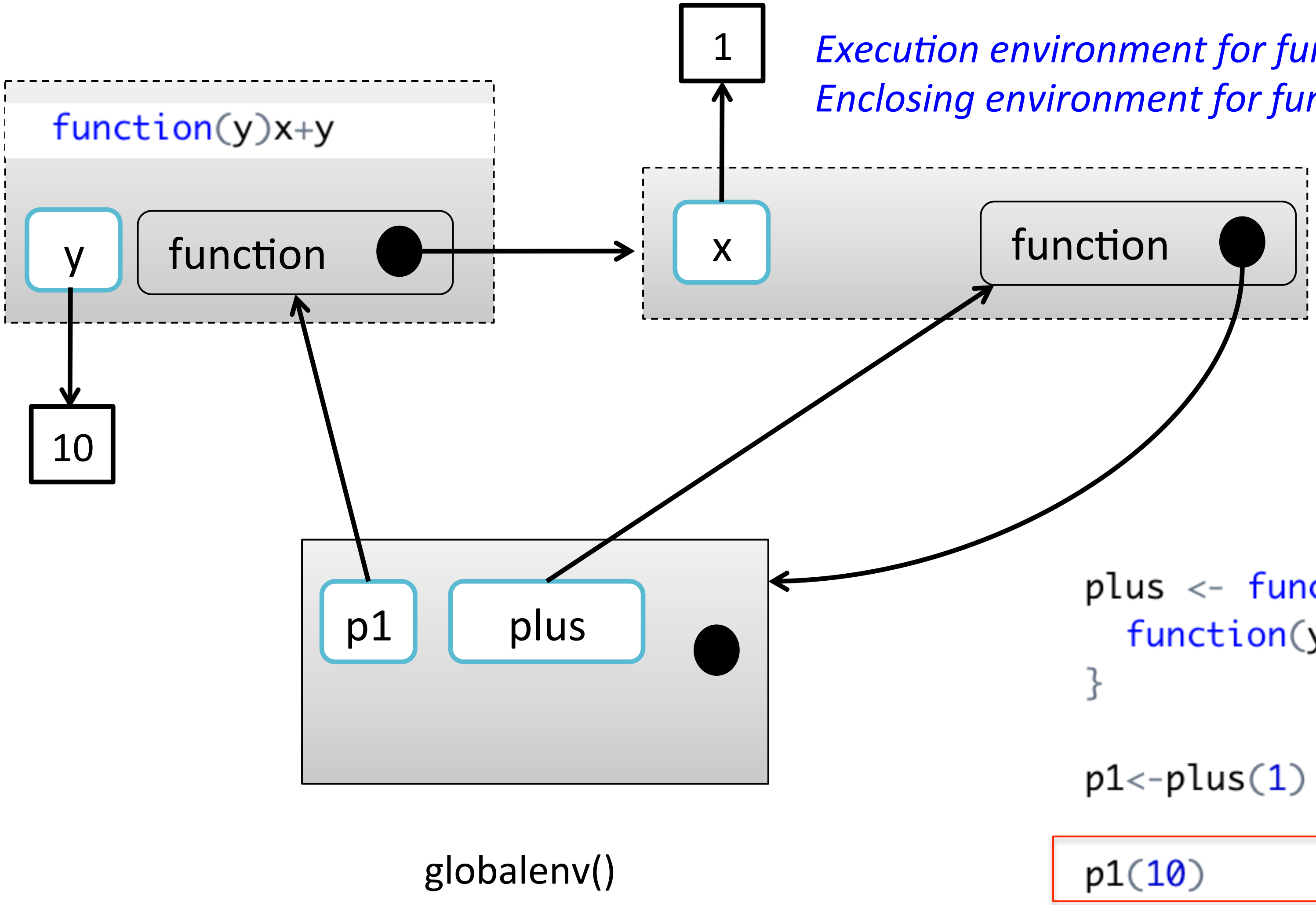
Return and Assign the value to p1



*Execution environment for function **plus***
*Enclosing environment for function **p1***

