CSC 121: Computer Science for Statistics

Radford M. Neal, University of Toronto, 2017

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

Week 3

Making Functions Do Different Things, Using `if`

When you call a function with different values for its arguments, it can compute different return values, or plot different data. That's more useful than a script that computes or does only one thing.

But what if the *way* the return value should be computed, or data should be plotted, depends on the arguments?

We can use `if` to do this:

```
# Function to compute how much of your income you should save.
money_to_save <- function (income)
    if (income < 30000) 0 else 0.1 * (income-30000)


# Plot data with lines, plus dots if no more than 100 points.
plot_data <- function (data) {
    if (length(data) > 100)
        plot (data, type="l")   # Plot lines only
    else
        plot (data, type="b")   # Plot lines, plus dots at the points
}
```

Comparisons and the Logical Data Type

In these `if` expressions, which thing to do is determined by comparing two numbers.

Comparisons in R produce values of *logical* data type — either `TRUE` or `FALSE`.

Here are some examples:

```
> a <- 12
> a < 10    # "less than" comparisons
[1] FALSE
> a < 20
[1] TRUE
> a > 0     # "greater than" comparisons
[1] TRUE
> 10 > a
[1] FALSE
> a == 12   # "equals" comparisons - note that it uses ==, not just =
[1] TRUE
> a == 9
[1] FALSE
> a != 9    # "not equal" comparison
[1] TRUE
```

More Comparisons

```
> a <- 12
> a >= 12   # "greater or equal" comparisons
[1] TRUE
> 13 >= a
[1] TRUE
> a <= 11   # "less or equal" comparison
[1] FALSE
```

You can compare strings too:

```
> time <- "morning"
> time == "evening"
[1] FALSE
> time == "morning"
[1] TRUE
> time != "evening"
[1] TRUE
```

Strings can also be compared with `<`, `>`, `<=`, and `>=`, according to "alphabetical order", but exactly how this works may depend on what "locale" R is operating in.

Some Notes on Spaces

Although R usually ignores spaces, an exception is that the `==`, `!=`, `<=`, and `>=` operators must be written without spaces inside them. Notice the syntax error below, for example:

```
> a < = 4
Error: unexpected '=' in "a < ="
```

This is true for the assignment operator, `<-`, as well. It's a good idea to always put spaces *around* the `<-` operator, however, because it makes programs easier to read. Spaces around other operators can sometimes improve readability too.

Compare the following:

```
> abc[123]<-xyz+456*pqr
```

```
> abc[123] <- xyz + 456*pqr
```

Warning: The expression `a<-9` assigns the value `9` to the variable `a`. If you want to compare the value in `a` to the number `-9`, you *must* put a space between `<` and `-`:

```
> a < -9
[1] FALSE
```

More on How `if` Works

An `if` expression has the form

`if (` *condition* `)` *true-option* `else` *false-option*

The *condition* produces a `TRUE` or `FALSE` value. If the value is `TRUE`, the *true-option* expression is done; if `FALSE`, the *false-option* expression is done.

The *true-option* and *false-option* expressions can have several steps, enclosed between `{` and `}`.

When the expression is evaluated for what it does, rather than producing a value, the `else` part can be omitted — that's the same as making *false-option* be `{ }`, which does nothing.

It's often useful for *false-option* to be another `if` expression. An example:

```
> edu_level <- ( if (school=="primary") 1
                 else if (school=="secondary") 2
                 else if (school=="university") 3
                 else 0 )
```

Doing Things Again and Again and Again... Using `for`

We often want to do the same thing, or similar things, many times. One way is to do arithmetic operations on vectors. But only some simple things can be done many times that way.

A more general way is to use a *loop*.

One kind of loop in R is a `for` loop, which has the form

`for (` *variable* `in` *vector* `)` *body*

The *body* can be one statement, or several enclosed in `{` and `}`. The `for` loop does the *body* as many times as there are elements in *vector*, with *variable* set to each element in turn.

