BIO 754 - Lecture 06

30-03-2017

Contents

Saving and writing files

We will continue working with the liver transcriptome dataset. To avoid repeating the same steps each time, we will learn how to save specific objects in an .Rdata file.

First let's recreate the objects:

```
# setwd("~/Documents/misc/metu/ders/2380754_comp_2017")
mat = read.table(
  "./GSE17274_ReadCountPerLane.txt", row.names=1, head=T, sep="\t" )
mat = as.matrix(mat)
species = factor( substr ( colnames(mat), 6, 7 ) )
sex = factor( substr(colnames(mat), 8, 8) )
run = factor( substr(colnames(mat), 2, 2) )
indv = factor( substr(colnames(mat), 6, 9) )
totalcounts = colSums(mat)
```

`save` and `load`

Now save these objects in a file called `liver_transcriptome_v1.Rdata`:

```
save(mat, species, sex, run, indv, totalcounts, file="liver_transcriptome_v1.Rdata")
```

It will be saved in the current working directory. Check:

```
list.files()
```

```
##  [1] "BIO754 Student list.docx"         "BIOL 754 Section 1 Grades.ods"
##  [3] "Empirical_Rule.png"               "GSE17274_ReadCountPerLane.txt"
##  [5] "GSE17274_ReadCountPerLane.txt.gz" "HW5.pdf"
##  [7] "all_objects.Rdata"                "attendance BIOL 754.ods"
##  [9] "homework_week_01.Rmd"             "homework_week_01.html"
## [11] "homework_week_02.Rmd"             "homework_week_02.html"
## [13] "homework_week_02_solution.Rmd"    "homework_week_02_solution.pdf"
## [15] "homework_week_03.Rmd"             "homework_week_03.html"
## [17] "homework_week_03_answers.Rmd"     "homework_week_03_answers.html"
## [19] "homework_week_03_answers.pdf"     "homework_week_04.Rmd"
## [21] "homework_week_04.pdf"             "homework_week_05.Rmd"
## [23] "homework_week_05.pdf"             "lecture_week_01.Rmd"
## [25] "lecture_week_01.pdf"              "lecture_week_02.Rmd"
## [27] "lecture_week_02.pdf"              "lecture_week_03.Rmd"
## [29] "lecture_week_03.pdf"              "lecture_week_04.Rmd"
## [31] "lecture_week_04.pdf"              "lecture_week_05.Rmd"
## [33] "lecture_week_05.pdf"              "lecture_week_06.Rmd"
## [35] "lecture_week_06.pdf"              "lecture_week_07.Rmd"
## [37] "lecturenotes"                     "liver_transcriptome_v1.Rdata"
## [39] "mat.txt"                          "notes"
## [41] "random.txt"                       "stat test.docx"
## [43] "stat test.pdf"                    "syllabus_2380754_2017.docx"
## [45] "syllabus_2380754_2017.pdf"
```

To save all objects in the current environment, you can also say:

```
save(list=ls(), file="all_objects.Rdata")
```

To load the objects in an .Rdata file in the environment, you use `load`. To see the effect, first let's delete the objects:

```
rm(list=ls())  # delete all objects
ls()
```

```
## character(0)
```

```
load("liver_transcriptome_v1.Rdata")
ls()
```

```
## [1] "indv"        "mat"         "run"         "sex"         "species"
## [6] "totalcounts"
```

write.table

We can also save matrices (or vectors) by writing them as into text files. Do this for `mat`:

```
write.table(mat, file="mat.txt",
            col.names=T, row.names=F)
```

Check the `mat.txt` file in your working directory (remember: `getwd`). The row names should be missing.

Now try:

```
write.table(mat, file="mat.txt", col.names=T, row.names=T,
            quote=F, sep="\t", append=F)
# quote = F suppresses quotation marks around text strings
# sep defines the column separators, which is space by default
# append = F overwrites any existing file with the same name, otherwise new text appended
```

Appending output to files

The `append=T` argument can be useful for adding data into the same file. This will be particularly useful in situations like this: you are running a long loop, and each step of the loop creates a bulky object (e.g. a very long vector). If you store the results in working memory this will slow R down as the simulation proceeds. To avoid this you may instead save the results of each step in a file on the hard drive, and overwrite the object in working memory. You can analyse the results once the loop is finished.

