# Chapter 8 - Further Programming Tools

# 1 Aims of the Chapter

By the end of this chapter you should know

- what scoping is and what its effects are
- how to write pseudocode
- how to avoid common mistakes
- how to find bottlenecks in your code (parts of your code that take a long time to run)

# 2 Scoping rules

When we refer to the workspace we mean the Global Environment. In all the following code chunks imagine we start with an empty workspace (use the *brush* symbol in the *Environment* window if you want to actually empty it). If we run the following code

```
add_x_and_y <- function(x, y) x + y
add_x_and_y(x = 3, y = y)
```

Error in add_x_and_y(x = 3, y = y) : object 'y' not found

we'll get an error message since we don't supply a numeric value for `y` in the function argument. But in the following code

```
y <- 1
add_x_and_y <- function(x, y) x + y
add_x_and_y(x = 3, y = y)
```

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

R finds `y` even though we don't give it a value in the function argument. But in the code below

```
y <- 1
add_x_and_y <- function(x, y) x + y
add_x_and_y(x = 3, y = y)
x
```

the function runs OK but it gives the error message

[1] 4
Error: object 'x' not found

and R cannot find `x` which was specified inside the function. This behaviour relates to the **scoping** rules in R. When R looks for an object it starts by looking in the active environment. If it can't find it then it looks in the parent of the active environment and continues looking up the tree of environments. If it doesn't find the object anywhere it gives an error message to say it can't find it.

So when we evaluate a **function** what is the active environment? When evaluating a function, R actually creates a separate offspring environment of the environment the function is called from. So when we evaluate the function `add_x_and_y` from the Global Environment R creates an environment that is an offspring of Global Environment to evaluate the function in. Because R looks up the tree of environments any object not found in the function environment will be searched for next in the Global Environment. Let's look at some more code to see what is going on

```
y <- 1
add_x_and_y <- function(x, y) x + y
add_x_and_y(x = 3, y = y)
y
x
```

- `y` is declared to be 1 in the workspace
- the function is evaluated in an offspring of the workspace
- the `x` and `y` variables are local to this function environment
- once the `add_x_and_y` function has finished evaluating, the active environment returns to being the the workspace.
- R can find y because there is a variable called `y` in the workspace (with the value 1)
- since `x` is an object only in the offspring function environment R will not find it (since it looks **up** the hierarchy of environments - not down).

The fact that R creates separate environments for every function ensures that there is no risk of a clash between variables within a function and variables in the workspace 9we had two versions of the variable `y` in the previous code chunk that were stored in different environments.

This is why you are strongly advised to make sure that all the variables you need in a function are made arguments of the function. Otherwise the function may be have in an unpredictable way (by reading values from the workspace). The code below is bad since it requires `y` to be found in the workspace which could lead to unpredictable results and won't run unless `y` is in the workspace.

```
add_x_and_y <- function(x) x + y
```

If `y` is needed in the function it should be a function argument.

There is a nice but very detailed description of the scoping rules in R in Chapter 6 of (Grolemund 2015), available here (https://d1b10bmlvqabco.cloudfront.net/attach/ighbo26t3ua52t/igp9099yy4v10/igz7vp4w5su9/OReilly_HandsOn_Programming_with_R_2014.pdf)

Environments are quite difficult to fully understand in R but mostly all you need to know is how they affect where R looks for variables when functions are evaluated.

