University of Glasgow

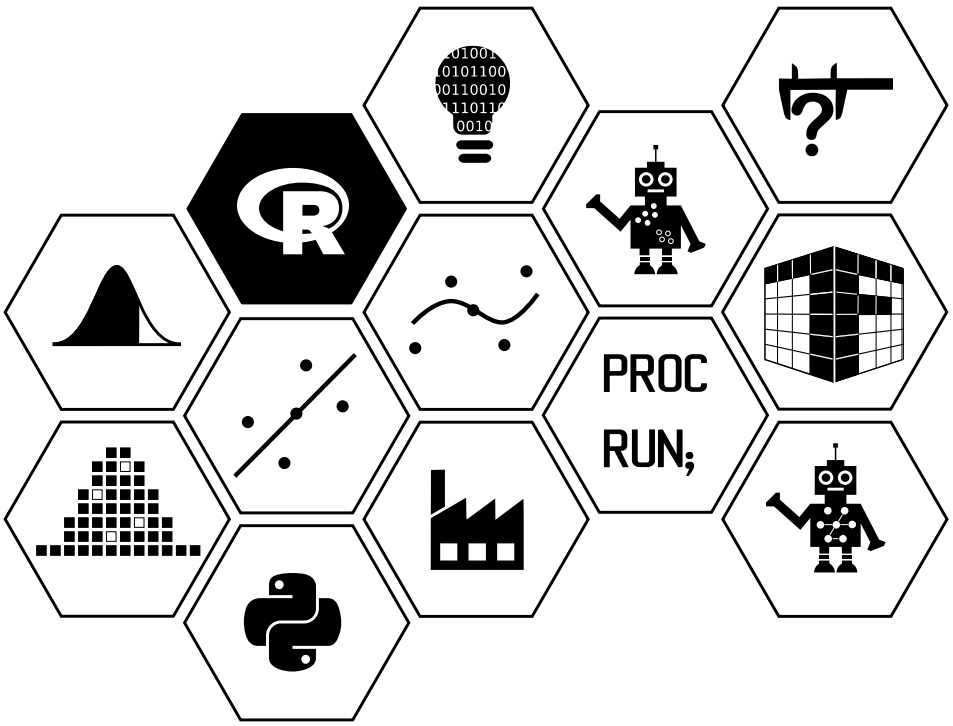
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

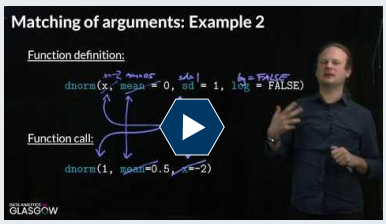
Week 8:

Functions



DATA ANALYTICS GLASGOW

Calling R functions



Calling an R function

https://youtu.be/IUhA7qpgs6Q

Duration: 8m36s

So far, we have used many different R functions: `rnorm`, `cbind`, etc. When we called these functions, we usually passed on some information ("arguments") to the function. For example we used the command `rnorm(10)` to create a sequence of white noise of length 10. This section explains the formal rules R uses to match arguments.

Consider the example of the function `dnorm`. According to its help page, its arguments are:

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
```

`dnorm` computes the p.d.f. of the N(`mean`, $\texttt{sd}^2$) distribution (or the logarithm if `log=TRUE`).

To evaluate the p.d.f. of the $N(0.5, 1)$ distribution at $x = -2$ we can use any of the following lines (and many other variants on the theme):

```
dnorm(-2, mean=0.5, sd=1)
dnorm(-2, 0.5, 1)
dnorm(0.5, 1, x=-2)
dnorm(x=-2, mean=0.5)
```

The reason as to why all the above commands work is that arguments in R are handled in a much more flexible way compared to other programming languages such as C or Java.

In R arguments can be given in two different forms:

- Named form: Arguments are given in the form `name=value`, i.e. `dnorm(x=-2, mean=0.5)`. Arguments in named form can be in any order. The name of the argument does not need to be given in full as long as it is unique, i.e. you can use `dnorm(x=-2, m=0.5)`: in this case R will interpret `m` as `mean`, as there are no other arguments starting with the letter "m".
- Positional form: In positional form no `names` are given. In this case, R matches the arguments by their position. For `dnorm` for example, the first argument is, according to the function definition, `x`, and the second argument is `mean`, thus R interprets `dnorm(-2, 0.5)` as `dnorm(x=-2, mean=0.5)`.

You can also use a mixture of named form and positional form. In this case R first matches the named arguments, and then matches the remaining arguments using their position. Let's look at the example `dnorm(0.5, 1, x=-2)` in more detail. R first matches the named arguments, in our case `x=-2`. The remaining arguments of the function `dnorm` are `mean`, `sd`, and `log`. The remaining arguments of our call are `0.5, 1`, so R interprets this as `mean=0.5` and `sd=1`.

Some arguments have default values and only need to be specified if you want to set this argument to a different value. This is typically (but not always) visible in the function declaration, e.g. `sd` of `dnorm` defaults to 1 ("`sd=1`"). Thus if we do not specify `sd`, we implicitly set `sd=1`.

Some functions allow unspecified arguments ("..."); we will come back to this later.

Task 1.

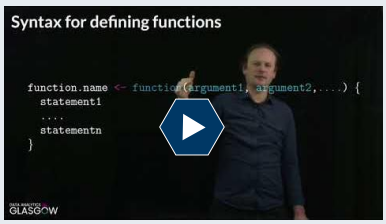
Consider a function taking arguments `a`, `b` and `c` and returning $(a + 2 \times b)^c$, defined as set out below.

```
func <-  function(a, b=0, c=1) {
    (a+2*b)^c
}
```

Rewrite the calls below into named form (i.e. using `parameter=value`) and give the output of `func`.