Let's run a toy example:

```
for (i in 1:5) {
  random = sample(5)
  print(random)
  write.table(random, file="random.txt", sep="\t", append=T, col.names = F, row.names = F)
}
```

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

```
read.table("random.txt")
```

```
##    V1
## 1   1
## 2   5
## 3   2
## 4   3
## 5   4
## 6   3
## 7   2
## 8   4
## 9   1
## 10  5
## 11  4
## 12  5
## 13  2
## 14  3
## 15  1
## 16  5
## 17  1
## 18  2
## 19  3
## 20  4
## 21  3
```

```
## 22  1
## 23  5
## 24  2
## 25  4
```

Data are written along the columns in this case. To write the data in rows, you can run the following. But first, delete the existing file using `unlink`:

```
unlink("random.txt")
```

Now create and fill in a new file with the same name:

```
for (i in 1:5) {
  random = sample(5)
  print(random)
  write.table(t(random), file="random.txt", sep="\t", append=T, col.names = F, row.names = F)
}
```

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

```
read.table("random.txt")
```

```
##   V1 V2 V3 V4 V5
## 1  2  1  3  4  5
## 2  2  4  3  5  1
## 3  1  5  2  4  3
## 4  5  4  3  2  1
## 5  2  1  4  5  3
```

How could you add column names to this file, from `a` to `e`?

```
# delete the existing file again
unlink("random.txt")
#
x = letters[1:5]
x
```

```
## [1] "a" "b" "c" "d" "e"
```

```
# write this as the first line
write.table(t(x), file="random.txt",
            col.names = F, row.names = F, sep="\t", quote=F)
# now fill in the numbers
for (i in 1:5) {
  random = sample(5)
  print(random)
  write.table(t(random), file="random.txt", sep="\t", append=T, col.names = F, row.names = F)
}
```

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

```
read.table("random.txt", head=T)
```

```
##   a b c d e
## 1 2 5 4 3 1
## 2 5 1 4 3 2
## 3 4 5 3 2 1
## 4 4 1 3 5 2
## 5 5 1 4 2 3
```

What if you had a text file table with non-uniform columns? E.g. we can create one by adding a longer row in the file "random.txt":

```
write.table(1:7, file="random.txt", sep="\t", append=T, col.names = F, row.names = F)
```

Now try to read, and `read.table` will complain:

```
read.table("random.txt", head=T)
```

```
## Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, : line 6 did not have 5
```

One way to overcome this is to use the `fill` argument:

```
read.table("random.txt", head=T, fill = T)
```

```
##    a  b  c  d  e
## 1  2  5  4  3  1
## 2  5  1  4  3  2
## 3  4  5  3  2  1
## 4  4  1  3  5  2
## 5  5  1  4  2  3
## 6  1 NA NA NA NA
## 7  2 NA NA NA NA
## 8  3 NA NA NA NA
## 9  4 NA NA NA NA
## 10 5 NA NA NA NA
## 11 6 NA NA NA NA
## 12 7 NA NA NA NA
```

Another useful trick about reading tables is `skip`. `quote = F` is also useful - it suppressed `write.tables` search for quotation marks in the table (which can cause trouble when you have text with quotation marks).

```
newmat = read.table(file="mat.txt", head=T, row.names=1, sep="\t", skip = 1000, quote = "")
dim(mat)
```

```
## [1] 20689    36
```

```
dim(newmat)
```

```
## [1] 19689    36
```

scan

If the data is too large, you can also use `scan` instead. But reading in data in this format will require quite a lot of processing. Let us do this:

```
newmat = scan("GSE17274_ReadCountPerLane.txt.gz",
     what="complex")
length(newmat)
```

```
## [1] 765530
```