# 3 Exercises A

1. By thinking about the scoping rules, predict what the following code chunks will produce (ie will they run or produce an error message) before running the code to check it. Assume that the workspace is empty before each chunk of code is run

```
remove_low_values <- function(x, threshold) x[-which(x < threshold)]
remove_low_values(x = rnorm(n = 100))
```

```
remove_low_values <- function(x, threshold) x[-which(x < threshold)]
some_threshold <- 1
remove_low_values(x = rnorm(n = 100), threshold = some_threshold)
```

```
remove_low_values <- function(x, threshold) x[-which(x < threshold)]
some_threshold <- 1
remove_low_values(x = rnorm(n = 100), threshold = some_threshold)
x
```

```
remove_low_values <- function(x) x[-which(x < threshold)] # note the change in arguments
remove_low_values(x = rnorm(n = 100), threshold = 1)
```

# 4 Pseudocode

Pseudocode is a description of what your code does using general programming conventions (like `if`, `else`, `return` etc). It should be written so that it needs no further explanation. Communicating what the code does is the only aim. It also helps to write pseudocode before you write actual code as you can see how everything fits together, what functions should return etc. We illustrate pseudocode for the process of simulating games of Snakes and Ladders described in the next section. Pseudocode is not unique - there are lots of ways of writing it.

# 5 Snakes and Ladders game

Snakes and Ladders is a simple game. A typical board is shown here (https://www.amazon.co.uk/Traditional-Wooden-Games-Snakes-Ladders/dp/B008646LAU) You start at square 1. You roll a die and move that many squares. If you land on a ladder foot you go to the top of the ladder. If you land on a snake head you go to the tail of the snake. You continue until you reach or pass square 100. It is traditionally played with two or more players taking it in turns to roll the die.

Suppose you are working for an online gambling company. They have developed an online game of Snakes and Ladders. Players pay say £3 per game and win £5 if they complete the game within $x$ throws of the die.

The question your boss has asked is “what is the value of $x$ such that the expected profit for the player is zero?” (this represents the break-even scenario; over a large number of games the player is expected to neither make nor lose money).

# 5.1 Pseudocode for snakes and ladders

We define a couple of matrices, create functions to make a move, climb a ladder, descend a snake and count the number of moves needed to finish the game. The idea with pseudocode is not to include any actual code. It should be readable by anyone who is used to programming but doesn’t know the particular syntax of R. If after reading the pseudocode below you don’t understand what the code does then I have failed (in which case think of a better way of doing it)

```
ladder_mat <- 8 by 2 matrix
for each row j of ladder_mat do
  first value <- jth ladders foot
  second value <- jth ladders top
end for

snake_mat is an by 8 by 2 matrix
for each row j of snake_mat do
  first value  <- jth snakes head
  second value  <- jth snakes tail
end for

function make_move
Input: starting square
Output: finishing square

function climb_ladder
Input: ladder foot
check there is a ladder with that foot
Output: ladder top

function descend_snake
Input: snake head
check there is a ladder with that head
Output: snake tail

function count_num_moves
  counter=0
  square=1
  finished_game=FALSE
  while finished_game=FALSE do
    counter -> counter + 1
    call make_move function
    if land on a ladder foot do
      call climb_ladder
      endif
      if land on snake head do
      call descend_snake
      endif
      update square
    if square exceeds 100 do
      finished_game=TRUE
    endif
  end while
```

