OO programming

Hadley Wickham
Chief Scientist
RStudio

August 2015

1. Motivation
2. Warmup
3. Classes
4. Inheritance
5. Methods
6. Generic functions
7. Other OO systems

Motivation

Motivation

- Understanding more code
- Extensibility
- Programming “in the large”

- Focus on S3 because it is most commonly used, and solves most problems.

```r
print(1:10)
print(mtcars)
print
# How does print work?

mean
# How does mean work?
```

Warmup

Your turn

What is an attribute? What types of objects can have attributes?

How do you *get* the value of an attribute? How do you *set* the value an attribute?

What are the three most important attributes? (Hint: what makes a matrix different from a vector?)

Brainstorm with your neighbours for 2 minute.

```r
# Attributes add arbitrary additional
# metadata to any base type
x <- 1:6
attr(x, "max") <- 5
attr(x, "max")
attributes(x)

# structure returns a modified object with attrs
structure(1:10, max = 5)

# Most important attributes are dim, class and
# names.  Should always use dim(), class() and
# names() respectively to get and set their
# values.
```

Your turn

Every S3 class is built on a basic type (like character, integer, function, list, …). The two most important S3 classes are factor and data frame.

What are factors built on top of? What attributes do they use?

What are data frames built on top of? What attributes do they use?

```r
f <- factor(c("a", "b", "c"))
typeof(f)     # Built on top of integer
attributes(f) # Use levels and class attributes

d <- data.frame(f)
typeof(d)     # Built on top of list
attributes(d) # names, row.names and class
```

Class	Base type	class()
Data frame	list	`data.frame`
Factor	integer	`factor`
Date	integer	`Date`
Time	double	`POSIXct, POSIXt`
Linear model	list	`lm`

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.

```
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

```
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)
```

```
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

```
# 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)
```

```
# 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.

```
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.

```r
# S3 and S4
method(object, 1, 2, 3)

# RC/R6
object$method(1, 2, 3)

# Many other languages
object.method(1, 2, 3)
```

Learning more

S4

Same basic style as S3, but formal and rigorous (and verbose and slow).

`setClass()` defines classes.

`setGeneric()` defines generic functions.

`setMethod()` defines methods.

R6

Package inspired by ReferenceClasses, but much faster & fixes major problem.

Class-based (message passing) OO. Much closer to Java/C#/Python/Ruby etc.

Has mutable state.

Recommendations

- Use **S3**, unless:
- You have complex network of classes and methods - use **S4**.
- You have objects with changing state - use **R6**.

http://adv-r.had.co.nz/OO-essentials.html

https://www.google.com/search?q=bioconductor+s4