You can compare how fast it worked relative to `read.table`, which can be considerably different when you have tables with rows on the order of millions - not unusual for e.g. genomic datasets. To compare the time, you can use the `system.time` command:

```
sapply(1:5, function(i) system.time(scan("mat.txt", what="integer")) )
```

```
##             [,1]  [,2]  [,3]  [,4]  [,5]
## user.self  0.332 0.333 0.288 0.291 0.316
## sys.self   0.013 0.008 0.006 0.005 0.006
## elapsed    0.355 0.350 0.298 0.298 0.329
## user.child 0.000 0.000 0.000 0.000 0.000
## sys.child  0.000 0.000 0.000 0.000 0.000
```

```
sapply(1:5, function(i) system.time(read.table(
  "./GSE17274_ReadCountPerLane.txt", row.names=1, head=T, sep="\t" ) ) )
```

```
##             [,1]  [,2]  [,3]  [,4]  [,5]
## user.self  0.361 0.337 0.353 0.349 0.455
## sys.self   0.016 0.012 0.010 0.013 0.012
## elapsed    0.386 0.356 0.370 0.373 0.481
## user.child 0.000 0.000 0.000 0.000 0.000
## sys.child  0.000 0.000 0.000 0.000 0.000
```

Exercise: converting a scan output into a matrix

But now we would have the task of reformatting this into a table. Let's do this as exercise:

```
dim(newmat)
```

```
## NULL
```

```
length(newmat)
```

```
## [1] 765530
```

```
head(newmat, 100)
```

```
##   [1] "EnsemblGeneID"   "R1L1.HSM1"       "R1L2.PTF1"
##   [4] "R1L3.RMM1"       "R1L4.HSF1"       "R1L6.PTM1"
##   [7] "R1L7.RMF1"       "R2L2.RMF2"       "R2L3.HSM2"
##  [10] "R2L4.PTF2"       "R2L6.RMM2"       "R2L7.HSF2"
##  [13] "R2L8.PTM2"       "R3L1.RMM3"       "R3L2.HSF2"
##  [16] "R3L3.PTM1"       "R3L4.RMF3"       "R3L6.HSM3"
##  [19] "R3L7.PTF3"       "R3L8.RMM1"       "R4L1.HSM3"
##  [22] "R4L2.HSF1"       "R4L3.RMM3"       "R4L4.PTF1"
##  [25] "R4L6.PTM2"       "R4L7.RMF3"       "R4L8.HSM2"
##  [28] "R5L1.RMF1"       "R5L2.HSM1"       "R5L3.PTF3"
##  [31] "R5L4.RMM2"       "R5L8.RMF2"       "R6L2.PTM3"
##  [34] "R6L4.PTM3"       "R6L6.PTF2"       "R8L1.HSF3"
##  [37] "R8L2.HSF3"       "ENSG00000000003" "60"
##  [40] "285"             "207"             "172"
##  [43] "176"             "259"             "299"
##  [46] "219"             "213"             "676"
##  [49] "147"             "316"             "338"
##  [52] "153"             "233"             "242"
##  [55] "199"             "180"             "217"
##  [58] "160"             "157"             "367"
##  [61] "289"             "369"             "238"
##  [64] "202"             "252"             "61"
##  [67] "206"             "672"             "165"
##  [70] "216"             "212"             "216"
##  [73] "78"              "90"              "ENSG00000000005"
##  [76] "0"               "1"               "1"
##  [79] "0"               "0"               "0"
##  [82] "1"               "1"               "1"
##  [85] "0"               "0"               "0"
##  [88] "0"               "0"               "0"
##  [91] "0"               "0"               "0"
##  [94] "1"               "0"               "0"
##  [97] "0"               "0"               "1"
## [100] "2"
```

```
# we can fit this in a table with 37 columns:
newmat2 = t(matrix(newmat, 37, ))
# this is a more elegant solution:
newmat2 = matrix(newmat, , 37, byrow = T)
head(newmat2)
```

```
##      [,1]              [,2]        [,3]        [,4]        [,5]
## [1,] "EnsemblGeneID"   "R1L1.HSM1" "R1L2.PTF1" "R1L3.RMM1" "R1L4.HSF1"
## [2,] "ENSG00000000003" "60"        "285"       "207"       "172"
## [3,] "ENSG00000000005" "0"         "1"         "1"         "0"
## [4,] "ENSG00000000419" "17"        "54"        "20"        "36"
## [5,] "ENSG00000000457" "50"        "61"        "68"        "41"
## [6,] "ENSG00000000460" "9"         "6"         "2"         "3"
##      [,6]        [,7]        [,8]        [,9]        [,10]
## [1,] "R1L6.PTM1" "R1L7.RMF1" "R2L2.RMF2" "R2L3.HSM2" "R2L4.PTF2"
## [2,] "176"       "259"       "299"       "219"       "213"
```

```
