SECTION 1:A

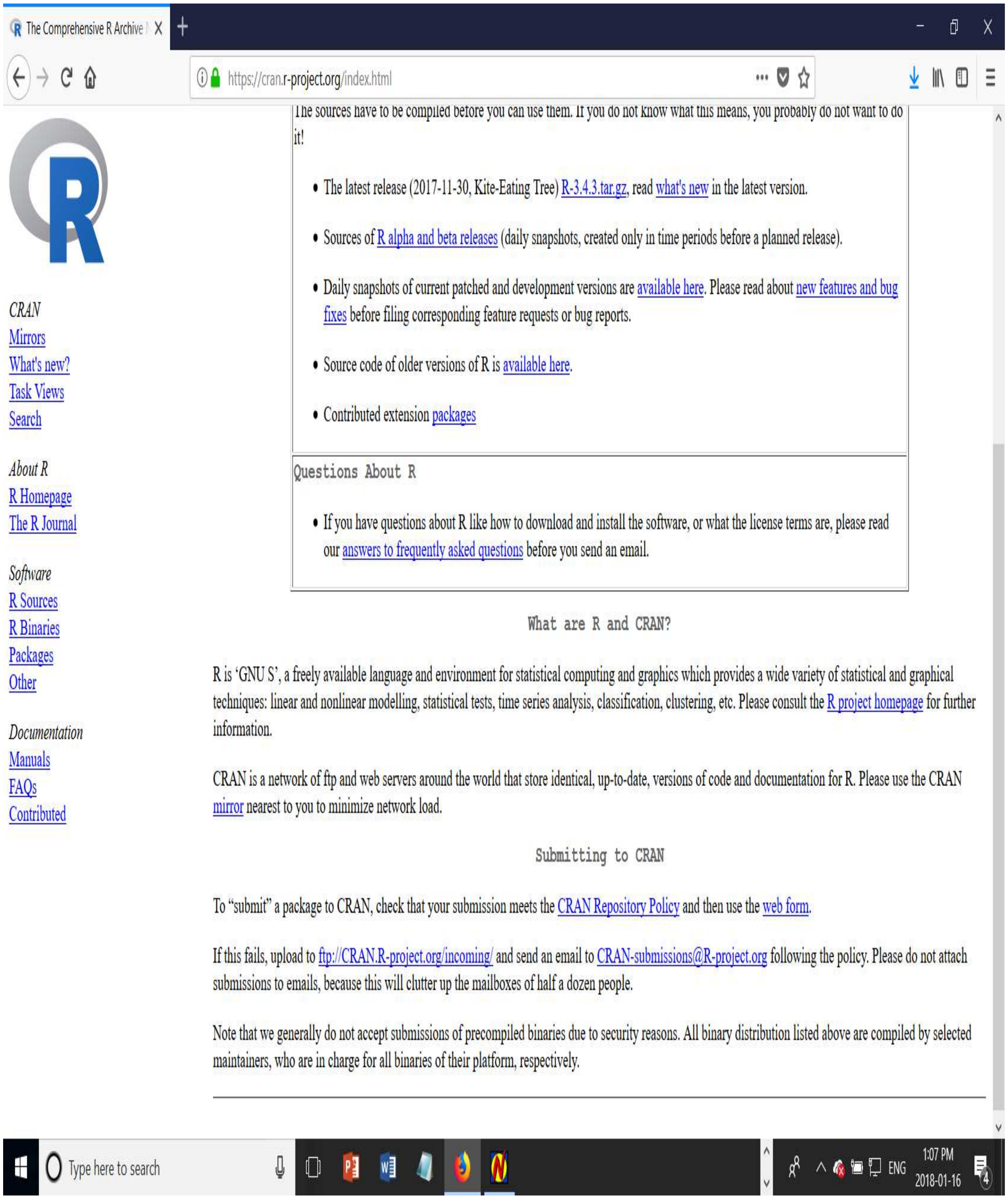
Software

1.1 Introduction to R http://cran.r-project.org

R is an interpretive statistical programming languge similar to the commercial products S+ and Matlab. These languages are based on matrices/vectors and enable us to get results in a concise manner.

