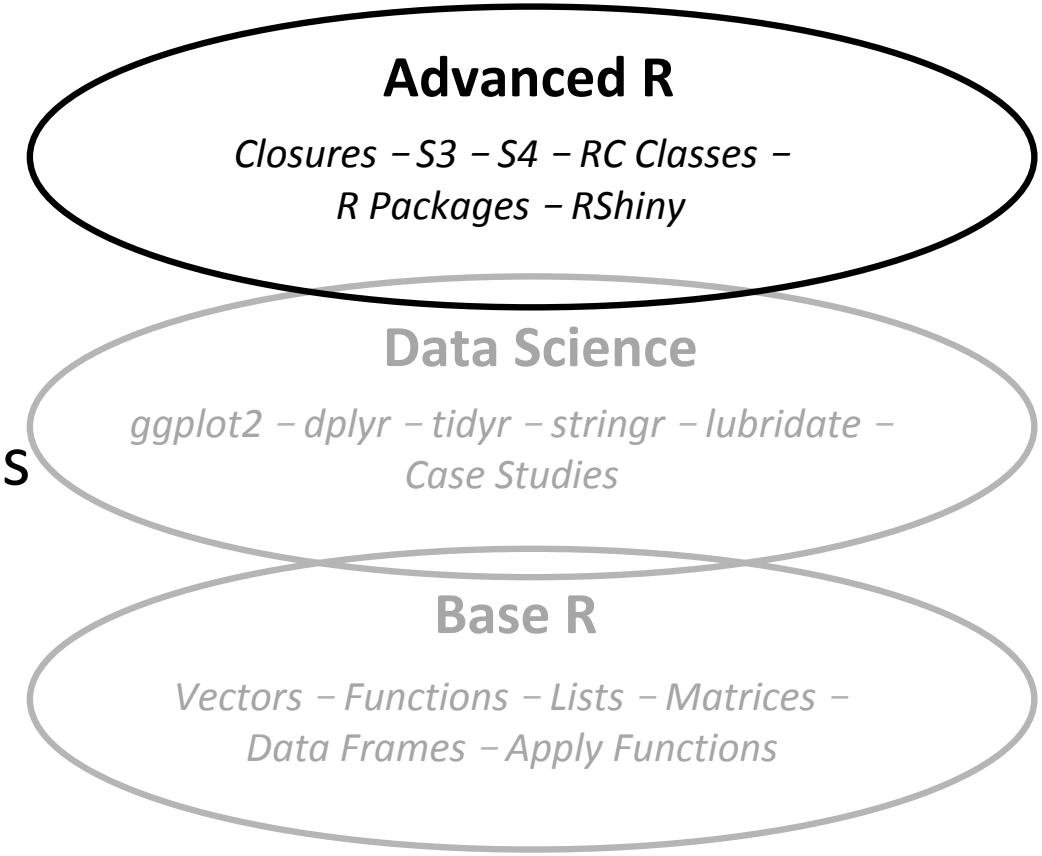
CT5102: Programming for Data Analytics

Lecture 11: Environments & Functions

Dr. Jim Duggan,
School of Engineering & Informatics
National University of Ireland Galway.
https://github.com/JimDuggan/CT5102

Overview

- Functions
- Environments
- Function Environments
- Closures



(1) Functions - Components

- The body(), the code inside the function
- The formals(), the list of arguments which controls how you call the function
- The environment, the "map" of the location of the function's variables
- Functions can also possess a number of attributes, "srcref" is one which points to the source code

```
> f <- function(x) x^2
>
> formals(f)
$x


>
> body(f)
x^2
>
> environment(f)
<environment: R_GlobalEnv>
>
```

Primitive Functions

- One exception to the rule that functions have three components.
- Primitive functions, like sum(), call C code directly with .Primitive() and contain no R code
- Primitive functions are only found in the base package.

```
> formals(sum)
NULL
>
> body(sum)
NULL
>
> environment(sum)
NULL
```

Functions – Lexical Scoping

- Scoping is the set of rules that govern how R looks up the value of a symbol
- If a name is not defined in a function, R will look one level up

```
x <- 2
g <- function(){
  y <- 1
  c(x,y)
}

g()
```

```
> g()
[1] 2 1
```

Values between invocation calls?