## [3,] "0"         "0"         "1"         "1"         "1"
## [4,] "23"        "38"        "40"        "42"        "35"
## [5,] "143"       "42"        "47"        "30"        "111"
## [6,] "3"         "2"         "1"         "2"         "5"
##      [,11]       [,12]       [,13]       [,14]       [,15]
## [1,] "R2L6.RMM2" "R2L7.HSF2" "R2L8.PTM2" "R3L1.RMM3" "R3L2.HSF2"
## [2,] "676"       "147"       "316"       "338"       "153"
## [3,] "0"         "0"         "0"         "0"         "0"
## [4,] "39"        "26"        "26"        "36"        "35"
## [5,] "55"        "28"        "110"       "76"        "34"
## [6,] "2"         "8"         "3"         "2"         "9"
##      [,16]       [,17]       [,18]       [,19]       [,20]
## [1,] "R3L3.PTM1" "R3L4.RMF3" "R3L6.HSM3" "R3L7.PTF3" "R3L8.RMM1"
## [2,] "233"       "242"       "199"       "180"       "217"
## [3,] "0"         "0"         "0"         "0"         "1"
## [4,] "36"        "22"        "33"        "35"        "29"
## [5,] "131"       "61"        "53"        "46"        "49"
## [6,] "9"         "5"         "11"        "2"         "5"
##      [,21]       [,22]       [,23]       [,24]       [,25]
## [1,] "R4L1.HSM3" "R4L2.HSF1" "R4L3.RMM3" "R4L4.PTF1" "R4L6.PTM2"
## [2,] "160"       "157"       "367"       "289"       "369"
## [3,] "0"         "0"         "0"         "0"         "1"
## [4,] "29"        "45"        "54"        "46"        "32"
## [5,] "42"        "50"        "87"        "70"        "129"
## [6,] "6"         "3"         "2"         "2"         "4"
##      [,26]       [,27]       [,28]       [,29]       [,30]
## [1,] "R4L7.RMF3" "R4L8.HSM2" "R5L1.RMF1" "R5L2.HSM1" "R5L3.PTF3"
## [2,] "238"       "202"       "252"       "61"        "206"
## [3,] "2"         "0"         "0"         "0"         "1"
## [4,] "35"        "36"        "29"        "22"        "30"
## [5,] "57"        "43"        "47"        "64"        "41"
## [6,] "5"         "5"         "2"         "6"         "3"
##      [,31]       [,32]       [,33]       [,34]       [,35]
## [1,] "R5L4.RMM2" "R5L8.RMF2" "R6L2.PTM3" "R6L4.PTM3" "R6L6.PTF2"
## [2,] "672"       "165"       "216"       "212"       "216"
## [3,] "0"         "0"         "1"         "0"         "3"
## [4,] "43"        "32"        "40"        "53"        "31"
## [5,] "59"        "22"        "71"        "78"        "118"
## [6,] "1"         "3"         "5"         "5"         "3"
##      [,36]       [,37]
## [1,] "R8L1.HSF3" "R8L2.HSF3"
## [2,] "78"        "90"
## [3,] "0"         "0"
## [4,] "16"        "40"
## [5,] "34"        "42"
## [6,] "7"         "5"
```

```
# column names, stored for later purposes
cn = newmat2[1, -1]
cn
```

```
##  [1] "R1L1.HSM1" "R1L2.PTF1" "R1L3.RMM1" "R1L4.HSF1" "R1L6.PTM1"
##  [6] "R1L7.RMF1" "R2L2.RMF2" "R2L3.HSM2" "R2L4.PTF2" "R2L6.RMM2"
## [11] "R2L7.HSF2" "R2L8.PTM2" "R3L1.RMM3" "R3L2.HSF2" "R3L3.PTM1"
```

```
## [16] "R3L4.RMF3" "R3L6.HSM3" "R3L7.PTF3" "R3L8.RMM1" "R4L1.HSM3"
## [21] "R4L2.HSF1" "R4L3.RMM3" "R4L4.PTF1" "R4L6.PTM2" "R4L7.RMF3"
## [26] "R4L8.HSM2" "R5L1.RMF1" "R5L2.HSM1" "R5L3.PTF3" "R5L4.RMM2"
## [31] "R5L8.RMF2" "R6L2.PTM3" "R6L4.PTM3" "R6L6.PTF2" "R8L1.HSF3"
## [36] "R8L2.HSF3"
```

```
# row names, stored for later purposes
rn = newmat2[-1, 1]
head(rn)
```

```
## [1] "ENSG00000000003" "ENSG00000000005" "ENSG00000000419" "ENSG00000000457"
## [5] "ENSG00000000460" "ENSG00000000938"
```

```
# remove the columns with row and column names
newmat3 = newmat2[-1, -1]
dim(newmat3)
```

```
## [1] 20689    36
```

```
# convert to integer - but this doesn't work - also converts into a vector
head(as.integer(newmat3))
```

```
## [1] 60  0 17 50  9 32
```

```
# this works
newmat4 = apply(newmat3, 2, as.integer)
dim(newmat4)
```

```
## [1] 20689    36
```

```
head(newmat4)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,]   60  285  207  172  176  259  299  219  213   676   147   316   338
## [2,]    0    1    1    0    0    0    1    1    1     0     0     0     0
## [3,]   17   54   20   36   23   38   40   42   35    39    26    26    36
## [4,]   50   61   68   41  143   42   47   30  111    55    28   110    76
## [5,]    9    6    2    3    3    2    1    2    5     2     8     3     2
## [6,]   32   50   44   23   99   46   18   26   53    20    30    16    29
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24]
## [1,]   153   233   242   199   180   217   160   157   367   289   369
## [2,]     0     0     0     0     0     1     0     0     0     0     1
## [3,]    35    36    22    33    35    29    29    45    54    46    32
## [4,]    34   131    61    53    46    49    42    50    87    70   129
## [5,]     9     9     5    11     2     5     6     3     2     2     4
## [6,]    35    99    28    32    39    36    33    21    43    34    15
##      [,25] [,26] [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35]
## [1,]   238   202   252    61   206   672   165   216   212   216    78
## [2,]     2     0     0     0     1     0     0     1     0     3     0
## [3,]    35    36    29    22    30    43    32    40    53    31    16
## [4,]    57    43    47    64    41    59    22    71    78   118    34
```

