

Unit 5 - Programming

Contents

1. Programming concepts in R	2
2. Vectorized calculations	3
2.1. Element-wise Operations on Vectors.....	3
2.2. Filtering.....	4
3. Control Structures	7
3.1. Statements that perform testing that shifts flow	8
3.1.1. Statement if	8
3.1.2. Switch statements	9
3.2. Statements that perform / control looping.....	11
3.2.1. Statement – for	11
3.2.2. Statement – while	11
3.2.3. Statement – repeat	12
3.2.4. Statement – break.....	12
3.2.5. Statement – next.....	13
3.3. Loops and vectortization	13
3.4. Functions of type 'apply'	14
4. Writing functions.....	16
5. Built-in functions	17
6. Scoping rules	23
7. Directing console output to a file	25
8. Reading from and writing to an external file	25
8.1. Handling files in R.....	25
8.2. Reading from a csv file	27
8.3. Reading from a fixed-width file	28
8.4. Writing to an external file.....	29
9. Debugging	30

1. Programming concepts in R

Basic R Concepts

- R is a computer language which is processed by a special program called an interpreter. This program reads and evaluates R language expressions, and prints the values determined for the expressions.
- The interpreter indicates that it is expecting input by printing its prompt at the start of a line. By default, the R prompt is a greater than sign >.
- On UNIX or LINUX machines you can start R by typing the command "R" to a command interpreter prompt in a terminal window.
- You can run R within an Emacs window by typing M-x R.

Using R as a Calculator

- Users type expressions to the R interpreter.
- R responds by computing and printing the answers.

```
> 1/2
[1] 0.5
> 25^4
[1] 390625
> 2^5
[1] 32
> 70/5
[1] 14
> 14 + 50
[1] 64
```

Grouping and Evaluation

- Normal arithmetic rules apply; multiplication and division occur before addition and subtraction.

```
> 10 + 2 * 5
[1] 20
```

- The default evaluation rules can be overridden by using parentheses.

```
> (10 + 2) * 5
[1] 60
```

Operator Precedence

- R's basic operators have the following precedences (listed in highest-to-lowest order)
 - ^ Exponentiation
 - + unary minus and plus
 - : sequence operator
 - /% %% integer division, remainder

* / multiplication, division
+ - addition, subtraction

- Operators with higher precedence take place before those with lower precedence.

Evaluation Order

- Evaluation of equal precedence takes place left-to-right (except for exponentiation, which takes place right-to-left)

```
> 2^3^2
[1] 512
> (2^3)^2
[1] 64
> 2^(3^2)
[1] 512
```

Benefits

- Clearer, more compact code.
- Potentially much faster execution speed.
- Less debugging (since you write less code).
- Easier transition to parallel programming.

2. Vectorized calculations

2.1. Element-wise Operations on Vectors

- Suppose we have a function $f()$ that we wish to apply to all elements of a vector x . In many cases, we can accomplish this by simply calling $f()$ on x itself.

Vectorized Functions

- Many operations are vectorized, such as $+$ and $>$:

```
> u<-c(5,2,8)
> v<-c(1,3,9)
> u+v
[1] 6 5 17
> u>v
[1] TRUE FALSE FALSE
> w<-function(x)
+ return(x+1)
> w(u)
[1] 6 3 9
> # If a R function uses vectorized operations, then it too is vectorized
> # w() uses +, which is vectorized, so w() is vectorized as well
> #This applies to many of R's built-in functions.
> round(c(1.2,3.1,4.5,9.8)
+ )
[1] 1 3 4 10
> round(1.1) # round() function was applied to a vector having a single element
[1] 1
```

- Here we used the built-in function `round()`, but you can do the same thing with functions that you write yourself.
- Note that the functions can also have extra arguments, e.g.

```
> f<-function(elt,s) return (elt+s)
> y<-c(1,2,4)
> f(y,10)
[1] 11 12 14
> y+4
[1] 5 6 8
> '+'(y,4)
[1] 5 6 8
```

- As seen above, even operators such as `+` are really functions. For example, the reason why element-wise addition of 4 worked is that the `+` is actually considered a function!

The Case of Vector-Valued Functions

```
> z12<-function(z) return(c(z,z^2))
> x<-1:5
> z12(x)
[1] 1 2 3 4 5 1 4 9 16 25
> matrix(z12(x),ncol=2)
      [,1] [,2]
[1,]    1    1
[2,]    2    4
[3,]    3    9
[4,]    4   16
[5,]    5   25
```

- The above operations work with vector-valued functions too:

```
> z12<-function(z) return(c(z,z^2))
> x<-1:5
> z12(x)
[1] 1 2 3 4 5 1 4 9 16 25
> matrix(z12(x),ncol=2)
      [,1] [,2]
[1,]    1    1
[2,]    2    4
[3,]    3    9
[4,]    4   16
[5,]    5   25
```

2.2. Filtering

- Another idea borrowed from functional programming is filtering, which is one of the most common operations in R.

On Vectors

For example:

```
> z <-c(5,2,-3,8)
> w <-z[z*z >20]
> w
[1] 5 8
```

- Here is what happened above: We asked R to find the indices of all the elements of *z* whose squares were greater than 20, then use those indices in an indexing operation on *z*, then finally assign the result to *w*.

```
> z<-c(5,2,-4,8)
> j<-z*z>20
> j
[1] TRUE FALSE FALSE TRUE
```

- We may just want to find the positions within *z* at which the condition occurs. We can do this using `which()`:

```
> which(z*z>20)
[1] 1 4
```

x is an array of numbers, mostly in non-decreasing order, but with some
violations of that order `nviol()` returns the number of indices *i* for
which $x[i+1] < x[i]$

```
> nviol
function(x)
{
  diff <-x[-1]-x[1:(length(x)-1)]
  return(length(which(diff < 0)))
}
> x<-c(4,3,8,2)
> nviol(x)
[1] 2
```

- You can also use this to selectively change elements of a vector, e.g.

```
> x <-c(1,3,8,2)
> x[x > 3] <-0
> x
[1] 1 3 0 2
```

On Matrices and Data Frames

- Filtering can be done with matrices and data frames too. Note that one must be careful with the syntax.

```
> m <-matrix(c(1,2,3,4,5,6),nrow=3)
> m
      [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m[m[,1]>1,]
      [,1] [,2]
[1,]    2    5
[2,]    3    6
> m[m[,1]>1&m[,2]>5,]
[1] 3 6
```

Combining Elementwise Operations and Filtering, with the ifelse() Function

- The form is ifelse(b,u,v) where b is a boolean vector, and u and v are vectors.
- The return value is a vector, element i of which is u[i] if b[i] is true, or v[i] if b[i] is false.

```
> x <-c(5,2,9,12)
> ifelse(x > 6,2*x,3*x)
[1] 15  6 18 24
```

- The advantage of ifelse() over the standard if-then-else is that it is vectorized. Thus it's potentially much faster. Due to the vector nature of the arguments, one can nest ifelse() operations.
- In the following example, involving an abalone data set, gender is coded as 'M', 'F' or 'I', the last meaning infant. We wish to recode those characters as 1, 2 or 3:

```
> g <-c("M","F","F","I","M")
> ifelse(g == "M",1,ifelse(g == "F",2,3))
[1] 1 2 2 3 1
```

- The inner call to ifelse(), which of course is evaluated first, produces a vector of 2s and 3s, with the 2s corresponding to female cases, and 3s being for males and infants. The outer call results in 1s for the males, in which cases the 3s are ignored.
- Remember, the vectors involved could be columns in matrices, and this is a very common scenario. Say our abalone data is stored in the matrix ab, with gender in the first column.

```
> ab<-matrix(c("M","F","I","M","F","I"),nrow=3)
> ab
      [,1] [,2]
[1,] "M"  "M"
[2,] "F"  "F"
[3,] "I"  "I"
> ab[,1] <-ifelse(ab[,1] == "M",1,ifelse(ab[,1] == "F",2,3))
> ab
      [,1] [,2]
[1,] "1"  "M"
[2,] "2"  "F"
[3,] "3"  "I"
```

