MAS115 R programming, Lab Class 2

Bryony Moody (Original notes by Prof. P. G. Blackwell) *

2020-21

1 Loops and control statements

1.1 Basic loops

Using a `for` loop

In many coding situations you will want to create an iterative loop so that your computer can cycle through a task automatically rather than you having to write it all out by hand. The most basic way to do this is using a `for` loop in R — there are some other ways we can do this using `while` and `repeat` which we shall cover in the next class.

```
# A basic loop to sum integers
total <- 0 # Total initialised to zero
a <- 1:8
for(i in a) {
    print(i)
    total <- total + i
}
```

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

```
total
```

```
[1] 36
```

Syntax

This construct should be recognisable to you from Python last term:

```
for (variable in sequence) statement
```

*Thanks to Dr. T Heaton & Dr. R Ripley for suggestions

There will be one iteration of the `statement` for each component of the vector `sequence`, with `variable` taking on the values of those components. The `statement` can itself be compound i.e. include several commands in which case you need to enclose it in brackets `{ }`

Shortening

In the above example we created the `sequence` first i.e. `a <- 1:n` and then set `i` to cycle through it in the `for()` statement. Generally this is overkill in coding and you can just enter the `sequence` you want to use directly in the loop itself i.e.

```
n <- 8
for(i in 1:n) {
  print(i)
}
```

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

Note that here the difference in how R and Python create sequences means that if you're not careful you can get different results.

1.2 `if` and `if else` statements

Often when in the loop of a function you will want to do different things depending upon the iteration you are on. You can do this using the `if` statement

```
# If Else statements
x <- c(-3, 5, 12)
y <- rep(NA, length(x))
for(i in 1:length(x)) {
    if(x[i] >= 0) {
        y[i] <- x[i]
    } else {
        y[i] <- -x[i]
    }
}
y
```

```
[1]  3  5 12
```

Syntax

Again the syntax is almost identical to Python:

```
if (condition) true.branch else false.branch
```

The `else` part is optional and the `true.branch` or `false.branch` can be compound statements enclosed in `{ }`.

Multiple conditions

In R (unlike Python) there is no `elif` command to split conditioning into multiple classes. Instead, if you want to check multiple expressions for `TRUE` and execute the first block of code where the accompanying condition is `TRUE` then you have to nest `else ... if` statements within one another. For example:

```
# This is only one value of x, try others to reach the different blocks
x <- -2
if (x < 0) {
  cat(x, "is less than zero \n")
} else if (x <= 4) {   # Note it will only consider this if (x < 0) is FALSE
  cat(x, "is between 0 and 4 \n")
} else { # It will only consider this if both (x < 0) and (x <= 4) are FALSE
  cat(x, "is bigger than 4 \n")
}
```

```
-2 is less than zero
```

```
# Note it will only consider the statement for any block if all the
# previous statements are FALSE so you only enter a single one of the blocks
```

There are other structures in R to address this, e.g. `switch`; we will not cover them formally, but you can of course investigate and use them.

1.3 Notes on conditions

Avoid vector conditions

You need to be careful when you are evaluating conditions in R particularly within an `if` statement. You need to make sure that the condition you are evaluating is only of length 1; anything longer than that doesn't really make sense. For example try running

```
# Conditions
x <- c(1,-2,3)
if(x < 0) print("Hi!")
```

```
Warning in if (x < 0) print("Hi!"): the condition has length > 1 and only the
first element will be used
```

```
x <- c(-1, 2, -3)
if(x < 0) print("Hi!")
```

```
Warning in if (x < 0) print("Hi!"): the condition has length > 1 and only the
first element will be used
```

```
[1] "Hi!"
```

This happens because the condition `x>0` is a vector of length 3 whereas R is wanting a single `TRUE` or `FALSE`. What R does in this situation is use only the first condition value and a warning is issued.

You should make sure that when you have any condition it takes a single `TRUE` or `FALSE` value.

1.4 Boolean Operations

We can create compound Booleans using the commands `&` `|` `&&` `||` which perform AND/OR operations.

Elementwise operations with `&` and `|`

Single operators `&` `|` work like normal arithmetic operators, acting elementwise on vectors. Have a look at the help file: `?"&"`.

```
# Booleans
a <- c(TRUE, TRUE)
b <- c(FALSE, TRUE)
a&b
```

```
[1] FALSE  TRUE
```

```
a|b
```

```
[1] TRUE TRUE
```

Note: This is not what we will want in an `if` condition statement

The longer operators `&&` and `||`

The longer form of the comparisons will examine only the first element of each vector. Compare the previous results with

```
a&&b
```

```
[1] FALSE
```

```
a||b
```

```
[1] TRUE
```

The terms here are evaluated from left to right and evaluation will stop once the result has been determined.

Good programming etiquette means that when you are using compound Boolean terms in `if` conditions you use the longer comparison e.g. `&&` and make sure that you are evaluating conditions of length 1.

1.5 The `break` and `next` commands

The `break` command

Sometimes you might want to end a loop early if a certain condition has been met. In this case we use the `break` command e.g.

```r
# The break command
for(i in 1:10) {
    print(i)
    if(i == 5) break
}
```

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

```r
i
```

```
[1] 5
```

This will cause the computer to jump out of the loop when the break is encountered.