One of the best ways to develop an ability to program in R is to cut and paste sample code and then see what happens when you change parts of the code.

A good source for such code is in the examples supplied in the R documentation. The documentation can be obtained by going to http://cran.r-project.org/ and, in the left hand margin, clicking on “Documentation > Manuals”. This brings up the following :


The Comprehensive R Archive
https://cran.r-project.org/index.html
Type here to search
ENG
1:07 PM
2018-01-16

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2017-11-30, Kite-Eating Tree) R-3.4.3.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to minimize network load.

Submitting to CRAN

To "submit" a package to CRAN, check that your submission meets the CRAN Repository Policy and then use the web form.

If this fails, upload to ftp://CRAN.R-project.org/incoming/ and send an email to CRAN-submissions@R-project.org following the policy. Please do not attach submissions to emails, because this will clutter up the mailboxes of half a dozen people.

Note that we generally do not accept submissions of precompiled binaries due to security reasons. All binary distribution listed above are compiled by selected maintainers, who are in charge for all binaries of their platform, respectively.

Figure 1.

If we select “**An Introduction to R**” we will find a good manual for learning R.

While there are many functions for doing statistical calculations, most problems require only a small subset of them. For this reason the functions (and data sets) are broken into libraries or packages. If we select “**Packages**” we will see all the packages that are installed in our version of R. It is likely that there is only a subset of all the possible packages installed and if we want to see what else is available we can find many others at **http**://**cran.r-project.org**. These packages are the work of statisticians around the world (as is R itself) and are referred to as “Contributed” packages.
If we are not sure which of our packages might contain the functions that we need, we can type, for example

```
??logistic
```

and the output

```
Help files with alias or concept or title matching logistic using
fuzzy matching:
MASS::polr Ordered Logistic or Probit Regression
nnet::multinom Fit Multinomial Log-linear Models
stats::glm Fitting Generalized Linear Models
stats::Logistic The Logistic Distribution
stats::SSfpl Self-Starting Nls Four-Parameter Logistic Model
stats::SSlogis Self-Starting Nls Logistic Model
survival::clogit Conditional logistic regression
Type '?PKG::FOO' to inspect entries 'PKG::FOO', or 'TYPE?PKG::FOO' for
entries like 'PKG::FOO-TYPE'.
```

tells us the name of the package, the function within the package, and a brief description of what the function does..

1.1.1 Libraries (Packages)

Once we have found which library we need, we must load it before we can use it. For example, if we wish to do Principal Component Analysis, we would find it in the `stats` package. In order to use it we type

```
library(stats)
```

and we can use any function in that library.

If we know the name of the function that we wish to use (and the library is loaded) we can type

```
?prcomp
```

This will bring up a window with a description of the function, its usage, its arguments, return values, and (usually) an example (or examples). Cutting and pasting these examples gives you an opportunity to explore the behaviour of the function. The documentation may also give references for the concepts behind the function and point to other functions that are related to it. Starting with version 2.4, the **help** window is a mini-browser for the entire package rather than just a text page for the requested function.

1.1.2 Assignments, sequences

To get a start on using R, we can look at some simple examples.

If you wish to assign a number (or more complicated object) to a variable the usual method is to use `<-` (although in most contexts it is also possible to use `=`) as in

```
a <- 5
```

R gives no response, but if we type

```
a
[1] 5
```

shows that `a` has the been assigned the value 5.

A simpler method is to enclose the expression in parentheses

```
(a <- 5)
[1] 5
```

We can assign a vector to a variable. (In fact there are different way to do this depending on the nature of the vector.)

```
(b <- c(1, 3, 2, 6, 5, 3, 2))
[1] 1 3 2 6 5 3 2
```

In this example, `c` concatenates the set of comma-delimited numbers into a vector.