3. Control Structures

- Computation in R consists of sequentially evaluating statements. Statements, such as `x <- 10:100` or `mean(x)`, can be separated by either a semi-colon or a new line. Whenever the evaluator is presented with a syntactically complete statement that statement is evaluated and the value returned. The result of evaluating a statement can be referred to as the value of the statement. The value can always be assigned to a symbol.
- Statements can be grouped together using braces '{' and '}'. A group of statements is called a block. Single statements are evaluated when a new line is typed at the end of the syntactically complete statement. Blocks are not evaluated until a new line is entered after closing the brace.

```
> { i <- 50
+ i /5
+ }
[1] 10
```

- The following are the basic control-flow constructs of the R language: They function in much the same way as control statements in any programming language.
 - a. Statements that perform testing that shifts flow**
 1. if statements
 2. switch statements
 - b. Statements that perform / control looping**
 1. for
 2. while
 3. repeat
 4. break
 5. next

3.1. Statements that perform testing that shifts flow

3.1.1. Statement if

- The if/ else statement conditionally evaluates two statements. There is a condition which is evaluated and if the value is TRUE then the first statement is evaluated; otherwise the second statement will be evaluated. The if/else statement returns the value, the value of the statement that was selected.

- The formal syntax is

```
if      (statement1)
  statement2
else
  statement3
```

- First, statement1 is evaluated to yield value1. If value 1 is a logical vector with first element **TRUE** then statement2 is evaluated.
- If the first element of value1 is FALSE then statement3 is evaluated.
- If value1 is a numeric vector then statement3 is evaluated when the first element of value1 is zero and otherwise statement2 is evaluated. Only the first element of value1 is used. All the other elements are ignored. If value1 has any type other than a logical or numeric vector an error is returned.
- The else clause is optional.
- When the **if** statement is not in a block the **else**, if present, must appear on the same line as the end of the statement2. Otherwise the new line at the end of statement2 completes the **if** and yields a syntactically complete that is evaluated. A simple solution is to use a compound statement wrapped in braces, putting the else on the same line as the closing brace that marks the end of the statement.
- The statement if /else can be nested.

```
if      (statement1) {
  statement2
} else if (statement3) {
  Statement4
} else if (statement5) {
  Statement6
} else
  Statement8
```

- One of the even numbered statements will be evaluated and the resulting value is returned.

- If the optional else clause is omitted and all the odd numbered statements evaluate to FALSE no statement will be evaluated and NULL is returned.
- The odd numbered statements are evaluated, in order, until one evaluates to TRUE and then the associated even numbered statement is evaluated.

```
> x <- 8
> if (x < 2)
+ {
+   x <- x + 4
+   cat( " increment that number \n")
+ } else
+ {
+   x <- x - 2
+   cat( " make it a smaller number \n")
+ }
make it a smaller number
```

3.1.2. Switch statements

- The switch takes an expression and returns a value in a list based on the value of the expression. How it does this depends on the data type of the expression.
- The syntax of the switch statement is

```
switch(object,
       "value 1" = {expr1},
       "value 2" = {expr2},
       "value 3" = {expr3},
       {other expressions}
)
```

- If the value of value1 then the expression expr1 is executed, if it has the value of value2 then the expression expr2 is executed and so on.
- If the object has no match, the switch will return NULL in case the object does not match any value.
- An expression expr1 can consist of multiple statements. Each statement should be separated with ; or on a separate line and surrounded by curly brackets.

Example 1:

Generate ten random values using normal distribution. Using switch statement evaluate mean, median, sd for these ten values.

Solution

```
> y <- rnorm(10) # this generates 10 random values using normal distribution
> y
[1] -1.4640718  0.7020654  0.1034881 -1.2059056  0.2833055  0.0294456 -0.3560928  0.7484389
[9]  0.8165163 -1.1524026
> x="mean"# for mean calculation
> x <- switch(x,
+ "mean" = {
+   Ans <- mean(y)
+   cat("mean = ",Ans,"\n")
+ },
+ "median" = {
+   Ans <- median(y)
+   cat("median = ",Ans,"\n")
+ },
+ "sd" = {
+   Ans <- sd(y)
+   cat("sd = ",Ans,"\n")
+ }
+ )
mean = -0.1495213
```

```
> x="median"# for median calculation
> x <- switch(x,
+ "mean" = {
+   Ans <- mean(y)
+   cat("mean = ",Ans,"\n")
+ },
+ "median" = {
+   Ans <- median(y)
+   cat("median = ",Ans,"\n")
+ },
+ "sd" = {
+   Ans <- sd(y)
+   cat("sd = ",Ans,"\n")
+ }
+ )
median = 0.06646683
```

```
> x="sd"# for standard deviation calculation
> x <- switch(x,
+ "mean" = {
+   Ans <- mean(y)
+   cat("mean = ",Ans,"\n")
+ },
+ "median" = {
+   Ans <- median(y)
+   cat("median = ",Ans,"\n")
+ },
+ "sd" = {
+   Ans <- sd(y)
+   cat("sd = ",Ans,"\n")
+ }
+ )
sd = 0.8589772
```

3.2. Statements that perform / control looping

3.2.1. Statement – for

- For loops can be used to loop through the values of an array. It is of the form
for <index> *in* <vector> {<statements>}
- The expressions between curly brackets are executed separately for each value in the vector.
- The following example should print a list with all the values in the vector 1:8.

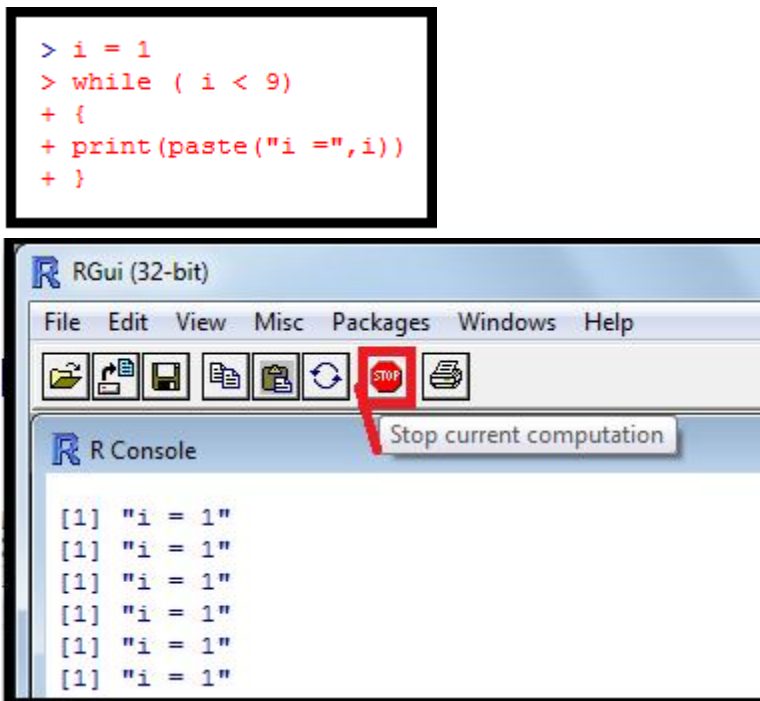
```
> for (i in 1:8)
+ {
+   print(paste("i =",i));
+ }
[1] "i = 1"
[1] "i = 2"
[1] "i = 3"
[1] "i = 4"
[1] "i = 5"
[1] "i = 6"
[1] "i = 7"
[1] "i = 8"
```

3.2.2. Statement – while

- One of the simplest looping structures is the while loop, which takes the form
while (<condition>) {statements}
- We shall try out the above example using while statement.