```
func(1, 2, 3)
func(4)
func(c=3, 2, 1)
```

Defining your own functions



Defining your own functions

https://youtu.be/R_h7IntJ6QU

Duration: 17m15s

Why should I write my own functions?

Functions are one of the most important concepts in programming. Functions allow you to structure complex code.

The idea behind functions is to break a complex problem into blocks of code, each of which is self-contained and performs a specified subtask.

Once we have established that a function performs the specified subtask correctly, we only need to remember how to use the function. We do not need to remember the details of how we implemented the subtask. For example, when you use the function `dnorm` you only need to know that it computes the p.d.f. of the $\mathrm{N}(\mu, \sigma^2)$ distribution, but you do not need to know how it does it.

The three main advantages of using functions are:

- Functions structure your code, thus making reading and maintaining the code much easier.
- Functions enable the reuse of code and thus help reducing the duplication of code. Duplication of code (i.e. copying and pasting of code from one task to another) is generally considered to be bad programming style. Rather than copying your code you should write a function that implements the common task and then call this function for each task. This approach is not only more efficient and less error-prone, but also avoids the so-called "update anomaly": If you find an error in duplicated code, you have to remember where you copied the code to, and correct every copy of the code. If however you had used a function, you would only need to fix the function.
- Functions can be "unit-tested". We can test whether the function "does what it says on the tin" before we integrate into a more complex project.

Furthermore, using a structured approach also makes implementing complex tasks much easier.

How to write your own functions

The syntax for defining functions in R is

```
function.name <- function(argument1, argument2, ...) {
    statement1
    ...
    statementn
}
```

Example 1 (Drawing a circle).

A circle can be drawn in R using the following commands

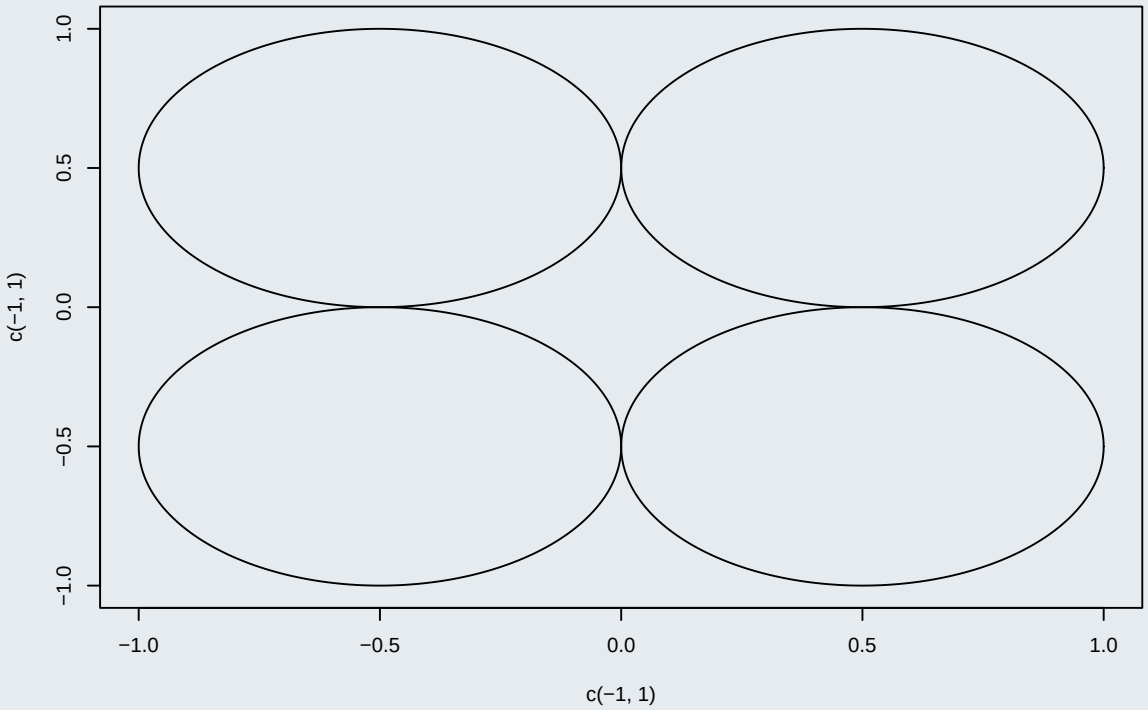
```
plot(c(-1,1), c(-1,1), type="n")    # Set up canvas
x <- 0                              # x-coordinate of centre
y <- 0                              # y-coordinate of centre
r <- 1                              # Radius
t <- seq(0, 2*pi, length=250)
lines(x+r*cos(t), y+r*sin(t))       # Draw circle
```

Suppose you want to draw four circles. Instead of repeating the above commands four times you might want to define a function `circle`. This function should have three arguments: x-coordinate, y-coordinate, and radius:

```
circle <- function(x, y, r) {
  t <- seq(0, 2*pi, length=250)
  lines(x+r*cos(t), y+r*sin(t))
}
```

Now we can draw four circles using

```
plot(c(-1,1), c(-1,1), type="n")                # Set up a canvas
circle(-0.5, -0.5, r=0.5)
circle(-0.5,  0.5, r=0.5)
circle( 0.5, -0.5, r=0.5)
circle( 0.5,  0.5, r=0.5)
```



In the PDF version of the notes, the circles look more like ovals. This is because the scales of the x axis and of the y axis are not equal. If we had used `eqscplot` from `MASS` instead of `plot`, the circles would have been perfect circles.

You can specify default values for some of the arguments using `argument=expression` in the list of arguments. If the user does not provide this argument, the default value (determined by `expression`) is used.

The default value can be a function of other arguments given to the function. Essentially, R evaluates arguments lazily (i.e. only if and when they are needed inside the function body), so when R evaluates `argument` for the first time, it must be able to evaluate `expression`.

Example 2 (Drawing a circle (continued)).

In our function `circle` we might want to set the radius by default to 1.