Here's an example:

```
> for (i in c(10,12,456)) cat ("The square of",i,"is",i^2,"\n")
The square of 10 is 100
The square of 12 is 144
The square of 456 is 207936
```

Using `for` with a Sequence Vector

We often want a `for` loop to go through a sequence of integers. We can create a vector containing such a sequence with the `:` operator. For example:

```
> 1:5
[1] 1 2 3 4 5
```

Here's an example of its use with `for`:

```
> for (i in 1:5) cat ("The square of",i,"is",i^2,"\n")
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

Example: Looking at and Modifying a Vector Using a Loop

The function below takes a vector argument and returns this vector modified so all elements are between 0 and 100. It also prints a message if any elements were outside this range:

```
make_in_0_to_100 <- function (vec) {
    below_0 <- 0; above_100 <- 0
    for (i in 1:length(vec)) {
        if (vec[i] < 0) {
            vec[i] <- 0
            below_0 <- below_0 + 1
        }
        else if (vec[i] > 100) {
            vec[i] <- 100
            above_100 <- above_100 + 1
        }
    }
    if (below_0 + above_100 > 0)
        cat ("Out of", length(vec), "elements,", below_0,
             "were below 0 and", above_100, "were above 100\n")
    vec
}
```

Calls of the Example Function

Here are some calls of this function:

```
> source("http://www.cs.utoronto.ca/~radford/csc121/make_in_0_to_100.r")
> make_in_0_to_100(c(33,55,77))
[1] 33 55 77
> original_vec <- c(12,-2,17,33,101,-3,104,-1,93)
> modified_vec <- make_in_0_to_100(original_vec)
Out of 9 elements, 3 were below 0 and 2 were above 100
> modified_vec
[1]  12   0  17  33 100   0 100   0  93
```

Notice the way we count how many elements were below zero using the `below_0` variable. This variable is set to zero before the loop. Inside the loop, whenever an element is found to be below zero, the count is increased by the assignment

```
below_0 <- below_0 + 1
```

This assigns a new value to `below_0` that's equal to the old value of `below_0` plus 1.

Later, we'll see how we can write this function more easily, without a loop, using more advanced vector-handling facilities. But avoiding loops isn't always possible.

When You Don't Know How Many Times... Using `while`

A `for` loop repeats its body only as many times as the length of the vector it is given at the start. But sometimes you can't know at the start how many repetitions will be needed.

Instead, we can use a `while` loop. It has the form

`while (` *condition* `)` *body*

This will repeat *body* as many times as necessary, until *condition* is `FALSE`. If *condition* is `FALSE` at the start, *body* is not done even once.

Here's an example, which searches for the smallest integer, i, greater than one, for which i^{20} is less than e^i:

```
> i <- 2
> while (i^20 >= exp(i)) i <- i + 1
> i
[1] 90
```

Now You Can Do Anything!

With what you now know about R programming, for *anything* that can possibly be computed, you can (in theory) write a program that can compute it!

The keys points:

- You know about vectors, which as they get longer (eg, by putting shorter ones together with `c (..., ...)`) will hold more and more data, with no upper limit (in theory — in practice you run out of memory on your computer at some point).
- You know about `while` loops, which can repeat operations as many times as necessary, with no upper limit (in theory — in practice your computer will wear out and fail after some number of years of computing).

The technical term for this is that the R language (even just the part you know now) is "Turing complete" (after famous computer scientist Alan Turing, who formalized the notion of what can and cannot be computed).

So That's the End of the Course?

Now that you know how to compute anything that's computable, what's left to do in the course?

- You may "know" how to do any computation, but you still need to develop the skills that will help you to actually do it. This takes both instruction and practice, practice, practice, ...
- Once you learn more about R, you'll know how to do some computations more easily than you could do them using only what you know now.
- You'll learn about some R features that aren't strictly computational, but are very useful (such as how to put titles on plots).
- We'll talk about how to write programs that run faster — so you won't have to wait hours or days for the results.
- We'll talk about how to write programs that are easier for yourself and other people to understand.
- We'll talk about how to test that your programs actually work correctly.
- We'll talk about how to keep track of different versions of your programs, as you change them to make them better, or work for various different problems.