CSC 120: Computer Science for the Sciences (R section)

Radford M. Neal, University of Toronto, 2016

http://www.cs.utoronto.ca/∼radford/csc120/

Week 10

Another Use for Classes — Factors

Recall that how R handles an object can be changed by giving it a "class" attribute. That's how lists become data frames. Another example is the "factor" class, which is used to convert a vector of strings to a vector of integers, along with a vector of just the *distinct* string values.

Here's an illustration:

```
> a <- as.factor(c("red","green","yellow","red","green","blue","red"))
> a
[1] red    green  yellow red    green  blue   red
Levels: blue green red yellow
> class(a)        # We can see that this object has the class "factor"
[1] "factor"
> unclass(a)      # Here's what it is without its class attribute
[1] 3 2 4 3 2 1 3
attr(,"levels")
[1] "blue"   "green"  "red"    "yellow"
```

One reason to use factors is that an integer uses less memory than a long string. R automatically converts strings to factors in `read.table`, unless you use the `stringsAsFactors=FALSE` option.

Operations on Factors

Factors look like strings for many purposes:

```
> a <- as.factor(c("red","green","yellow","red","green","blue","red"))
> a == "red"
[1]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Even though factors are represented as integers, mathematical operations on them are not allowed:

```
> sqrt(a)
Error in Math.factor(a) : sqrt not meaningful for factors
```

This is because the integers representing the “levels” of the factor are arbitrary, so treating them like numbers would be misleading. (Unfortunately, R isn’t completely consistent in this, and will sometimes use a factor as a number without a warning.)

Another Use of Classes — Dates and Time Differences

R also defines classes for dates, and for differences in dates. Some of what you can do with these is illustrated below:

```
> d1 <- as.Date("2015-03-24")  # d1 will be an object of class "Date"
> d1
[1] "2015-03-24"     # Adding an integer to a date gives a new date
> d1+2
[1] "2015-03-26"
> d1+10              # Addition will automatically change the month
[1] "2015-04-03"
>
> d2 <- as.Date("2015-02-24")
> d1-d2              # The difference has class "difftime"
Time difference of 28 days
> as.numeric(d1-d2)  # We can convert a "difftime" object to a number
[1] 28
```

Defining Your Own Classes

You can attach a class attribute of your choice to any object. If that's all you do, the object gets handled just as before, except the class attribute is carried along:

```
> x <- 9
> class(x) <- "mod17"
> x + 10
[1] 19
attr(,"class")
[1] "mod17"
```

But you can now redefine some operations (ones that are "generic") to operate specially on your class:

```
> ‘+.mod17‘ <- function (a,b) {
+     r <- (unclass(a) + unclass(b)) %% 17
+     class(r) <- "mod17"
+     r
+ }
> x + 10
[1] 2
attr(,"class")
[1] "mod17"
```

Defining Your Own Generic Functions

You can also create new generic functions, that you can define “methods” for, that are used when they are called with objects of particular classes. For example:

```
> picture <- function (x) UseMethod("picture")
> picture.default <- function (x)
+                            cat(x,"\n")
> picture.mod17   <- function (x)
+                            cat(rep("-",x-1),"O",rep("-",17-x),"\n")
>
> picture(9)
9
> picture(x)
- - - - - - - - O - - - - - - - -
> picture(x+3)
- - - - - - - - - - - O - - - - -
```

The definition of `picture` just says it’s generic. If no special method is defined for a class, `picture.default` is used. By defining `picture.mod17`, we create a special method for class `mod17`. R finds the method to use based on the class of the first argument to the generic function.

The Object-Oriented Approach to Programming

R's classes are designed to support what is called "object-oriented" programming.

This approach to programming has several goals:

- Allow manipulation of "objects" without having to know exactly what kind of object you're manipulating — as long as the object can do the things that you need to do (it has the right "methods").

  Benefit: We can write one just function for all objects, not many functions, that all do the same thing but in somewhat different ways.

- Separate *what* the methods for an object do from *how* they do it (including how the object is represented).

  Benefit: We can change how objects work without having to change all the functions that use them.

- Permit the things that can be done with objects ("methods") and the kinds of objects ("classes") to be extended without changing existing functions.

  Benefit: We can more easily add new facilities, without having to rewrite existing programs.

Generic Functions for Drawing, Rescaling, and Translating

Let's see how we can define a set of generic functions for drawing and transforming objects like circles and boxes.

We start by setting up the generic functions we want:

```
draw <- function (w) UseMethod("draw")
rescale <- function (w,s) UseMethod("rescale")
translate <- function (w,tx,ty) UseMethod("translate")
```

Then we need to define methods for these generic functions for all the classes of objects we want. We also need functions for creating such objects.

Note: We might not have done things in this order. For example, we might have first defined only `draw` and `translate` methods, and then later added the `rescale` method. We would then need to implement a `rescale` method for a class only if we actually will use `rescale` for objects of that class.

Implementing a Circle Object

We'll represent a circle by the x and y coordinates of its centre and its radius.

