Lecture 3:

Simple programming techniques in R

Programming: the basic toolkit

Conditional execution: if some condition holds, *do this* else *do that*

Repeat an operation a fixed number of times: for each value in a set
do this

Repeat an operation until some condition is satisfied: while the condition isn't satisfied, *do this*

Packaging commonly-used code into single commands: function to perform a specific task

`if` statements

- **Purpose:** do different things in different situations depending on some condition(s) hold

`if` statements

- **Purpose:** do different things in different situations depending on some condition(s) hold
- **Syntax:** `if (`*condition*`)` *statement1* `else` *statement2*

`if` statements

- **Purpose:** do different things in different situations depending on some condition(s) hold
- **Syntax:** `if (`*condition*`)` *statement1* `else` *statement2*
  - *condition* is an expression that evaluates to either `TRUE` or `FALSE` (**NB** a single value, not a vector!)

`if` statements

- **Purpose:** do different things in different situations depending on some condition(s) hold
- **Syntax:** `if (`*condition*`)` *statement1* `else` *statement2*
  - *condition* is an expression that evaluates to either `TRUE` or `FALSE` (**NB** a single value, not a vector!)
  - *statement1* is either one command, or a group of commands enclosed in braces `{ }`. It is only executed if *condition* is `TRUE`.

`if` statements

- **Purpose:** do different things in different situations depending on some condition(s) hold
- **Syntax:** `if (`*condition*`)` *statement1* `else` *statement2*
  - *condition* is an expression that evaluates to either `TRUE` or `FALSE` (**NB** a single value, not a vector!)
  - *statement1* is either one command, or a group of commands enclosed in braces `{ }`. It is only executed if *condition* is `TRUE`.
  - The `else` *statement2* part is optional, but when used *statement2* is executed when *condition* is `FALSE`.

`if` statements

- **Purpose:** do different things in different situations depending on some condition(s) hold
- **Syntax:** `if (`*condition*`)` *statement1* `else` *statement2*
  - *condition* is an expression that evaluates to either `TRUE` or `FALSE` (**NB** a single value, not a vector!)
  - *statement1* is either one command, or a group of commands enclosed in braces `{ }`. It is only executed if *condition* is `TRUE`.
  - The `else` *statement2* part is optional, but when used *statement2* is executed when *condition* is `FALSE`.

Simple example

```
> x <- 3
> if (x>0) sqrt(x)
[1] 1.732051
> x <- -4
> if (x>0) sqrt(x)
>
```

`if` statements: another example

Assigning a value to a group

```
> x <- 23
> if (x<10) {
+   Group <- 1
+ } else if (x<20) { # Only get here if
+   Group <- 2        # x is at least 10
+ } else if (x<30) { # And only get here if
+   Group <- 3        # x is at least 20
+ }
> Group
[1] 2
```


- **NB** repeated `else` clauses for different conditions, with braces and spacing used to help readability
- **NB** also: as here, `if` construction is often clumsy: avoid if possible! Alternative for this example:

```
> Group <- x%/%10 + 1
```

`if` statements: even more examples

Testing whether an object exists

```
> if (!exists("ustemp")) load("UStemps.rda")
```

- `exists()` command returns `TRUE` if object exists, `FALSE` otherwise
- **NB** '`!`' means 'not' so `!exists("ustemp")` is `TRUE` if `ustemp` *doesn't* exist, `FALSE` otherwise
- No `else` clause used here

`if` statements: even more examples

Testing whether an object exists

```
> if (!exists("ustemp")) load("UStemps.rda")
```

- `exists()` command returns `TRUE` if object exists, `FALSE` otherwise
- **NB** '`!`' means 'not' so `!exists("ustemp")` is `TRUE` if `ustemp` *doesn't* exist, `FALSE` otherwise
- No `else` clause used here

Avoiding opening too many graphics windows

```
> if (dev.cur()==1) x11(width=8,height=6)
```

- `dev.cur()` is number of current graphics device: 1 means 'no graphics device open'.
- Remember use of `==` to test that two values are the same: `dev.cur()==1` is `TRUE` if there is no graphics device open, `FALSE` otherwise.
- Similar code used in Workshop 1

