Functional programming

Hadley Wickham
Chief Scientist
RStudio

May 2015

1. Motivation
2. Warmups
3. Functionals
4. Friends of `lapply()`
5. Anonymous functions
6. Function factories

Motivation

DRY principle: Don’t Repeat Yourself

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Popularised by the “Pragmatic Programmers”

```
# Fix missing values
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -99] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$f == -99] <- NA
df$g[df$g == -98] <- NA
df$h[df$h == -99] <- NA
df$i[df$i == -99] <- NA
df$i[df$j == -99] <- NA
df$k[df$k == -99] <- NA
```

DRY principle prevents inconsistency

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$f)
df$g <- fix_missing(df$g)
df$h <- fix_missing(df$h)
df$h <- fix_missing(df$i)
df$j <- fix_missing(df$j)
df$k <- fix_missing(df$k)
```

More powerful abstractions lead to less repetition

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}

df[] <- lapply(df, fix_missing)
```

And easier generalisation

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}

numeric <- vapply(df, is.numeric, logical(1))
df[numeric] <- lapply(df[numeric], fix_missing)
```

And easier generalisation

```
missing_fixer <- function(missing) {
  function(x) {
    x[x == missing] <- NA
    x
  }
}

numeric <- vapply(df, is.numeric, logical(1))
df[numeric] <- lapply(df[numeric], missing_fixer(-99))
```

Warmups

What does this function return?

```
x <- 5
f <- function() {
  x <- 20
  y <- 10
  c(x = x, y = y)
}
f()
```

```r
x <- 5
f <- function() {
  x <- 20
  y <- 10
  c(x = x, y = y)
}
f()
```

```
 x  y
20 10
```

What does this function return?

```
x <- 5
g <- function() {
  y <- 10
  c(x = x, y = y)
}
g()
```

```r
x <- 5
g <- function() {
  y <- 10
  c(x = x, y = y)
}
g()
```

```
 x  y
 5 10
```

What does this function return?

```
x <- 5
h <- function() {
  y <- 10
  i <- function() {
    z <- 20
    c(x = x, y = y, z = z)
  }
  i()
}
h()
```

```r
x <- 5
h <- function() {
  y <- 10
  i <- function() {
    z <- 20
    c(x = x, y = y, z = z)
  }
  i()
}
h()
```

```
 x  y  z
 5 10 20
```

```r
j <- function() {
  if (!exists(”a”)) {
    a <- 5
  } else {
    a <- a + 1
  }
  print(a)
}
j()

j()
```

What does this function return the first time you run it? The second time?

```r
j <- function() {
  if (!exists("a")) {
    a <- 5
  } else {
    a <- a + 1
  }
  print(a)
}
j()
[1] 5
j()
[1] 5
```

What does this function return the first time you run it? The second time?

Functionals

Functionals

- A **functional** is a function that takes a function as input and returns a vector.
- Functionals are used to abstract over common patterns of looping.
- Common functions are `lapply()`, `apply()`, `tapply()`, ...
- Reduce bugs by better communicating intent.

```r
set.seed(1014)

# Create some random output:
# 20 random vectors with random lengths

l <- replicate(20, runif(sample(1:10, 1)),
  simplify = FALSE)
str(l)
l
```

```r
# Extract length of each element
lengths <- vector("list", length(l))
for (i in seq_along(l)) {
  lengths[[i]] <- length(l[[i]])
}
lengths
```

Preallocating space for output saves a lot of time

```
# Extract length of each element
lengths <- vector("list", length(l))
for (i in seq_along(l)) {
  lengths[[i]]
}
lengths
```

Safe shortcut for `1:length(l)`

How would you change this to compute the mean of each element?

```
# Extract length of each element
lengths <- vector("list", length(l))
for (i in seq_along(l)) {
  lengths[[i]] <- length(l[[i]])
}
lengths
```

```
# Compute mean of each element
means <- vector("list", length(l))
for (i in seq_along(l)) {
  means[[i]] <- mean(l[[i]])
}
means
```

Your turn

What are the common parts of this pattern? How could you extract them into a function?

