Advanced R

Hadley Wickham
Chief Scientist
RStudio

August 2015

What is a function?

```r
add <- function(x, y) {
  x + y
}
formals(add)
body(add)
environment(add)

# Environment is important part of scoping.
# If can't find names inside function, next
# looks in the parent environment.

# Is the name of a function important?
```

What does this function return?

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

What does this function return?

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

```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
# What does this function do? How do you use it?
`last<-` <- function(x, value) {
  x[length(x)] <- value
  x
}

# What does this function do? How do you use it?
`%+%` <- function(a, b) {
  paste0(a, b)
}
```

Why write a function?

```
passionn <- min(comp$passion,na.rm=T)
passionx <- max(comp$passion,na.rm=T)-passionn

leadershipn <- min(comp$leadership,na.rm=T)
leadershipx <- max(comp$leadership,na.rm=T)-leadershipn

loyaltyn <- min(comp$loyalty,na.rm=T)
loyaltyx <- max(comp$loyalty,na.rm=T)-loyaltyn

basicServn <- min(comp$basicServ,na.rm=T)
basicServx <- max(comp$basicServ,na.rm=T)-basicServn

educationn <- min(comp$education,na.rm=T)
educationx <- max(comp$education,na.rm=T)-educationn

safetyn <- min(comp$safety,na.rm=T)
safetyx <- max(comp$safety,na.rm=T)-safetyn

...
```

```
cityagg <- ddply(dat,.(city),summarise,
  wt=sum(svywt),
  people=length(svywt),
  passion=sum(svywt*((passion-passionn)/passionx),na.rm=T)/sum(svywt[!is.na(
  leadership=sum(svywt*((leadership-leadershipn)/leadershipx),na.rm=T)/sum(s
  loyalty=sum(svywt*((loyalty-loyaltyn)/loyaltyx),na.rm=T)/sum(svywt[!is.na(
  basicServ=sum(svywt*((basicServ-basicServn)/basicServx),na.rm=T)/sum(svywt
  education=sum(svywt*((education-educationn)/educationx),na.rm=T)/sum(svywt
  safety=sum(svywt*((safety-safetyn)/safetyx),na.rm=T)/sum(svywt[!is.na(safe
  aesthetic=sum(svywt*((aesthetic-aestheticn)/aestheticx),na.rm=T)/sum(svywt
  economy=sum(svywt*((economy-economyn)/economyx),na.rm=T)/sum(svywt[!is.na(
  socialOff=sum(svywt*((socialOff-socialOffn)/socialOffx),na.rm=T)/sum(svywt
  civicInv=sum(svywt*((civicInv-civicInvn)/civicInvx),na.rm=T)/sum(svywt[!is
  openness=sum(svywt*((openness-opennessn)/opennessx),na.rm=T)/sum(svywt[!is
  socialCap=sum(svywt*((socialCap-socialCapn)/socialCapx),na.rm=T)/sum(svywt
  domains=sum(svywt*((domains-domainsn)/domainsx),na.rm=T)/sum(svywt[!is.na(
  comOff=sum(svywt*((comOff-comOffn)/comOffx),na.rm=T)/sum(svywt[!is.na(comO
  comAttach=sum(svywt*((comAttach-comAttachn)/comAttachx),na.rm=T)/sum(svywt
)
```

```r
# Your turn: turn this into a function.
# What variables do you need?

passionn <- min(comp$passion,na.rm=T)
passionx <- max(comp$passion,na.rm=T)-passionn

sum(svywt*((comp$passion-passionn)/
passionx),na.rm=T)/sum(svywt[!is.na(comp$passion)])
```

```r
f <- function(x, wt) {
  min_x <- min(x, na.rm = TRUE)
  rng_x <- max(x, na.rm = TRUE) - min_x

  sum(wt * ((x - min_x)/rng_x), na.rm = TRUE) /
    sum(wt[!is.na(x)])
}

# How could you make the intent clearer?
```

```r
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1]
}

f <- function(x, wt) {
  sum(wt * rescale01(x), na.rm = TRUE) /
    sum(wt[!is.na(x)])
}
```

```r
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1]
}

# Is this better?
f <- function(x, wt) {
  wt <- wt[!is.na(x)]
  x <- x[!is.na(x)]

  sum(wt * rescale01(x)) / sum(wt)
}
```

```r
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1]
}

# Is this better?
f <- function(x, wt) {
  weighted.mean(rescale01(x), wt, na.rm = TRUE)
}
```

How do you write a good function?