```
circle <- function(x, y, r=1) {
  t <- seq(0, 2*pi, length=250)
  lines(x+r*cos(t), y+r*sin(t))
}
```

The special argument `...` captures all arguments that are not matched otherwise. It is very useful to pass on additional arguments to a function that you call inside a function without having to worry about the details of these arguments.

Example 3 (Drawing a circle (continued)).

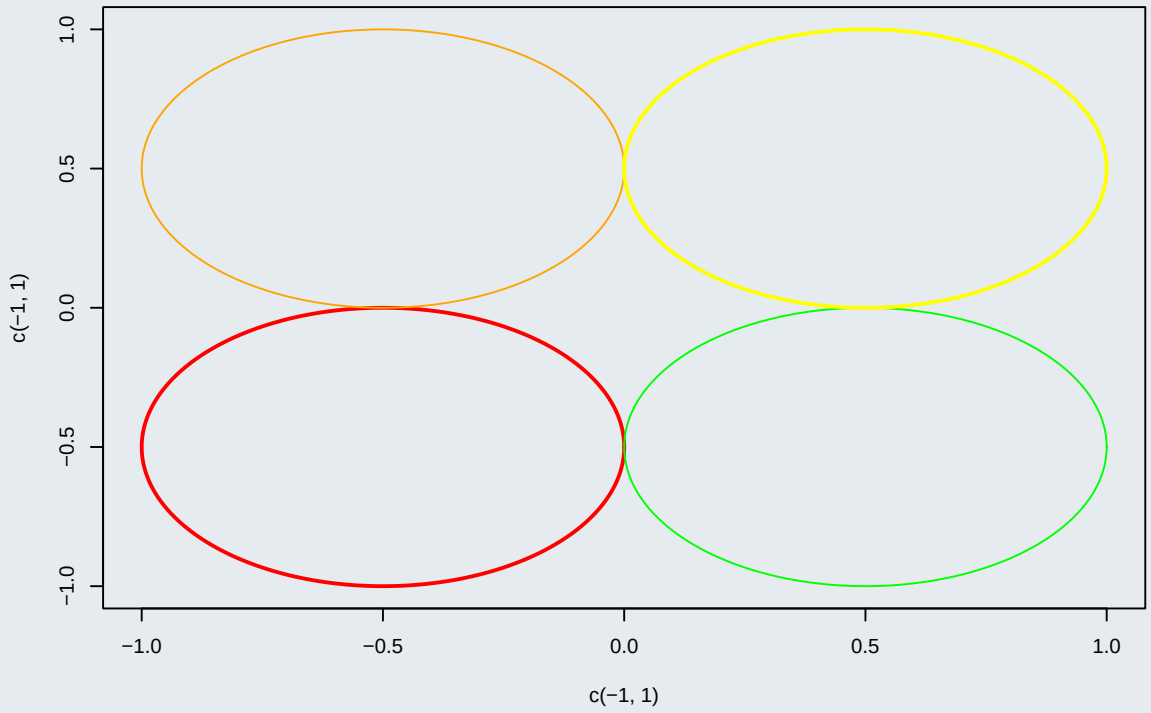
Suppose we want to use the function `circle` to draw circles in different colours and using different line types and widths using arguments like `col`, `lty` and `lwd`. We can then pass on these arguments when we call the `lines` command.

Rather than specifying all possible parameters that can be used to customise lines, we can simply use the special argument `...`.

```
circle <- function(x, y, r=1, ...) {
  t <- seq(0, 2*pi, length=250)
  lines(x+r*cos(t), y+r*sin(t), ...)
}
```

Now we can draw the circles in different colours and line widths:

```
plot(c(-1,1), c(-1,1), type="n")                # Set up a canvas
circle(-0.5, -0.5, r=0.5, col="red", lwd=2)
circle(-0.5,  0.5, r=0.5, col="orange")
circle( 0.5, -0.5, r=0.5, col="green")
circle( 0.5,  0.5, r=0.5, col="yellow", lwd=2)
```



Task 2.

Set up a "canvas" using

```
plot(c(-3,3), c(-3,3), type="n")
```

and set the variables

```
x <- 0
y <- 0
width <- 4
```

Use the function `rect` to draw a square with bottom left corner $(\texttt{x} - \texttt{width}/2, \texttt{y} - \texttt{width}/2)$ and top right corner $(\texttt{x} + \texttt{width}/2, \texttt{y} + \texttt{width}/2)$.

Turn (some of) your code into a function `square` that takes the arguments `x`, `y` and `width` and that draws a square with bottom left corner $(\texttt{x}-\texttt{width}/2, \texttt{y}-\texttt{width}/2)$ and top right corner $(\texttt{x}+\texttt{width}/2, \texttt{y}+\texttt{width}/2)$. Additional arguments (such as `col` or `lty`) provided on to your function `square` should be passed on to `rect`.

Use your function to draw three squares in different colours with centre at $(0, 0)$ and widths 2, 4, and 6.

Returning objects

Often we want functions not only to perform certain tasks, but also to return a value (e.g. the result of a calculation). We can return values using the function `return(object)`. The function `return` terminates the function and returns `object`. When we return `object` from the function we can "capture" this value from outside the function and store it in a variable. If we do not assign the returned result from a function to a variable it will be printed on the screen.

If we do not use `return`, then R returns the value of the last statement of the function body.

Example 4 (Stirling's formula).

Suppose we want to write a function that computes Stirling's approximation to $n!$, which is $\sqrt{2\pi n}n^n \exp(-n)$.