`for` loops

- **Purpose:** Repeat a statement (or group of statements) several times, with different variable / object values at each iteration

`for` loops

- **Purpose:** Repeat a statement (or group of statements) several times, with different variable / object values at each iteration
- **Syntax:** `for (`*index variable* `in` *vector*`)` *statement(s)*

`for` loops

- **Purpose:** Repeat a statement (or group of statements) several times, with different variable / object values at each iteration
- **Syntax:** `for (`*index variable* `in` *vector*`)` *statement(s)*

Example: a simple `for` loop

```
> for(i in 1:5) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

`for` loops

- **Purpose:** Repeat a statement (or group of statements) several times, with different variable / object values at each iteration
- **Syntax:** `for (index variable in vector) statement(s)`

Example: a simple `for` loop

```
> for(i in 1:5) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```


. . . and a better way without a loop!

```
> print(1:5)
[1] 1 2 3 4 5
```

Blocks and braces

- To execute more than one statement in a loop, use blocks within braces `{...}` in the same way as for `if` statements:

Example: cumulating sums

```
> sum1 <- 0
> sum2 <- 0
> for (i in 1:5) {
+   sum1 <- sum1 + i
+   sum2 <- sum2 + i^2
+   cat("i =",i," Sum =",sum1,
+       " Sum of Squares =",sum2,"\n")
+ }
i = 1  Sum = 1  Sum of Squares = 1
i = 2  Sum = 3  Sum of Squares = 5
i = 3  Sum = 6  Sum of Squares = 14
i = 4  Sum = 10  Sum of Squares = 30
i = 5  Sum = 15  Sum of Squares = 55
```

`for` loops: more examples

- Note that the "`in`" vector can be numeric, character or logical.

Example: transforming a vector of values

```
> for (theta in c(0, pi, 2*pi)) print(sin(theta))
[1] 0
[1] 1.224606e-16
[1] -2.449213e-16
```

`for` loops: more examples

- Note that the "`in`" vector can be numeric, character or logical.

Example: transforming a vector of values

```
> for (theta in c(0, pi, 2*pi)) print(sin(theta))
[1] 0
[1] 1.224606e-16
[1] -2.449213e-16
```


Example: looping over a character vector

```
> LETTERS   # R knows the alphabet!
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" [snip] "T"
[21] "U" "V" "W" "X" "Y" "Z"
> for(let in LETTERS[c(8,5,12,16)]) cat(let); cat("\n")
HELP
```


- **NB** in this example, only `cat(let)` is part of the loop.

`while` loops

- **Purpose:** Repeat a procedure while some condition holds (or 'until the condition no longer holds')

`while` loops

- **Purpose:** Repeat a procedure while some condition holds (or 'until the condition no longer holds')
- **Syntax:** `while (`*condition*`)` *statement(s)*
  - *condition* is an expression that evaluates to either `TRUE` or `FALSE`

`while` loops

- **Purpose:** Repeat a procedure while some condition holds (or 'until the condition no longer holds')
- **Syntax:** `while (`*condition*`)` *statement(s)*
  - *condition* is an expression that evaluates to either `TRUE` or `FALSE`

Example: what is the first factorial number that is greater than 10 000?


```
> n <- 0
> prod.sofar <- 1
> while (prod.sofar<10000) {
+   n <- n+1
+   prod.sofar <- prod.sofar*n
+ }
> prod.sofar
[1] 40320
>  n
[1] 8
```

Beware the Evil Loop of No Return

What happens if you try this?

Just an innocent little loop

```
> x <- 2.1
> y <- 2.5
> while (x<y) print(x)
```


Beware the Evil Loop of No Return

What happens if you try this?

Just an innocent little loop

```
> x <- 2.1
> y <- 2.5
> while (x<y) print(x)
```


Some remedies if you find yourself on the infinite staircase

- If you think it might happen, build a stopping criterion into the `while` *condition* (see workshop); or use `break` statements inside the loop
- In an emergency, press the  button in RStudio

Loops and `if`s: caveats

- R is an interpreted language: each line of code is interpreted as it is encountered and then executed
  - Compare with compiled languages, where entire programs are compiled into machine code before execution

Loops and `if`s: caveats

- R is an interpreted language: each line of code is interpreted as it is encountered and then executed
  - Compare with compiled languages, where entire programs are compiled into machine code before execution
- In an R loop, each line of the loop gets interpreted at every iteration!