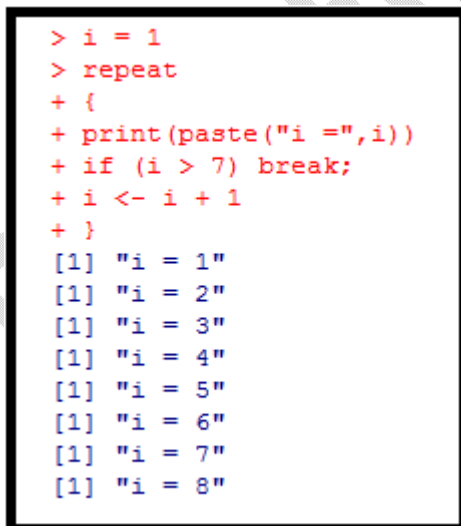
```
> i = 1
> while ( i < 9)
+ {
+   print(paste("i =",i))
+   i = i + 1
+ }
[1] "i = 1"
[1] "i = 2"
[1] "i = 3"
[1] "i = 4"
[1] "i = 5"
[1] "i = 6"
[1] "i = 7"
[1] "i = 8"
```

- Use while loop with caution; never write an infinite loop. Whenever you encounter an infinite loop, you can stop execution of the program using the stop just below the menu bar.



3.2.3. Statement – repeat

- Another loop structure provided by R is the **repeat** structure. This structure repeats the commands in its body until a **break** statement is reached.
- We shall try out the above example using **repeat** statement.



3.2.4. Statement – break

- The **break** statement breaks out of a **for**, **while** or **repeat** loop.
- Control is transferred to the first statement outside the inner-most loop.
- The **break** statement applies only to the innermost of the nested loops.

3.2.5. Statement – next

- The **next** statement halts the processing of the current iteration and advances the looping index.
- The **next** statement applies only to the innermost of the nested loops.
- This example skips printing the number 5 in letters.

```
> i = 0
> num_ch <- c("One", "Two", "Three", "Four", "Five", "Six", "Seven")
> a <- c("", "", "", "", "", "", "")
> while (i < 7)
+ {
+   i <- i + 1
+   if (i == 5) {
+     next
+   }
+   a[i] <- num_ch[i]
+ }
>
> a
[1] "One"   "Two"   "Three" "Four"  ""       "Six"   "Seven"
```

3.3. Loops and vectortization

- However, loops and control structures can be avoided in most situations because of the feature vectorization.
- Vectorization makes loops implicit in expression.
- For example, consider the addition of two vectors x and y.

```
> x <- c(1,2,3,4,5)
> y <- c(5,4,3,2,1)
> z <- x + y # add two vectors x and y
> z
[1] 6 6 6 6 6
> ex_3 <- function(x,y) {
+   z <- numeric(length(x))
+   for (i in 1:length(x))
+     z[i] <- x[i] + y[i]
+   return (z)
+ }
> ex_3(x,y)
[1] 6 6 6 6 6
```

- The explicit loop will work only if x and y are of same length; whereas the first expression `z <- x + y` will work in all situations.
- *In addition to being simpler, vectorized expressions are computationally more efficient, particularly with large quantities of data.*

3.4. Functions of type 'apply'

- Several functions of the type 'apply' which avoids writing loops can be used very effectively.
- The function apply acts on rows and / or columns of a matrix.
- The function apply returns a vector or list of values obtained by applying a function to margins of an array or matrix.

Syntax: apply(X, MARGIN, FUN,...)

- Arguments for the function apply

X	An array including a Matrix
MARGIN	for a matrix, this indicates whether to consider the rows (1) or columns (2) or both (c(1,2)). Where X has named dimnames, it can be a character vector selecting dimension names.
FUN	the function to be applied
....	optional arguments to FUN

```
> matrix1<-matrix(rep(seq(5), 4), ncol = 4)
> matrix1
      [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    2    2    2    2
[3,]    3    3    3    3
[4,]    4    4    4    4
[5,]    5    5    5    5
> # row sum of matrix1
> apply(matrix1,1,sum)
[1]  4  8 12 16 20
> # column sum of matrix1
> apply(matrix1,2,sum)
[1] 15 15 15 15
> # Using a user defined function
> # Add 5 to the row sums
> apply(matrix1,1,function(x,y) sum(x)+ y, y= 5)
[1]  9 13 17 21 25
> # Add 5 to the column sums
> apply(matrix1,2,function(x,y) sum(x)+ y, y= 5)
[1] 20 20 20 20
```

- The function `lapply` acts on a list; its syntax is similar to `apply` and it returns a list
Syntax: `lapply(list, function, ...)`
- In R, the data frame is considered as a list and the variables in the data frame are the elements of the list.

```
> # creating a data frame using matrix1
> matrix1.df<-data.frame(matrix1)
> matrix1.df
  X1 X2 X3 X4
1  1  1  1  1
2  2  2  2  2
3  3  3  3  3
4  4  4  4  4
5  5  5  5  5
> class(matrix1.df) # to find the class of matrix1
[1] "data.frame"
> # Obtaining the sum of each variable in matrix1.df
> lapply(matrix1.df,sum)
$X1
[1] 15

$X2
[1] 15

$X3
[1] 15

$X4
[1] 15

> # User defined function with multiple arguments
> # function defined inside the lapply function
> # displaying the first two results in the list
> lapply(matrix1.df,function(x,y) sum(x)+ y, y= 5)
$X1
[1] 20

$X2
[1] 20

$X3
[1] 20

$X4
[1] 20
```

- The function `supply` applies function to elements in a list and returns the results in a vector, matrix or a list.

Syntax: `supply (list, function,..., simplify = T)`

where `simplify = T` returns the results in a simplified form if at all possible.

```
> y2<- supply(matrix1.df, function(x,y) sum(x) + y, y = 5)
> matrix1.df
  X1 X2 X3 X4
1  1  1  1  1
2  2  2  2  2
3  3  3  3  3
4  4  4  4  4
5  5  5  5  5
> y2
  X1 X2 X3 X4
20 20 20 20
```

4. Writing functions

- Writing your own functions allows an efficient, flexible, and rational use of R.
- To execute a function, the function must be loaded in memory.
 1. The lines of function can be typed directly on the key board, like any other command, or
 2. Copied and pasted from an editor.
 3. If this function is saved in a text file, it can be loaded with `source()` like another program.
 4. If the user wants some function to be loaded each time when R starts, they can be saved in a workspace ".Rdata" which will be loaded in memory if it is in the working directory.
 5. Another possibility is to configure the file '.Rprofile' or 'Rprofile'.
 6. Finally, it is possible to create a package.
- Once the function is loaded into the memory, we can access the function with the single command. This function creates less than and more than cumulative frequencies.
- The syntax for writing a function is

function (arglist) body

- The first component of the function declaration is the keyword ***function*** which indicates to R that you want to create a function.
- An argument list is a comma separated list of formal arguments. A formal argument can be a symbol, a statement of the form '*symbol = expression*', or the special formal argument '*....*'.
- The *body* can be any valid R expression. Generally, the body is a group of expressions contained in curly braces '{' and '}'.
- Generally functions are assigned to symbols but they don't need to be. The value returned by the call to ***function*** is a function.
- Anonymous functions are most frequently used as arguments to other functions such as the `apply` family or `outer`.
- Here is a simple function: `echo <- function (x) print(x)`
- So, `echo` is a function that takes a single argument and when `echo` is invoked it prints its argument.
- The function to create a cumulative and reverse cumulative frequency table is shown below:


```

R Information

freqmatrix <- function(x) {
  cumfreq = cumsum(x)
  sumv = sum(x)
  morethanfrq = sumv
  xlen = length(x)
  rvcumfrq = rep(0,xlen)
  for ( i in 1:xlen) {
    rvcumfrq[i] = morethanfrq
    morethanfrq = morethanfrq - x[i]
  }
  m <- as.matrix(cbind(x,cumfreq,rvcumfrq))
  return (m)
}