```
## [5,]     5     5     2     6     3     1     3     5     5     3     7
## [6,]    26    17    27    41    37    14    10    30    28    51   112
##      [,36]
## [1,]    90
## [2,]     0
## [3,]    40
## [4,]    42
## [5,]     5
## [6,]    98
```

```
# add back the names
colnames(newmat4) = cn
rownames(newmat4) = rn
head(newmat4)
```

```
##                 R1L1.HSM1 R1L2.PTF1 R1L3.RMM1 R1L4.HSF1 R1L6.PTM1
## ENSG00000000003        60       285       207       172       176
## ENSG00000000005         0         1         1         0         0
## ENSG00000000419        17        54        20        36        23
## ENSG00000000457        50        61        68        41       143
## ENSG00000000460         9         6         2         3         3
## ENSG00000000938        32        50        44        23        99
##                 R1L7.RMF1 R2L2.RMF2 R2L3.HSM2 R2L4.PTF2 R2L6.RMM2
## ENSG00000000003       259       299       219       213       676
## ENSG00000000005         0         1         1         1         0
## ENSG00000000419        38        40        42        35        39
## ENSG00000000457        42        47        30       111        55
## ENSG00000000460         2         1         2         5         2
## ENSG00000000938        46        18        26        53        20
##                 R2L7.HSF2 R2L8.PTM2 R3L1.RMM3 R3L2.HSF2 R3L3.PTM1
## ENSG00000000003       147       316       338       153       233
## ENSG00000000005         0         0         0         0         0
## ENSG00000000419        26        26        36        35        36
## ENSG00000000457        28       110        76        34       131
## ENSG00000000460         8         3         2         9         9
## ENSG00000000938        30        16        29        35        99
##                 R3L4.RMF3 R3L6.HSM3 R3L7.PTF3 R3L8.RMM1 R4L1.HSM3
## ENSG00000000003       242       199       180       217       160
## ENSG00000000005         0         0         0         1         0
## ENSG00000000419        22        33        35        29        29
## ENSG00000000457        61        53        46        49        42
## ENSG00000000460         5        11         2         5         6
## ENSG00000000938        28        32        39        36        33
##                 R4L2.HSF1 R4L3.RMM3 R4L4.PTF1 R4L6.PTM2 R4L7.RMF3
## ENSG00000000003       157       367       289       369       238
## ENSG00000000005         0         0         0         1         2
## ENSG00000000419        45        54        46        32        35
## ENSG00000000457        50        87        70       129        57
## ENSG00000000460         3         2         2         4         5
## ENSG00000000938        21        43        34        15        26
##                 R4L8.HSM2 R5L1.RMF1 R5L2.HSM1 R5L3.PTF3 R5L4.RMM2
## ENSG00000000003       202       252        61       206       672
## ENSG00000000005         0         0         0         1         0
## ENSG00000000419        36        29        22        30        43
```

```
## ENSG00000000457        43        47        64        41        59
## ENSG00000000460         5         2         6         3         1
## ENSG00000000938        17        27        41        37        14
##                 R5L8.RMF2 R6L2.PTM3 R6L4.PTM3 R6L6.PTF2 R8L1.HSF3
## ENSG00000000003       165       216       212       216        78
## ENSG00000000005         0         1         0         3         0
## ENSG00000000419        32        40        53        31        16
## ENSG00000000457        22        71        78       118        34
## ENSG00000000460         3         5         5         3         7
## ENSG00000000938        10        30        28        51       112
##                 R8L2.HSF3
## ENSG00000000003        90
## ENSG00000000005         0
## ENSG00000000419        40
## ENSG00000000457        42
## ENSG00000000460         5
## ENSG00000000938        98
```

```
# check they are the same
identical(newmat4, mat)
```

```
## [1] TRUE
```