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

p1<-plus(1)

p1(10)
```

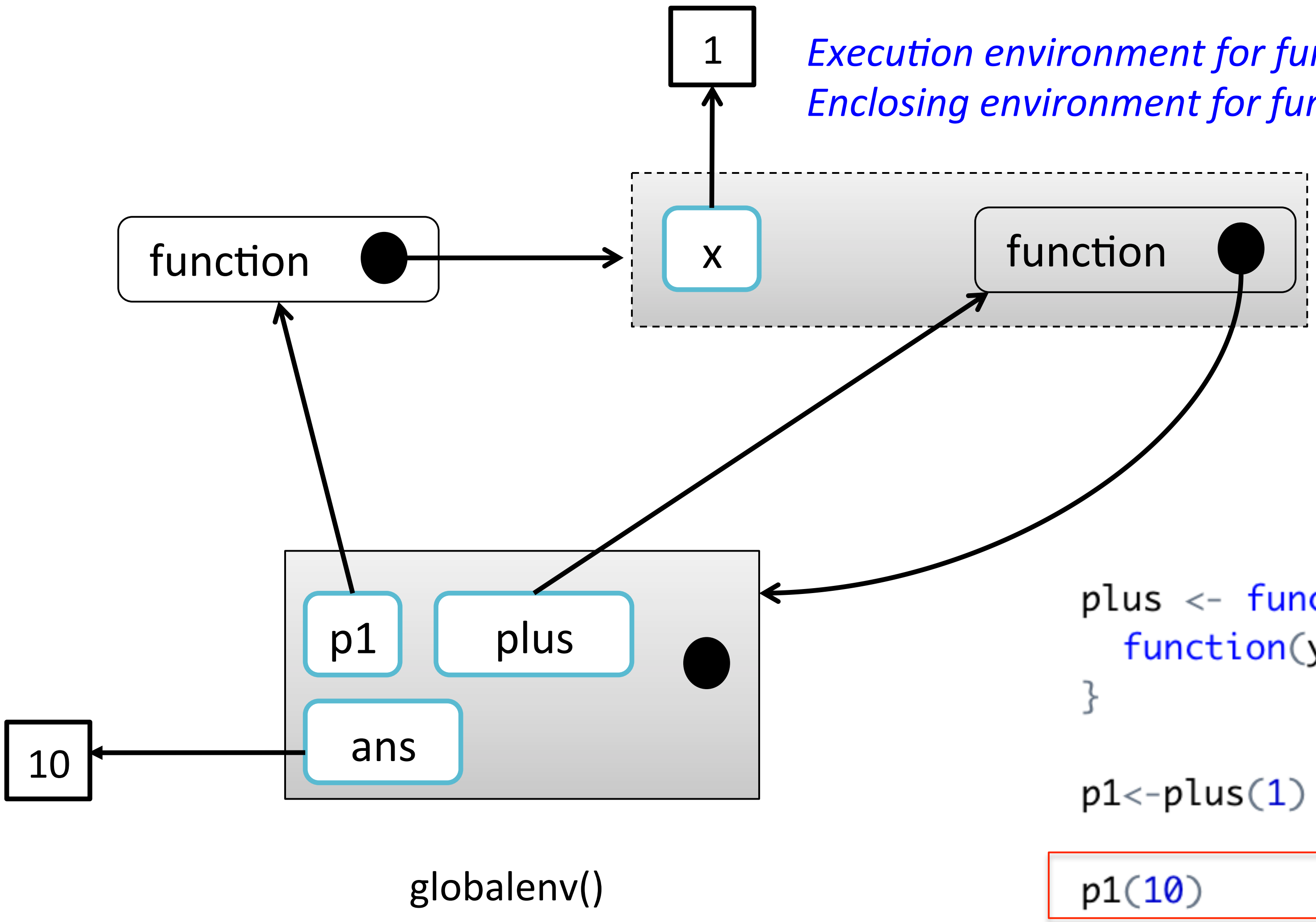
Call ans <- p1(10)



*Execution environment for function **plus***
*Enclosing environment for function **p1***

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

p1<-plus(1)

p1(10)
```

After call returns...