In the following, the vector is created by repeating a number or set of numbers

```
(c <- rep(5, 7))
[1] 5 5 5 5 5 5 5
(c.1 <- rep(c(1,3,2),4))
[1] 1 3 2 1 3 2 1 3 2 1 3 2
```

The above shows a way of creating variable names with the use of the `.`.

We can create vectors by sequencing operations

```
(d <- 1:6)
[1] 1 2 3 4 5 6

(e <- 6:1)
[1] 6 5 4 3 2 1

(f <- 1:10/10)
[1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

(g <- seq(10, 9, -.2))
[1] 10.0 9.8 9.6 9.4 9.2 9.0
```

We can perform operations on these vectors

```
a - b
[1] 4 2 3 -1 0 2 3
a*b
[1] 5 15 10 30 25 15 10
```

(No parentheses are needed because there is no assignment.)

1.1.3 Matrices

```
a%*%b
Error in a %*% b : non-conformable arguments
```

The `%*%` represents matrix multiplication and the above tried to multiply a 1×1 and a 7×1 vector together.

We can use the `t` operator (transpose) to give a 1×1 and a 1×7.

```
a%*%t(b)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    5   15   10   30   25   15   10

sum(a%*%t(b))
[1] 110
```

We can also assign a matrix to a variable.

Simple matrices could be created as

```
(m.1 <- matrix(0, 3, 2))
     [,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    0    0

(m.2 <- matrix(1:12, nrow=3))
     [,1] [,2] [,3] [,4]
```

```
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

as well as some special matrices such as

```
(I.3 <- diag(1,3))
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

We could create a matrix from the vector b with

```
(h <- matrix(b, nrow=1))
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    3    2    6    5    3    2
```

We can multiply matrices in different ways

```
h*h
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    9    4   36   25    9    4
```

(the same as b*b).

```
h%*%t(h)
[1,]   88
```

```
(h.m <- t(h)%*%h)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    3    2    6    5    3    2
[2,]    3    9    6   18   15    9    6
[3,]    2    6    4   12   10    6    4
[4,]    6   18   12   36   30   18   12
[5,]    5   15   10   30   25   15   10
[6,]    3    9    6   18   15    9    6
[7,]    2    6    4   12   10    6    4
```

To access an entry (or submatrix)

```
h.m[5, 2]
[1] 15
```

```
h.m[1:3, 5:4]      # Note the reverse order of the columns
     [,1] [,2]
[1,]    5    6
[2,]   15   18
[3,]   10   12
```

In the above input, the # indicates that the remainder of the line is a comment.

```
h.m[c(3,5,1), c(6,2,7)]
     [,1] [,2] [,3]
[1,]    6    6    4
[2,]   15   15   10
[3,]    3    3    2
```

It is also possible to change the values

```
(h.m[c(3,5,1), c(6,2,7)] <- -10)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1  -10    2    6    5  -10  -10
```

```
[2,]    3    9    6   18   15    9    6
[3,]    2  -10    4   12   10  -10  -10
[4,]    6   18   12   36   30   18   12
[5,]    5  -10   10   30   25  -10  -10
[6,]    3    9    6   18   15    9    6
[7,]    2    6    4   12   10    6    4
```

and determine some values -

```
sum(h.m)
[1] 330

mean(h.m)
[1] 6.734694

apply(h.m, 1, sum)       # Row sums
[1] -16  66  -2 132  40  66  44
apply(h.m, 1, mean)      # Row means
[1] -2.2857143  9.4285714 -0.2857143 18.8571429  5.7142857  9.4285714  6.2857143

apply(h.m, 2, sum)       # Column sums
[1]  22  12  44 132 110  12  -2
apply(h.m, 2, mean)      # Column means
[1]  3.1428571  1.7142857  6.2857143 18.8571429 15.7142857  1.7142857 -0.2857143
```

etc.

