Before you start

1) Remember that everything is CASE SENSITIVE in R. This means that caps make a difference.

2) Also, functions in R mostly work ELEMENTWISE by default! This means that a lot of operations can be accomplished with a single line of code.

3) Several functions can be used in the same line of code (i.e. output of one function is used as (perhaps partial) input for another) and if desired, lines of code can be "joined" by putting a semicolon between them.

4) It is not necessary to leave any spaces anywhere, this is just for display purposes.

5) Character # can be used for commenting the code (i.e. everything following this symbol on a line is ignored by R). Make sure that you do COMMENT YOUR CODE. It is invaluable when you are returning to use your code later on.

Getting Started

`variable <- value` setting a *value* for *variable*, often **=** is used instead

```
a <- 3
b = 9
```

`?function` documentation of a function

```
?sum
```

`apropos("string")` lists all the functions that contain *string* in its name

```
apropos("sum")
```

`install.packages("packagename")` installs the package named *packagename* into the users computer

```
install.packages("gsl")
```

`library("packagename")` loads that package

`library(help="packagename")` opens an overview of the package and the functions in it

Basic Operators

`+ - * /` addition, substraction, multiplication, division,

`** ^` raising to a power (two variants),

`%/%` **`%%`** integer division, remainder

```
a * b #27
b ** (1 / 2) #3
b ** 1 / 2 #4.5
a %% b #3
```

Vectors

`c(...)` combines arguments into a vector

```
c(1, 2) #1 2
c(2 * c(a, b), a) #6 18 3
```

`from : to` produces a vector of integers with increment (plus or minus) one

```
a : b  #3 4 5 6 7 8 9
a : -1  #3 2 1 0 -1
1 : 2 + 3  #4 5
```

`seq(from, to, by)` identical but *by* specifies the increment; instead of *by* argument *length.out* can be used to specify the desired length of the sequence

```
seq(a, -1, -2) #3 1 -1
seq(a, -1, -3) #3 0
seq(a, -1, length.out=5) #3  2  1  0 -1
```

`rep(x, times)` replicate argument *x times* times; *each* can be used to replicate each element of *x each* times

```
rep(c(2, 5), times=2) #2 5 2 5
```

rep(c(2, 5), each=4) #2 2 2 2 5 5 5 5

Math & Stat

length(x) number of elements in argument *x*
abs(x) absolute value of elements of argument *x*
max(...) maximal element of all elements in the arguments
    max(a, 6) #6
    max(c(a, b), 6) #9
min(...) minimal element of all elements in the arguments
sum(...) sum of all elements in the arguments
prod(...) product of all elements in the arguments
log(x) natural logarithm of elements of argument *x*; argument *base* can be used to set a different base
    log*(9, base=3) #2*
exp(x) exponent of elements of argument *x*
mean(x) arithmetic mean of elements of argument *x*
    mean*(c(a, b)) #6*
sd(x) standard deviation of elements of argument *x*
    var*(c(a, b)) #18*
var(x) variance of argument *x*
cor(x, y) correlation between vectors *x* and *y*
cov(x, y) covariance between vectors *x* and *y*
round(x, digits) round the elements of *x* to the number of decimal places specified by *digits*
pmax(...) positionwise maxima
    pmax(c(4, 5), c(a, b)) #4 9
pmin(...) positionwise minima
cumsum(x) cumulative sum of the elements of argument *x*
    cumsum(c(a, b, a)) #4 13 17
cumprod(x) cumulative product of the elements of argument *x*
cummax(x) cumulative maximum of the elements of argument *x*
    cummax(c(a, b, a)) #4 9 9
cummin(x) cumulative minimum of the elements of argument *x*

Matrices

matrix(x, nrow, ncol) creating a matrix with *nrow* rows and *ncol* columns from elements of *x* by first filling the first column (top to bottom), then the second, etc
    matrix(c(2, 3), nrow=2, ncol=2) #2 2
    #3 3
diag(x) forms a diagonal matrix with elements of *x* on the main diagonal; argument *nrow* can still be used
%*% binary operator for matrix multiplication
t(x) transposes argument *x*
solve(x) inverse of the square matrix *x*
dim(x) dimensions of argument *x* i.e. number of rows and columns for a matrix*;* returns *NULL* for vectors
    dim(diag(2, nrow=2)) #2 2
rowSums(x) sum the elements of *x* by rows
colSums(x) sum the elements of *x* by columns
cbind(...) combine arguments side-by-side
    cbind(c(2, 3), c(4, 5), c(a, b)) #2 4 3
    #3 5 9
rbind(...) combine arguments top to bottom
    rbind(c(2, 3), c(4, 5)) #2 3
    #4 5
apply(X,MARGIN,FUN) apply function *FUN* to the object *X* by rows if *MARGIN=1* or by columns if