Robust code

- Spend time now to save time later
- Be explicit, e.g. `TRUE` and `FALSE`, not `T` and `F`
- Avoid functions that have different types of output (avoid `sapply`, beware `[`)
- Avoid functions that use non-standard evaluation (no `subset`, `with`, `transform`)
- Check preconditions and fail fast

Vocabulary

- A broad R vocabulary lets you make use of existing R functions
- Existing functions are documented, better tested, often more general, ...
- But more importantly they will often have a standard name

Code = communication

- Rewrite important code: your first attempt isn't usually the best approach.
- Consider the audience; what vocabulary should you assume?
- Being obviously correct is better than just being correct, but it may take a lot of time to get there.

Debugging

http://adv-r.had.co.nz/Exceptions-Debugging.html

Steps

1. Realise that you have a bug
2. Make it repeatable
3. Figure out where it is
4. Fix it and test the fix

Tools

1. RStudio error inspector/ `traceback()`
2. RStudio's rerun with debug/ `options(error = browser)`
3. RStudio's breakpoints/`browser()`

Testing

- Debugging gets it working now; testing ensure that it keeps working in the future. Really important!
- Recommend that you learn how to use `testthat`: http://r-pkgs.had.co.nz/tests.html

Functional programming

Hadley Wickham
Chief Scientist
RStudio

August 2015

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))
```

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]] <- length(l[[i]])
}
lengths
```

Safe shortcut for `1:length(l)`

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

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

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

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

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

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

```r
# How would you reduce the duplication here?

f1 <- function(x) x + 1
f2 <- function(x) x + 2
f3 <- function(x) x + 3
```

Functions can be arguments!

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

compute(l, length)
compute(l, mean)
compute(l, median)
```

Placeholder for “any other” arguments

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

compute(l, mean, trim = 0.5)
compute(l, mean, na.rm = TRUE)
```

Classes

Goal

Make a class that allows us to easily work with discrete random variables.

A discrete rv connects probabilities to numbers. Probabilities all greater than 0 and add to one; finite number of numbers.

Want to be able to plot, sample, take expectations, compute probabilities, combine etc.

```r
source("rv.r")

dice <- rv(1:6)
mean(dice)
min(dice)
max(dice)
range(dice)

P(dice > 3)
plot(dice + dice + dice)
```

discreteRV

discreteRV	Casella and Berger
`E(X)`	E(X)
`P(X == x)`	$P(X = x)$
`P(X >= x)`	$P(X \geq x)$
`P((X < x1) %AND% (X > x2))`	$P(X < x_1 \cap X > x_2)$
`P((X < x1) %OR% (X > x2))`	$P(X < x_1 \cup X > x_2)$
`P((X == x1) | (X > x2))`	$P(X < x_1 \| X > x_2)$
`probs(X)`	$f(x)$
`V(X)`	$Var(X)$

For a fuller exploration of these ideas, see `discreteRV` by Eric Hare.

http://journal.r-project.org/archive/2015-1/hare-buja-hofmann.pdf

Random variables

x	-1	0	1	2	3
P(x)	0.2	0.1	0.3	0.1	0.3

How might we record the information in the random variable shown above? (There are at least three ways)

What two things do we need to store to model a discrete random variable?

x	-1	0	1	2	3
P(x)	0.2	0.1	0.3	0.1	0.3

```
x <- c(-1, 0, 1, 2, 3)
p <- c(0.2, 0.1, 0.3, 0.1, 0.3)

# Ways to store
structure(x, prob = p)
structure(p, val = x)
list(x = x, p = p)
```

S3

No formal definition of the attributes that make up a class.

Instead, just set `class()` attribute of base type.