The elements of a matrix are not restricted to numerical values. For example,

```
matrix(letters[1:6], ncol=3)
     [,1] [,2] [,3]
[1,] "a"  "c"  "e"
[2,] "b"  "d"  "f"
```

Suppose we have a system of equations $A\mathbf{x} = \mathbf{b}$ with

```
(A <- matrix(c(3, 2, 5, 4, 1, 9, -1, 6, 8), 3, 3))
     [,1] [,2] [,3]
[1,]    3    4   -1
[2,]    2    1    6
[3,]    5    9    8
```

and

```
(bT <- c(-1, 3, 2))
[1] -1  3  2
```

we can create the **augmented matrix** by “binding” `bT` to `A`

```
cbind(A, bT)
            bT
[1,] 3 4 -1 -1
[2,] 2 1  6  3
[3,] 5 9  8  2
```

(We could also `rbind`.)

The entries in the matrix can be of different types

```
(Mixed <- matrix(c("Height", "Width", 25, 30), 2, 2))
```

```
     [,1]     [,2]
[1,] "Height" "25"
[2,] "Width"  "30"
```

but the entries are all made the same type - in this case strings.

If we try to multiply the entries in the second column

```
Mixed[1,2]*Mixed[2,2]
Error in Mixed[1, 2] * Mixed[2, 2] : non-numeric argument to binary operator
```

It is possible to convert a string to a number with

```
as.numeric(Mixed[1,2])*as.numeric(Mixed[2,2])
[1] 750
```

1.1.4 Lists

A matrix can only be used for *rectangular* arrays. The `list` is a data structure that is more flexible.

We can create a simple list

```
(L.1 <- list(first.name = "John", last.name = "Smith", sn = 345678, mark = "A-"))
$first.name
[1] "John"
$last.name
[1] "Smith"
$sn
[1] 345678
$mark
[1] "A-"
```

We can refer to the components of the list by

```
L.1[2]
$last.name
[1] "Smith"

L.1[[2]]
[1] "Smith"
```

(Notice that the first form gives the name of the component.)

```
L.1$last.name
[1] "Smith"
```

It is better to refer to the components by name because that means that if the order is changed to `list(last.name="Smith", first.name="John", sn=345678, mark="A-")` that we still get the correct values.

Many functions have return values in the form of lists.

We can also build a list by appending components. This is often useful in situations in which we are iterating

```
L.2 <- {}
L.2 <- c(L.2, list(1))
```

```
L.2 <- c(L.2, list("x"))
L.2 <- c(L.2, list("x^2/2!"))
(L.2 <- c(L.2, list("x^3/3!")))
[[1]]
[1] 1
[[2]]
[1] "x"
[[3]]
[1] "x^2/2!"
[[4]]
[1] "x^3/3!"
```

It is sometimes useful to remove things from the list structure and that can be done by

```
unlist(L.2)
[1] "1"      "x"      "x^2/2!" "x^3/3!"
```

1.1.5 paste

In the above, we use the `c` to concatenate a set of numbers into a vector. If we wish to concatenate strings (numbers get converted to strings), we use `paste` as in

```
paste("John", "Smith", 345678)
[1] "John Smith 345678"
```

Notice that there is a space between the names. We can change that with

```
(str.1 <- paste("John", "Smith", 345678, sep=","))
[1] "John,Smith,345678"

(str.2 <- paste("Jane", "Jones", 234567, sep=","))
[1] "Jane,Jones,234567"
```

In other words, `sep=","` controls what separates the quantities that are being pasted together (it defaults to a space but can be more than a single character). If we do not want the space, we can remove it with the `sep = ""`. On the other hand we may wish to insert some other character

```
(str.3 <- paste("D:", "DATA", "Data Mining R-code",sep="/"))
[1] "D:/DATA/Data Mining R-code"
```

If we have a vector of strings, `paste` does nothing unless we tell it to collapse the vector (and what to put between the elements).

```
(str.4 <- paste(unlist(L.2), collapse=" + "))
[1] "1 + x + x^2/2! + x^3/3!"
```

1.1.6 stringsplit

There may be times when we need to “unpaste” a string. We do this with

```
strsplit(str.1, ",")
[[1]]
[1] "John"   "Smith"  "345678"
```

which produces a list of one element, or

```
strsplit(rbind(str.1, str.2), ",")
[[1]]
[1] "John"    "Smith"   "345678"
[[2]]
[1] "Jane"    "Jones"   "234567"
```

which produces a list element for each row of the matrix.