```

```

> m= freqmatrix(c(7,11,24,32,9,14,2,1))
> m
      x cumfreq rvcumfrq
[1,]  7        7      100
[2,] 11       18       93
[3,] 24       42       82
[4,] 32       74       58
[5,]  9       83       26
[6,] 14       97       17
[7,]  2       99        3
[8,]  1      100        1

```

5. Built-in functions

Refer to <http://www.statmethods.net/management/functions.html>

Numeric functions	
abs(x)	Absolute value
sqrt(x)	Square root
ceiling(x)	ceiling(3.475) is 4
floor(x)	floor(3.475) is 3
trunk(x)	trunk(6.99) is 6
round(x, digits=n)	round(3.475,digits=2) is 3.48
signif(x, digits=n)	signif(3.475, digits=2) is 3.5
log(x)	Natural logarithm
log10(x)	Common algorithm
exp(x)	Exponential of x: e^x
cos(x), sin(x), tan(x)	

Character functions	
<code>substr(x,start=n1,stop=n2)</code>	Extract or replace substrings in a character vector. a <- "uvwxyz" substr(x,4,6) is "xyz"
<code>grep(pattern,x,ignore.case=FALSE, fixed=FALSE)</code>	Search for <i>pattern</i> in x. If fixed = FALSE then <i>pattern</i> is a regular expression. If fixed = TRUE then <i>pattern</i> is a text string. Returns matching indices. grep("A",c("b","A","c"), fixed=TRUE) returns 2
<code>sub(pattern, replacement, x, ignore.case=FALSE, fixed=FALSE)</code>	Find <i>pattern</i> in x and <i>replace</i> with <i>replacement</i> text. If fixed = FALSE then <i>pattern</i> is a regular expression. If fixed = TRUE then <i>pattern</i> is a text string. sub("There","Sir",Hello There) returns "Hello Sir"
<code>strsplit(x, split)</code>	Split the elements of character vector x at <i>split</i> . Strsplit("abc","") returns 3 element vector "a","b","c"
<code>paste(...,sep="")</code>	Concatenate strings after using <i>sep</i> string to a separate them. paste("x",1:3,sep="") returns <code>"x1" "x2" "x3"</code>
<code>toupper(x)</code>	Converts x into upper case. <code>> toupper("uvwxyz")</code> <code>[1] "UVWXYZ"</code>
<code>tolower</code>	Converts x into lower case. <code>> tolower("ABCDEF")</code> <code>[1] "abcdef"</code>

Statistical Probability Functions

The following table describes functions related to probability distributions. For random number generators below, you can use `set.seed(1234)` or some other integer to create reproducible pseudo-random numbers.

Function	Description
<code>dnorm(x)</code>	normal density function (by default $m=0$ $sd=1$) # plot standard normal curve <code>x <- pretty(c(-3,3), 30)</code> <code>y <- dnorm(x)</code> <code>plot(x, y, type="l", xlab="Normal Deviate", ylab="Density", yaxs="i")</code>
<code>pnorm(q)</code>	cumulative normal probability for q (area under the normal curve to the right of q) <code>pnorm(1.96)</code> is 0.975
<code>qnorm(p)</code>	normal quantile. value at the p percentile of normal distribution <code>qnorm(.9)</code> is 1.28 # 90th percentile
<code>rnorm(n, m=0, sd=1)</code>	n random normal deviates with mean m and standard deviation sd . #50 random normal variates with mean=50, $sd=10$ <code>x <- rnorm(50, m=50, sd=10)</code>
<code>dbinom(x, size, prob)</code> <code>pbinom(q, size, prob)</code> <code>qbinom(p, size, prob)</code> <code>rbinom(n, size, prob)</code>	binomial distribution where $size$ is the sample size and $prob$ is the probability of a heads (π) # prob of 0 to 5 heads of fair coin out of 10 flips <code>dbinom(0:5, 10, .5)</code> # prob of 5 or less heads of fair coin out of 10 flips <code>pbinom(5, 10, .5)</code>
<code>dpois(x, lamda)</code> <code>ppois(q, lamda)</code> <code>qpois(p, lamda)</code> <code>rpois(n, lamda)</code>	poisson distribution with $m=std=lamda$ #probability of 0,1, or 2 events with $lamda=4$ <code>dpois(0:2, 4)</code> # probability of at least 3 events with $lamda=4$ <code>1- ppois(2,4)</code>
<code>dunif(x, min=0, max=1)</code> <code>punif(q, min=0, max=1)</code> <code>qunif(p, min=0, max=1)</code> <code>runif(n, min=0, max=1)</code>	uniform distribution, follows the same pattern as the normal distribution above. #10 uniform random variates <code>x <- runif(10)</code>

```
> # dt is t distribution function
> x <- seq(-20,20,by=0.5)
> y <- dt(x,df=50)
> head(y)
[1] 1.843125e-25 5.788839e-25 1.859976e-24 6.118128e-24 2.061807e-23 7.124011e-23
> # t probability distribution function
> x = c(-3, -4, -2, -1)
> pt((mean(x) - 2)/sd(x),df=40)
[1] 0.000603064
> # t quantile distribution function
> a = c(0.005, 0.025, 0.05)
> qt(a,df=50)
[1] -2.677793 -2.008559 -1.675905
> # t random number
> rt(5,df=50)
[1] -0.82781293 0.18295457 0.70438375 0.05774089 -0.40249570
```

```
> #
> # dchisq is chi-square distribution function
> x <- seq(-20,20,by=0.5)
> y <- dchisq(x,df=50)
> head(y)
[1] 0 0 0 0 0 0
> # chi-square probability distribution function
> x = c(-3, -4, -2, -1)
> dchisq((mean(x) - 2)/sd(x),df=40)
[1] 0
> # chi-square quantile distribution function
> a = c(0.005, 0.025, 0.05)
> qchisq(a,df=50)
[1] 27.99075 32.35736 34.76425
> # chi-square random number
> rchisq(5,df=50)
[1] 59.78357 48.56802 55.62401 47.09176 44.28001
```

```
> #
> # df is F distribution function
> x <- seq(-400,400,by=5)
> y <- df(x^2,2,50)
> head(y)
[1] 1.090326e-99 2.096921e-99 4.066538e-99 7.953861e-99 1.569417e-98 3.124691e-98
> # F probability distribution function
> x = c(-3, -4, -2, -1)
> pf((mean(x^2) - 2)/sd(x^2),1,10)
[1] 0.6186886
> # F quantile distribution function
> qf(0.95,df1=5,df2=2) # 95th percentile of F distribution with (5,2) df
[1] 19.29641
> # F random number
> rf(5,2,10)
[1] 0.4450600 0.2332314 0.3360383 1.2393196 2.7716963
> #
```