The simplest OO system that might possibly work. Adequate for 95% of R programming.

```r
# No checks for object correctness, so easy to abuse
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
mod

class(mod) <- "table"
mod

# But surprisingly, this doesn't cause that
# many problems - instead of the language enforcing
# certain properties you need to do it yourself
```

```r
# Start by defining a constructor function. It
# uses structure to set the class attribute.
rv <- function(x, probs = NULL) {
  if (is.null(probs)) {
    probs <- rep(1, length(x)) / length(x)
  }
  structure(x, probs = probs, class = "rv")
}
```

```r
# Also customary to create function to test if
# an object is of that class:
is.rv <- function(x) {
  inherits(x, "rv")
  # equivalent to "rv" %in% class(x)
}

# And we'll also write a helper to extract
# the probabilities
probs <- function(x) attr(x, "probs")
```

Your turn

What's wrong with the following objects?

```
rv(1:3, c(-1, 2))
```

```
rv(c(1, 1), c(0.5, 0.5))
```

What constraints are there on the probabilities? Can you fix `rv()` to verify or fix the constraints?

What's wrong with each of the following rvs?
How could you write code to detect the problem?

x	-1	0	1
P(x)	0.5	0.5	0.5

x	-1	0	1
P(x)	"a"	FALSE	😀

x	-1	0	1
P(x)	-0.25	1	0.25

x	-1	0	1
P(x)	0.5	0.4	0.1

x	-1	0	1
P(x)	0.33	0.67	

x	-1	0	1
P(x)	NA	0.5	0.5

```r
check_probs <- function(x) {
  if (!is.numeric(x)) {
    stop("`prob` must be numeric.")
  }
  if (any(is.na(x))) {
    stop("`prob` must not contain any NA")
  }
  if (any(x < 0)) {
    stop("All `prob` must be >= 0")
  }
  if (sum(x) != 1) {
    stop("`sum(prob)` must equal 1")
  }
}
x <- rep(1/49, 49)
check_probs(x)
```

```r
check_probs <- function(x) {
  if (!is.numeric(x)) {
    stop("`prob` must be numeric.")
  }
  if (any(is.na(x))) {
    stop("`prob` must not contain any NA")
  }
  if (any(x < 0)) {
    stop("All `prob` must be >= 0")
  }
  if (abs(sum(x) - 1) > 1e-6) {
    stop("`sum(prob)` must equal 1")
  }
}
x <- rep(1/49, 49)
check_probs(x)
```

```
rv <- function(x, probs = NULL) {
  if (is.rv(x)) x <- as.numeric(x)
  if (is.null(probs)) {
    probs <- rep(1, length(x)) / length(x)
  } else {
    if (length(x) != length(prob)) stop("Values and probability...")
    check_probs(x)
 }

  # Simplify by summing probabilities with equal x's. Need to use
  # addNA since otherwise tapply silently drops groups with missing values
  grp <- addNA(x, ifany = TRUE)
  x_new <- as.vector(tapply(x, grp, "[", 1))
  probs <- as.vector(tapply(probs, grp, sum))

  # Set probs and class attributes
  structure(x_new, probs = probs, class = "rv")
}
```

Strict

Helpful

Methods

Methods

- To make a class act differently, need to supply **methods** for **generic functions**.
- Most commonly provided methods are for: `print()` (202!), `format()` (63), `summary()` (32), `as.data.frame()`, `plot()`

Methods belong to **functions**, not classes

```r
# See what methods are defined for print and summary
methods("print")
methods("summary")

# See what methods are defined for data.frame
# and factor
methods(class = "data.frame")
methods(class = "factor")

`[.factor`
print.factor
getS3method("[", "factor")
```

Methods belong to **functions**, not classes

	factor	Date	data frame
relevel	✓		
mean		✓	
rep	✓	✓	
print	✓	✓	✓

```
# First method is usually a print method. Always
# look at the generic first so that you can match
# the arguments correctly.
print

# Can tell it's a generic function because it uses
# UseMethod()

# Methods follow simple naming scheme
print.rv <- function(x, ...) {
  ...
}
```

Your turn

Fill in the template to create a print method for rv objects.

