# MAS115 R programming, Lab Class 6

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

2020-21

## 1 Writing functions

### 1.1 Introduction to Functions

R has a lot of built in functions that you have already used e.g. `sum()`, `nrow()`, `runif()`, .... Sometimes however we might wish to write our own. We would typically do this if we are going to be reusing bits of code, potentially with different input parameters. Defining our own functions allows us to do this simply and in a way that makes our code easier to organise and keep track of.

There is nothing very special about functions; they just provide a nice wrapper to keep your lines of code contained. Here is an example that we will explain step by step:

```r
# A hand written version of a two-sided t-test
# Inputs: x vector of observed data
# Outputs: p-value of the test
my_t_test <- function(x) {
  n <- length(x)
  t <- sqrt(n) * (mean(x))/sd(x)
  pval <- 2 *(1-pt(abs(t), n-1))
  return(pval)
}
```

Run this code in R. Initially no output will be produced but the object `my_t_test`, the function itself, will now be stored in your workspace. To actually use the function you need to run it using `()` brackets. Note that the internal name of the input, here `x`, is a local 'dummy' name; it does not have to match the external name of the object it operates on. Try the following code:

```r
my_t_test # See the function itself
```

```
function(x) {
  n <- length(x)
  t <- sqrt(n) * (mean(x))/sd(x)
  pval <- 2 *(1-pt(abs(t), n-1))
  return(pval)
}
```

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

```
a <- c(0.41, 1.03, -0.52, -0.36, 2.18, 0.07, 2.32, 1.62, 0.95, 0)
my_t_test(a) # Run the function on your observed data
```

```
[1] 0.04076888
```

**Syntax**

The general syntax of a function in R is as follows:

```
# Comments explaining function
my_function_name <- function(x)
{
  do first thing with x
  do next thing with x
  . . .
  return the result
}
```

Inside the `{ }` brackets we place the lines of code we wish to run, which can be any commands that you wish. There are however some special commands that are worth explaining.

- `my_function_name` — you can give your function any name that you wish, as long as it doesn't clash with reserved words (although one that relates to its use is a good idea). In the initial example we called our function `my_t_test`. When we want to run the function it is this name which we need to call e.g. `my_t_test(a)`.
- `<- function(x)` — the first part of this tells R that we are creating a function. We then have to specify which variables we are going to be using in the function (we call these the *inputs* or *arguments*). In our example the input is going to be a single vector `x` but we can specify multiple inputs if we want as we will illustrate later.
- `return()` — when our function has finished and we want to output the result we use the `return()` command. In the brackets we specify the name of the variable (*output*) that we want returned. In our example, we are returning the p-value for the t test.
  *If you do not use the `return()` command then R will return the last evaluation it performed, by default. Some coders use this style of programming but explicit `return()` calls make code easier to understand.* For a very short function e.g. a one-line calculation, it may be clearer to omit the textttreturn command.
- `# Comments` — it is good programming practice to explain what your function does when you are writing it. As in our t test example I usually give a brief description of the function's aim and then describe what the inputs and outputs are going to be.

## 1.2 Multiple arguments and default values

Often we will wish to pass more than one input to our function. Suppose for example that we wish to be able to perform a two-sided t test for a general $H_0 : \mu = \mu_0$ as opposed to solely $\mu_0 = 0$ which was all our first example was capable of testing. We can do this by listing multiple inputs when we define the function:

```
# Adding more arguments (with a default value)
my_t_test2 <- function(x, mu = 0) {
  n <- length(x)
  t <- sqrt(n) * (mean(x - mu))/sd(x)
  pval <- 2 *(1-pt(abs(t), n-1))
  return(pval)
}
```

This means that the function `my_t_test2` can now take two arguments: `x` which will be the vector of observed values and `mu` which will be the value for the mean under the null hypothesis. Try running:

```
restest <- my_t_test2(a, mu = 1)
restest
```

```
[1] 0.4939469
```

### Default Values

We may wish to have default values for our parameters, in which case we can specify them when defining the function. In the case above we have given `mu` a default value of 0.