*Execution environment for function **plus***
*Enclosing environment for function **p1***

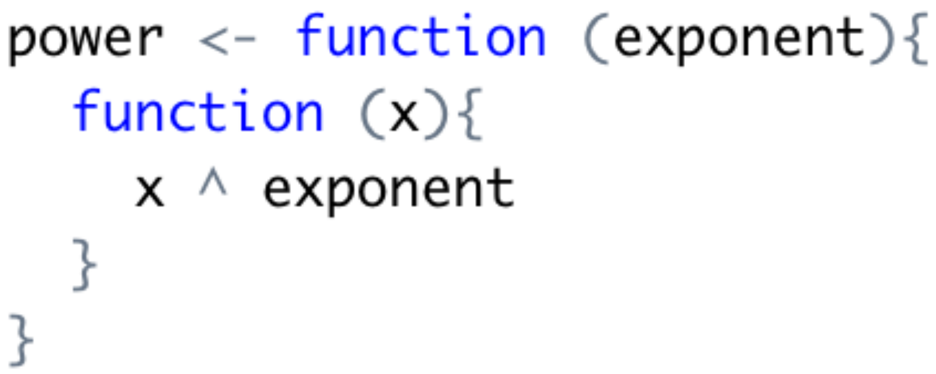
```
plus <- function (x){
  function(y)x+y
}

p1<-plus(1)

p1(10)
```

Closures

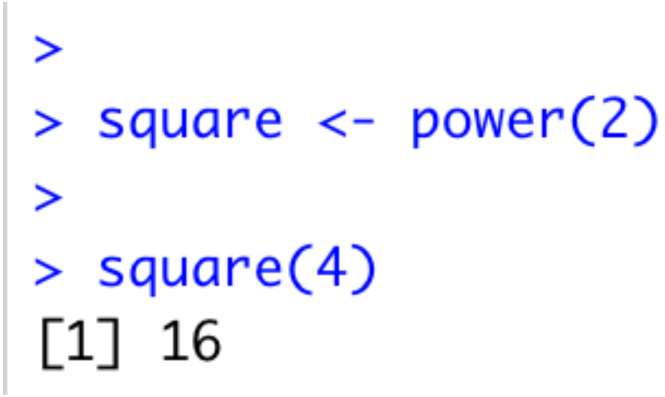
"An object is data with functions. A closure is a function with data." John D. Cook.

- Anonymous functions can be used to create closures, functions written by functions
- Closures get their name because the enclose the environment of the parent function and can then access all its variables

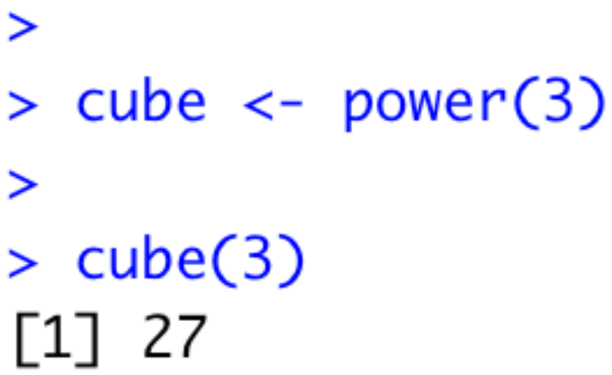
Example



```
power <- function (exponent){
  function (x){
    x ^ exponent
  }
}
```



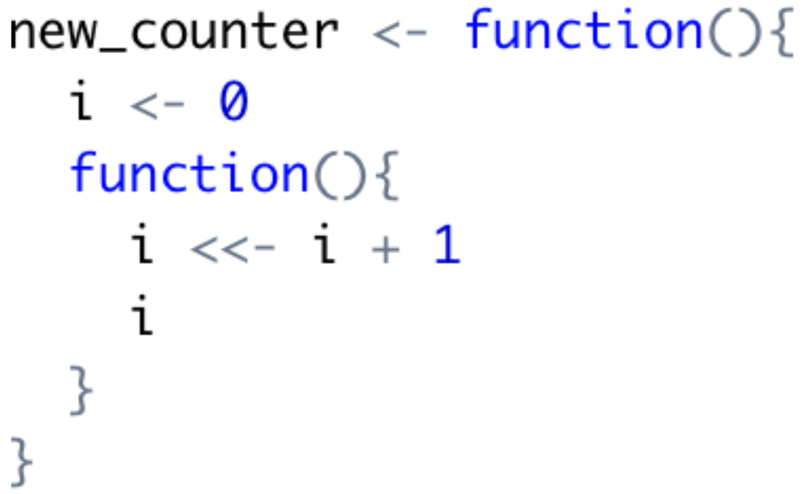
```
>
> square <- power(2)
>
> square(4)
[1] 16
```