If we try the same thing on

```
strsplit(str.4, " + ")
[[1]]
[1] "1 + x + x^2/2! + x^3/3!"
```

nothing happens. The reason is that "+" is a metacharacter in regular expressions - along with "`\ | ( ) [ { ^$ * ?`" and we need to change it to an ordinary character with

```
strsplit(str.4, " \\+ ")
[[1]]
[1] "1"      "x"      "x^2/2!" "x^3/3!"
```

1.1.7 Control Structures

R has the usual control structures.

If we let

```
a <- 1
b <- 2
if (a < b) print("a < b")
[1] "a < b"
if (a == b)
   print("a < b")
else
Error: syntax error in "else"
   print("a >= b")
```

tells us that there is a syntax error in "`else`".

The correct form is

```
if (a == b) {
   print("a < b")
} else {
   print("a >= b")
}
[1] "a >= b"
```

Note that the { and } are used as the beginning and ending of blocks of code.

```
for (i in 1:5) {
   print (i)
}
[1] 1
[1] 2
[1] 3
[1] 4
```

```
[1] 5
```

Instead of `1:5` we could have things such as `i in c(5, 3, 7, 2, -4, -9)`.

```
n <- 1
f <- 1
while (n < 5) {
   f <- f * n
   n <- n + 1
   print(f)
}
[1] 1
[1] 2
[1] 6
[1] 24
```

R also has `repeat`, `break`, and `next`.

1.1.8 Functions

In the preceding, we have used several of the built-in functions of R (`paste`, `strsplit`, `print`). We will often need to write our own functions, so we need to look at the structure of functions.

Suppose that we have a random set of numbers and wish to find the mean (there is, of course, a built-in function for this)

```
(num <- runif(20, -1, 1)) # 20 random numbers from a uniform distribution
[1] -0.439628823  0.233594691 -0.266024740 0.007379845 0.466801666
[6] -0.092201637  0.683620457 -0.003777758  0.040208154 0.276043486
[11] -0.236744290  0.258244825  0.262341596 -0.518254284 -0.584123774
[16] -0.300963114  0.129197590  0.941738119  0.073712909 -0.991614198
```

We could write the function as

```
my.mean <- function (x) {
   len <- length(x)
   sum(x)/len
}
```

Note that this says that the name `my.mean` has the `function` assigned to it. We can see what the function is if we enter the name

```
my.mean
function (x) {
   len <- length(x)
   sum(x)/len
}
```

We call the function in the usual manner

```
my.mean(num)
```

```
[1] -0.003022464
```

A type of function that we will often need is the **recursive** function. A common example that is used in recursion is the *factorial* function (it is NOT the best way to find the factorial, but it is easy to program and to understand.)

If you are not familiar with recursion in programming, the following might help.

```
fact <- function (m) {
   if (m > 1) {
      f <- fact(m - 1) * m
   } else {
      return(1)
   }
   f
}
```

1.1.9 Debugging

We will use the

```
debug(fact)
```

to enable us to trace through the factorial function.

(We can display the values of the variables within the function by typing the name of the variable. If we have a variable called `n` we need to type `print(n)` - this is why I use `m` as the variable. If you have seen enough, you can use `c` to continue without debugging or `Q` to quit.)