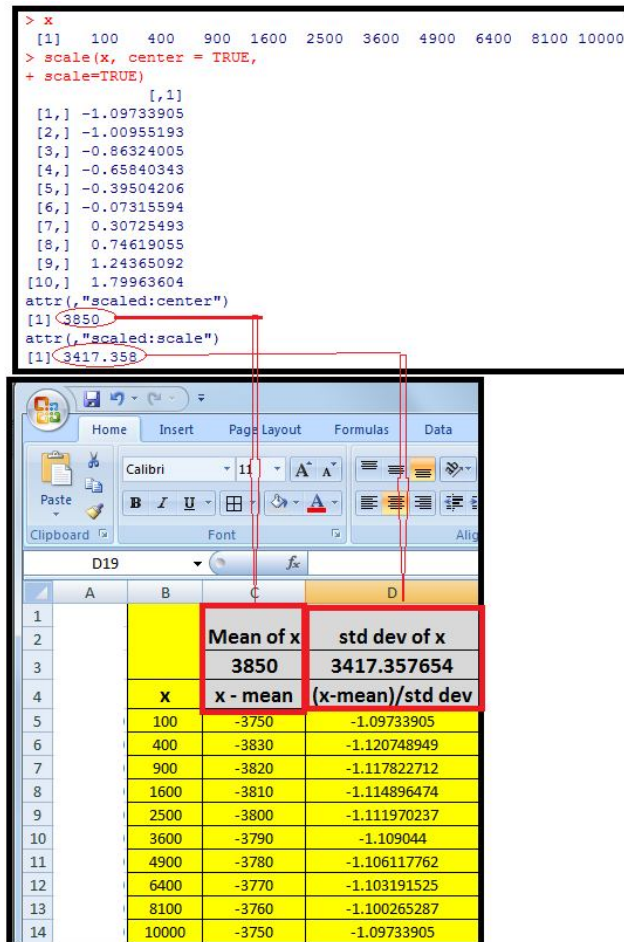
In R console, to see all the statistical functions type as follows:

```
> help(package=stats)
```

Also see http://www.sr.bham.ac.uk/~ajrs/R/r-function_list.html

Other Statistical functions	
mean(x, trim=0, na.rm=FALSE)	Mean of object x # trimmed mean, removing any missing values and 5 percent o highest and lowest scores mx <- mean(x, trim=0.5, na.rm=TRUE)
sd(x)	Standard deviation of the object x
var(x)	Variance of the object x
median(x)	Median
quantile(x, probs)	Quantiles where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0,1]
range(x)	Range of the x values
sum(x)	Sum of the x values
diff(x, lag=1)	Lagged differences, with lag indicating which lag to use
min(x)	Minimum of the x values
max(x)	Maximum of the x values
scale(x, center = TRUE, scale=TRUE)	Column center or standardize a matrix
cov	Covariance Matrix
cor	Correlation
lm	Simple linear Regression, Multiple Linear Regression
aov, lm.aov	Analysis of Variance
xtabs, table	Contingency table
t.test	T tests for means
prop.test	Tests for proportions
chisq.test	Chi-square for independence
Wilcox.test, Friedman.test, kruskal.test	Various non-parameter functions
hist	Histogram
stem	Stem and leaf display
boxplot	Box plot
ts.plot	Time-series plot

Example of using a scale function:



Other Useful functions	
<code>seq(from, to, by)</code>	Generate a sequence indices <- seq(1,10,2) #indices is c(1,3,5,7,9)
<code>rep(x, times)</code>	Repeat x n times y <- rep(1:3,2) #y is c(1,2,3,1,2,3)
<code>cut(x,n)</code>	Divide continuous variable in factor with n levels y <- cut(x,5)

```
> x
[1] 1 1 1 1 1 1 3 3 3 3 5 5 5 5 7 7 7 7 7 7 7 7 9 9 9 9
> z <- c(1,4,8,12)
> table(cut(x,z))

(1,4] (4,8] (8,12]
     5    13      4
```

6. Scoping rules

- The symbols which occur in the body of a function can be divided into three classes:
 - formal parameters
 - local variables
 - free variables
- The formal parameters of a function are those occurring in the argument list of the function. Their values are determined by the process of binding the actual function arguments to the formal parameters.
- Local variables are those whose values are determined by the evaluation of expression in the body of the functions.
- Variables which are not formal parameters or local variables are called free variables.

```
> f <- function(x) {
+ y <- x + 2
+ x+y+a
+ }
> a <- 4
> f(44)
[1] 94
```

- *In the above function, x is a formal parameter, y is a local variable and a is a free variable*
- In R, it is not necessary to declare the variables used within a function.
- When a function is executed, R uses a rule called lexical scoping to decide whether an object is local to the function or global.

```
> x <- 1
> ex1 <- function() {x <-2; print (x *x)}
> ex1
function() {x <-2; print (x *x)}
> ex1()
[1] 4
> x
[1] 1
```

- In the above example, x is used as the name of the object within our function, the value of x in the global environment is not used. In this example, print() uses the object x that is defined within its environment of ex1.

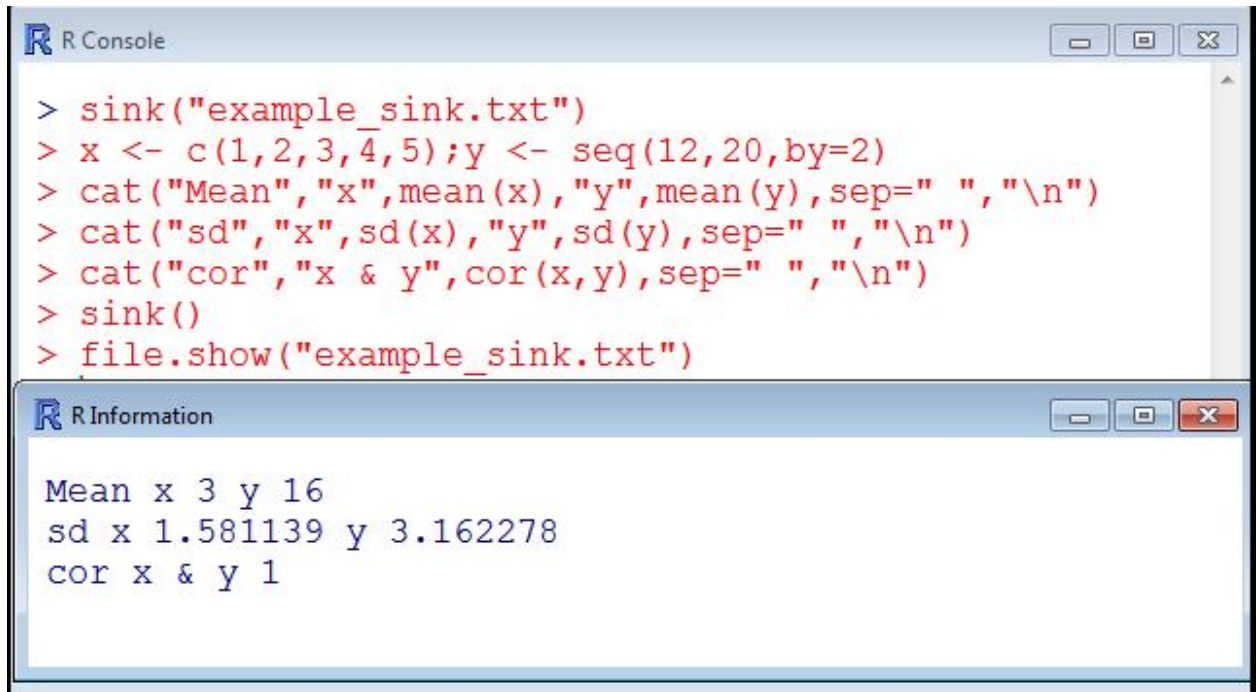
- There are two ways to specify arguments to a function: by their positions or by their names (also called tagged arguments).
ex <- function(arg1,arg2,arg3) {.....}
- Now ex can be executed without the names of the arg1, arg2,...if the corresponding objects are placed in the correct position. for instance ex(x,y,z).
- The position has no importance if the names of the arguments are used, for instance ex(arg3=z,arg1=y,arg2=y).
- Another possibility is use the default values in the function definition, for instance ex <- function(arg1 = 4, arg2 = 3, arg3 = 5) {....}
- The commands ex(x,3,5) or ex(4,y,z) or ex(a,y) produce the same result.
- If you do want to write to global variables (or more precisely, to variables one level higher than the current scope), you can use the super-assignment operator, >>.

```
> cubex <- function(x) {  
+ x <- x * x * x  
+ u <- u * u * u  
+ v <- v * v * v  
+ a <- c(x,u,v)  
+ return (a)  
+ }  
> x <- 4  
> u <- 5  
> v <- 6  
> cubex(x)  
[1] 4 125 216  
> x  
[1] 64  
> u  
[1] 125  
> v  
[1] 6
```

- *Note the values of x and u are changed after calling the function cubex and the value of v is not changed.*

7. Directing console output to a file

- `sink(file)` diverts R output to a connection; where `file` is a writable connection or a character string naming the file to write to, or `NULL` to stop sink-ing.
- The command `file.show` displays one or more files.