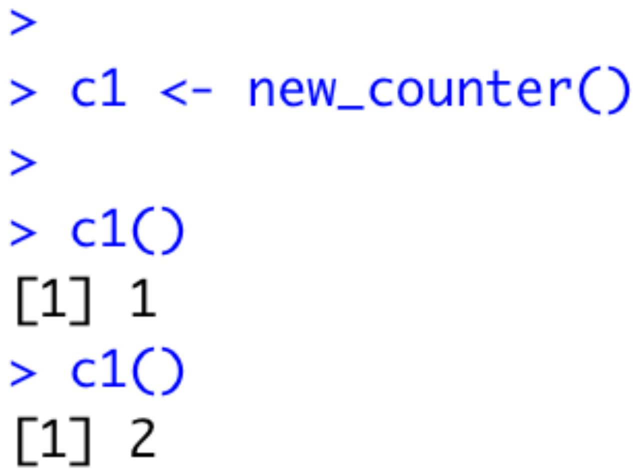
```
>
> cube <- power(3)
>
> cube(3)
[1] 27
```

Closures - Mutable State

- Having variables at two levels allows you to ***maintain state across function invocations***
- This is possible because the enclosing environment is constant
- Managing variables at different levels is possible using the super-assignment operator <<-



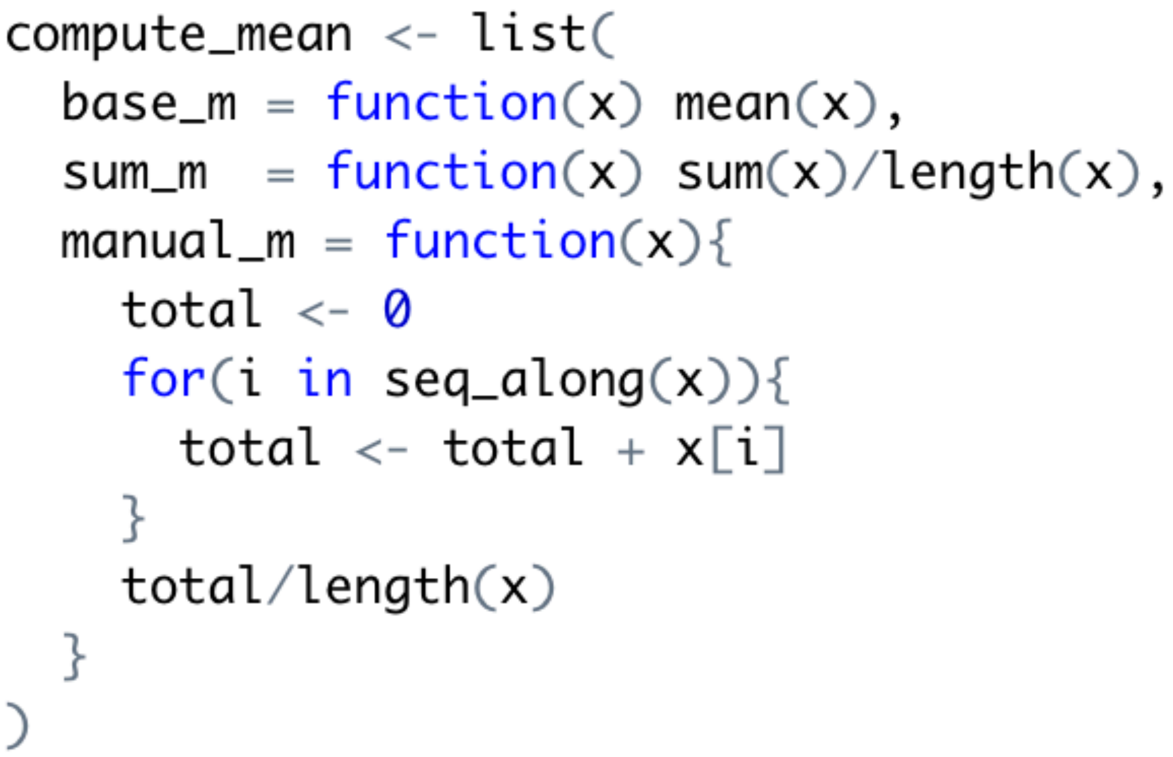
```
new_counter <- function(){
  i <- 0
  function(){
    i <<- i + 1
    i
  }
}
```



```
>
> c1 <- new_counter()
>
> c1()
[1] 1
> c1()
[1] 2
```