```
stirling <- function(n) {
  approx <- sqrt(2*pi*n) * n^n * exp(-n)
  return(approx)
}
```

Actually, there is no need to use `return` in the above example, as an R function will always return the value of the last statement. Thus we could have used as well

```
stirling <- function(n) {
  sqrt(2*pi*n) * n^n * exp(-n)
}
```

We can now calculate the Stirling approximation to $10!$ using

```
stirling(10)
```

```
## [1] 3598696
```

As the function `stirling` now returns a value, we can assign its result to a variable.

```
exact <- factorial(10)      # Store exact result in exact
exact
```

```
## [1] 3628800
```

```
approx <- stirling(10)      # Store Stirling's approximation in approx
approx
```

```
## [1] 3598696
```

```
exact-approx                # Print the error of the approximation
```

```
## [1] 30104.38
```

Lists are typically used to return more than one value or object.

Example 5 (Stirling's formula (revisited)).

Suppose we want to write a function `stirling.bounds` that returns the bounds for $n!$ obtained from the double inequality

$$\sqrt{2\pi n}n^n \exp\left(-n + \frac{1}{12n+1}\right) < n! < \sqrt{2\pi n}n^n \exp\left(-n + \frac{1}{12n}\right).$$

The function should return both bounds. We can do this using a list with two entries `lower` and `upper`.

```
stirling.bounds <- function(n) {
  approx <- sqrt(2*pi*n) * n^n * exp(-n)
  list(lower=approx * exp(1/(12*n+1)),
```

```
        upper=approx * exp(1/(12*n)))
}
```

We can then calculate the bounds for $10!$ using

```
stirling.bounds(10)
```

```
## $lower
## [1] 3628560
##
## $upper
## [1] 3628810
```

We can calculate the difference between the upper and the lower bound using

```
bounds <- stirling.bounds(10)       # Store result (a list) in variable bounds
bounds$upper-bounds$lower           # Compute difference between bounds
```

```
## [1] 249.9094
```

Task 3.

Write an R function `all.means` that returns for a given vector `x` the arithmetic mean $\frac{\sum_{i=1}^{n} x_i}{n}$, the harmonic mean $\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$, and the geometric mean $\sqrt[n]{\prod_{i=1}^{n} x_i}$. The function should return a list with the elements `arithmetic`, `harmonic`, and `geometric`.

Hint: You might find the function `prod`, which computes the product of a vector, helpful when calculating the harmonic mean.

Inside a function you can use all the control structures we have learned about last week (`if`, loops, …).

Task 4.

Write an R function `quadratic` which computes the real-valued solutions of the quadratic equation

$$ax^2 + bx + c = 0$$

Your function should take a, b, and c as arguments and should return the unique solutions to the above quadratic equation.

- Your function should start with computing the discriminant $\Delta = b^2 - 4ac$.
- If $\Delta < 0$, then the quadratic equation has no solution. In this case, return a vector of length 0.
- If $\Delta = 0$, then the quadratic equation has exactly one ("double") solution, $-\frac{b}{2a}$. In this case return this value only.
- If $\Delta > 0$, then the quadratic equation has two solutions, $-\frac{b+\sqrt{\Delta}}{2a}$ and $-\frac{b-\sqrt{\Delta}}{2a}$. In this case return a vector of length two containing both solutions.

What happens if $a = 0$? Can you also handle this case correctly?

Supplementary material:
Invisible returns

Sometimes we want a function to return some values *only* if its output is assigned to another variable. For instance a function may potentially return a large vector or list which we do not wish to be printed on the console. To achieve this one can use the function `invisible`, i.e. use `return(invisible(object))` rather than just `return(object)`.

Consider the following toy example.

```r
f <- function(x) {
    return(invisible(x^2))
}

f(2)                # prints nothing ...
a <- f(2)           # ... but we can assign the output to a variable ...
a                   # ... which we can then print
```

```
## [1] 4
```

Supplementary material:
Documenting functions

Few functions are fully self-explanatory, so it is always a good idea to document them. At the very least you should put a comment before the function definition (or at the top of the function body) describing what the function does.

If you use the syntax used by the R package roxygen2 it is very easy to later convert your comment into the R documentation files required for R packages. `roxygen2` has been inspired by the doxygen tool for C++, which is similar to the Javadoc format for documenting Java classes. The example below illustrates the format.

```r
#' Compute Stirling's approximation to the factorial
#' @param n numeric vector for which to calculate the Stirling approximation
#' @return the Stirling approximation to $n!$.
#' @details This function is a naive implementation of the formula and for
#'          large n numerically not stable
stirling <- function(n) {
  sqrt(2*pi*n) * n^n * exp(-n)
}
```

More information is available in the package vignette and in the chapter Object documentation from Hadley Wickham's book on R packages.

Background reading: Chapter 19 of R for Data Science

http://r4ds.had.co.nz/functions.html

Chapter 19 of *R for Data Science* gives an introductory overview over functions in R.

Background reading: Advanced R

http://adv-r.had.co.nz/Functions.html

The chapter on functions from *Advanced R* is worth a read if you want to delve little deeper into how functions work in R.

Scoping

Scoping

https://youtu.be/vvAeaoB47V8

Duration: 8m27s

When writing your own functions it is important that you distinguish between variables available locally in the function and variables available in your workspace (or in a parent function, from which your function is being called), i.e. outside your function.

Every variable that exists outside the function can be referenced and used inside the function. However, as soon as you change any of these variables, a local copy will be created: the changes you make to this variable will be only temporary. Once the function has finished running, the changes will be rolled back. Similarly, if you create a new variable, it will only be created temporarily and will be deleted once you exit from the function, unless you return it using the function `return` or by putting the object in the last line of the function body.

In R, all arguments (with the exception of environments) are always passed on to the function by value.

Example 6.

Consider the following function.