The screenshot shows two R windows. The 'R Console' window contains the following commands:

```
> sink("example_sink.txt")
> x <- c(1,2,3,4,5); y <- seq(12,20,by=2)
> cat("Mean", "x", mean(x), "y", mean(y), sep=" ", "\n")
> cat("sd", "x", sd(x), "y", sd(y), sep=" ", "\n")
> cat("cor", "x & y", cor(x,y), sep=" ", "\n")
> sink()
> file.show("example_sink.txt")
```

The 'R Information' window displays the output of the `cat` commands:

```
Mean x 3 y 16
sd x 1.581139 y 3.162278
cor x & y 1
```

8. Reading from and writing to an external file

8.1. Handling files in R

We can open the file to read or write or both by using the function `file()`.

Various file opening modes:

#	Mode	Description	Syntax
1	r or rt	Open to read in text mode	<code>file(filename, open = "r")</code> or <code>file(filename, open = "rt")</code>
2	w or wt	Open to write in text mode	<code>file(filename, open = "w")</code> or <code>file(filename, open="wt")</code>
3	a or at	Open to append in text mode	<code>file(filename, open = "a")</code> or <code>file(filename, open="at")</code>
4	rb	Open to read in binary file	<code>file(filename, open = "rb")</code>
5	wb	Open to write in binary file	<code>file(filename, open = "wb")</code>
6	Ab	Open to append in binary file	<code>file(filename, open = "ab")</code>

7	"r+","r+b"	Open to read and write file	file(filename, open = "r+") file(filename, open = "r+b")
8	"w+","w+b"	Open to read and write, truncate file	file(filename, open = "w+") file(filename, open = "w+b")
9	"a+","a+b"	Open to reading and appending file	file(filename, open = "a+") file(filename, open = "a+b")

```
> setwd("D:/R")
> file("graph1.R", open = "r")
description      class      mode      text      opened      can read
"graph1.R"      "file"      "r"      "text"      "opened"      "yes"
can write
"no"
> file("graph1.R", open = "w")
description      class      mode      text      opened      can read
"graph1.R"      "file"      "w"      "text"      "opened"      "no"
can write
"yes"
> file("graph1.R", open = "a")
description      class      mode      text      opened      can read
"graph1.R"      "file"      "a"      "text"      "opened"      "no"
can write
"yes"
> file("graph1.R", open = "w+")
description      class      mode      text      opened      can read
"graph1.R"      "file"      "w+"      "text"      "opened"      "yes"
can write
"yes"
> file("graph1.R", open = "r+")
description      class      mode      text      opened      can read
"graph1.R"      "file"      "r+"      "text"      "opened"      "yes"
can write
"yes"
```

File manipulation commands

#	Command	Description
1	file.create(filename)	Creates a file with the file name given as the argument. It provides the output "TRUE" if the file is successfully created or "FALSE" other wise.
2	file.exists(filename)	Checks whether the file with the filename specified in the argument exists in the directory. It provides the output "TRUE" if the file exists or "FALSE" other wise.
3	file.access(filename, mode=0)	Checks the permission to access a file. The mode value can be 0 or 1 or 2 or 4 based on the requirement to test for the existence, execute, write, and read respectively. Will return 0 if permitted and -1 for failure.
4	file.edit(filename)	Will open the file in a text editor for editing

		purpose.
5	<code>file.rename(file1, file2)</code>	Will rename file1 with file2. Will return TRUE if renamed successfully and FALSE if vice versa.
6	<code>file.remove(filename)</code>	Remove the file named in the argument. if successfully removed returns "TRUE"; else returns "FALSE"

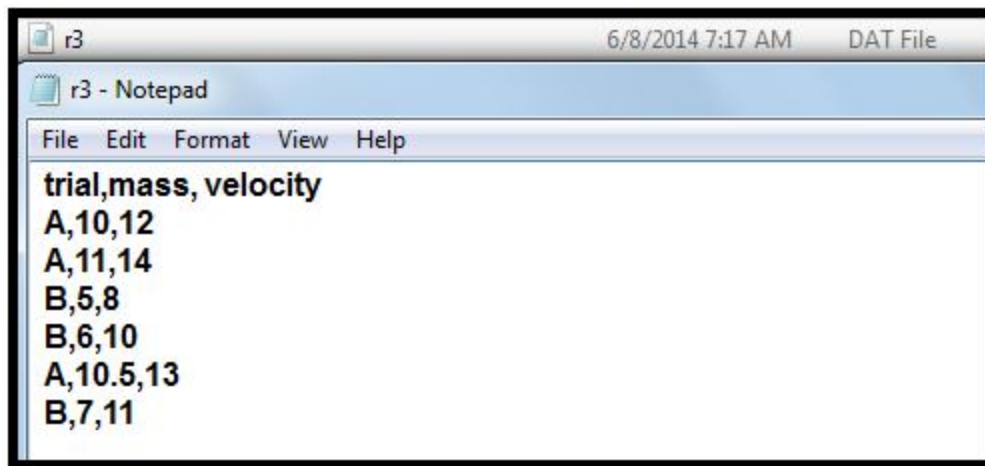


```
> setwd("D:/R")
> file.create("file1.txt")
[1] TRUE
> #
> file.exists("file1.txt")
[1] TRUE
> #
> file.exists("file11.txt")
[1] FALSE
> #
> file.access("file1.txt",mode=4)
file1.txt
0
> #
> file.edit("file1.txt")
> #
> file.rename("file1.txt", "copyfile1.txt")
[1] TRUE
> #
> file.exists("file1.txt", "copyfile1.txt")
[1] FALSE TRUE
> #
> file.remove("copyfile1.txt")
[1] TRUE
> #
> file.exists("copyfile1.txt")
[1] FALSE
```

The screenshot shows an R console window on the left with the above commands and their outputs. On the right is an R Editor window titled "file1.txt - R Editor". A red arrow points from the `file.edit("file1.txt")` command in the console to the R Editor window. The title bar of the R Editor window is circled in red.

8.2. Reading from a csv file

- Unfortunately, it is rare to have just a few data points that you do not mind typing in at the prompt. It is much more common to have a lot of data points with complicated relationships.
- Here we will examine how to read a data set from a file using the `read.csv` function but first discuss the format of a data file.
- We assume that the data file is in the format called "comma separated values" (csv). That is, each line contains a row of values which can be numbers or letters, and each value is separated by a comma.
- We also assume that the very first row contains a list of labels. The idea is that the labels in the top row are used to refer to the different columns of values.
- First we read a very short, somewhat silly, data file. The data file is called [r3.dat](#) and has three columns of data and six rows. The three columns are labeled "trial," "mass," and "velocity." We can pretend that each row comes from an observation during one of two trials labeled "A" and "B."



```
trial,mass, velocity
A,10,12
A,11,14
B,5,8
B,6,10
A,10.5,13
B,7,11
```

- The command to read the data file is `read.csv`. We have to give the command at least one arguments, but we will give three different arguments to indicate how the command can be used in different situations.
- The first argument is the name of file.
- The second argument indicates whether or not the first row is a set of labels. The third argument indicates that there is a comma between each number of each line.
- The following command will read in the data and assign it to a variable called “a”.
- Note: Last line in `r3.dat` is a blank line.

```
> a <- read.csv(file='r3.dat',head=TRUE,sep=",")
> a
  trial mass velocity
1    A 10.0      12
2    A 11.0      14
3    B  5.0       8
4    B  6.0      10
5    A 10.5      13
6    B  7.0      11
```

8.3. Reading from a fixed-width file

- It is common to come across data that is organized in flat files and delimited at preset locations on each line. This is often called a “fixed width file.”
- The command to deal with these kind of files is `read.fwf`.
- If you would like more information on how to use this command enter the following command:
 - `help(read.fwf)`
- The **`read.fwf`** command requires at least two options.
- The first is the name of the file and the second is a list of numbers that gives the length of each column in the data file.
- A negative number in the list indicates that the column should be skipped.