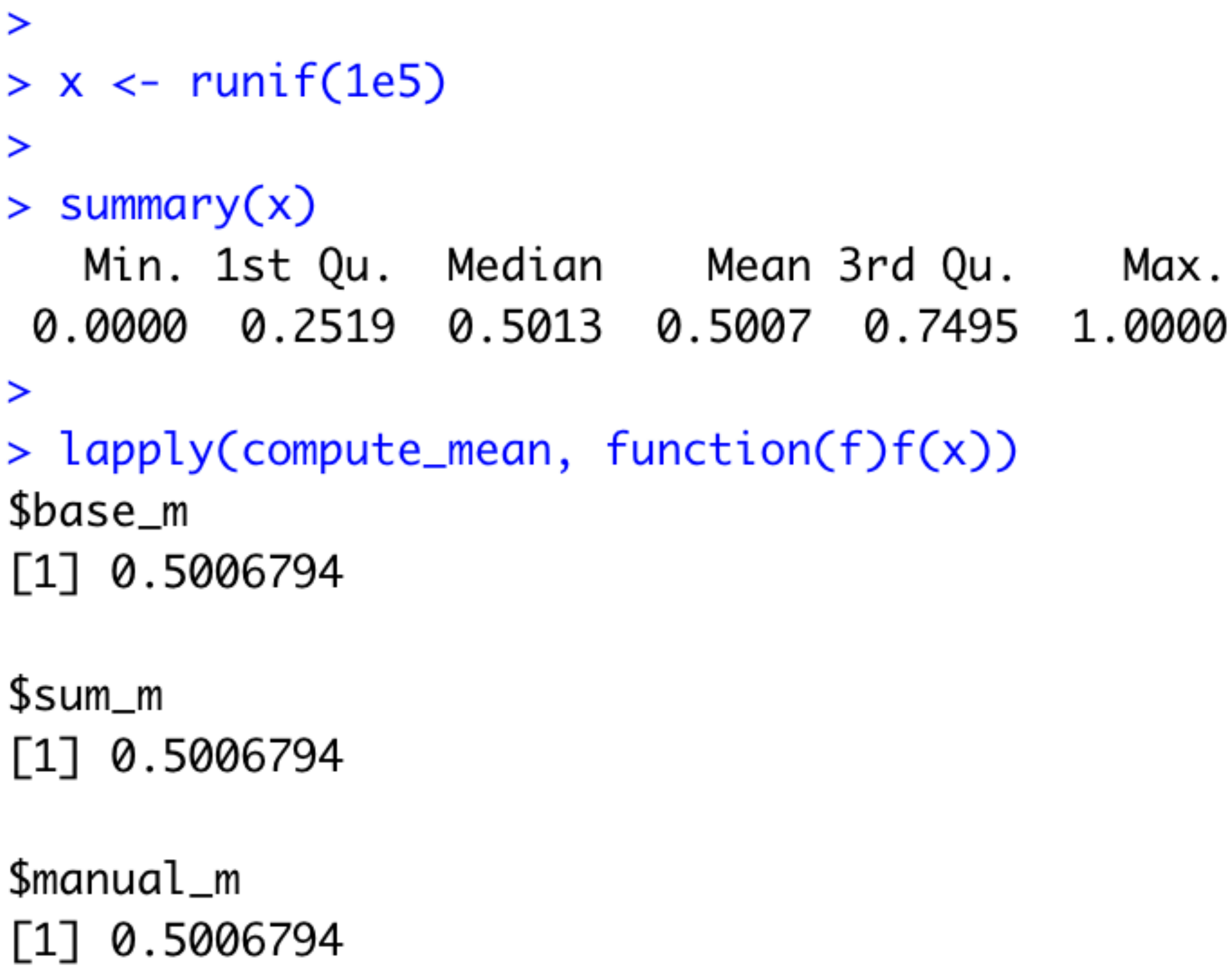
Lists of Functions

- In R, functions can be stored in lists.
- This makes it easier to work with groups of related functions



```
compute_mean <- list(
  base_m = function(x) mean(x),
  sum_m  = function(x) sum(x)/length(x),
  manual_m = function(x){
    total <- 0
    for(i in seq_along(x)){
      total <- total + x[i]
    }
    total/length(x)
  }
)
```

Use of lapply(flist,f)



```
>
> x <- runif(1e5)
>
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 0.0000  0.2519  0.5013  0.5007  0.7495  1.0000 
>
> lapply(compute_mean, function(f)f(x))
$base_m
[1] 0.5006794

$sum_m
[1] 0.5006794

$manual_m
[1] 0.5006794
```

Closure Example

Write a function factory (named stock) that represents state information on a stock keeping unit (SKU), and provides methods to change the state.