### Calling functions with multiple arguments

When a function has multiple arguments then you can either call the function keeping the arguments in the same order:

```
my_t_test2(a, 1)
```

```
[1] 0.4939469
```

or use names for each argument

```
my_t_test2(mu = 1, x = a)
```

```
[1] 0.4939469
```

If any inputs are not specified then they will take the default values (so long as they have them), so the following are equivalent:

```
my_t_test2(a)
```

```
[1] 0.04076888
```

```
my_t_test2(a, mu = 0)
```

```
[1] 0.04076888
```

**IMPORTANT:**

- When you run the function you should provide it with values for all inputs which do not have default values as otherwise there will be problems when it tries to perform calculation with those undefined variables. If the input is not referred to *at all*, for example because it is not relevant given the values of other inputs, then it can be omitted without causing an error. Another possibility that sometimes make sense, in more elaborate functins, is to have a default value of `NULL`.
- Every function should be self-contained and only get information through the arguments you pass it. Do NOT use any variables which you have not directly passed yourself to the function through its arguments. For example do not create a variable `z` in your workspace which is then used in a function unless you specifically pass that variable into the function via its inputs. It should be possible to work out what a function will do just from its code and the values of its inputs.

## 1.3 Multiple return values

The first time that your computer encounters the `return` command it will end the function and return to the workspace. This might seem like a problem if we wish to return more than one output from our function. In such cases we can either create a vector of values to return using `c()` or create a list. Normally we will give this object names so that it is then easier to understand.

In our t-test example, suppose that as well as the p-value we would also like to return the $t$-statistic calculated and also a boolean which tells us whether or not the p-value is significant at the 5% level. To do this we will create a list containing the relevant bits of information.

```
# Multiple return values with a list
my_t_test3 <- function(x, mu = 0) {
  n <- length(x)
  t <- sqrt(n) * (mean(x - mu))/sd(x)
  pval <- 2 *(1-pt(abs(t), n-1))
  return(list(pval = pval, t = t, signif = (pval < 0.05)))
}
```

Note that we have also given the elements in our list useful names (`pval`, `t` and `signif`) so that when the list is returned we know what it contains. The names can be the same as the variables that fill them if we wish, as in our example of `t = t`. When we run this code we now have access to all the different elements in the list

```
res2 <- my_t_test3(a)
res2
```

```
$pval
[1] 0.04076888

$t
[1] 2.386827

$signif
[1] TRUE
```

```
res2$pval
```

```
[1] 0.04076888
```

## 2 Calculations within your function are local

When we pass variables to functions through the inputs, R makes local copies of them for use in the function. Any changes to those variables which occur within the function do not propagate to the larger global environment. This can be seen in by running the following code:

```r
## Local copies of variables do not change external values
z <- 5
# Add 10 to a number
addten <- function(z) {
    z <- z + 10
    return(z)
}
addten(z)
```

```
[1] 15
```

```r
# But have we changed the external variable z?
z
```

```
[1] 5
```

While sometimes this is beneficial, other times you might actually like to change the variable you pass. In order to do this we need to explicitly return the changed variable through the `return()` argument and then perform a reassignment. To continue the previous (trivial) example, we can increment `z` as follows.

```r
z <- 5
z <- addten(z)
z
```

```
[1] 15
```

This does not depend on the internal naming within the function; the fact that the internal and external names both happen to be `z` is irrelevant. This is of course the same approach we often use with built-in functions. For example suppose that we have a vector $y$ of numbers and we wish to sort them into increasing order. Look at the output of the following:

```r
## Changing variables by return and reassignment
y <- sample(1:100, 10)
y
```

```
 [1] 66 57 79 75 41 85 94 71 19  3
```

```r
y <- sort(y)
y
```

```
 [1]  3 19 41 57 66 71 75 79 85 94
```

The original order is *lost* when doing this, as `y` is over-written. If we need to preserve that information, we can make the assignment to a new variable instead.

```r
y <- sample(1:100, 10)
y_sorted <- sort(y)
```

## 3 Lab class tasks

### 3.1 Prime numbers

Adapt your code from Lab 2 for testing whether a number is prime, turning it into a function that takes a single number as its argument and returns a boolean variable. Make sure your function deals correctly with the special cases when the argument takes the value 1, 2 or 3.

### 3.2 The Fibonacci Sequence

Write a function to generate a Fibonacci sequence. Obviously the sequence is infinite, so you should think about how to specify where the sequence should be truncated; there is more than one possible approach. Try allowing sequences starting with different values.

### 3.3 Dispersion of a pollutant

A fluid dynamics model describes concentration of a pollutant at any point in a region following release from a point source,

$$C(y,z) = \frac{Q}{2\pi u_{10}\sigma_z\sigma_y} \exp\left[-\frac{1}{2}\left\{\frac{y^2}{\sigma_y^2} + \frac{(z-h)^2}{\sigma_z^2}\right\}\right], \tag{1}$$

where the variables have the following meanings: $C$: air concentration of pollutant; $Q$: release rate; $u_{10}$: wind speed at 10m above ground; $\sigma_y$, $\sigma_z$: diffusion parameters in horizontal and vertical directions; $h$: release height; $(y,z)$: coordinates along wind direction and above ground.