- Here we give the command to read the data file r2.dat. In this data file there are three columns. The first column is 2 characters wide, the second column is 7 characters wide, and the last column is 9 characters wide.
- In the example below we use the optional col.names option to specify the names of the columns:

```
> atemp = read.fwf('r2.dat',widths=c(-2,7,9),col.names=c('temp','offices'),nrows=4)
> atemp
  temp offices
1 17.0      35
2 18.0     117
3 17.5      19
4 17.5      28
```

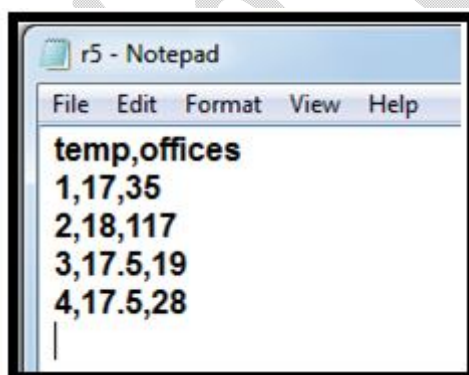
File contents of r2.dat

```
1 17.000000035
2 18.000000117
3 17.500000019
4 17.500000028
<blank line>
```

8.4. Writing to an external file

- The function write.table writes file, an object typically a data frame but this could well be another kind of object vector, matrix,...).

```
> write.table(atemp,file="r5.dat",append=FALSE,quote=FALSE,sep=",",eol="\n",
na="NA",row.names=TRUE,col.names=TRUE,qmethod=c("escape","double"))
```



- To write in a simple way an object in a file, the command write(x,file="data.txt") can be used, where the x is the name of the object (vector, matrix, sn array ..).

```
> write(x,file="data.txt")
> x
[1] 1 1 2 2 3 4 5 5 5 5
```



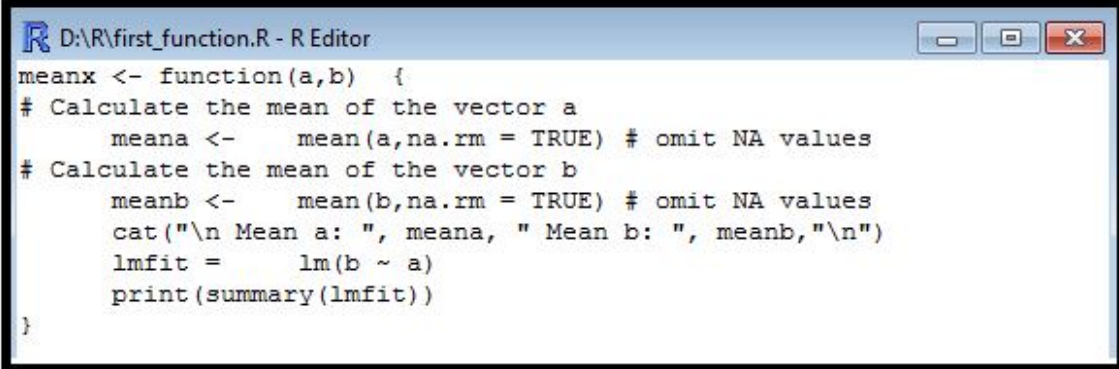
9. Debugging

- Let us make a mistake in the following R script, by replacing `meana` with `meanc`, which is non-existent.
- The `traceback()` function prints call stack, the list of functions which were called before the error occurred.

```
R Console
> a<-c(1,2,3,4,5,6,7,8,NA)
> b<-c(8,7,6,5,4,3,2,1,NA)
> # Calculate the mean of vector a
> meana<-mean(a,na.rm = TRUE) # omit NA values
> # Calculate the mean of vector b
> meanb<-mean(b,na.rm = TRUE) # omit NA values
> cat("\n Mean a: ",meanc," Mean b: ",meanb,"\n")
Error in cat("\n Mean a: ", meanc, " Mean b: ", meanb, "\n") :
  object 'meanc' not found
> traceback()
1: cat("\n Mean a: ", meanc, " Mean b: ", meanb, "\n")
>
```

Using `debug()` for debugging

- Let us create a R script called "first_function.R"

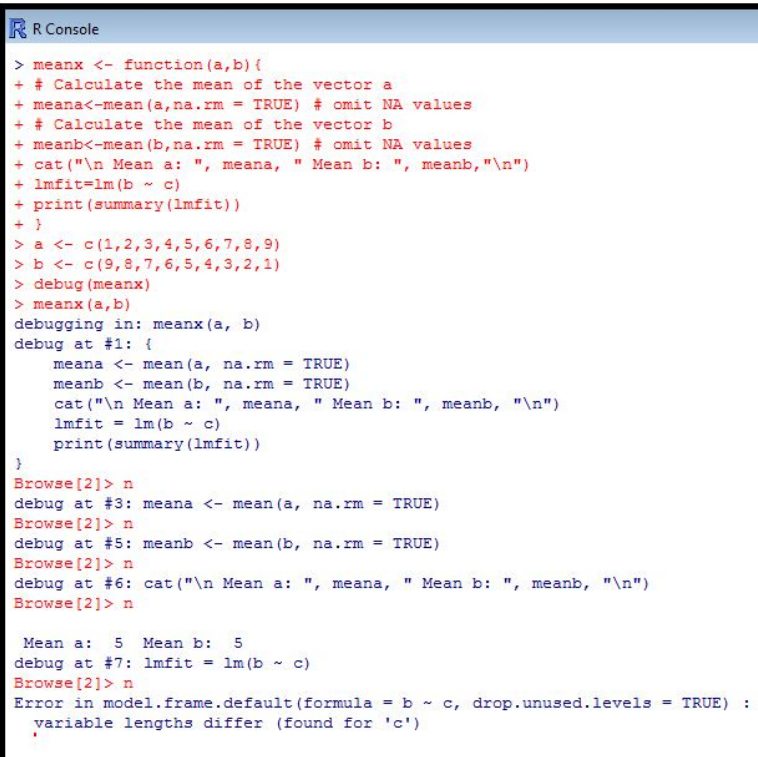


```
D:\R\first_function.R - R Editor

meanx <- function(a,b) {
# Calculate the mean of the vector a
  meana <- mean(a,na.rm = TRUE) # omit NA values
# Calculate the mean of the vector b
  meanb <- mean(b,na.rm = TRUE) # omit NA values
  cat("\n Mean a: ", meana, " Mean b: ", meanb,"\n")
  lmfit = lm(b ~ a)
  print(summary(lmfit))
}
```

- The following R session shows how meanx runs in the debugger:
 - Now, we try debug, which takes a single argument - the name of the function.
 - When you pass the name of a function to debug, that function is flagged for debugging. In order to unflag a function, there is the corresponding undebug function.
 - When a function is flagged for debugging, each statement is executed one at a time and the user can control when each statement gets executed.
 - After a statement is executed, the function suspends and the user is free to interact with the environment.
 - The first thing that happens when you execute a function in the debugger is the body of the function is printed and the following prompt appears:
 - You are now in what is called the "browser".