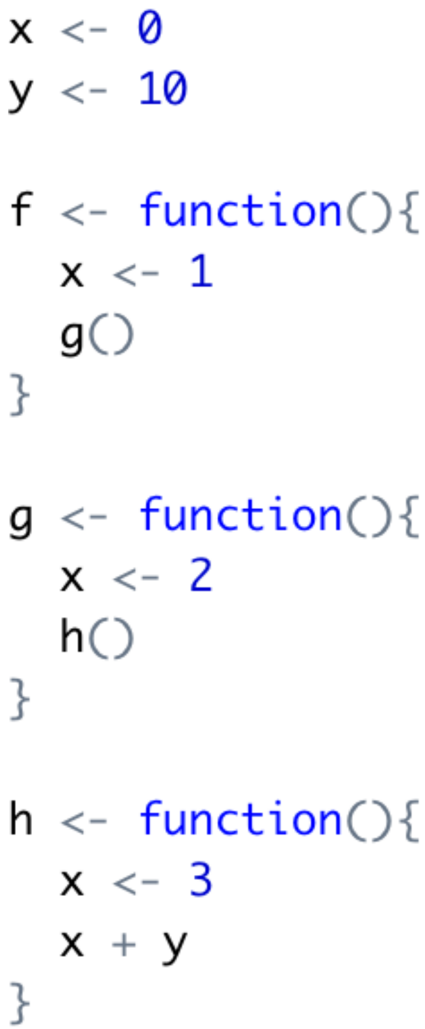
The state for the closure is *stock_id* and *quantity_on_hand.* Assume that the initial quantity on hand is zero.

There are three functions (returned in a list by the function factory) that can change/enquire of the state. These are:

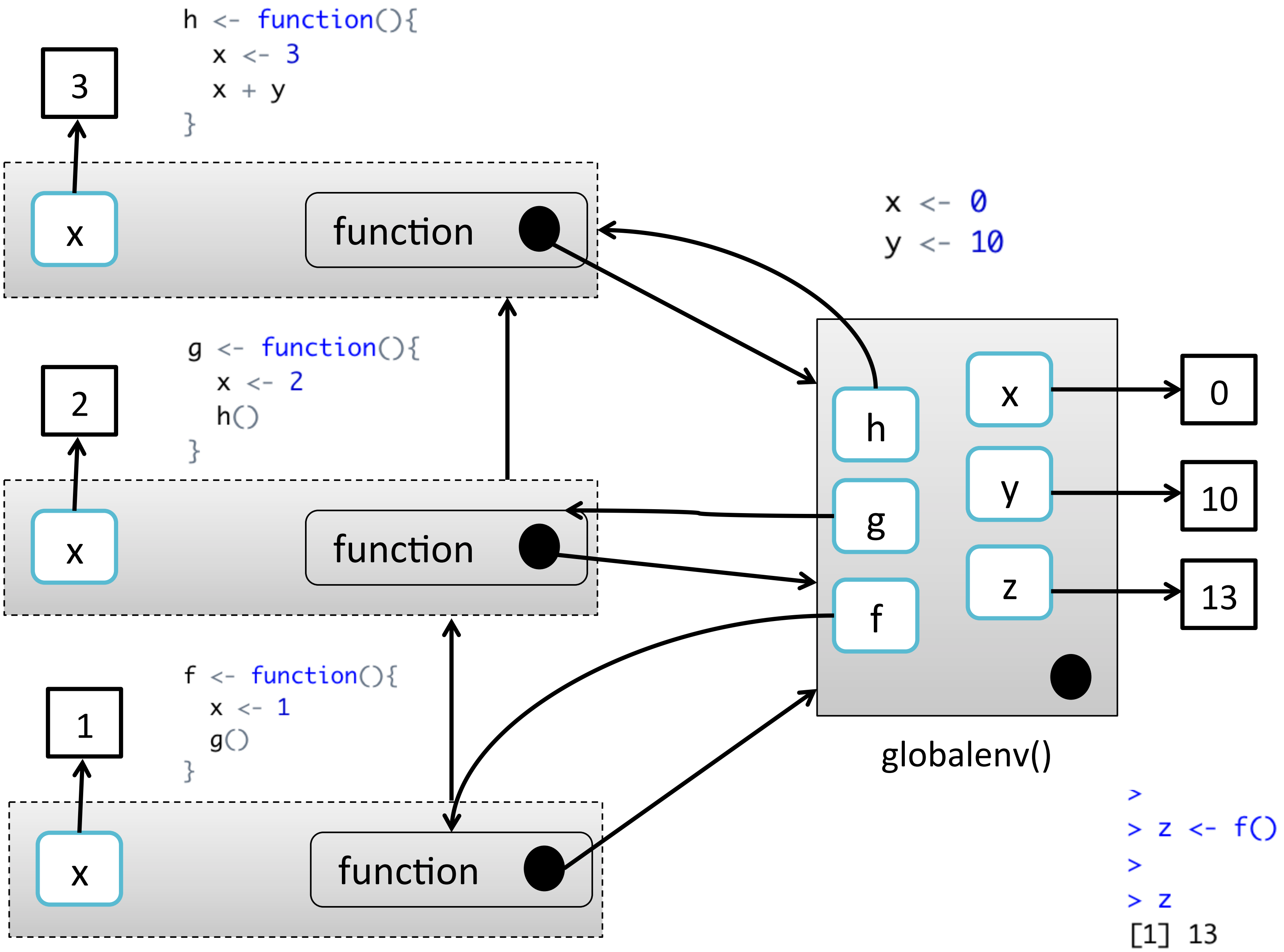
- *increment(n),* which increases the quantity on hand by *n*
- *decrement(n),* which reduces the quantity on hand by *n*
- *onhand(n),* which returns the number of items in stock