```
do_each <- function(?, ?) {
  ...
}

do_each(l, length)
do_each(l, mean)
```

No peeking until you’ve made an attempt!

Functions can be arguments!

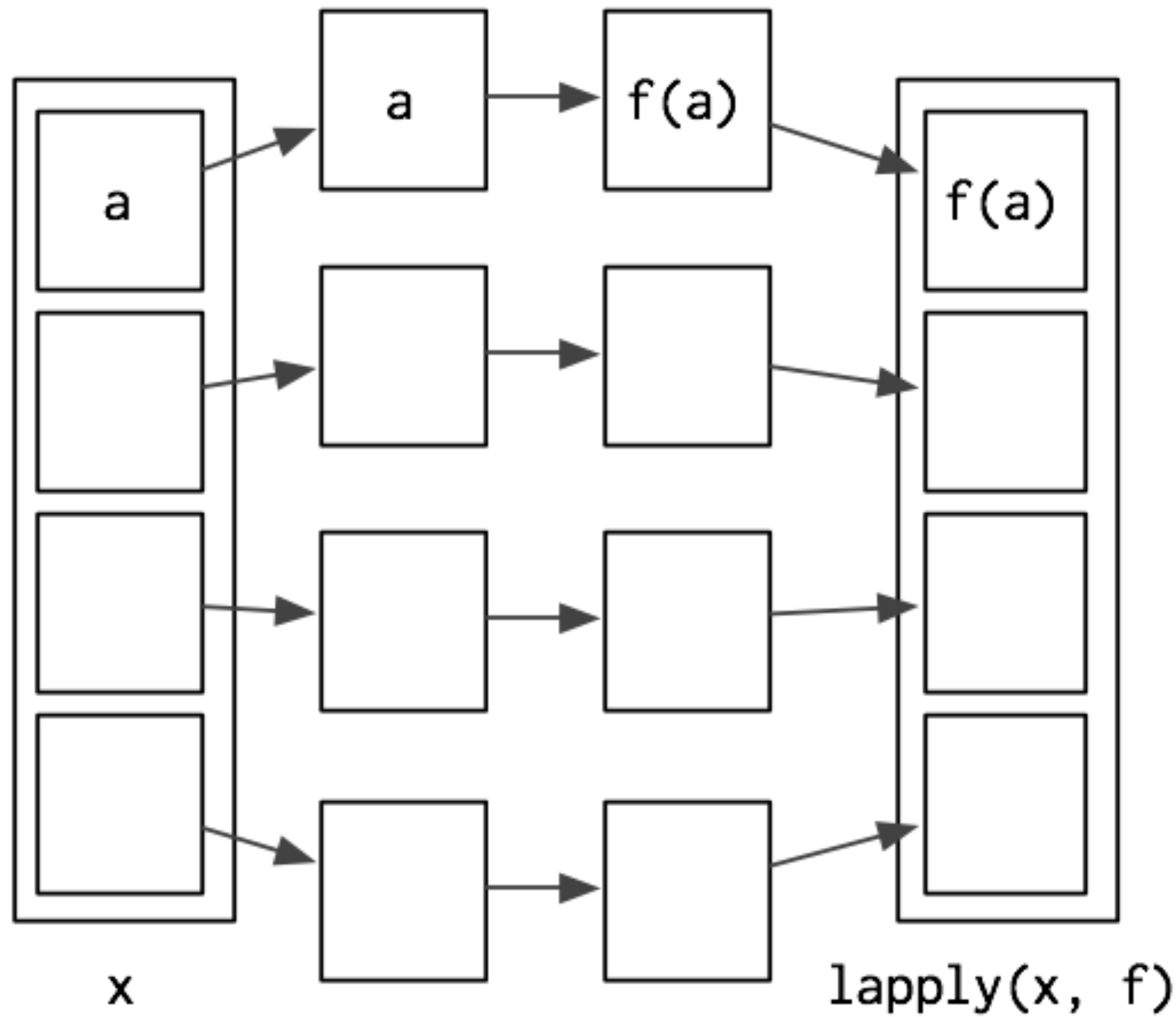
```
do_each <- function(x, f) {
 out <- vector("list", length(x))
  for(i in seq_along(x)) {
    out[[i]] <- f(x[[i]])
  }
  out
}

do_each(l, length)
do_each(l, mean)
do_each(l, min)
```

```
# BUT WAIT...

