CSC 121: Computer Science for Statistics

Radford M. Neal, University of Toronto, 2017

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

Week 10

Many Ways to Write a Simple Function

In this lecture, we'll look at many ways of writing a simple function called `is_not_decreasing`, which takes one argument, a vector, and returns TRUE if the elements in the vector are in non-decreasing order, and FALSE otherwise. We'll see some new R features along the way.

Examples:

```
> is_not_decreasing (c(4,8,8,9))
[1] TRUE
> is_not_decreasing (c(5,1,3))
[1] FALSE
> is_not_decreasing (7)
[1] TRUE
```

We'll assume that the vector has no NA values. What would be a reasonable thing to do if it did?

Ending a Loop Using a Logical Flag Variable

Here's one solution, that uses the setting of a logical variable as a way of terminating a `while` loop:

```
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:

```
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:

```
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.

```
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:

```
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:

```
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 `is_not_decreasing`

We could write the `is_not_decreasing` function using recursion. Here's one way:

```
is_not_decreasing <- function (v) {
    if (length(v) <= 1)
        TRUE
    else if (v[2] < v[1])
        FALSE
    else
        is_not_decreasing(v[-1])
}
```

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):

```
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)
}
```

Operations on Vectors

We've seen before that R can do many operations on entire vectors (or matrices), not just on single numbers. For example, we can add 1 to all elements of a vector:

```
> u <- c(3,5,1,9)
> v <- u + 1
> v
[1]  4  6  2 10
```

Instead of the statement `v <- u + 1` we could have written a loop:

```
> v <- u
> for (i in 1:length(v)) v[i] <- v[i] + 1
> v
[1]  4  6  2 10
```

But `v <- u + 1` is easier to write, easier to read, and also faster in R.

This isn't magic, though — there still **is** a loop hidden within the implementation of R, and in some other languages writing a loop yourself would be just as fast.

R has many other facilities for doing operations on vectors, matrices, or lists without having to write a loop, which often are also faster.

Replacing Loops with "apply" Functions

Functions in the "apply" family take as arguments both a data structure and a function to apply to parts of the data structure — an example of "functional programming", using functions to construct more complex operations.

The `lapply` function operates on a list, and returns a list of results of applying a given function to each element of the list. Here's an example using the `is.numeric` function, which says whether something is a numeric vector:

```
> L <- list ("abc", c(123,456), TRUE)
> lapply(L,is.numeric)
[[1]]
[1] FALSE

[[2]]
[1] TRUE

[[3]]
[1] FALSE
```

Using "apply" on Matrices

You can use `apply` to apply a function to all rows or to all columns of a matrix.

If the function applied returns a single value, the result is a vector of these values:

```
> M <- matrix (1:6, nrow=2, ncol=3)
> M
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> apply (M, 1, function (v) sum(v^2)) # 2nd arg of 1 means apply to rows
[1] 35 56
> apply (M, 2, function (v) sum(v^2)) # 2nd arg of 2 means apply to cols
[1]  5 25 61
```

If the function returns a vector of length greater than one, the result is a matrix:

```
> apply (M, 1, function (v) c(sum(v), prod(v)))
     [,1] [,2]
[1,]    9   12
[2,]   15   48
```

Logical Operators

Some previous slides have mentioned logical operations on vectors. These operate on operate on vectors of logical values, returning a vector of logical values.

For one logical value, the operators are defined as follows:

- `!` Logical "not": `TRUE` if its operand is `FALSE`, `FALSE` if its operand is `TRUE`.
- `&` Logical "and": `TRUE` only if both operands are `TRUE`.
- `|` Logical "or": `TRUE` if either operand is `TRUE`.

When applied to logical vectors, the operations are done on each element in turn:

```
> a <- c (TRUE,  TRUE,  FALSE, FALSE)
> b <- c (TRUE,  FALSE, TRUE,  FALSE)
> a & b
[1]  TRUE FALSE FALSE FALSE
> a | b
[1]  TRUE  TRUE  TRUE FALSE
> !a
[1] FALSE FALSE  TRUE  TRUE
```

An Example of `apply` Using Logical Operations

Here's how `apply` can be used to see which columns in a matrix have values that are all in the range of the first value to the last value:

```
> A <- matrix (c(2,9,0,1,3,8,4,9), nrow=4, ncol=2)
> A
     [,1] [,2]
[1,]    2    3
[2,]    9    8
[3,]    0    4
[4,]    1    9
> apply (A, 2, function (v) all (v >= v[1] & v <= v[length(v)]))
[1] FALSE  TRUE
```