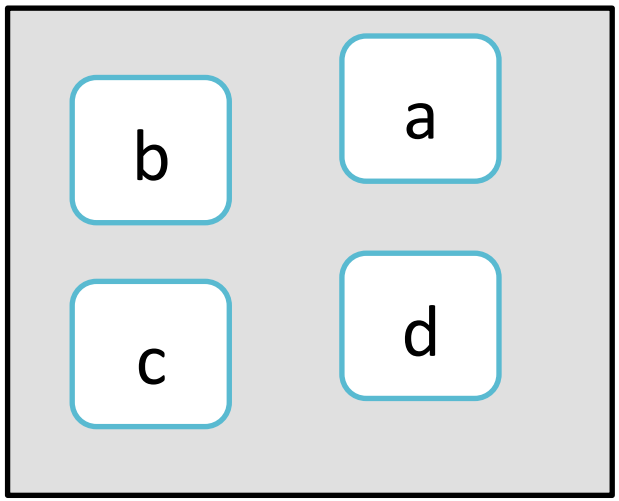
- Every time a function is called, a new environment is created to host execution
- A function has no way to know what happened the last time it was run
- Each invocation is completely independent

```
j <- function(){
  if(!exists("a")){
    a <- 1
  } else{
    a <- a + 1
  }
  print(a)
}
```

```
> j()
[1] 1
>
> j()
[1] 1
```

(2) 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)