```
test <- function(a) {
  print(b)
  b <- a
  print(b)
}
```

This function is certainly not best practice, but a valid R function.

```
b <- 19
test(13)
```

```
## [1] 19
## [1] 13
```

```
b
```

```
## [1] 19
```

Whilst the variable `b` visible inside the function is 13, the variable `b` in the workspace has retained the value 19.

The table below shows how the variables in the workspace and the local variables in the function change as the statements inside the function get run.

Commands	Commands in function	Workspace vars	Local vars	Comment
`b <- 19`	-	b=19	-	-
`test(13)`	`print(b)`	b=19	a=13	Prints 19
	`b <- a`	b=19	a=13, b=13	Local `b` masks `b`
	`print(b)`	b=19	a=13, b=13	Prints 13
`print(b)`	-	b=19	-	Prints 19

The fact that you can access objects available outside functions does not mean you should do this.

Good practice

Functions should be self-contained units, so all inputs should be passed on as arguments and all outputs should be `returned`. Thus when writing a function, you should use inside the function only:

- variables that are passed on to the function as arguments, and
- local variables you have defined inside the function.

You can access variables available only in the workspace, but this is typically (and for good reason) considered to be bad programming style. The following example illustrates why.

Example 7.

Suppose you want to write your own function for computing the mean of a vector `x`.

```
x <- 1:10                           # Set x
n <- length(x)                      # Determine length of x
my.mean <- function(x) {
  x.bar <- sum(x) / n               # Compute mean
  x.bar
}
```

When you want to compute the mean of `x`, you can now run

```
my.mean(x)
```

```
## [1] 5.5
```

which returns the correct mean. So, at first sight everything seems fine. However when we run

```
y <- 1:3
my.mean(y)
```

```
## [1] 0.6
```

rather than 3, the correct value. Why did this happen? When looking at the implementation of `my.mean` we computed the length `n` *outside* the function, thus we assumed that there is a variable `n` in the workspace which holds the length of the vector whose mean we want to compute. This was the case for `x`, but is not the case for `y`.

According to the advice given above, we should not have used `n` in the function without having it first set to the length of `x` inside the function. Thus we should have coded the function `my.mean` as follows:

```
my.mean <- function(x) {
  n <- length(x)                    # Determine length of x - now inside function
  x.bar <- sum(x) / n               # Compute mean
  x.bar
}
```

Designing functions

Programming is, just like mathematical proofs, an art.

The Art of Computer Programming

The Art of Computer Programming is also the title of a monograph by the American Computer Scientist Donald E. Knuth (who also developed the typesetting system TeX). Although written almost 50 years ago, it still is the single most influential book in Computer Science. It is however far from being an easy read. Bill Gates (former CEO of Microsoft and until 2010 the wealthiest man in the world) once said: "If you think you're a really good programmer, or if you want to challenge your knowledge, read The Art of Computer Programming by Donald Knuth. I studied 20 pages, put it away for a week, and came back for another 20 pages. If somebody is so brash that they think they know everything, Knuth will help them understand that the world is deep and complicated. If you can read the whole thing, send me a resume."

Plan before you code

When working on more complex programming tasks, take the time to plan before you start coding. Structuring a problem is often the key step to getting things right. Once you have broken down a problem into small, more manageable pieces it is a lot easier to code these up. There are no cook-book recipes that can guarantee you success. Below are a few hints which I found useful.

Keep in mind that your code doesn't just need to work. It also has to be maintainable both by others and future you. If you structure your code in terms of short well-defined (and documented) functions, your code is a lot easier to read, test, maintain and debug.

Often, going through the following steps helps tackling a complex programming problem:

1. Try to understand the problem you are trying to solve. If you don't know where to start, think of a simple special case.
2. Work out which steps are required to solve the problem. When working on a complex problem, think how you could divide the problem into smaller (and hopefully easier) sub-tasks. It is often a good idea to use a "top-down" design: split your main task into at most a dozen sub-steps. If necessary, split these sub-steps again into "sub-sub-steps" etc. until you end up with small enough steps which are straightforward to implement. It might help writing up your steps as "instructions" in plain English, or using flow diagrams or pseudo-code (use whatever works best for you). For complex problems, this step is by far the most difficult.
3. Translate each of these (sub-)steps into code. Make sure you document your functions properly. For each function explain at least what the function does, what arguments it takes, and what it returns.
4. Check your program. Does it work? Try out a few examples. If you can think of difficult special cases, try your program with these. Do not only test the final main function, but also test the functions implementing the sub-steps individually: this way you are not only more likely to find a mistake, it is also easier to locate what has gone wrong. If any of the functions does not do what it should, you need to find the mistake (more on this in the next section).

Task 5 (The Wheels on the Bus (harder - uses 'stringr')).

The Wheels on the Bus is an American folk song which is nowadays a popular nursery song in the English speaking world. There are many variants, a common one is

```
The wheels on the bus go round and round,
round and round, round and round.
the wheels on the bus go round and round,
all day long.

The wipers on the bus go swish swish swish,
swish swish swish, swish swish swish.
the wipers on the bus go swish swish swish,
all day long.

The horn on the bus goes beep beep beep,
```

```
beep beep beep, beep beep beep.
the horn on the bus goes beep beep beep,
all day long.