Browse[2]>



```
R Console

> meanx <- function(a,b){
+ # Calculate the mean of the vector a
+ meana<-mean(a,na.rm = TRUE) # omit NA values
+ # Calculate the mean of the vector b
+ meanb<-mean(b,na.rm = TRUE) # omit NA values
+ cat("\n Mean a: ", meana, " Mean b: ", meanb,"\n")
+ lmfit=lm(b ~ c)
+ print(summary(lmfit))
+ }
> a <- c(1,2,3,4,5,6,7,8,9)
> b <- c(9,8,7,6,5,4,3,2,1)
> debug(meanx)
> meanx(a,b)
debugging in: meanx(a, b)
debug at #1: {
  meana <- mean(a, na.rm = TRUE)
  meanb <- mean(b, na.rm = TRUE)
  cat("\n Mean a: ", meana, " Mean b: ", meanb, "\n")
  lmfit = lm(b ~ c)
  print(summary(lmfit))
}
Browse[2]> n
debug at #3: meana <- mean(a, na.rm = TRUE)
Browse[2]> n
debug at #5: meanb <- mean(b, na.rm = TRUE)
Browse[2]> n
debug at #6: cat("\n Mean a: ", meana, " Mean b: ", meanb, "\n")
Browse[2]> n

  Mean a:  5 Mean b:  5
debug at #7: lmfit = lm(b ~ c)
Browse[2]> n
Error in model.frame.default(formula = b ~ c, drop.unused.levels = TRUE) :
  variable lengths differ (found for 'c')
```

- Four basic debugging commands are:
 - ✓ **n** - executes the current line and prints the next one
 - ✓ **c** - executes the rest of the function without stopping and causes the function to return.
 - ✓ **Q** - quits debugging and completely halts execution of this function.
 - ✓ **where** - shows where you are in the function call stack.
- Other relevant commands, you can type are `ls()` , `print(x)`
- Now rectify the error and again try debug.

```
> a <- c(1,2,3,4,5,6,7,8,9)
> b <- c(9,8,7,6,5,4,3,2,1)
> debug(meanx)
> meanx(a,b)
debugging in: meanx(a, b)
debug at #1: {
  meana <- mean(a, na.rm = TRUE)
  meanb <- mean(b, na.rm = TRUE)
  cat("\n Mean a: ", meana, " Mean b: ", meanb, "\n")
  lmfit = lm(b ~ a)
  print(summary(lmfit))
}
Browse[2]> n
debug at #3: meana <- mean(a, na.rm = TRUE)
Browse[2]> n
debug at #5: meanb <- mean(b, na.rm = TRUE)
Browse[2]> n
debug at #6: cat("\n Mean a: ", meana, " Mean b: ", meanb, "\n")
Browse[2]> n

  Mean a:  5  Mean b:  5
debug at #7: lmfit = lm(b ~ a)
Browse[2]> c

Call:
lm(formula = b ~ a)

Residuals:
    Min       1Q   Median       3Q      Max
-1.88e-15 -2.09e-16  1.08e-17  8.76e-17  1.94e-15

Coefficients:
            Estimate Std. Error  t value Pr(>|t|)
(Intercept)  1.00e+01   7.49e-16  1.34e+16  <2e-16 ***
a           -1.00e+00   1.33e-16 -7.51e+15  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.03e-15 on 7 degrees of freedom
Multiple R-squared:  1,    Adjusted R-squared:  1
F-statistic: 5.64e+31 on 1 and 7 DF,  p-value: <2e-16

exiting from: meanx(a, b)
```


Use browser() command to debug

- Interrupt the execution of an expression and allow the inspection of the environment where browser was called from.

```
> a<-c(1.1,1.2,1.3,1.4,1.5,NA)
> b<-c(2.1,2.2,2.3,2.4,2.5,NA)
> c<-c(21,22,23,24,25,NA)
> browser()
Called from: top level
Browse[1]> mean(a)
[1] NA
Browse[1]> mean(b)
[1] NA
Browse[1]> mean(c)
[1] NA
Browse[1]> mean(a,na.rm= TRUE)
[1] 1.3
Browse[1]> mean(b,na.rm= TRUE)
[1] 2.3
Browse[1]> mean(c,na.rm= TRUE)
[1] 23
Browse[1]> n
```

Use trace() command to debug

- The trace() function modifies a function to allow debug code to be temporarily inserted.
- The untraced() function turns off the tracing.

```
> varx<-function(a,b,c) {
+ mean(a,na.rm=TRUE)
+ mean(b,na.rm=TRUE)
+ mean(c,na.rm=TRUE)
+ var(a+b,na.rm=TRUE)
+ }
> a<-c(1.1,1.2,1.3,1.4,1.5,NA)
> b<-c(2.1,2.2,2.3,2.4,2.5,26)
> c<-c(21,22,23,24,25,26)
> trace("varx", quote(if(any(is.na(a))) { browser() } ), at=2, print=T)
[1] "varx"
> varx(a,b,c)
Tracing varx(a, b, c) step 2
Called from: eval(expr, envir, enclos)
Browse[1]> n
debug: mean(a, na.rm = TRUE)
Browse[2]> a
[1] 1.1 1.2 1.3 1.4 1.5 NA
Browse[2]> n
debug at #3: mean(b, na.rm = TRUE)
Browse[2]> n
debug at #4: mean(c, na.rm = TRUE)
Browse[2]> c
[1] 0.1
```

- The code we've decided to insert simply invokes the browser function if any elements of a have the value NA.
- This expression has to be put into the quote function so that R does not evaluate the code, rather it simply inserts the statements in to the varx function. Without the quote function, R would try to evaluate the if statement within the trace function and it wouldn't make any sense.
- The at argument tells trace where to insert the new code. Here we've instructed trace to insert the code before the second statement.

Use recover() command to debug

- The recover() function allows you to modify the error behaviour so that you can browse the function call stack and check & modify the variables in upper level functions.
- It helps you to suspend the execution of a function in one location and jump up to a higher level in the function of a call stack.

```
> varx<-function(a,b,c)  {
+ mean(a,na.rm=TRUE)
+ mean(b,na.rm=TRUE)
+ mean(c,na.rm=TRUE)
+ var(a+b,na.rm=TRUE)
+ }
> a<-c(1.1,1.2,1.3,1.4,1.5,NA)
> b<-c(2.1,2.2,2.3,2.4,2.5,26)
> c<-c(21,22,23,24,25,26)
> trace("varx", quote(if(any(is.na(a))) { recover() })), at=2, print=T)
[1] "varx"
> varx(a,b,c)
Tracing varx(a, b, c) step 2

Enter a frame number, or 0 to exit

1: varx(a, b, c)

Selection: 1
Called from: .doTrace(if (any(is.na(a))) {
  recover()
}, "step 2")
Browse[1]> a <- c(1.1,1.2,1.3,1.4,1.5,20)
Browse[1]> mean(a)
[1] 4.416667
Browse[1]> c

Enter a frame number, or 0 to exit

1: varx(a, b, c)

Selection: 0
[1] 299.7067
.
```

Warning and stop functions

- Error and warning messages are produced while executing programs. In such situations, one can use the function warning to produce warnings.

For example,

```
> variation <- function(X) {  
+ if (min(X) < 0)  
+ {  
+ warning("Variation only useful for positive data")  
+ }  
+ sd(X) / mean(X)  
+ }  
>  
> variation(rnorm(100))  
[1] -9.841735  
Warning message:  
In variation(rnorm(100)) : Variation only useful for positive data
```

- If you want to raise an error, use the function stop in the above example by replace warning with stop function; this would halt R execution.

```
> variation <- function(X) {  
+ if (min(X) < 0)  
+ {  
+ stop("Variation only useful for positive data")  
+ }  
+ sd(X) / mean(X)  
+ traceback()  
+ }  
>  
> variation(rnorm(100))  
Error in variation(rnorm(100)) : Variation only useful for positive data  
.
```

- R will treat your warnings and errors as normal R warnings and errors and the function traceback can be used to see the call stack when an error occurred.