MAS115 R programming 2016-17
Lab Class 3

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
x <- 0 # Counter initialised to zero


a <- 1:10
for(i in a) {
   print(i)
   x <- x + i
}
x
```

Syntax

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

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

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.

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

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
```

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")
}
# 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
```

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. For example try running

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

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

```
# Booleans
?'&' # Have a look at the help file first
a <- c(TRUE, TRUE)
b <- c(FALSE, TRUE)

a&b
a|b
```

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
a||b
```

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

```
(0>10) & ("a"/2)
(0>10) && ("a"/2)
```

Note that in the second `&&` command the nonsense second evaluation isn't performed since we already know the compound statement must be `FALSE` from the first condition.
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.

```
# The break command
for(i in 1:10) {
   print(i)
   if(i == 7) break
}
i
```

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,

```
# 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)
}
```

2 Inputting values

So far, we have set values in our R code by using assignment within a script (i.e. using '`<-`'). When programming, we want to be able to set values interactively; we also want to be able to read in data sets that are not already available in R. Values read in can be used in calculations and to control the program flow. In R there are several functions for different input tasks; for now, we will look briefly at some important ones.

2.1 scan()

The most basic input function is `scan()`. It has many optional arguments to control its behaviour, but at its simplest it reads in numerical values, one at a time and each followed by Return, until a blank line is read. To read a fixed number of values, use the optional argument `n`; to reduce the amount of output, try `quiet=TRUE`. The argument `what` determines the type of input expected, defaulting to double; for other types, set `what` to a *call* to the function to create that type, e.g. `what=integer()`, `what=logical()`, or `what=character()`; the final case can be conveniently abbreviated to `what=""`.

For example, the following command expects to read in a single character string (which can by typed by the user without quotes), and will store it as `my.name`.

```
my.name <- scan(n=1,what="")
```

The `scan` function can also be used to read in much more complex data, usually from files, with the details controlled by the various optional arguments. However, in such cases it is often easier to use higher-level functions, such as `read.table` (described below in §2.3).

2.2 readline()

The `readline()` function prints out a prompt to the user, and then reads in a single line of input. The value is treated as a character string, but note that `as.numeric` can be used to convert a strong to a number, where that makes sense.

2.3 read.table()

The function `read.table` is designed to read data in a regular format from a file, creating a data frame. For this task it is easier to use, but a bit less flexible, than `scan`.

To reproduce the same data frame as in Lab Class 2, create a text file (not a script), called `Salary.dat` say, containing the following information.

```
subject age uni salary
Maths 25 Sheffield 27000
English 47 Leeds 45000
Physics 64 Manchester 60000
Maths 28 Nottingham 42000
```

Then use the command

```
grads <- read.table("Salary.dat", header=TRUE)
```

What happens if you omit the `header` argument?

3 Lab class tasks

3.1 Printing names

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 consider the following

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

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 is the newline character and means the output will jump onto a newline 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.

3.2 Square Numbers

Using a for loop, create a vector `ssquares` of length 10 whereby element `[i]` contains the sum of the squares of the first `i` natural numbers i.e.

$$\texttt{ssquares[i]} = \sum_{j=1}^{i} j^2$$

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.

3.3 Altering value of a vector

We have a function

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

- Create a vector `x` of length 1000 containing evenly spaced values between -1 and 3 (have a look at `?seq`).
- Using a for loop and nested if statements create a vector `f` containing the values $f(x)$.
- Plot the graph of $f(x)$ (have a look at `?plot`)

4 Homework — due practical class week 4

Your solutions must be clearly structured and be written in such a way that they are readable and understandable to the marker. Do **NOT** simply submit raw R code and output without any real world explanation.

4.1 Part I - Estimating π again

While you may wish to use loops to solve this problem, it doesn't require their use if you remember that R can work with vectors — for a general vector `a`, commands such as `a < 3` will give another vector checking if each element in `a` is less than 3 in turn.
Consider the unit quarter-disc in the first quadrant, which sits inside the unit square as shown in Figure 1.

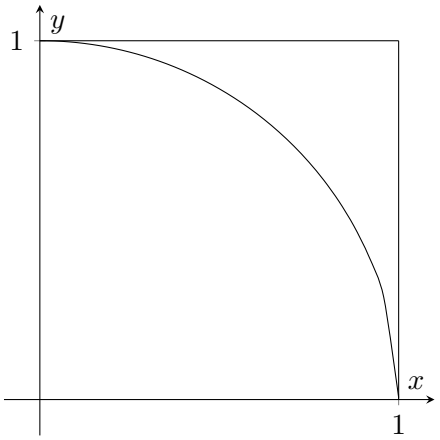

Figure 1: The unit quarter-disc in the unit square

4.1.1 Task

- 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 quarter-disc to obtain an estimate for $\pi/4$, and hence π.

4.2 Part II - Prime Numbers

4.2.1 The problem and pseudo-code

Given a natural number $n > 3$ we want to write some code to find out if it is prime or not.

Pseudo-code

```
INITIALIZE variable prime <- TRUE

FOR (each natural number i = 2, ..., ⌊√n⌋ )
```

```
  IF (n mod i == 0)
    THEN set  prime <- FALSE
    BREAK out of for loop
  ENDIF
ENDFOR
RETURN  prime
```

Note 1: Here you should note that the pseudo-code suggests that if you find any number that divides n you don't need to go on through the `for` loop but can instead break it using the command `break`.

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

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

4.2.2 Tasks

- Write code that implements the above pseudo-code for a given integer. `n`
- 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`.
- 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)`.
- Optional, and *not* for handing in: write pseudo-code for a program that repeatedly reads in a number from the user, and then checks whether it is prime, until it reads a number that is zero or less, at which point it stops. Modify your R code to operate in this way.