(4) Calling environments

What will f() return when the code is run?



```
x <- 0
y <- 10

f <- function(){
  x <- 1
  g()
}

g <- function(){
  x <- 2
  h()
}

h <- function(){
  x <- 3
  x + y
}
```



```
h <- function(){
  x <- 3
  x + y
}
```

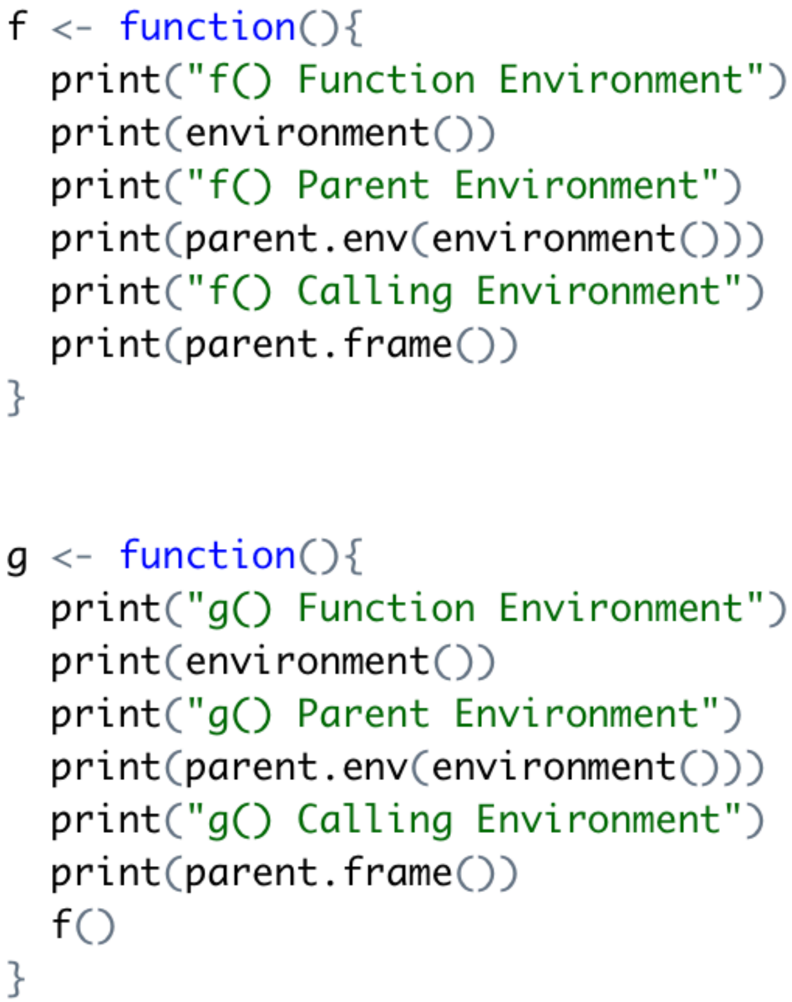
```
g <- function(){
  x <- 2
  h()
}
```

