CSC 121: Computer Science for Statistics

Radford M. Neal, University of Toronto, 2017

http://www.cs.utoronto.ca/∼radford/csc121/

Week 4

Combining Data of Different Types in a List

We've seen how we can put several numbers into a vector of numbers. Or we can put several strings into a vector of strings. But what if we want to combine both types of data? Let's try...

```
> c(123,"fred",456)
[1] "123"  "fred" "456"
```

R converts the numbers to character strings, so that the elements of the vector will all be the same type (character).

But we *can* put together data of different types in a *list*:

```
> list(123,"fred",456)
[[1]]
[1] 123

[[2]]
[1] "fred"

[[3]]
[1] 456
```

Lists Can Contain Anything

Elements of a list can actually be anything, including vectors of different lengths:

```
> list (1:4, 3:10)
[[1]]
[1] 1 2 3 4


[[2]]
[1]  3  4  5  6  7  8  9 10
```

You can even put lists within lists (though these are hard to read when printed):

```
> list(4,list(5,6))
[[1]]
[1] 4


[[2]]
[[2]][[1]]
[1] 5


[[2]][[2]]
[1] 6
```

Extracting and Replacing Elements of a List

You can get a single element of a list by subscripting with the `[[ ...]]` operator:

```
> L <- list (c(3,1,7), c("red","green"), 1:4)
> L[[2]]
[1] "red"   "green"
> L[[3]]
[1] 1 2 3 4
```

You can replace elements the same way. Continuing from above...

```
> L[[3]] <- c("x","y","z")
> L
[[1]]
[1] 3 1 7


[[2]]
[1] "red"   "green"


[[3]]
[1] "x" "y" "z"
```

Notice that the new value can have a type different from that of the old value.

Looking at All Elements of a List; Extending Vectors

You can look at all elements of a list with the `for` statement, using `length` to find out how many elements there are.

Suppose we have a list of vectors of strings or numbers. For example, we might create such a list as follows:

```
> L <- list (c("a","b"), 2:4, c("x","y","z"))
```

The following will create a single vector of strings, called `v`, containing all the elements of all the vectors from the list `L`:

```
> v <- character(0)      # creates a string vector with zero strings
> for (i in 1:length(L)) v <- c (v, L[[i]])
> v
[1] "a" "b" "2" "3" "4" "x" "y" "z"
```

Note how we can start with a vector with no elements, and then extend it using the `c` function. Also note how the vector of numbers was automatically converted to a vector of strings, so they could be combined with a string vector.

Extending Lists

You can also build up lists starting with a list containing zero elements, which we can create with `list()`.

One way to extend the list is to just assign to an element that doesn't exist yet (usually the one just after the last existing element):

```
> a <- list()
> a[[1]] <- 1:3; a[[2]] <- TRUE; a[[3]] <- "hello"
> a
[[1]]
[1] 1 2 3

[[2]]
[1] TRUE

[[3]]
[1] "hello"
```

You can also combine lists with the `c` function.

More on Logical Values

We've seen that R uses *logical* values to represent the result of a comparison, such as below:

```
> a <- 10
> a < 3
[1] FALSE
> a < 30
[1] TRUE
```

We can save logical values in variables, and then use them as `if` or `while` conditions:

```
> b <- a < 30
> if (b) cat("It's TRUE!\n")
It's TRUE!
```

We can also just assign `TRUE` or `FALSE` to a variable.

Using Logical Variables to Stop a While Loop

Logical variables we set to `TRUE` or `FALSE` can be useful for stopping `while` loops.

This bit of a program checks for values in `vec` outside the range 0 to 100, stops with a message if it finds one, or stops with no message if there are none:

```
i <- 0
keep_going <- TRUE
while (keep_going) {
    i <- i + 1
    if (i > length(vec))
        keep_going <- FALSE
    else if (vec[i] < 0) {
        cat("Found a value less than 0\n")
        keep_going <- FALSE
    }
    else if (vec[i] > 100) {
        cat("Found a value greater than 100\n")
        keep_going <- FALSE
    }
}
```

The Logical "AND" Operator — &&

Suppose we want to print a message if the number in `next_value` is within the range 0 to 100 (and do nothing if it is not).

Here's one way we could do this:

```
if (next_value >= 0)
    if (next_value <= 100)
        cat("Next value is OK\n")
```

Instead, we can do this with just one `if` by using R's *logical AND* operator, which is written `&&`:

```
if (next_value >= 0 && next_value <= 100)
    cat("Next value is OK\n")
```

An expression such as `X && Y` produces `TRUE` only if `X` and `Y` are both `TRUE`, and `FALSE` if either (or both) of `X` and `Y` are `FALSE`.

The Logical "OR" Operator — ||

Similarly, R has a *logical OR* operator, written `||`.

An expression such as `X || Y` produces `TRUE` if either `X` and `Y` (or both) are `TRUE`, and `FALSE` if both `X` and `Y` are `FALSE`.

We could use it to print a message if the number in `next_value` is not in the range 0 to 100:

```
if (next_value < 0 || next_value > 100)
    cat("Next value is out of range\n")
```

The `&&` and `||` operators can both be used in a condition, with `&&` having higher precedence.

There's also a "NOT" operator, written `!`.

Shortcuts when Evaluating && and ||

When R evalues somethinge like `X && Y`, it first finds the value of `X`, and if it is `FALSE`, it doesn't bother to find the value of `Y`, since the result must be `FALSE` regardless of `Y`.

This can be useful if evaluating `Y` would cause an error:

```
if (i <= length(L) && L[[i]] > 0) ... # do something
```

Trying to get element `i` of the list `L` results in an error message if `i` is greater than the length of the list, but this won't happen with the condition above.

Similarly, the value of `X || Y` will be `TRUE` if `X` is `TRUE`, regardless of `Y`, so if `X` is `TRUE`, R doesn't try to evaluate `Y`.