lapply(l, length)
lapply(l, mean)
lapply(l, min)

# We've just reinvented lapply :)
# Two differences:
# * lapply() uses some C tricks to be faster
# * lapply() passes ... on to f
```


a
a
f(a)
f(a)
x
lapply(x, f)

Placeholder for “any other” arguments

```
do_each <- function(x, f, ...) {
  out <- vector("list", length(x))
  for(i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}

do_each(l, mean, trim = 0.5)
```

Your turn

The function below scales a vector so it falls in the range [0, 1]. How would you apply it to every column of a data frame?

```
scale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

```r
mtcars <- lapply(mtcars, scale01)
mtcars # a list :(
rm(mtcars)

mtcars[] <- lapply(mtcars, scale01)
mtcars # a data frame :)
rm(mtcars)

for(i in seq_along(mtcars)) {
  mtcars[[i]] <- scale01(mtcars[[i]])
}
```

Friends of `lapply()`

Variation	Function
I want an atomic vector, not a list	`sapply()`, `vapply()`
I have more than one input	`Map()`, `mapply()`
I have lots of computation to do	`mclapply()`

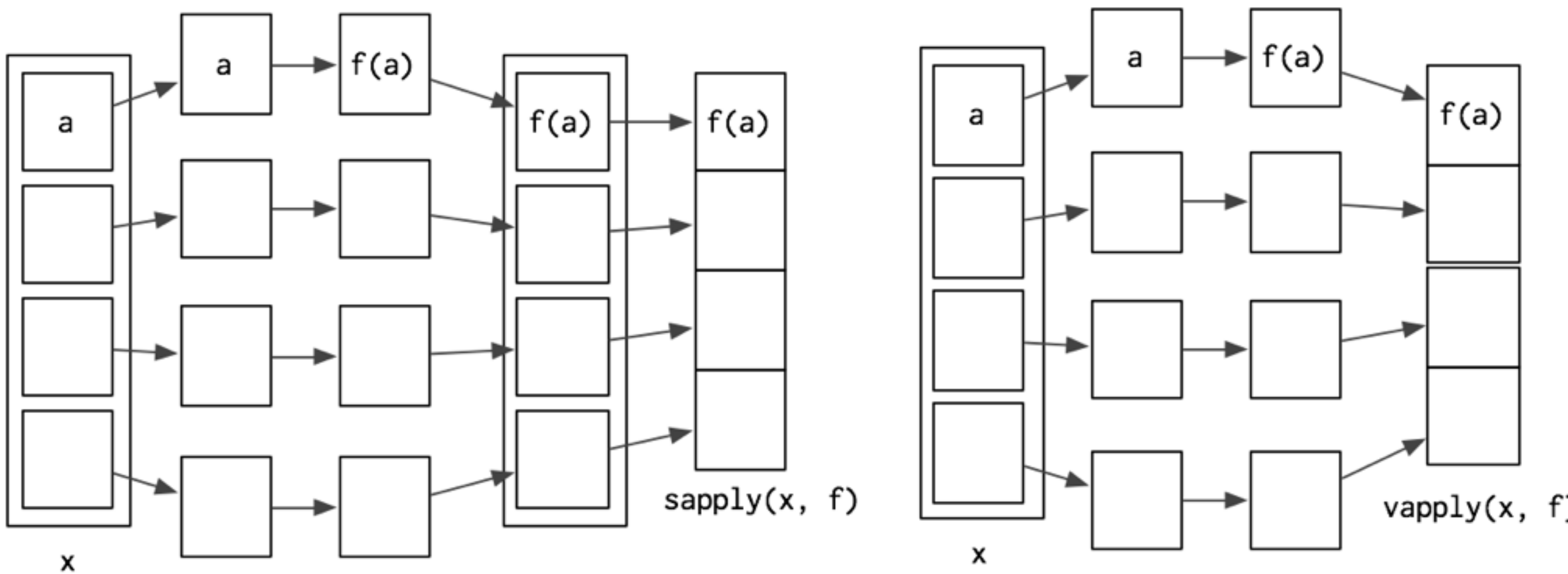
```r
# Output is annoyingly long!
lapply(mtcars, is.numeric)
# We'd prefer a vector instead of a list

sapply(mtcars, is.numeric)
vapply(mtcars, is.numeric, logical(1))

# sapply() is useful interactively, but
# dangerous in a function because it has
# to guess the output type.
```

```
sapply(mtcars, class)

mtcars$x <- Sys.now()
sapply(mtcars, class)
```