```
new_circle <- function (x, y, r) {
    w <- list (centre_x=x, centre_y=y, radius=r)
    class(w) <- "circle"
    w
}
draw.circle <- function (w) {
    angles <- seq (0, 2*pi, length=100)
    lines (w$centre_x + w$radius*cos(angles),
           w$centre_y + w$radius*sin(angles))
}
rescale.circle <- function (w,s) {
    w$radius <- w$radius * s;
    w
}
translate.circle <- function (w,tx,ty) {
    w$centre_x <- w$centre_x + tx; w$centre_y <- w$centre_y + ty
    w
}
```

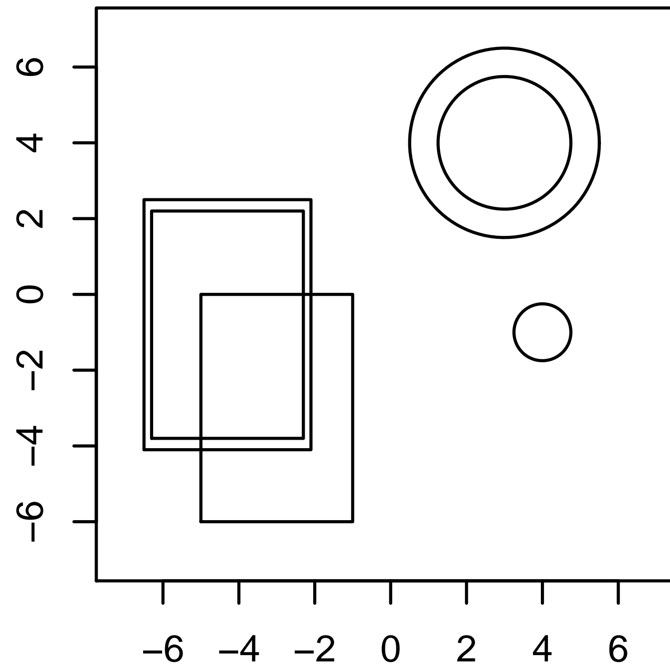
Implementing a Box Object

We'll represent a box by the x and y coordinates at its left/right top/bottom. But to create a box we'll give coordinates for its centre and offsets to the corners.

```
new_box <- function (x, y, sx, sy) {
    w <- list (x1=x-sx, x2=x+sx, y1=y-sy, y2=y+sy)
    class(w) <- "box"
    w
}
draw.box <- function (w) {
    lines (c(w$x1,w$x1,w$x2,w$x2,w$x1), c(w$y1,w$y2,w$y2,w$y1,w$y1))
}
rescale.box <- function (w,s) {
    xm <- (w$x1+w$x2) / 2
    w$x1 <- xm + s*(w$x1-xm); w$x2 <- xm + s*(w$x2-xm)
    ym <- (w$y1+w$y2) / 2
    w$y1 <- ym + s*(w$y1-ym); w$y2 <- ym + s*(w$y2-ym)
    w
}
translate.box <- function (w,tx,ty) {
    w$x1 <- w$x1 + tx; w$x2 <- w$x2 + tx
    w$y1 <- w$y1 + ty; w$y2 <- w$y2 + ty
    w
}
```

An Example of Drawing Objects This Way

```
> plot(NULL,xlim=c(-7,7),ylim=c(-7,7),xlab="",ylab="",asp=1)
> c <- new_circle(3,4,2.5)
> draw(c); draw(rescale(c,0.7)); draw(translate(rescale(c,0.3),1,-5))
> b <- new_box(-3,-3,2,3)
> b2 <- translate(b,-1.3,2.2)
> draw(b); draw(b2); draw(rescale(b2,1.1))
```



Defining a Function That Works On Both Circles and Boxes

Here is a function that should work for circles, boxes, or any other class of object that has `draw`, `rescale`, and `translate` methods:

```
smaller <- function (w, n)
    for (i in 1:n) { draw (w); w <- rescale(translate(w,1,0),0.9) }
```

Here are two uses of it:

```
> plot(NULL,xlim=c(-7,7),ylim=c(-7,7), xlab="",ylab="",asp=1)
> smaller (new_circle(-3,3.1,3),10)
> smaller (new_box(-3,-3,3.1,3),10)
```