The people on the bus go up and down,
up and down, up and down.
the people on the bus go up and down,
all day long.
```

Write an R function or a set of R functions which produce the lyrics as efficiently as possible. Ideally make your function also customisable, that verses can be added.

You can use the base function `paste` or the function `str_c` from `stringr` to concatenate strings.

Unit testing

When programming it is important that we make sure that our programmes work as intended. So every time you have written a piece of code (especially a function), you should test it systematically. This way you can ensure that it is indeed working as intended.

It pays off to confront possible issues as early as possible. It is much easier to test small blocks of code, rather than big blocks of code. Thus it is much better to use many small functions than a few big ones. This way, each of the functions can be tested individually, and once a bug is found, it can be identified fairly quickly.

There are in principle two ways how software can be checked:

- "White-box testing": One can inspect and review the source code of the programme. This is typically only useful if the code is checked by someone other than the author of the programme. However, this makes code review expensive and thus it is often only used for safety-critical or security-critical software. It is however also used for statistical programmes used in clinical trials.
- "Black-box testing": Alternatively, one can apply the function to a number of test cases for which the correct output and behaviour of the function can be predicted. This form of testing is much simpler. However the test cases used might not cover every possible scenario, so some bugs might remain unnoticed.

For more complex projects, unit testing can be automated and structured using the functions from the package testthat (see also the chapter on Testing in Hadley Wickham's book on R packages).

In some settings (e.g. statistical analysis of clinical trials) it is not uncommon to have two programmers implement the same tasks independently. When finished, the code produced by the two programmers is then run and the results are compared. This allows spotting mistakes, unless both programmers make the same mistake.

Warnings and Errors

Exceptions are conditions which do not allow continuing the normal flow of execution. R has two major types of exceptions:

Warnings

Warnings are less serious exceptions. R has encountered a problem processing some part of your commands, but can continue processing the other commands. However, the result might not be what you would expect, so you should carefully examine all variables used in the command that triggered the warning. Before returning to the command prompt R will display a short warning message describing the problem that has occurred.

Examples of code that produces warnings are:

- taking the logarithm of negative numbers,

```
x <- log(-1:1)
```

```
## Warning in log(-1:1): NaNs produced
```

```
x
```

```
## [1]  NaN -Inf    0
```

- trying to use recycling rules when the lengths of the vectors involved are not multiples of each other.

```
x <- 1:3
y <- 4:5
z <- x+y
```

```
## Warning in x + y: longer object length is not a multiple of
## shorter object length
```

```
z
```

```
## [1] 5 7 7
```

Errors

Errors are more serious exceptions. They are that serious that R cannot continue processing your commands and returns to the command prompt showing a brief description of the error.

Examples of code causing errors are:

- syntax errors like mismatched parentheses

```
sin(x[i)]
```

```
## Error: <text>:1:8: unexpected ')'
## 1: sin(x[i)
##            ^
```

- trying to use a variable that does not exist,

```
print(I.do.not.exist)
```

```
## Error in eval(expr, envir, enclos): object 'I.do.not.exist' not found
```

- calling a function with the wrong arguments.

```
rnorm(sample.size=10)
```

```
## Error in rnorm(sample.size = 10): unused argument (sample.size = 10)
```

A warning or an error message does not necessarily imply that there is something wrong with this specific line of code. It might well be that you made a mistake earlier on, but somehow R manages to carry on for a while before producing an error or a warning message.

In the debug section we will look at techniques for identifying the root cause of errors.

Raising warnings and errors in your code

When writing your own R functions, you can (and should) make use of functions like `warning` and `stop` to handle exceptions (do *not* use `print` or `cat` for these purposes). `warning` will display a warning message, but not abort the function, whereas `stop` will display an error message and return immediately to the command prompt.

Example 8.

Consider the following function `moments`, which calculates the expected value and standard deviation for a discrete distribution with range space `x` and associated probabilities `p`.

```
moments <- function(x, p) {
    e.x <- sum(x*p)
    var.x <- sum((x-e.x)^2*p)
    sd.x <- sqrt(var.x)
    list(e.x=e.x, sd.x=sd.x)
}
```

The calculations above are only sensible if:

- `x` and `p` are vectors of the same length.
- The entries in `p` are non-negative.
- The entries in `p` sum to one. However in this case we can simply rescale `p` so that it sums to 1.

This suggests that we should raise an error in the first two scenarios and raise a warning if we need to rescale `p`.

```
moments <- function(x, p) {
    if (length(x)!=length(p))
        stop("p and x must be of the same length.")
    if (any(p<0))
        stop("p is a vector probabilities and cannot have negative entries")
    if (abs(sum(p)-1)>1e-8) {                  # Allow for some numerical error
        p <- p/sum(p)
        warning("p has been rescaled so that probabilities sum to 1")
    }
    e.x <- sum(x*p)
    var.x <- sum((x-e.x)^2*p)
    sd.x <- sqrt(var.x)
    list(e.x=e.x, sd.x=sd.x)
}
```

Suppressing warnings and handling errors

Sometimes you want to call an R function that will create a warning which you know is safe to ignore. In these cases it can be useful to tell R to suppress the warning messages. You can do so by using the function `suppressWarnings`.

```
suppressWarnings(x <- log(-1))
```

will not show any warning messages.

The function `try` allows the user to "catch" errors, i.e. it allows us to deal with the error ourselves instead of having R deal with it. If an error occurs within a `try` block, R does not abort, but `try` simply returns an object of the class `try-error`.