The `next` command

Alternatively we may wish to sometimes just skip to the next iteration of the loop in which case we would want to use the `next` command.

```r
# The next command (look carefully at the output)
for(i in 1:10) {
    if(i == 7) next # Skip all lines below this and start next loop iteration
    print(i)
}
```

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

1.6 Tasks

1. Create a vector `names` containing the names `"Kate"`, `"Bob"` and `"Ioana"`. With a `for` loop create the output:

   ```
   Kate studies MAS115
   Bob studies MAS115
   Ioana studies MAS115
   ```

   Hint: To do this efficiently you will want to use the `cat` command so have a look at `?cat` beforehand. This command allows you to *concatenate and print* strings together. For example, try the following

```
a <- 1
cat(a, "is a number", "\n")
```

We initially define `a` in the first line. We then wish to join this variable with two strings `"is a number"` and `"\n"` and print the result to the screen. The `"\n"` is important — it represents the newline character and means the output will jump onto a new line once you have finished. Note also the fact that you don't need quotes `""` around the `a` since it is a defined variable but that you do around the other strings.

2. Define s_i to be the sum of the squares of the first i natural numbers i.e.

$$s_i = \sum_{j=1}^{i} j^2$$

Using a `for` loop, create a vector `ssquares` of length 10 whereby element `[i]` contains s_i.

Note that you will need to tell the computer that you want it to create a `ssquares` vector of length 10 **before** you start filling it in. This can be done by

```
ssquares <- rep(NA, 10)
```

If you don't do this the computer won't know where you are wanting it to store the values you create.

3. We have a function

$$f(x) = \begin{cases} 0 & x < 0 \\ 3x & 0 < x < 1 \\ 5x^2 - 2 & x > 1 \end{cases}$$

   (a) Create a vector `x` of length 1000 containing evenly spaced values between -1 and 3 (have a look at `?seq`).

   (b) Using a for loop and nested if statements create a vector `f` containing the values $f(x)$.

   (c) Plot the graph of $f(x)$ (have a look at `?plot`).

2 Vectorization

We have already seen that many calculations can be carried out on a whole vector at a time. For example, for a general vector `a`, commands such as `a < 3` will give another (logical) vector checking if each element in `a` is less than 3 in turn.

Such calculations could of course be done in a loop instead. For example

```
n<-length(x)
z<-rep(NA,n)
for (j in 1:n)
   z[j]<-x[j]+y[j]
```

is equivalent to `z <- x+y`, assuming `x,y` are the same length. In general, the vectorized form is faster to write, easier to read, and will run more quickly.

If we want to do two different things to elements in the vector(s), depending on some Boolean calculation, then usually that can be vectorized too. Again, we saw an example in Lab 1. It is often clearer to break down the calculation into several vectorized steps, for example to calculate some Boolean variable and then act on it; this is still likely to be (much) faster then calculation in a loop.

In addition, there are some built-in functions that vectorize common cases. You should be aware of `pmax` (for 'parallel maximum'); experiment and compare it with the `max` function, when comparing vectors.

2.1 Tasks

Repeat the tasks from §1.6 without using loops. You may want to look up the functions `paste` (and possibly `paste0`) and `cumsum`.

3 Homework — due 12pm Thursday week 3

Your solutions must be in the form of a PDF document produced from an R markdown file, including a suitable title and your name in the header and code, results and explanation for each task. Please submit your PDF via blackboard by **12pm Thursday 25th February 2021**.

1. **Estimating π again**

   While you may wish to use loops to solve this problem, it doesn't require their use—try vectorizing!

   Consider a unit circle, which sits inside the unit square as shown in Figure 1.

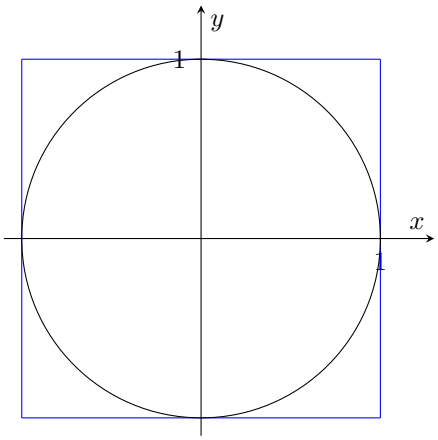

Figure 1: The unit circle in the unit square

   (a) Write an R script which randomly generates 10,000 points (x, y) in the unit square (that is, with $0 \leq x \leq 1$ and $0 \leq y \leq 1$) and uses the number that sit inside the circle to obtain an estimate for $\pi/4$, and hence π.

   Try to write your script in a way that makes it easy to use a different number of sample points.

   (b) Based on the discussion of Monte Carlo methods in the lecture, how accurate would you expect your estimate to be?

   You may wish to try coding in R some of the other things you explored in the Semester 1 mini-project, but that is *not* required for this assessment.

2. **Prime numbers** Given a natural number $n > 3$ we want to write some code to find out if it is prime or not. Pseudocode was given in the lecture.

   Note 1: The pseudocode suggests that if you find any number that divides n you don't need to go on through the `for` loop but can instead leave it by using the command `break`.

   Note 2: To work out $\lfloor x \rfloor$ in R we use the `floor()` command e.g. `floor(3.5)` will return 3.

   Note 3: To calculate $n \mod i$ in R we use the `%%` command e.g. `7 %% 2` will return 1

   (a) Write code that implements the pseudocode from the lecture for a given integer n.

(b) Does your code work if you choose $n = 2, 3$? If not then write a simple `if` statement which will check for these two numbers and return `prime` as `TRUE`.

(c) Modify your code to return all the prime numbers less than 10000 in a vector. How many are there? What is the 1000^{th} prime? *Answer: There should be 1229 and the* 1000^{th} *is 7919.* **Hint:** It's easiest to create a vector called `pvec` to store them in. Initialise this vector to store the first primes (2 and 3) before checking each number from 4 to 10000 for "primeness". If you find any number `i` to be prime then append it to the vector of primes `pvec` using the command `c(pvec, i)`.