Loops and `if`s: caveats

- R is an interpreted language: each line of code is interpreted as it is encountered and then executed
  - Compare with compiled languages, where entire programs are compiled into machine code before execution
- In an R loop, each line of the loop gets interpreted at every iteration!
- Therefore, loops in R are computationally inefficient (i.e. slow) and should be avoided if possible.

Loops and `ifs`: caveats

- R is an interpreted language: each line of code is interpreted as it is encountered and then executed
  - Compare with compiled languages, where entire programs are compiled into machine code before execution
- In an R loop, each line of the loop gets interpreted at every iteration!
- Therefore, loops in R are computationally inefficient (i.e. slow) and should be avoided if possible.

Ways to avoid loops and `ifs`

- Object-oriented thinking: operate on entire objects where possible, not their individual parts
  - Exploit existing R functions such as `apply()`, `tapply()`, `lapply()`, `sapply()`, `aggregate()`, `sum()`, `prod()`, `cumsum()`, `cumprod()` etc.
- Use subsetting rules (square brackets `[]`) and clever arithmetic to avoid `if()` statements.

Functions

- **Purpose:** define a single command to carry out some procedure so that it can easily be repeated in many different situations

Functions

- **Purpose:** define a single command to carry out some procedure so that it can easily be repeated in many different situations
- **Syntax:** `function(`*arguments*`)` *code to perform procedure*
  - *arguments* are named "inputs" to function, separated by commas
  - *code to perform procedure* may be a block enclosed in braces {}

Functions

- **Purpose:** define a single command to carry out some procedure so that it can easily be repeated in many different situations
- **Syntax:** `function(`*arguments*`)` *code to perform procedure*
  - *arguments* are named "inputs" to function, separated by commas
  - *code to perform procedure* may be a block enclosed in braces {}

Example: to compute logarithm of `x` to base `a`

```
> loga <- function(x, a=10) log(x)/log(a)
> loga(10)
[1] 1
> Pig.In.A.Wig <- c(10,32,81)
> Fish.In.A.Dish <- c(10,2,3)
> loga(Pig.In.A.Wig, Fish.In.A.Dish)
[1] 1 5 4
```

- Two arguments: `x` and `a`.
- If no value given for `a`, function uses *default* value `a=10`.

Functions: notes

- Enable you to do similar things repeatedly without having to type them each time
- Enable you to implement complex procedures with a single command
- Make your code more readable by referring to a large chunk of code with a sensible name
- Help prevent bugs and errors: only one copy of code for a procedure
- Enable you to develop programs and algorithms using 'building blocks'

Functions: notes

- Enable you to do similar things repeatedly without having to type them each time
- Enable you to implement complex procedures with a single command
- Make your code more readable by referring to a large chunk of code with a sensible name
- Help prevent bugs and errors: only one copy of code for a procedure
- Enable you to develop programs and algorithms using 'building blocks'

All R commands are functions!

- Hence use of brackets `()` in all commands
- Gives opportunity to customise existing R commands: make a copy and edit the function definition.

Functions: a longer example

Finding out whether `x` is a factorial number

```
is.factorial <- function(n) {
    i <- 0; prod.sofar <- 1 # Initialise values
    while (prod.sofar < n){
        i <- i+1
        prod.sofar <- prod.sofar*i
    }
    prod.sofar==n
}
> is.factorial(6)
[1] TRUE
> is.factorial(34)
[1] FALSE
```


- Value of `is.factorial(x)` is result of last statement executed in function
- 'Value' of function is specified on all R help pages

Functions: Tricks and Hints — leaving early

- Can leave early using `return(value)`

Example: square root of any real number

```
general.sqrt <- function(x) {
#
#  this function returns the square root of
#  any real number x, positive or negative.
#
  if (x>=0) return(sqrt(x))
#
# No "else" needed because if x is
# positive then we don't get this far
#
  complex(real=0,imaginary=sqrt(-x))
}
> general.sqrt(-1)
[1] 0+1i
```