```
success <- try({read.table("file.does.not.exist.txt")})
success
```

but the execution is not aborted. It is now up to us to decide how we want to proceed. The function `try` has a sibling `tryCatch`, which allows for more fine-grain control.

Bugs

Bugs in computer code

For more than a century, engineers have been referring to small glitches in their devices as "bugs". In 1947 Grace Hopper and her colleagues at the University of Harvard found a moth stuck in a faulty relay in the computer. They taped the insect in their logbook and wrote "first actual case of bug being found" next to it. Grace Hopper is said to have coined the term of the system having been "debugged". The logbook with the moth is now an exhibit in the Smithsonian National Museum of American History in Washington DC and also available online.

A bug is a fault in a computer program that will cause the program either to produce an error or a warning message, and/or to produce an incorrect or unexpected result. Whilst it is relatively easy to find bugs that lead to an error or a warning message, it is much harder to locate bugs that "just" lead to an incorrect result: often such bugs remain unnoticed for some time.

Bugs can be due to many factors: a faulty design ("semantic error"), an error in the code, numerical problems, or, however not very often, a bug in R (or the operating system) itself.

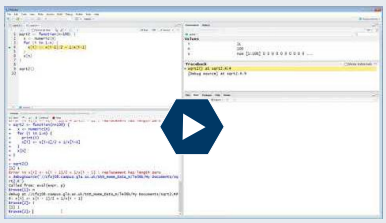
It is almost impossible (at least for me) to write code that is free from errors. Having a well thought-out design and having structured and commented your code will not only help avoiding mistakes, but also help simplify debugging. Nonetheless, every program will (at least initially) contain bugs. Thus it is important to know how to find bugs and how to fix them.

It is an unfortunate fact of life that it often takes as least as much time (or even more) to debug a program as it takes to write it in the first place. When programming I typically spend around one third of my time on designing the program, one third on actual coding, and another third on testing and debugging.

Identifying and fixing bugs usually involves going through the following steps:

1. Realise that your program does not (always) work as intended.
2. Find out how you can trigger the bug. Without being able to reproduce the bug, you have little chance of being able to locate the bug. Thinking about the conditions under which you can trigger the bug might help you get an idea of where the bug might be. This step can be very difficult, especially if your code uses random numbers (simulation studies, bootstrap, MCMC, etc.)
3. Once you know how to trigger the bug, you have to identify where it is and what is causing the bug. R has special tools for this (more on this in the video).
4. Once you have identified the actual mistake, you "just' ' have to correct it.
5. Look out for similar bugs. If for example your bug was due to a design fault, it is likely that other parts of the program also turn out to be faulty.

Debugging in RStudio



Debugging in RStudio

https://youtu.be/hS482Goj1t8

Duration: 9m38s

Task 6.

The function below is meant to compute the variance $\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2$ of a sample $(x_1,\ldots,x_n)$.

```
compute.variance <- function(x) {
  x.bar <- mean(x)
  variance <- sum(x-x.bar)^2 / n-1
  variance
}
```

Can you fix it?

Background reading: Advanced R

http://adv-r.had.co.nz/Exceptions-Debugging.html

The chapter on debugging, condition handling and defensive programming from *Advanced R* is worth a read if you want to delve little deeper into the nitty-gritty details.

Answers to tasks

Answer to Task 1. The call

`func(1,2,3)`

is equivalent to

`func(a=1, b=2, c=3)`

and calculates $(1 + 2 \times 2)^3 = 5^3 = 125$.

The call

`func(4)`

is equivalent to

`func(a=4, b=0, c=1)`

(default values are used for `b` and `c`) and calculates $(4 + 0 \times 2)^1 = 4^1 = 4$.

The call

`func(c=3, 2, 1)`

is equivalent to

`func(a=2, b=1, c=3)`

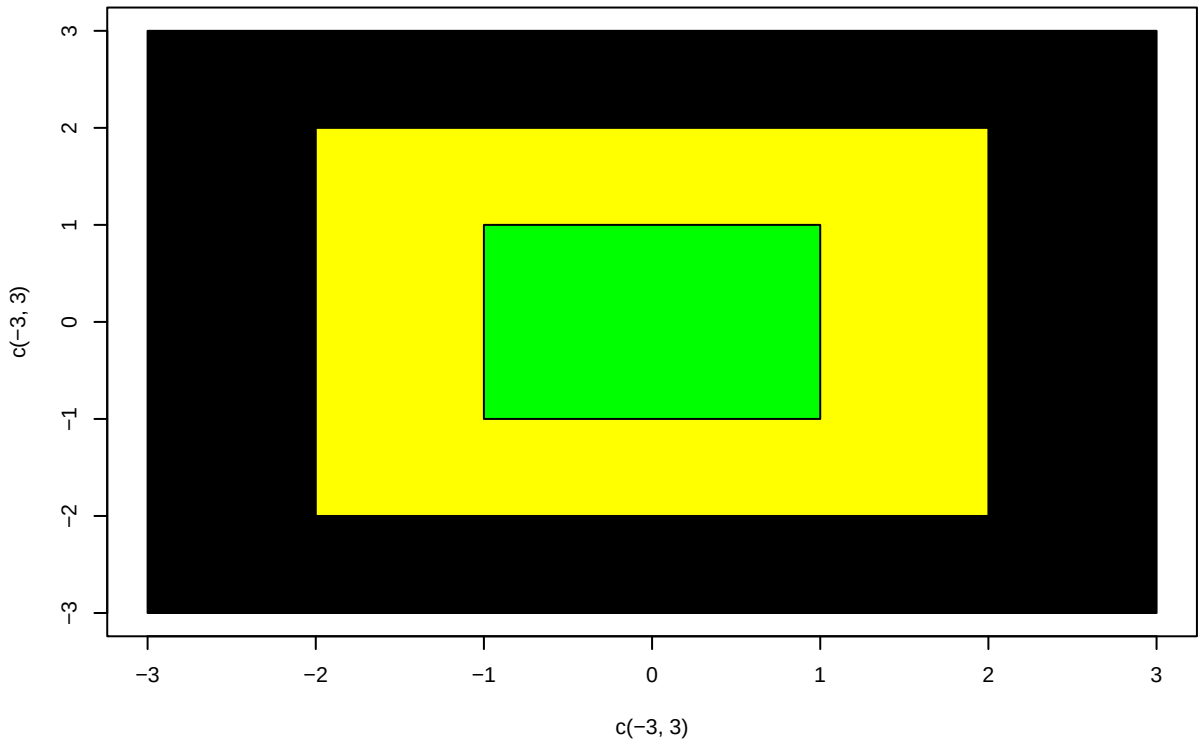
and calculates $(2 + 1 \times 2)^3 = 4^3 = 64$.

Answer to Task 2. We can define the function square as follows.