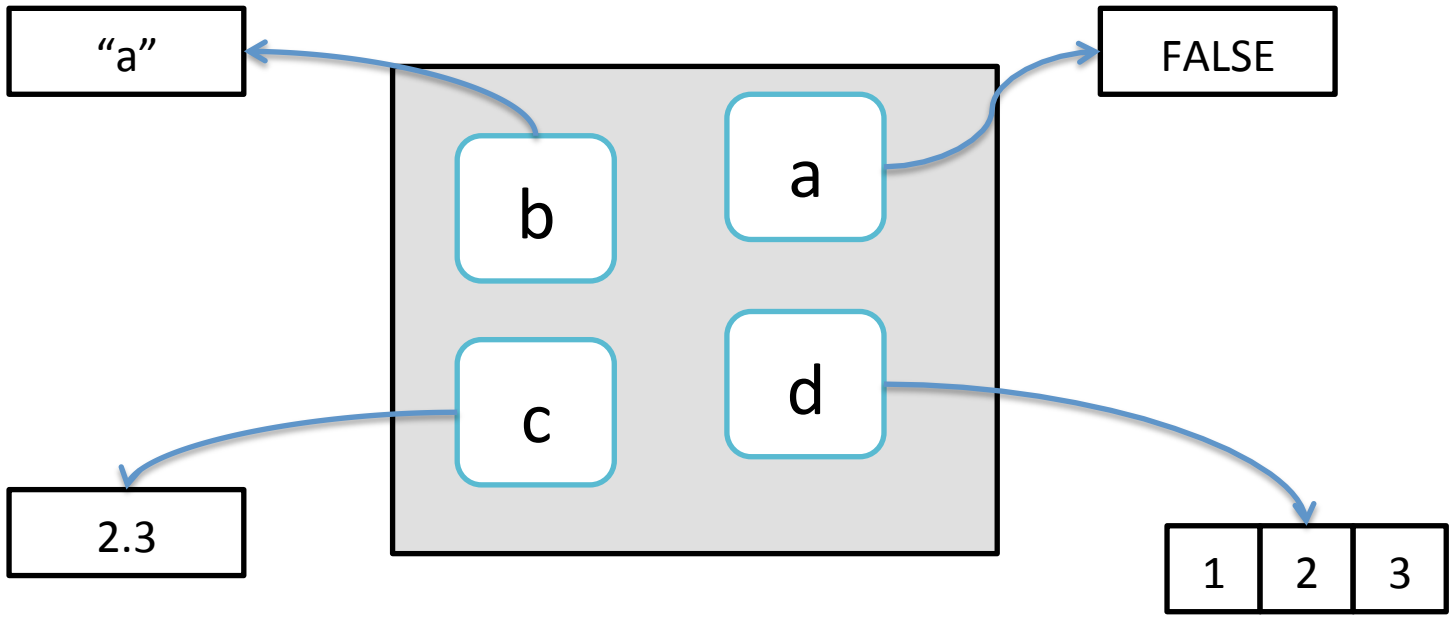


Each name points to an object stored elsewhere in memory

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

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

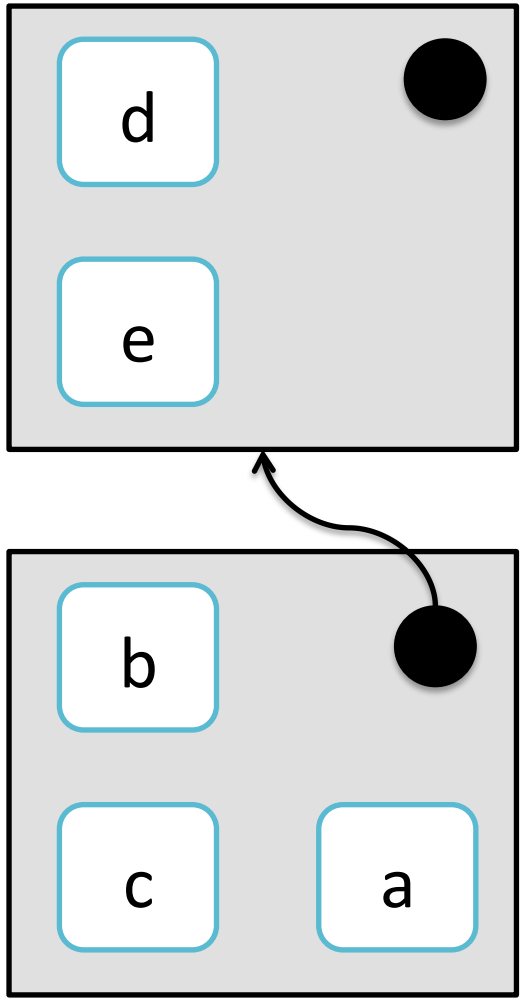
```
e$c <- 2.3
e$d <-1:3
```




```
> 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.

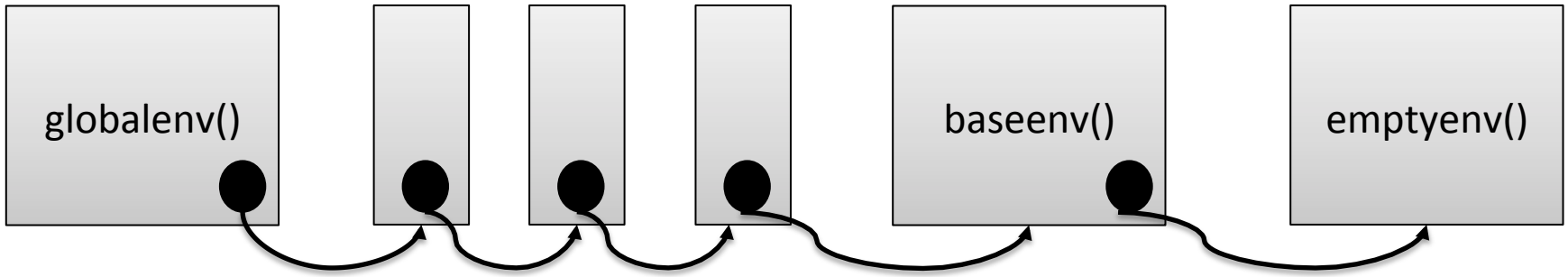
There are 4 special environments

- **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: basenv()

```
> 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

```
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

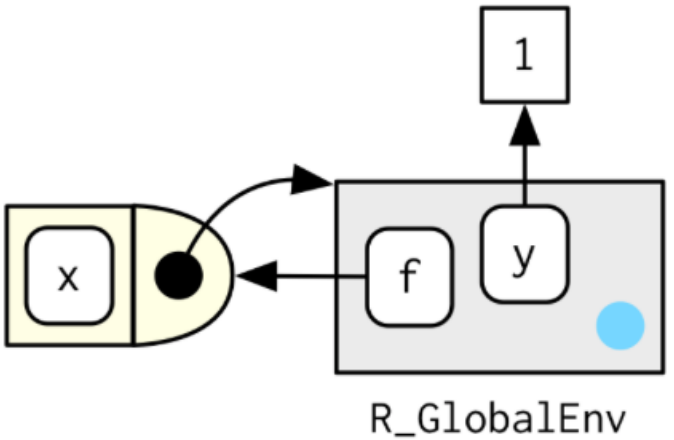
```
x <- 10

