R Commands & flow control

3

Basic R 'Built-in' functions for working with variables

- list & remove objects

```
ls(), rm()⌉
rm(list=ls()) # get rid of everything
```

- Add rows or columns to a data frame, ***df***. Row bind, column bind

```
rbind(df,...), cbind(df,...)⌉
```

- Remove a row, or column, from a data frame.

```
df[-1,] # remove first row
df[,-1] # remove first column
```

- Sort a data frame. There are three functions, ***rank***, ***sort*** and ***order***. Order is the hardest one to understand, but is best suited to reordering data frames.

```
 phoneBook[order(phoneBook$secondName) , ]
```

- Missing values, uses ***is*** family of functions
- (NA is different from "want to record missing values")

```
 is.na(…)⌉
```

- Names of objects

```
names(…)⌉
colnames(...)
rownames(...)
```

- Return length of an object, number of rows or columns of a dataframe or matrix

```
length(…)⌉
nrow(…)⌉
ncol(…)⌉
```

A note on data sort ...

```
A<-1:7
B<-c("Mon","Tue","Wed","Thur","Fri","Sat","Sun")⌉
names(A)<-B

sort(A) ; sort(names(A))⌉
order(A); order(names(A))⌉
```

sort() returns the sorted data. order() returns a permutation vector showing how to reorder the data
Use sort() to sort a single vector, order() to sort multiple dependent vectors (e.g. columns in a dataframe)

Basic R 'Built-in' functions for working with matrices

- Adding rows and columns to matrix or data frame
  - Make a sample matrix from vector

```
x <- matrix(x, ncol=4) # converts vector X to matrix, with 4 columns and 5 rows
y <- matrix(y, ncol=4) # converts vector Y to matrix, with 4 columns and 5 rows
```

- Addition of rows with rbind

```
xx <- rbind(x,y)↰
```

- Addition of columns with cbind

```
yy <- cbind(x,y)↰
```

- Examine the output

```
xx
yy
```

- Arithmetic with matrices
  - Means & medians of values by row

```
rowMeans(xx) ; rowMedians(xx)↰
```

  - Means & medians of values by column

```
colMeans(yy) ; colMedians(yy)↰
```

  - Name rows and columns

```
rownames(xx) <- c(...)↰
colnames(yy) <- c(...)↰
```

Basic R 'Built-in' functions for working with objects

- Arithmetic with vectors
  - Min / Max value number in a series

```
min(x) ; max(x)⌉
```

  - Sum of values in a series

```
sum(x)⌉
```

  - Average estimates (mean / median)

```
mean(x) ; median(x)⌉
```

  - Range of values in a series

```
range(x)⌉
```

  - Correlation, variance & covariance, of series (e.g. heights & yields) of vectors

```
var(x)   # variance of x
cor(x,y) # correlation of x and y
cov(x,y) # covariance of x and y
```

- Arithmetic with vectors
  - Rank ordering

```
rank(x) # returns positions (placement) of
   elements
```

  - Quantiles

```
quantile(x); boxplot(x)⌉
```

  - Tukey's 5 number summary

```
fivenum(x)⌉
```

  - Square Roots

```
sqrt(x)⌉
```

  - Standard deviations

```
sd(x)⌉
```

  - Median average distance

```
mad(x)⌉
```

  - Trigonometry functions

```
tan(x) ; cos(x) ; sin(x)⌉
```

```
x <- sample(10000,20)/rnorm(20,5)⌉
```

```
y <- sample(10000,20)/rnorm(20,7)⌉
```

Useful vector functions

Commands & flow control

- repeat: generates a vector of repeating data units

```
rep(data, number of repeats)↵

> rep(1:5, 5)↵
[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Useful to create empty vectors, e.g.

```
> rep(0, 10)↵
[1] 0 0 0 0 0 0 0 0 0 0
```

- sequence: generates a vector of a data sequence

```
seq(from, to, by)↵

> seq(1,10,2)↵
[1] 1 3 5 7 9
> seq(2,10,2)↵
[1]  2  4  6  8 10
```

Looping - informal introduction

- Consider a problem where we need to execute a certain command many times
- e.g. we want to print numbers 1, 2, .. 10 one-by-one

one solution:

```
print(1)
print(2)
...
print(9)
print(10)
```

a better solution:

```
for(i in 1:10){
  print(i)
}
```

Both give exactly the same output!
The **for** loop is just more compact.
Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

R language elements
Commands & flow control

- Looping
  - Iterate over a set of values (**for** loop)
  - or while a condition is met (**while** loop)

- Remember that all operations in R are vectorized. No need to use loops to e.g. make a sum of two vectors.

- Loops are multi-line commands. R will execute them only after the whole loop has been typed in. Use Rstudio editor to type it all in, don't do it in R console!

LOOPS
Commands & flow control

- For loops - iterate over a set of values, in this case 1..10

```
for (f in 1:10) {
   print(f)
}
```

- While loops - iterate while a condition is met

```
f <- 1
while ( f <= 10 ) {
   print(f)
   f <- f + 1
}
```

when this condition is false the loop stops

Loops with breaks

Commands & flow control

Any loop can be prematurely broken with **break**