```
square <- function(x, y, width, ...) {
  rect(x-width/2, y-width/2, x+width/2, y+width/2, ...)
}
```

We can then draw the three rectangles using

```
plot(c(-3,3), c(-3,3), type="n")
square(0, 0, 6, col="black")
square(0, 0, 4, col="yellow")
square(0, 0, 2, col="green")
```



Answer to Task 3. We start by defining functions which calculate the harmonic mean and the geometric mean.

```
#' Compute the harmonic mean
#' @param x numeric vector containing the data
#' @return the harmonic mean of x
harmonic.mean <- function(x) {
  n <-length(x)
  n / sum(1/x)
}

#' Compute the geometric mean
#' @param x numeric vector containing the data
#' @return the geometric mean of x
geometric.mean <- function(x) {
  n <- length(x)
  prod(x)^(1/n)
}
```

We then combine these two functions and `mean` into the function `all.means`:

```
#' Compute the arithmetic, geometric and harmonic mean
#' @param x numeric vector containing the data
#' @return a list containing the arithmetic, geometric and harmonic mean of x
all.means <- function(x) {
  n <- length(x)
  list(arithmetic = mean(x),
       harmonic = harmonic.mean(x),
       geometric = geometric.mean(x))
}
```

Answer to Task 4. The case $a = 0$ corresponds to a linear equation (as the coefficient in front of the quadratic term is zero), so the solution is given by $-\frac{c}{b}$.

```
#' Solve quadratic equation ax^2 + bx + c = 0
#' @param coefficient of the quadratic term (scalar)
#' @param coefficient of the linear term (scalar)
#' @param coefficient of the intercept term (scalar)
#' @return the real solution(s) as a vector of length 0, 1 or 2.
quadratic <- function(a, b, c) {
  if (abs(a)<1e-10) {             # Linear case (for really small a the quadratic
                                  # formula is unstable, so we use the linear one)
     return(-c/b)
  }
  delta <- b^2 - 4*a*c            # Discriminant
  if (abs(delta)<1e-10) {         # Exactly one solution (allowing for rounding errors)
    return(-b/(2*a))
  }
  if (delta>0) {                  # Two real solutions
    sol1 <- -(b+sqrt(delta)) / (2*a)
    sol2 <- -(b-sqrt(delta)) / (2*a)
    return(c(sol1, sol2))
  }
  return(c())                     # Otherwise no solution
}
```

Answer to Task 5 (The Wheels on the Bus (harder - uses 'stringr')). We will write two functions: the first one generates a single verse and the second one uses the first function to produce all the lyrics.

The prototype for a verse is

```
The <object> on the bus <go or goes> <what>,
<what>, <what>.
The <object> on the bus <go or goes> <what>,
```

`all day long.`

So the function generating a single verse (`wheels.verse`) needs three inputs, `object`, whether the object is `plural` or not (which determines whether we use the verb "go" or "goes") and `what` the object does.

The second function (`wheels`) then uses a loop to use `wheels.verse` to generate the lyrics one verse after the other and then puts the verses together and returns them. The second function takes a vector of `objects`, `plurals` and `whats` as arguments (which defaults set to generate the above lyrics).

```
library(stringr)

#' Generate a single verse from the Wheels on the Bus
#' @param object object of the verse (defaults to "wheels")
#' @param plural whether the object is a plural (defaults to TRUE)
#' @param what what the object does without the word "go"
#'             (defaults to "round and round")
#' @return one verse about the \code{object} going code{what}
wheels.verse <- function(object="wheels", plural=TRUE, what="round and round") {
    go.goes <- ifelse(plural, "go", "goes")
    str_c("The ",object ," on the bus ", go.goes, " ", what, ",\n", what, ", ", what,".\n",
          "the ",object, " on the bus ", go.goes, " ", what, ",\nall day long.\n")
}

#' Generates all verses of the Wheels on the Bus
#' @param objects objects of the verse (defaults to the standard lyrics)
#' @param plurals whether the objects are a plural word
#'                (defaults to the standard lyrics))
#' @param what what the object does without the word "go"
#'             (defaults to the standard lyrics)
#' @return all verses of the lyrics of the song
wheels <- function(objects=c("wheels", "wipers", "horn", "people"),
                   plural=c(TRUE, TRUE, FALSE, TRUE),
                   what=c("round and round", "swish swish swish",
                          "beep beep beep", "up and down")) {
    verses <- character(length(objects))
    for (i in seq_along(verses))
        verses[i] <- wheels.verse(objects[i], plural[i], what[i])
    # The function wheels.verse accepts vector input so we could have
    # just used
    # verses <- wheels.verse(objects, what)
    str_c(verses, collapse="\n")
}

cat(wheels())
```

Answer to Task 6. First of all we need to define `n` as the length of `x`. There are also two mistakes in the line `variance <- sum(x-x.bar)^2 / n-1`. As it stands it computes

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

instead of

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

Thus this line of code should be `variance <- sum((x-x.bar)^2) / (n-1)`, i.e. the function should be

```
compute.variance <- function(x) {
  x.bar <- mean(x)
  n <- length(x)
  variance <- sum((x-x.bar)^2) / (n-1)
  variance
}
```