Functions: Tricks and Hints — returning multiple objects

Some options for returning more than one object

- Return a vector of values (must all be of the same type!)
- Return a data frame (to return several vectors of equal length)
- Return a list with named components that can be accessed subsequently with $:

```
function {
    ...
    main body of function
    ...
    list(temp1, lm.res=lm.obj)
}
```


- **Example:** see the help page for `boxplot()`, under 'Value'.

```
> GroupStats <- boxplot(Petal.Length ~ Species, data=iris)
> GroupStats$stats
[snip]
```

Functions: Tricks and Hints — multiple arguments and '. . .'

Example: converting from Fahrenheit to Centigrade

```
convert.temp <- function(degrees.F, plot.wanted=TRUE, ...) {
  degrees.C <- (degrees.F - 32) * 5/9
  if (plot.wanted) plot(degrees.F, degrees.C, ...)
  degrees.C
}
> convert.temp(10*(0:10),
+              xlab=expression(degree*F),
+              ylab=expression(degree*C),
+              main="Centigrade against Fahrenheit")
[snip]
> convert.temp(plot.wanted=FALSE,degrees.F=10*(0:10))
[snip]
```


- Argument '. . .' stands for 'any other user-supplied arguments' (here `xlab`, `ylab` and `main`, passed through to `plot()`)
- Arguments can be supplied in any order when calling function, but must be named if in 'wrong' order

Programming: good practice and recommendations (1)

- Make your code easy for a human to read — it helps when looking for errors! Suggestions:
  - Code should always be well commented using # lines
  - Code should be well spaced (see examples – **NB** also use of indentation to show where loops / functions start and end (RStudio can do this automatically: use `Reindent Lines` on Code menu)
  - Use meaningful object names: **NOT** `a`, `b`, `c`, `d`, ... !

Programming: good practice and recommendations (1)

- Make your code easy for a human to read — it helps when looking for errors! Suggestions:
  - Code should always be well commented using # lines
  - Code should be well spaced (see examples – NB also use e indentation to show where loops / functions start and end (RStudio can do this automatically: use `Reindent Lines` on Code menu)
  - Use meaningful object names: **NOT** `a`, `b`, `c`, `d`, ... !
- Avoid object names that already exist in R: e.g. `mean`, `sum`, `t` etc.
  - To find out if a name already exists in R, type it.


```
> sd
function (x, na.rm = FALSE)
sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
    na.rm = na.rm))
[snip]
> SD
Error: object 'SD' not found
```

Programming: good practice and recommendations (1)

- Make your code easy for a human to read — it helps when looking for errors! Suggestions:
  - Code should always be well commented using # lines
  - Code should be well spaced (see examples – NB also use of indentation to show where loops / functions start and end (RStudio can do this automatically: use `Reindent Lines` on Code menu)
  - Use meaningful object names: **NOT** `a`, `b`, `c`, `d`, ... !
- Avoid object names that already exist in R e.g. `mean`, `sum`, `t` etc.
  - To find out if a name already exists in R, type it.

```
> sd
function (x, na.rm = FALSE)
sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
    na.rm = na.rm))
[snip]
> SD
Error: object 'SD' not found
```

- Look for efficient ways of doing things — e.g. avoid loops unless absolutely necessary

Programming: good practice and recommendations (2)

- Write function definitions in an R script, then use `source()` to define them to your R session.
- Think about possible values of inputs that could cause problems when writing functions, and try to 'trap' them.

```
MeanExcess <- function(x, threshold) {
#
#  Calculate mean of exceedances of a vector x over
#  a threshold i.e. the mean of the values (x-threshold)
#  where these values are positive
#
   BigX <- (x>threshold)      # Elements are TRUE or FALSE
   if (!any(BigX)) return(0) # There may be no exceedances!
   mean(x[BigX])             # Only get here if there *are* some
}
```

- Always name arguments in complicated function calls, so that there's no ambiguity about what you intend
- Clear workspace using `rm(list=ls())` before testing anything: then you know that R isn't using information from old objects without telling you