Good print methods are really hard, so aim to get the important data out, even if it doesn't look great.

```
print.rv <- function(x, ...) {
  X <- format(x, digits = 3)
  P <- format(probs(x), digits = 3)
  out <- cbind(X = X, "P(X)" = P)
  rownames(out) <- rep("", nrow(out))
  print(out, quote = FALSE)
}

dice <- rv(1:6)
print(dice)
```

```r
# Another common method is plot
plot.rv <- function(x, ...) {
  name <- deparse(substitute(x))
  ylim <- range(0, probs(x))

  plot(as.numeric(x), probs(x), type = "h", ylim = ylim,
    xlab = name, ylab = paste0("P(", name, ")"), ...)
  points(as.numeric(x), probs(x), pch = 20)
  abline(h = 0, col = "gray")
}
```

Mean & variance

The **mean** summarises the "middle" of the distribution. Mean = E(X) = "Sum" of all outcomes, weighted by their probability.

x	-1	0	1	2	3
P(x)	0.2	0.1	0.3	0.1	0.3

Your turn

Implement a mean method.

```r
mean.rv <- function(x, ...) {
  sum(x * probs(x))
}
```

Inheritance

S3 inheritance

- Multiple elements in class attribute.
- First looks method for first class, then second, and so on.
- Then looks for method for *implicit* class.
- Then looks for default method.

Class	class()	Implicit class
Time	`POSIXct`, `POSIXt`	`numeric`, `double`
Generalised linear model	`glm`, `lm`	`list`
Data frame	`data.frame`	`list`

```r
iclass <- function(x) {
  c(
    if (is.matrix(x)) "matrix",
    if (is.array(x) && !is.matrix(x)) "array",
    if (is.double(x) || is.integer(x)) "numeric",
    typeof(x)
  )
}
method_names <- function(generic, x) {
  paste0(generic, ".", c(class(x), iclass(x), "default"))
}
s3_dispatch <- function(call) {
  call <- substitute(call)
  generic <- as.character(call[[1]])
  object <- eval(call[[2]], parent.frame())
  methods <- method_names(generic, object)
  exists <- vapply(methods, exists, logical(1))
  cat(paste0(ifelse(exists, "*", " "), " ", methods,
    collapse = "\n"), "\n", sep = "")
}
```

```r
x <- 1:10
class(x) <- c("c", "b", "a")

print.c <- function(x) {
  cat("C\n")
  NextMethod()
}
print(x)
s3_dispatch(print(x))

# See s3-inheritance.R
```

```
dice <- rv(1:6)

# Why do these work?
min(dice)
range(dice)

# What's wrong with these?
dice * 2
abs(dice)
abs(dice - 2)
dice[1:3]
```

Inheritance

Want to use the default behaviour for abs, [, etc.

`NextMethod()` call the next method in the sequence.

You never supply arguments - it uses non-standard evaluation magic to figure them out.

```r
sumto1 <- function(x) x / sum(x)

`[.rv` <- function(x, i, ...) {
  rv(NextMethod(), sumto1(probs(x)[i]))
}

abs.rv <- function(x) {
  y <- NextMethod()
  rv(y, probs(y))
}

# What would methods for sqrt, log and exp
# look like?
```

```
abs.rv <- function(x) {
  y <- NextMethod()
  rv(y, probs(y))
)
log.rv <- function(x) {
  y <- NextMethod()
  rv(y, probs(y))
)
exp.rv <- log.rv
sqrt.rv <- log.rv
```

Your turn

Look at `rv.R`. What other methods are implemented? What do they do?

Generic functions

S3 generics

- As well as creating methods for existing generics, you can also create your own.
- Creating a generic is very very simple!
- Just call `UseMethod(”generic name”).`
- Other arguments figured out by NSE magic.

```
# Creating your own generics
mean2 <- function (x, ...) {
  UseMethod("mean2")
}

mean2.numeric <- function(x, ...) sum(x) / length(x)
mean2.data.frame <- function(x, ...)
  sapply(x, mean2, ...)
mean2.matrix <- function(x, ...) apply(x, 2, mean)

mean2.default <- function(x, ...) {
  stop("mean2 not implemented for objects of type ",
    class(x))
}
```

Encapsulation

In Java/C#/Ruby/Python etc., often have many small methods, even if only used by one class.

This is not useful in R – only useful to define methods that are used by multiple classes.

Use namespaces for the equivalent encapsulation.