```
for (f in 1:10) {
      if (f==6) break
      print( f )
}
```

stops when f == 6

Output:

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

What would happen if we switched lines 2 and 3 (i.e. break after print)?

Loop exercise
Commands & flow control

1. Write a for loop that prints the letters of the alphabet [a to z]

2. Write a while loop that does the same, in reverse [z to a]

3. Afterwards, add a break statement to stop the loops when letter 'n' is reached

- Hints :
  - think about how would you solve this problem without using loops
  - base your solution on the for/while examples from previous slides
  - try to modify them so they print letters of the alphabet (instead of numbers), use the built-in vector **letters** to get individual letters

Solution to loop problem

Commands & flow control

For loop

```
  for (f in 1:26){
→    print(letters[f])↵
  }
```

While loop

```
  f <- 26
→ while(f!=0){
     print(letters[f])↵
     f <- f-1
  }
```

Add this (→) to make it break

```
if (letters[f]=="n") break
```

Example code:
06_loopExercise.R

Code formatting avoids bugs!

- Code formatting is crucial for readability of loops

```
 f<-26
 while(f!=0){

print(letters[f])↰
 f<-f-1 }
```

BAD!!!

```
f <- 26
while( f != 0 ){
   print(letters[f])↰
   f <- f-1
}
```

GOOD!

- The code between brackets {} **always** is indented, this clearly separates what is executed once, and what is run multiple times
- Trailing bracket } always alone on the line at the same indentation level as the initial bracket {
- Use white spaces to divide the horizontal space between units of your code, e.g. around assignments, comparisons

Conditional branching
Commands & flow control

- Some commands might need to be executed only if a condition is met.

- `if` allows different outcomes to be selected based up on a calculation result within brackets.

```
if (condition) {
… do this …
} else {
… do something else …
}
```

  - condition is any logical value, and can contain multiple conditions
    - e.g. `(a==2 & b <5),` this is a compound conditional argument

Exercise: functions, loops & branches

- Write a function that computes the first *i* elements of the Fibonacci series
  - The Fibonacci series is simply a vector in which the proceeding number is an addition of the two prior items
    - 1, 1, 2, 3, 5, 8, 13, 21 …
    - think vector arithmetic x[***i***] = x[***i***-2] + x[***i***-1] ; where ***i*** is the index number
- Solve the problem as follows:
  - Work out the steps needed to compute the first 10 items of the series
    - Steps
      - Define a vector to hold answer
      - Use a for loop to iterate 10 times
      - Use an if branch to do something different for index numbers <2 and >2
  - Later we will embody the steps in a function that takes ***i*** as an argument

Fibonacci problem: procedural solution

07_fibonacci.R
(solution1, procedure)

i is a For loop counter that iterates 10 times

if ***i*** is less than 2, then Fibonacci can only be 1, Otherwise the Fibonacci equation is computed . The result is stored in the ***i*** **-**th element of **fib**

fibResult will store the Fibonacci series, as it is computed

```
fib <- rep(0, 10)↰

for (i in 1:10){
       if (i>2){
           fib[i] <- fib[i-2] + fib[i-1]
       } else {
           fib[1] <- 1
           fib[2] <- 1
       }
}
```

There are 2 sets of procedural braces in the if … else statement, and these are nested inside the For loop { } braces.

{ } → if is TRUE procedure; ***i*** counter is greater than 2
{ } → if is FALSE procedure

It is important to ensure there are equal numbers of opening and closing procedural braces!

Q. The if else isn't needed. Show why?
A. See code sheet.

Defining user functions

Commands & flow control

- User functions are objects, they are assigned like any other R object!

```
myFunction <- function(args=...){
    ...code...
    return(...)↰
}
```

- User functions may pass any number of named or unnamed arguments, with or without default values
- User functions may only return a single object
  - They automatically return the last assigned object. Hence, a return statement is not required unless the object you want to return isn't the last object referenced

Fibonacci problem revistited: User function solution

- Modify your Fibonacci code as follows

07_fibonacci.R
(solution1, function)

```
fibonacci <- function(n=10){
        fib <- rep(0, n)

        for (i in 1:n){
            if (i>2){
                fib[i] <- fib[i-2] + fib[i-1]
            } else {
                fib[1] <- 1
                fib[2] <- 1
            }
      }

        return(fib)
}
```

Now type ***fibonacci()***

Add an argument ***fibonacci(25)***

n is the named argument passed to your user defined **fibonacci** function. It specifies the upper index number to which Fibonnaci will be calculated. We define the function with a default value 10, which will be automatically passed If the user doesn't enter any arguments

Fibonacci problem: Streamlined function

- This version of the function computes the value at position ***n***, not up to position ***n***
- It is more efficient than previous versions because results are not incrementally stored

```
fibonacciAtPosition <- function(n=10){
      nextVal <- 1
      prevVal <- 0
      while(n > 1){
          currentVal <- nextVal
          nextVal <- nextVal + prevVal
          prevVal <- currentVal
          n <- n-1
      }
      return(nextVal)↰
}
```

To get a range of values, it's necessary to use the `sapply()` function:

```
sapply(1:10,fibonacciAtPosition)
↰
```

prevVal and **nextVal** are lower and upper elements of the Fibonacci sequence.
n counts down. Each decrement of **n** sees the values of **prev** and **next** swap, via **current**. Prior to the swap, **next** is assigned the value of **next + prev**, which is equivalent to the Fibonacci formula described earlier.

07_fibonacci.R
(solution2, function)