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

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

(3) Function Environments

- A function binds the current environment when it is created.
- In diagrams, functions are depicted as rectangles with a rounded end that binds an environment
- Also referred to as the binding environment

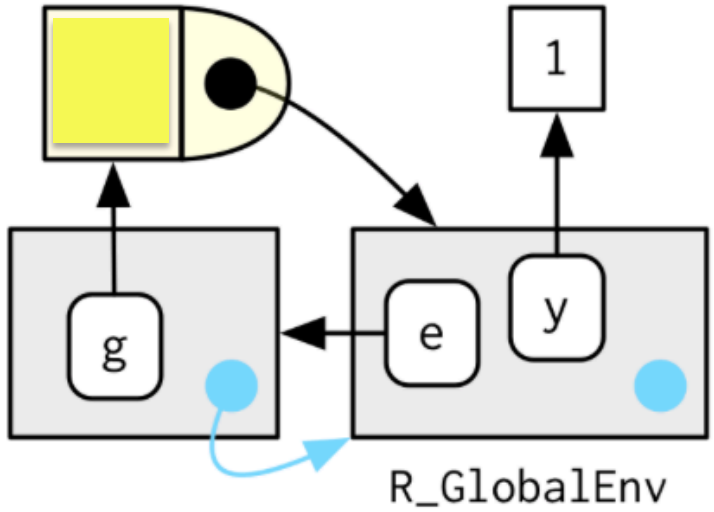
```
> y <- 1
>
> f <- function(x)x+y
>
> f(10)
[1] 11
```



https://adv-r.hadley.nz/environments.html#the-function-environment

Binding Environment Example

```
> e <- new.env()
>
> e$g <- function()1
>
> e$g()
[1] 1
```



https://adv-r.hadley.nz/environments.html#the-function-environment

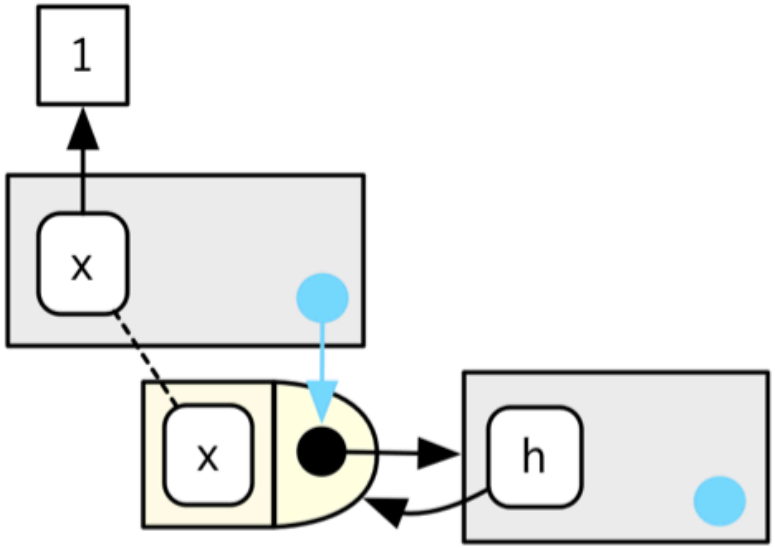
Execution Environments

- 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)
```

Step 1

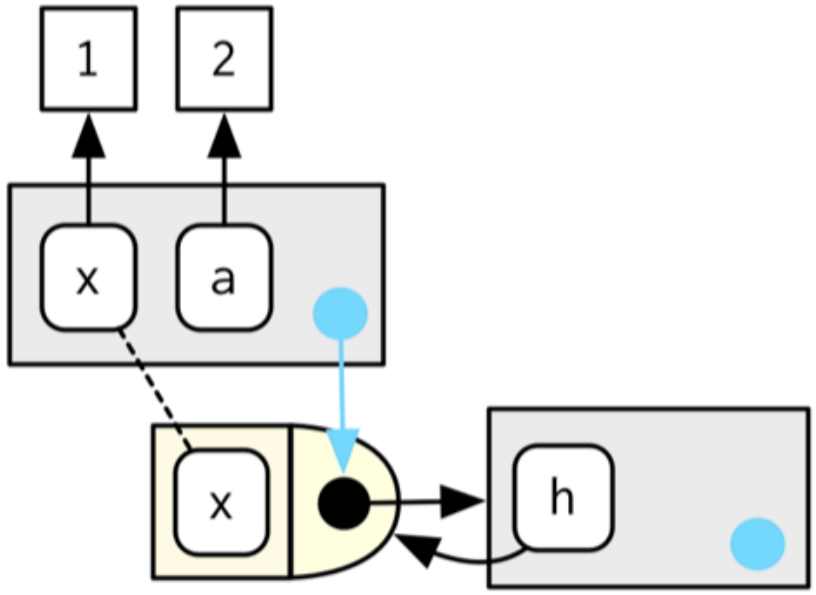
1. Function called with x = 1




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

y <- h(1)
```

