First class functions

Hadley Wickham
Assistant Professor / Dobelman Family Junior Chair
Department of Statistics / Rice University

August 2011

1. Motivation
2. First class functions
3. Closures
4. Higher-order functions
5. Lists of functions

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
```

```r
# 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)
```

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

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

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

```
mean(df$a)
median(df$a)
sd(df$a)
mad(df$a)
IQR(df$a)

mean(df$b)
median(df$b)
sd(df$b)
mad(df$b)
IQR(df$b)

mean(df$c)
median(df$c)
sd(df$c)
mad(df$c)
IQR(df$c)
```

What are the two sources of repetition in this code? Discuss with your neighbour for 1 minute.

```
summary <- function(x) {
 c(mean(x, na.rm = TRUE),
   median(x, na.rm = TRUE),
   sd(x, na.rm = TRUE),
   mad(x, na.rm = TRUE),
   IQR(x, na.rm = TRUE))
}

summary(df$a)
summary(df$b)
summary(df$c)
```

In this session we'll learn new tools for dealing with repetition of functions

```
summary <- function(x) {
 c(mean(x, na.rm = TRUE),
   median(x, na.rm = TRUE),
   sd(x, na.rm = TRUE),
   mad(x, na.rm = TRUE),
   IQR(x, na.rm = TRUE))
}

summary(df$a)
summary(df$b)
summary(df$c)
```

```
summary <- function(x) {
 c(mean(x), median(x), sd(x), mad(x), IQR(x))
}

summary(df$a)
summary(df$b)
summary(df$c)
```

First class functions

1. Functions don't need names (anonymous functions)
2. Functions can be written by other functions (closures)
3. Functions can take functions as arguments (higher-order functions)
4. Functions can be stored in other data structures

```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
# Useful for small, one-off tasks that don't
# merit creating a named function

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

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

Your turn

Given a name, how do you find that function? Given a function, how do you find its name?

Brainstorm with your neighbour for 1 minute.

Closures

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

g <- function() {
  x <- 20
  y <- 10
  c(x = x, y = y)
}
g()

# What do these functions return?
# How does variable lookup in R work?
```

```
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?
```

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”.

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

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)
parent.env(environment(square))

# Or inspect objects defined in that environment
ls(environment(square))
get("exponent", environment(square))
environment(square)$exponent
as.list(environment(square))
```

Your turn

Run the code on the following page. What does it do? How does it work? Why do the different counters not interfere with each other?

```
new_counter <- function() {
  i <- 0
  function() {
    # do something useful, then ...
    i <<- i + 1
    i
  }
}

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

counter_one()
counter_one()
counter_two()
```

Mutable state

Closures are one way of creating mutable state - the usual copy on modify semantics do not seem to apply here.

```
# Built in functions that make closures

Negate(is.numeric)("abc")
Negate

vrep <- Vectorize(rep.int, "times")
vrep(42, times = 1:4)
vrep
as.list(environment(vrep))

e <- ecdf(runif(1000))
str(e)
e(0.5)
class(e) # Functions can have classes too!
```

Higher-order functions

HOFs

Better vocab

Closures are most useful in conjunction with functions that take functions as arguments.

You're probably already familiar with a few: `lapply`, `sapply`, `apply`, `optimise`, ...

Two main camps: data structure manipulation and mathematical

```r
# Data structure HOFs
# Provide basic tools for when you have a predicate
# function instead of a logical vector.

# Filter: keeps true
# Find: value of first true
# Position: location of first true

Filter(is.factor, iris)
Find(is.factor, iris)
Position(is.factor, iris)

# One function I use a lot:
compact <- function(x) Filter(Negate(is.null), x)
```

```r
samples <- replicate(5, sample(10, 20, rep = T),
  simplify = FALSE)

# Want to find intersection of all values
int <- intersect(samples[[1]], samples[[2]])
int <- intersect(int, samples[[3]])
int <- intersect(int, samples[[4]])
int <- intersect(int, samples[[5]])

# Reduce recursively applies a function in this way
Reduce(intersect, samples)
```

```r
# Mathematical HOFs

integrate(sin, 0, pi)
uniroot(sin, pi * c(1 / 2, 3 / 2))
optimise(sin, c(0, 2 * pi))
optimise(sin, c(0, pi), maximum = TRUE)
```

```r
# Combination of closures and HOF particularly useful.
# For statistics, maximum likelihood estimation is a
# great example.

poisson_nll <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  function(lambda) {
    n * lambda - sum_x * log(lambda) # + ...
  }
}

nll1 <- poisson_nll(c(41, 30, 31, 38, 29, 24, 30, 29))
nll2 <- poisson_nll(c(6, 4, 7, 3, 3, 7, 5, 2, 2, 7))

optimise(nll1, c(0, 100))
optimise(nll2, c(0, 100))
```

Lists of functions

```r
compute_mean <- list(
  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x),
  manual = function(x) {
    total <- 0
    n <- length(x)
    for (i in seq_along(x)) {
      total <- total + x[i] / n
    }
    total
  }
)

call_fun <- function(f, ...) f(...)
x <- runif(1e6)
lapply(compute_mean, call_fun, x)
lapply(compute_mean, function(f) system.time(f(x)))
```

```
summary <- function(x) {
 c(mean(x, na.rm = TRUE),
   median(x, na.rm = TRUE),
   sd(x, na.rm = TRUE),
   mad(x, na.rm = TRUE),
   IQR(x, na.rm = TRUE))
}

summary(df$a)
summary(df$b)
summary(df$c)
```

Your turn

Modify the summary function to take a user specified list of functions.