The text in `( )` are comments on the process.

```
fact(5)
debugging in: fact(5)          (tells us that we have just entered fact)
debug: {
    if (m > 1) {               (the next block of the function to be evaluated)
       f <- fact(m - 1) * m
    }
    else {
        return(1)
    }
    f
}
Browse[1]> m;f;n                (a command to print the value of m & f and take the next
[1] 5     (m)
[1] 0     (f)
debug: if (m > 1) {             (next block - we stepped past the {)
    f <- fact(m - 1) * m
} else {
    return(1)
}
Browse[1]> m;f;n
[1] 5
[1] 0
debug: f <- fact(m - 1) * m    (m > 1 so we execute this)
Browse[1]> m;f;n
[1] 5
[1] 0
debugging in: fact(m - 1)       (we have stepped into fact again - with m = 4 - see below
debug: {
    if (m > 1) {
        f <- fact(m - 1) * m
    }
    else {
      return(1)
    }
    f
}
Browse[1]> m;f;n
[1] 4     (m = 4)
[1] 0
debug: if (m > 1) {
    f <- fact(m - 1) * m
} else {
    return(1)
}
Browse[1]> m;f;n
[1] 4
[1] 0
debug: f <- fact(m - 1) * m    (m > 1 so we execute this)
Browse[1]> m;f;n
[1] 4
[1] 0
debugging in: fact(m - 1)      (we have stepped into fact again - with m = 3 - see below
debug: {
    if (m > 1) {
        f <- fact(m - 1) * m
    }
    else {
        return(1)
    }
    f
}
Browse[1]> m;f;n
[1] 3
[1] 0
debug: if (m > 1) {
    f <- fact(m - 1) * m
} else {
    return(1)
}
Browse[1]> m;f;n
[1] 3
[1] 0
debug: f <- fact(m - 1) * m
Browse[1]> m;f;n
[1] 3
[1] 0
debugging in: fact(m - 1)    (we have stepped into fact again - with m = 2 - see below)
debug: {
    if (m > 1) {
        f <- fact(m - 1) * m
    }
    else {
        return(1)
    }
    f
}
Browse[1]> m;f;n
[1] 2
[1] 0
debug: if (m > 1) {
    f <- fact(m - 1) * m
} else {
    return(1)
}
Browse[1]> m;f;n
[1] 2
[1] 0
debug: f <- fact(m - 1) * m
Browse[1]> m;f;n
[1] 2
[1] 0
debugging in: fact(m - 1)    (we have stepped into fact again - with m = 1 - see below)
debug: {
    if (m > 1) {
        f <- fact(m - 1) * m
    }
    else {
        return(1)
    }
    f
}
Browse[1]> m;f;n
[1] 1
[1] 0
debug: if (m > 1) {
    f <- fact(m - 1) * m
} else {
    return(1)
}
Browse[1]> m;f;n
[1] 1
[1] 0
debug: return(1)             (this time m = 1 so we do not take the fact path)
Browse[1]> n
exiting from: fact(m - 1)    (this is the first time that we have done this
                              we return a 1 to the function that called this
                              and use that value as the multiplier of m = 2)
debug: f
Browse[1]> m;f;n
[1] 2
[1] 2
exiting from: fact(m - 1)    (return from fact with the value 2 = 2·1
                              and use this as the multiplier of m = 3)
debug: f
Browse[1]> m;f;n
[1] 3
[1] 6
exiting from: fact(m - 1)    (return from fact with the value 6 = 3·2·1
                              and use this as the multiplier of m = 4)
debug: f
Browse[1]> m;f;n
[1] 4
[1] 24
exiting from: fact(m - 1)    (return from fact with the value 24 = 4·3·2·1
                              and use this as the multiplier of m = 5)
debug: f
Browse[1]> m;f;n
[1] 5
[1] 120
exiting from: fact(5)       (return from fact with the value 120 = 5·4·3·2·1)
```

```
[1] 120
```

If you have a long complicated function with a small section that you wish to investigate, it is possible to insert the command `browser()` into the code. In this case, the use of `c`ontinue will allow the code to be executed until you reach the `browser()` command again.

This gives a brief look at some of the concepts in R. We will look at others as we need them.