Step 2

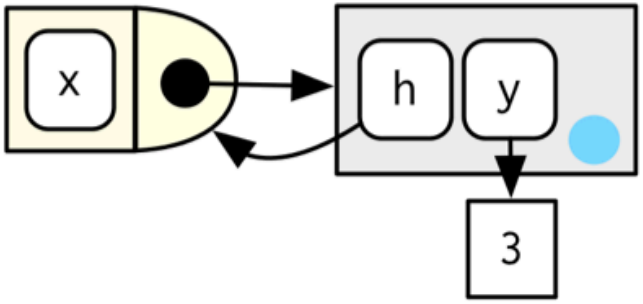
2. a bound to value 2




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

y <- h(1)
```

Step 3

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



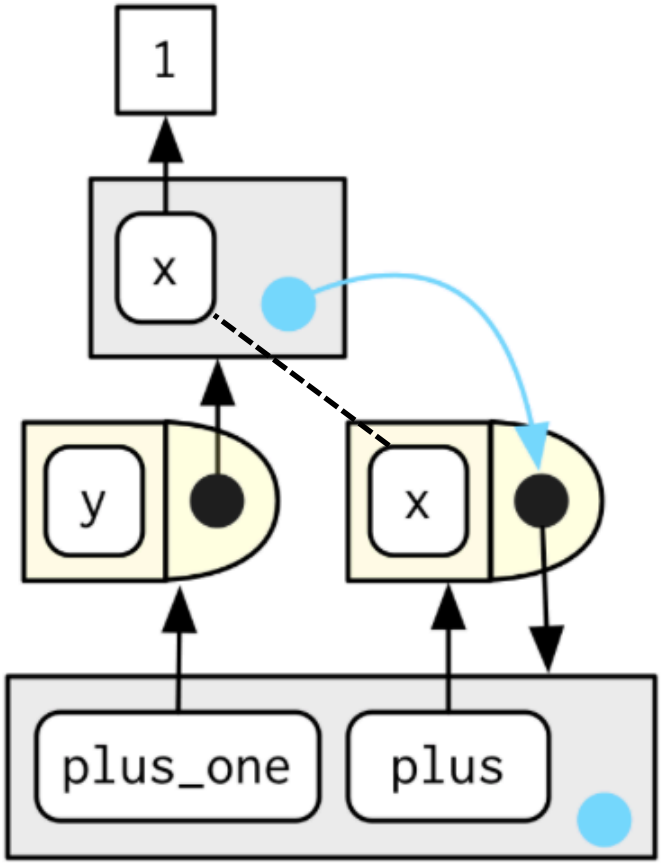

```
h <- function(x){
  a <- 2
  x + a
}

y <- h(1)
```

Additional 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

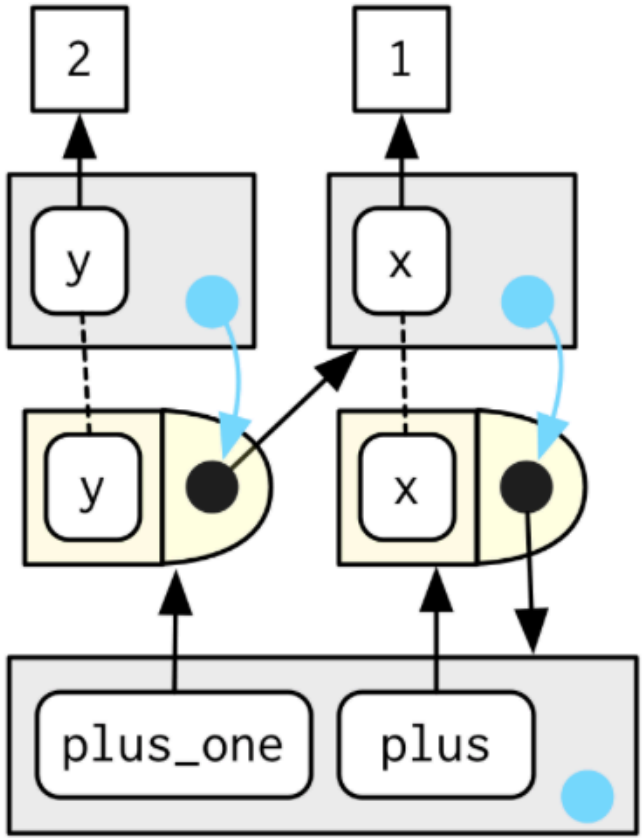
```
plus <- function(x){
  function(y)x+y
}