# 5.2 Writing the code for snakes and ladders

We will use the board shown here (https://www.amazon.co.uk/Traditional-Wooden-Games-Snakes-Ladders/dp/B008646LAU)

```
################################################################################
#################### Kevin Walters Sheffield 22 November 2018 ##################
################################################################################

# Create matrix of ladder feet and tails
ladder_mat <- matrix(NA , nrow = 8, ncol = 2)
ladder_mat[1,] <- c(4, 14)
ladder_mat[2,] <- c(9, 31)
ladder_mat[3,] <- c(20, 38)
ladder_mat[4,] <- c(28, 84)
ladder_mat[5,] <- c(40, 59)
ladder_mat[6,] <- c(51, 67)
ladder_mat[7,] <- c(63, 81)
ladder_mat[8,] <- c(71, 91)

# Create matrix of snake heads and tails
snake_mat <- matrix(NA , nrow = 8, ncol = 2)
snake_mat[1,] <- c(17, 7)
snake_mat[2,] <- c(54, 34)
snake_mat[3,] <- c(62, 19)
snake_mat[4,] <- c(64, 60)
snake_mat[5,] <- c(87, 24)
snake_mat[6,] <- c(93, 73)
snake_mat[7,] <- c(95, 75)
snake_mat[8,] <- c(99, 78)

# Function to move the counter
make_move <- function(start_sq){
  finish_sq <- start_sq + sample(x = 1:6, size = 1)
}

# Function that moves the player up a ladder if they land on the ladder foot
climb_ladder <- function(ladder_foot, ladder_mat){
  row <- which(ladder_mat[, 1] == ladder_foot)
  ladder_mat[row, 2]
}

# Function that moves a player down a snake if they land on a snake head
descend_snake <- function(snake_head, snake_mat){
  row <- which(snake_mat[, 1] == snake_head)
  snake_mat[row, 2]
}

# Function to play the game until the player reaches 100 or more.
# The function returns the number of moves taken.
# If `to_print = T` then the function prints the squares visited in each move
count_num_moves <- function(snake_mat, ladder_mat, to_print){
  num_moves <- 0
  start_sq <- 1
  finished_game = FALSE
  while(finished_game == FALSE){
    if(to_print == T) cat("start at square = ", start_sq, "\t")
    num_moves <- num_moves + 1
    next_sq <- make_move(start = start_sq)
    if(to_print == T) cat("move to square= ", next_sq, "\n")
    if(any(ladder_mat[, 1] == next_sq)) {
      next_sq <- climb_ladder(next_sq, ladder_mat)
      if(to_print == T) cat("climb ladder to",next_sq, "\n")
    }
    if(any(snake_mat[, 1] == next_sq)) {
      next_sq <- descend_snake(next_sq, snake_mat)
      if(to_print == T) cat("descend snake to", next_sq, "\n")
    }
    if(next_sq >= 100) finished_game = TRUE else{
      start_sq <- next_sq
    }
  }
  return(num_moves)
}
```

# 5.3 Using `cat` to check code inside functions

As we have seen, the variables in a function are local to the function (R can't find them from the Global Environment). Because of this it is difficult to check code inside a function without first copying it to the Global Environment.

An alternative that sometimes works is just to print out parts of the code inside a function to check it is working as expected. I prefer to use `cat` for this but `print` is better if you want to retain the object structure when printing. See the difference between `cat` and `print` when showing the contents of a matrix

```
x <- matrix(1:4, nrow = 2)
print(x)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
cat(x)
```

```
## 1 2 3 4
```

We can now run the function to check it is doing what we expect

```
count_num_moves(snake_mat, ladder_mat, to_print = T)
```