MARGIN=2

Logicals

`== >= > < <= != %in%` binary operators equal to, greater than or equal to, greater than, less than, less than or equal to, not equal to, and is contained in producing logical objects consisting of *TRUE* and/or *FALSE*

```
a > b #FALSE
a + 6 == b #TRUE
a != b #TRUE
c(a, 2) %in% c(2, b, 1) #FALSE TRUE
```

`!` unary operator for negating the logical object

```
!c(FALSE, TRUE) #TRUE FALSE
```

`& |` binary operators AND and OR for combining logical objects

```
a == 3 | b == 3 #TRUE
```

Probability distributions

`rnorm(n)` random generation of *n* numbers from a standard normal (Gaussian) distribution; arguments *mean* and *sd* can be used to specify distribution parameters

```
set.seed(16)
rnorm(2) #1.147829 -0.468412
rnorm(2, mean=3) #4.096216 1.555771
```

`dnorm(x)` value of the probability density function (pdf) of a standard normal distribution at elements of *x*; arguments *mean* and *sd* can be used to specify distribution parameters

`pnorm(q)` value of the cumulative distribution function (cdf) of a standard normal distribution at elements of *q*; arguments *mean* and *sd* can be used to specify distribution parameters

```
pnorm(0) #0.5
```

`qnorm(p)` value of the quantile function i.e. inverse cdf of a standard normal distribution at elements of *q*; arguments *mean* and *sd* can be used to specify distribution parameters

```
qnorm(0.95) #1.644854
```

Similar functions exist for many other distributions e.g. **runif** generates uniformly distributed random numbers (from the unit interval) and **pexp** is the cdf of exponentially distributed random numbers; arguments for specifying the parameters have different names

Data extraction

```
x = 2 : 10
y = 9 : 1
z = c(5, 2, 3)
```

`y[3] #7`	element at a specific position
`z[-2] #5 3`	all elements except one at a specific position
`y[1 : 3] #9 8 7`	elements at specific positions
`y[-(1 : 7)] #2 1`	all elements except those at specific positions
`x[c(2, 4, 6)] #3 5 7`	elements at specific positions
`z[c(TRUE, FALSE, TRUE)] #5 3`	elements at positions *TRUE*
`x[y > 4] #2 3 4 5 6`	elements at positions *TRUE*
`x[x > 3 & x < 5] #4`	elements at positions *TRUE*

```
m = rbind(x, y)
```

`m[2, 3] #7`	element at a specific position
`m[1, ] #2 3 4 5 6 7 8 9 10`	specified row
`m[ , 2] #3 8`	specified column
`m[-1, 1] #9`	specified sub-matrix

`m[1, c(1, 3)] #2 4` specified sub-matrix

Plotting

`plot(x, y)` plot the points coordinates of which are defined by the vectors *x* and *y* elementwise; argument *type* can be used to change the plotting style e.g. *"p"* for points, *"l"* for lines, *"o"* for overplotted points and lines

```
plot(x, y, type="l")
```

`matplot(x, y)` plot the series of points coordinates of which are defined by the matrices *x* and *y* elementwise with series in columns; if only one matrix is given then this is assumed to be *y*; *type* can be used as before

```
matplot(m, type="l")
x1 = (1 : 200) / 100; y1 = matrix(0, length(x1), 2)
y1[ , 1] = cos(x); y1[ , 2] = sin(x); matplot(x1, y1, type="l")
```

Programming

function definition; functions are usually named and not used "on the spot"; giving default values to arguments allows execution of a function without specifying values for all the arguments; values can be arbitrary (e.g. function names); when the function body consists of a single expression then the curly braces can be omitted

```
mltpl_tbl = function(x=1:9, y=1:9){x %*% t(y)}
mltpl_tbl(1:2, 1:2)    #1 2
                       #2 4
univ_func = function(obj, f) f(obj)
```

`if(condition){expressions}` *expressions* are executed only if *condition* is *TRUE*

```
if(runif(1) > runif(1)) print("first was bigger")
```

`if(condition){expressions1}else{expressions2}` if *condition* is TRUE then *expressions1* are executed, if not then *expressions2*

```
if(runif(1) > runif(1)){
      print("first was bigger")
      }else{
      print("second was bigger")
}
```

`for(variable in sequence){expressions}` a cycle where a *variable* takes the value of the first element of the *sequence* and *expressions* are then executed, then the *variable* takes the value of the second element of the *sequence* and *expressions* are again executed; this continues until the *sequence* is exhausted or *expressions* cause the cycle to end prematurely

```
N = 1000
n = 500
meanv = rep(0, N)
for(i in 1 : N){
      meanv[i] = mean(runif(n))
}
var(meanv) #should be approximately 1/(12*n)
```

`while(condition){expressions}` a cycle where *expressions* are executed if the *condition* is *TRUE* initially; then the *condition* is re-checked and if it is still true then *expressions* are executed again; this continues until the *condition* is *FALSE* or *expressions* cause the cycle to end prematurely

```
init = 1:10
summand = init
while(max(summand) > 0.00001){
      summand = summand / 2
      init=init+summand
}
```