plus_one <- plus(1)
```

Calling plus(1)




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

plus_one <- plus(1)
```

Calling plus_one(2)




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

plus_one <- plus(1)

plus_one(2)
```

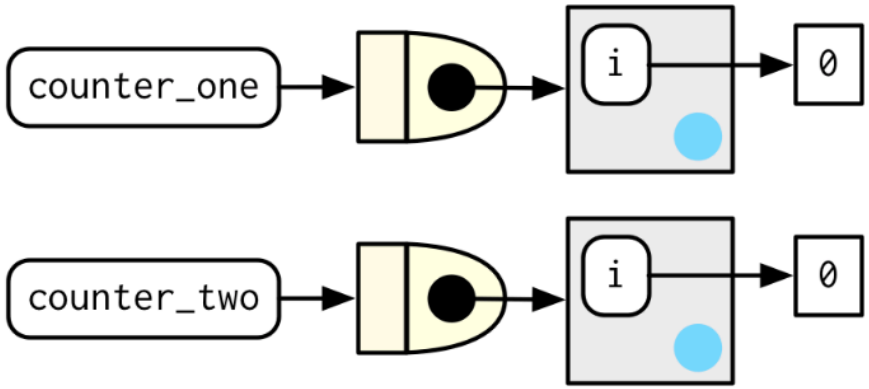
Closures

"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
- "Stateful functions (closures) are best used in moderation. As soon as your function starts managing the state of multiple variables, it's better to switch to R6" (Wickham)

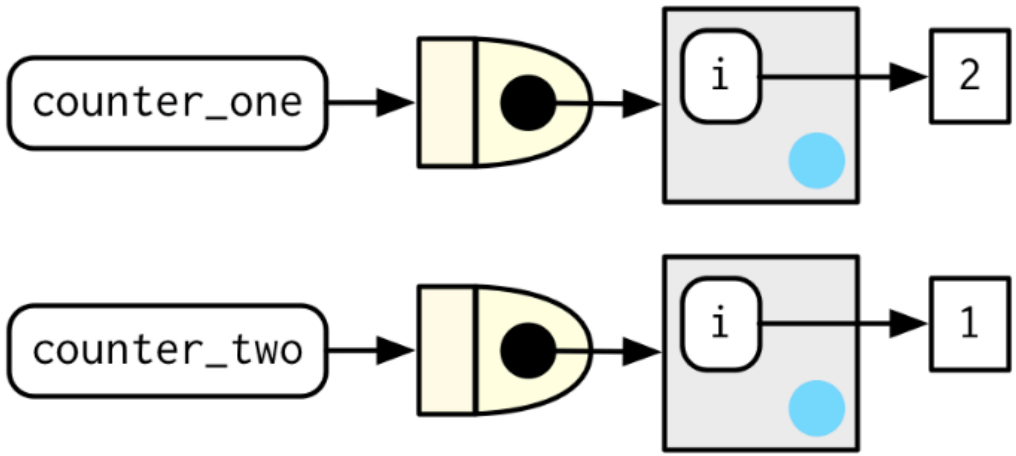
Provides mutable state

```
new_counter <- function() {
  i <- 0

  function() {
    i <<- i + 1
    i
  }
}

counter_one <- new_counter()
counter_two <- new_counter()
```



State is maintained between function calls…

```
> counter_one()
[1] 1
>
> counter_one()
[1] 2
>
> counter_two()
[1] 1
```



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
```

References

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