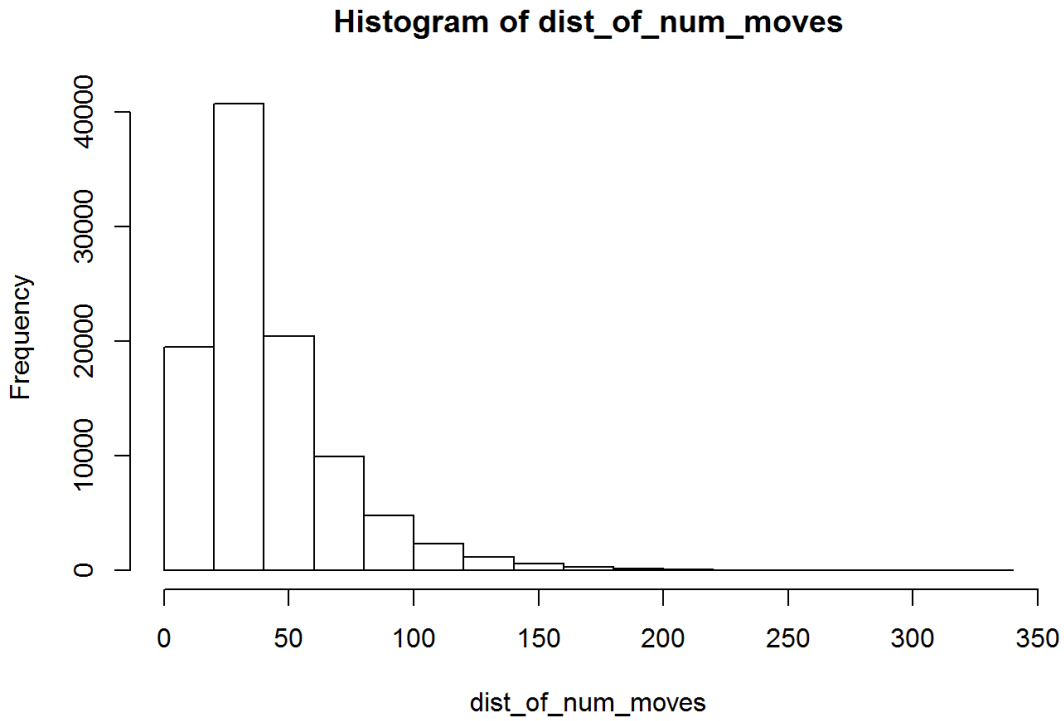
```
## start at square =  1     move to square=  5
## start at square =  5     move to square=  11
## start at square =  11    move to square=  17
## descend snake to 7
## start at square =  7     move to square=  12
## start at square =  12    move to square=  18
## start at square =  18    move to square=  24
## start at square =  24    move to square=  25
## start at square =  25    move to square=  29
## start at square =  29    move to square=  32
## start at square =  32    move to square=  34
## start at square =  34    move to square=  36
## start at square =  36    move to square=  42
## start at square =  42    move to square=  48
## start at square =  48    move to square=  51
## climb ladder to 67
## start at square =  67    move to square=  68
## start at square =  68    move to square=  72
## start at square =  72    move to square=  76
## start at square =  76    move to square=  80
## start at square =  80    move to square=  81
## start at square =  81    move to square=  85
## start at square =  85    move to square=  88
## start at square =  88    move to square=  94
## start at square =  94    move to square=  95
## descend snake to 75
## start at square =  75    move to square=  80
## start at square =  80    move to square=  84
## start at square =  84    move to square=  89
## start at square =  89    move to square=  91
## start at square =  91    move to square=  96
## start at square =  96    move to square=  101
```

```
## [1] 29
```

# 5.4 Answering the online gambling company's question

We can now run the function thousands of times to see the distribution of the number of moves needed to complete the game.

```
dist_of_num_moves <- replicate(100000, count_num_moves(snake_mat, ladder_mat, to_print = F))
hist(dist_of_num_moves)
```




How do we use this information to answer the question? The player pays £3 to play. They win £5 is they win and nothing is they lose. So their profit is

- -£3 if the player loses
- £2 if the player wins

Let $w$ be the probability that the player wins. Then the expected profit for the player (in £) is $-3(1-w)+2w = 5w-3$ For this expected profit to be zero requires $w = 0.6$ so we require the probability that the player wins to be 0.6

Since the player wins if they complete the game in fewer than $x$ moves, we require the 60th percentile of our distribution of the number of moves needed to complete the game.

```
quantile(dist_of_num_moves, 0.60)
```

```
## 60%
##  40
```

If we run this a few times we consistently get an answer around 40 moves. If the company specifies that the player must complete the game in 40 or fewer throws of the dice, then the probability that the player wins a game is 0.6 and their expected profit is zero.