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

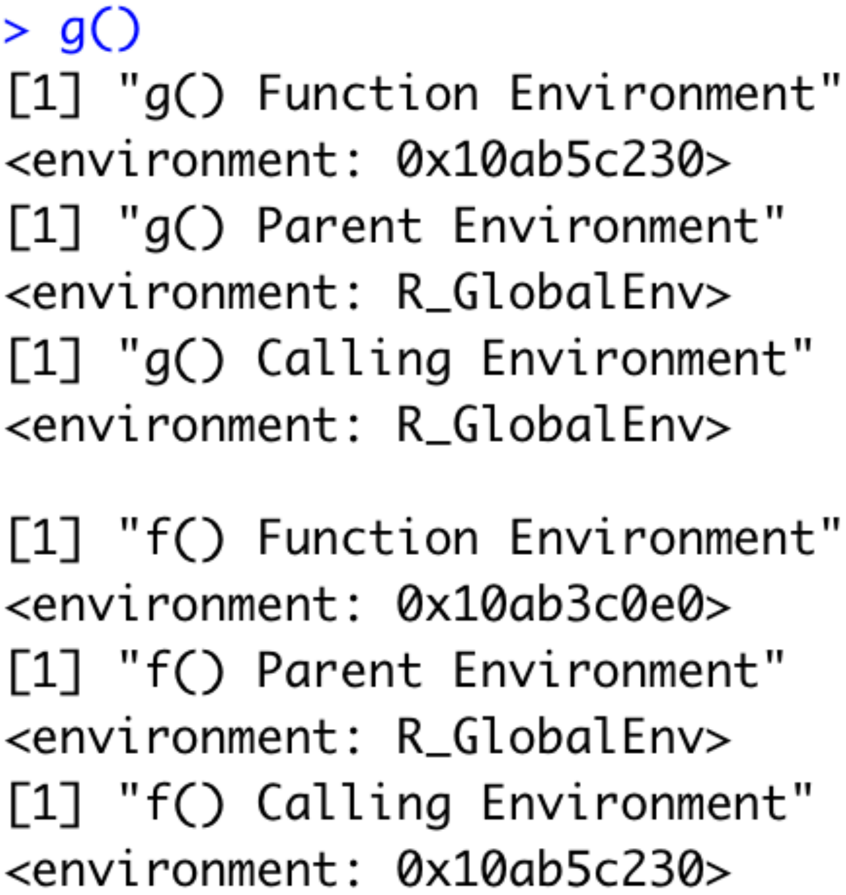
```
x <- 0
y <- 10
```

```
>
> z <- f()
>
> z
[1] 13
```

Displaying Environment info.



```
f <- function(){
  print("f() Function Environment")
  print(environment())
  print("f() Parent Environment")
  print(parent.env(environment()))
  print("f() Calling Environment")
  print(parent.frame())
}

g <- function(){
  print("g() Function Environment")
  print(environment())
  print("g() Parent Environment")
  print(parent.env(environment()))
  print("g() Calling Environment")
  print(parent.frame())
  f()
}
```



```
> g()
[1] "g() Function Environment"
<environment: 0x10ab5c230>
[1] "g() Parent Environment"
<environment: R_GlobalEnv>
[1] "g() Calling Environment"
<environment: R_GlobalEnv>

[1] "f() Function Environment"
<environment: 0x10ab3c0e0>
[1] "f() Parent Environment"
<environment: R_GlobalEnv>
[1] "f() Calling Environment"
<environment: 0x10ab5c230>
```

References

- Wickham, H. 2015. Advanced R. Taylor & Francis