a
a
f(a)
f(a)
f(a)
sapply(x, f)
x
a
a
f(a)
f(a)
vapply(x, f)
x

```r
sapply2 <- function(x, f, ...) {
  res <- lapply2(x, f, ...)
  simplify2array(res)
}

vapply2 <- function(x, f, f.value, ...) {
  out <- matrix(rep(f.value, length(x)), nrow = length(x))
  for (i in seq_along(x)) {
    res <- f(x[[i]], ...)
    stopifnot(
      length(res) == length(f.value),
      typeof(res) == typeof(f.value)
    )
    out[i, ] <- res
  }
  out
}
```

`sapply()`

Never use inside a function!

```r
col_means <- function(df) {
  numeric <- vapply(df, is.numeric, logical(1))
  numeric_cols <- df[, numeric, drop = FALSE]

  data.frame(lapply(numeric_cols, mean))
}
```

Anonymous functions

```r
# Don't have to name functions to use with
# apply etc. Can create an inline anonymous
# function:

lapply(mtcars, function(x) length(unique(x)))

integrate(function(x) sin(x)^2, 0, pi)
```

```r
# Creating an anonymous function
function(x) 3

# Calling an anonymous function
(function(x) 3)()
# Not:
function(x) 3 ()

# Anonymous functions work just like ordinary
# functions
formals(function(x = 4) g(x) + h(x))
body(function(x = 4) g(x) + h(x))
environment(function(x = 4) g(x) + h(x))
```

```r
# Functions that input and output
# functions are called function operators.
# They abstract away common uses of anonymous
# functions

library(pryr)

sapply(mtcars, function(x) length(unique(x)))
sapply(mtcars, compose(length, unique))

sapply(mtcars, function(x) mean(x, trim = 0.2))
sapply(mtcars, partial(mean, trim = 0.2))

# Allows very expressive code, where the
# focus is on the operations, not the arguments
```

	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operators

Function factories

```r
x <- 0
y <- 10
f <- function() {
  x <- 1
  function() {
    y <- 2
    x + y
  }
}

# What does f() return?
# What does f()() mean? What does it do?
# How does it work?
```

```r
f <- function() {
  x <- sample(1000, 1)
  function() {
    x + 2
  }
}

f1 <- f()
f2 <- f()
f1()
f1()
f2()
```

Scoping

R uses lexical scoping: variable lookup is based on where functions were created.

If a variable isn’t found in the current environment, R looks in the parent: the environment where the function was created.

Anonymous functions remember their parent environment, even if it has since “disappeared”.

```r
# Closures are useful when you want a function
# that can create a whole class of functions:
# a function factory

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

square <- power(2)
square(2)
square(4)

cube <- power(3)
cube(2)
cube(4)
```

```
square

# We can find the environment and its parent
environment(square)
as.list(environment(square))
as.list(environment(cube))

pryr::unenclose(square)
pryr::unenclose(cube)
```

```
missing_fixer <- function(missing) {
  force(missing)
  function(x) {
    x[x == missing] <- NA
    x
  }
}

missing_vals <- c(-99, -99, -9000, -90)
fixers <- lapply(missing_vals, missing_fixer)
invoke <- function(f, ...) f(...)

Map(invoke, fixers, df)
->
invoke(fixers[[1]], df[[1]]) -> fixers[[1]](df[[1]])
invoke(fixers[[2]], df[[2]]) -> fixers[[2]](df[[2]])
```

Learning more

Advanced R

http://adv-r.had.co.nz/Functions.html

http://adv-r.had.co.nz/Functional-programming.html

http://adv-r.had.co.nz/Functionals.html

http://adv-r.had.co.nz/Function-operators.html