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:

```r
> 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:

```r
> 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:

```r
> 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:

```r
> 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:

```r
> 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:

```r
> `+.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:

```r
> picture <- function (x) UseMethod("picture")
> picture.default <- function (x)
+ cat(x,"
")
> picture.mod17 <- function (x)
+ cat(rep("-",x-1),"0",rep("-",17-x),"
")
>
> picture(9)
9
> picture(x)
- - - - - - - - 0 - - - - - - -
> picture(x+3)
- - - - - - - - - - - 0 - - - - -
```

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:

```r
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.

```r
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.

```r
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:

```r
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:

```r
> 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)
```
Ending a Loop Using a Logical Flag Variable

This week’s lab exercises included writing a function that will return TRUE if the elements in its argument (a numeric vector) do not decrease. Here’s one solution:

```r
is.not.decreasing <- function (v) {
  answer_is_known <- FALSE
  i <- 2
  while (!answer_is_known) {
    if (i > length(v)) {
      answer <- TRUE
      answer_is_known <- TRUE
    }
    else if (v[i] < v[i-1]) {
      answer <- FALSE
      answer_is_known <- TRUE
    }
    i <- i + 1
  }
  answer
}
```
Using a `repeat` Loop and `break` Statement

This function used two logical variables — one to hold the answer returned, the other to indicate when the answer is now known, and hence the loop can end. We can instead use a loop written using `repeat`, which continues indefinitely, until a `break` statement is done:

```r
is.not.decreasing <- function (v) {
  i <- 2
  repeat {
    if (i > length(v)) {
      answer <- TRUE
      break
    }
    if (v[i] < v[i-1]) {
      answer <- FALSE
      break
    }
    i <- i + 1
  }
  answer
}
```
Using **break** Within a **for** Loop

We can use **break** to immediately exit any kind of loop. Here’s another way to write this function:

```r
is.not.decreasing <- function (v) {
    answer <- TRUE
    if (length(v) > 1)
        for (i in 2:length(v)) {
            if (v[i] < v[i-1]) {
                answer <- FALSE
                break
            }
        }
    answer
}
```

In this version, we initially set `answer` to **TRUE**, which will be the answer if we don’t find a place where the elements decrease. If we do find a decrease, we set `answer` to **FALSE**, and also immediately exit the **for** loop.

**Caution:** The **break** statement exits from the innermost loop that contains it. If you’re inside two loops, you can’t use **break** to exit both of them at once.
Returning a Value for a Function Immediately

Rather than exit a loop with `break` after setting `answer`, and then making `answer` the value of the function by putting it as the last thing, we can instead use `return` to exit the whole function, and specify the value it returns.

```r
is.not.decreasing <- function (v) {
  if (length(v) > 1) {
    for (i in 2:length(v)) {
      if (v[i] < v[i-1])
        return(FALSE)
    }
  }
  return(TRUE)
}
```

At the end, we could just have written `TRUE` instead of `return(TRUE)` — they do the same thing at the end of a function.

Why is the check for `length(v) > 1` needed?
Avoiding Loops with a Vector Comparison

We can write `is.not.decreasing` without an R loop using a vector comparison and the `all` function:

```r
is.not.decreasing <- function (v) all (v[-length(v)] <= v[-1])
```

In this version, `v[-length(v)]` will contain all of `v` except the last element, and `v[-1]` will contain all of `v` except the first element. So `v[-length(v)] <= v[-1]` compares each element except the last to the next element. The vector `v` is non-decreasing if all these comparisons are `TRUE`.

Here’s another way to do the same thing:

```r
is.not.decreasing <- function (v) {
  if (length(v) < 2)
    TRUE
  else
    all (v[1:(length(v)-1)] <= v[2:length(v)])
}
```

Why is the check for `length(v) < 2` needed here, but not in the version above?
Recursion — When a Function Calls Itself

As you know, an R function can call another R function, which can call yet another R function, etc.

Indeed, an R function can even call itself. This is called “recursion”.

Of course, a function had better not always call itself, or it will just keep calling, and calling, and calling, without end.

But having a function sometimes call itself can be useful. Here’s a recursive function to compute factorials in R:

```
fact <- function (n) if (n == 0) 1 else n * fact(n-1)
```

(Although R already has a pre-defined factorial function.)

In fact, anything computable can be computed using if and recursion, without any loops or assignment statements. That’s not a typical style of programming in R, but it is typical for some other programming languages.
Two Recursive Versions of \texttt{is.not.decreasing}

We could write the \texttt{is.not.decreasing} function using recursion. Here’s one way:

\begin{verbatim}
is.not.decreasing <- function (v) {
    if (length(v) <= 1)
        TRUE
    else if (v[2] < v[1])
        FALSE
    else
        is.not.decreasing(v[-1])
}
\end{verbatim}

Here’s another way that doesn’t copy parts of \(v\), and also extends the function’s meaning so it checks only from a certain point forward (default, from the start):

\begin{verbatim}
is.not.decreasing <- function (v, from=1) {
    if (length(v) <= from)
        TRUE
    else if (v[from+1] < v[from])
        FALSE
    else
        is.not.decreasing(v,from+1)
}
\end{verbatim}
Reminders: R Features Useful for Assignment 3

The March 18 lab exercise uses the `sample` function, which you’ll also need for Assignment 3. In its simplest form, it randomly permutes a vector of values.

For example:

```r
> v <- c(9,8,1,8,2)
> sample(v)
[1] 8 1 2 9 8
> sample(v)
[1] 8 2 8 1 9
```

You’ll also need to get a column from a data frame that has a certain name.

When you know exactly what the name is, you can use `$` — for example, `df$age` to get the `age` column. But if the name is in a variable, you have to use the name as an index (see Week 8 lectures). For example:

```r
> df <- data.frame (list (abc=c(4,1,2), def=c(8,3,7)))
> n <- "def"
> df[,n]
[1] 8 3 7
```