Write a function to implement the calculation of concentration given by this equation. The function should have two required arguments, corresponding to $y$ and $z$, and optional arguments corresponding to the other variables with suitable default values: $Q = 100,\ h = 50,\ \log u_{10} = 2,\ \log \sigma_y^2 = 10,\ \log \sigma_z^2 = 5$.

Check that your function works with vector values for $y$ and $z$.

## 4 Homework due 12pm Thursday Week 7

Please submit your PDF via blackboard by **12pm Thursday 25th March 2021**.

### 4.1 Calculation of population variance

The following expression defines a version of the variance of a vector $(x_1,\dots,x_n)$, treating the values as the (equally likely) values of a random variable rather than a sample from a distribution:

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where $\bar{x} = n^{-1}\sum x_i$ vas usual. The following R function calculates this variance.

```r
# Find the `empirical' (or `population' or `uncorrected') variance of a vector
EmpVar <- function(x)
{
  n <- length(x) # Find the length of x
  return ((sum(x^2)-n*mean(x)^2)/n) # Calculate and return empirical variance
}
```

(a) Use this function to calculate the variance of the roll of an ordinary fair 6-sided die.

(b) Modify this function by adding an optional argument so that the user can choose to calculate either the empirical variance or the usual sample variance as the value returned. The default behaviour should be as in the original version of your function.

(c) What happens when the values in `x` are very large? Use your function to calculate the empirical variance of $m+1, \ldots, m+6$ for various large values of $m$, say between $10^6$ and $10^9$. Explain which (if any) of the numerical answers that you get are mathematically correct.

(d) Optional: improve your function in the light of your previous answer.

## 4.2 Newton-Raphson Algorithm

The Newton-Raphson Algorithm is an iterative scheme to find the root of an equation using the first order approximation given by a Taylor series. Suppose that we wish to solve

$$f(x) = 0$$

for a specific function $f(\cdot)$. The Newton-Raphson algorithm proceeds by choosing an initial starting point $x_0$ in the search, and then iteratively updating this value using the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

The sequence of numbers $x_0, x_1, \ldots$ will generally converge to the true root quite quickly although there can be instability if we try values that are near a horizontal asymptote or a local extremum. As the algorithm is sometimes unstable we determine when to stop the Newton-Raphson algorithm if either the $x_i$ stop changing significantly (i.e. if $|x_i - x_{i+1}| < \tau$) or we exceed a pre-specified number of iterations.

### Pseudocode

We can write this in pseudocode as follows:

```
# Pseudocode for Newton-Raphson
SPECIFY tol and maxit
INITIALISE value x = x_0
SET iteration = 0
SET eps = infinity
WHILE (iteration < maxit and |eps| > tol)
  eps =  f(x)/f'(x)
  x = x - eps
  iteration = iteration + 1
ENDWHILE
IF(iteration != maxit)
  THEN converged
  ELSE not converged
ENDIF
```

We will commence by using Newton-Raphson to get an estimate for $\sqrt{2}$ i.e. the root of $f(x) = x^2 - 2$. To start with we will write two very simple functions which work out $f(x)$ and $f'(x)$ for us:

1. Write a function called `f1.func` which takes an argument `x` and returns $f(x) = x^2 - 2$.
2. What is $f'(x)$? Write another function called `f1deriv.func` which takes an argument `x` and returns $f'(x)$.

Having written these functions we can then call them from the main Newton Raphson function when we are working out `eps`. We can now put together our complete Newton-Raphson algorithm:

1. Write a function called `myNR.func` which performs the Newton-Raphson algorithm. It should take the following arguments:
   - `x` — the starting point for the algorithm,
   - `funct` — the function to find zeros of,
   - `deriv` — the derivative of the above function,
   - `maxit` - the maximum number of iterations to perform (with default 100),
   - `tol` - the point at which convergence is assumed to have been reached (default 0.000001).

   You should implement the above pseudo-code and use calls to the functions you just created for $f(x)$ and $f'(x)$. The function should output a list containing the following elements:
   - `x` — the final estimate for the root
   - `iter` — the number of iterations performed,
   - `converged` — a boolean (i.e. TRUE/FALSE) stating whether convergence has been achieved.
2. Use your code to find the solution to $\cos(x) = x$. What is the answer?