# 6 General programming tips

Finding errors in code can be painful so try to avoid making them. Here are some general tips that will help

- if you are going to write a long program then create the pseudocode first
- if you have code that you need to run several times then put it in a function and call the function when needed
- when creating functions first get the code to work in the workspace before putting it inside a function
- always look for shortcuts (for example is there an existing R function that does what you want to do - internet searches are useful for finding this out)
- annotate your code if you won't remember or easily be able to understand what it does later (perhaps several years later)

# 7 Common errors

- using `=` instead of `==` when testing equality
- forgetting to ensure all the objects you need to evaluate a function, that aren't created inside the function, are function arguments
- forgetting to use `drop= FALSE` when subsetting matrices (to preserve dimensions)
- using `*` not `%*%` for matrix multiplication
- forgetting to create a vector before assigning values to its elements in a `for` loop
- getting `&&` and `&` mixed up
- not using `()` when needed; eg the difference between `1:5+1` and `1:(5+1)`

```
1:5 + 1
```

```
## [1] 2 3 4 5 6
```

```
1:(5 + 1)
```

```
## [1] 1 2 3 4 5 6
```

# 8 Finding potential bottlenecks

Suppose your code is running but it is slow. How can you tell which parts of your code are slowing it down? First you need to decide if your code is really too slow. Hadley Wickham (Wickham 2015) points out that spending three hours to make your code 15 minutes faster is not a good use of your time unless you are going to run the code many times. Let's see how we can profile the code-execution time by considering the following function which calculates $\boldsymbol{x}^T\Sigma^{-1}\boldsymbol{x}$ for some column vector $\boldsymbol{x}$ and matrix $\Sigma$

```
size = 100
large_mat <- matrix(rnorm(size ^ 2), ncol = size)
large_vec <- matrix(rnorm(size), ncol = 1)
t(large_vec)%*% solve(large_mat) %*% large_vec
```

```
##         [,1]
## [1,] 3.20759
```

```
Rprof(tmp <- tempfile())
size = 1000
large_mat <- matrix(rnorm(size ^ 2), ncol = size)
large_vec <- matrix(rnorm(size), ncol = 1)
t(large_vec)%*% solve(large_mat) %*% large_vec
```

```
##          [,1]
## [1,] 15.71188
```

```
Rprof()
summaryRprof(tmp)$by.self
```

```
##                 self.time self.pct total.time total.pct
## "solve.default"      0.50    86.21       0.50     86.21
## "rnorm"              0.06    10.34       0.06     10.34
## "matrix"             0.02     3.45       0.08     13.79
```

The `self.pct` column tells us what percentage of the total time taken was spent in each function. Finding the 1000 by 1000 matrix inverse with `solve` takes around 90% of the time and the random normal realisations take around 10% of the time. In this case there isn't much you can do to speed this up (that I know about) but sometimes you can identify the bottlenecks and improve your code run-time.

# 9 Exercises B

1. The rules of a simplified game of blackjack are as follows:
    - There is a dealer (assumed female) and one other player (assumed male)
    - The dealer deals two cards to both herself and the player (neither player can see the value of the other person's cards)
    - For the player to win he must score **higher** than the dealer but no more than 21 (otherwise he goes bust and loses)
    - All picture cards (jack, queen and king) are worth 10
    - Aces are worth either 1 or 11, the person whose hand contains an ace can choose its value
    - scoring 21 in two cards (ace plus a card worth 10) is called blackjack and beats all other hands
    - a hand of 5 cards is called a five-card trick and beats everything except black jack
    - the player looks at his two cards and can choose to keep his hand as it is or ask the dealer to give him another card
    - if he scores 22 or more he is bust. Whilst he has less than 21 he can choose to add another card.
    - once the player decides he wants no more cards the dealer then has her go - with the same rules applied as for the player

    - if the dealer draws with the player the dealer wins

Think about how to code playing this in an efficient way and write the pseudocode for it. You will need to chhose a suitable rule to decide whether the player (and dealer) choose another card or not. You could choose a simple rule, eg they don't take another card if they have 16 or more, or something that assignns probabilities of picking another card that depend on their current score. The dealer and player could have different rules as well.

2. Write some R code to play the game based on the above.

# References


Grolemund, G. 2015. *Hands-on Programming with R*. O'Reilly Media Inc.

Wickham, H. 2015. *Advanced R*. CRC Press.