Custom functions

Frequently you will need to run the same task on different occasions with different input.

In this case, you will define a custom function that will perform the same task, that accepts a specific set of arguments.

A function is defined like this:

```
myfunction = function(argument1, argument2, ...) { # arguments are the input for the func ...
  # your algorithm goes here
}
myfunction
```

```
## function(argument1, argument2, ...) { # arguments are the input for the func ...
##   # your algorithm goes here
## }
```

```
# a simple example
myfunction = function(x, y, ...) {
  x + y
}
myfunction(1, 2)
```

```
## [1] 3
```

So for instance, let's write a function that calculates the number of unique elements of a vector. We will name our function `luniq`; the name `ee` for argument is an arbitrary choice:

```
luniq = function(ee) {
  # function to calculate no. of unique elements in vector
  length( unique( ee ) )
}
```

After this definition, if we write name of the function only, R prints the function. If we also provide the parantheses with necessary arguments, function does its job.

```
luniq
```

```
## function(ee) {
##   # function to calculate no. of unique elements in vector
##   length( unique( ee ) )
## }
```

```
x = sample(1:10, 15, replace = T)
x
```

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

```
luniq( x )
```

```
## [1] 10
```

```
# number of unique runs in the transcriptome experiment
luniq( run )
```

```
## [1] 7
```

Now count the number of unique elements of each factor in our transcriptome dataset: `run`, `species`, `sex`, and `indv`. For this, we can form a list of these vectors, and use `sapply` over `luniq`. Let's write the same line in 3 ways. First the classical way, at each round, one of the vectors is passed to `x`:

```
sapply(list(run, species, sex), function(x) {
  luniq(x)
})
```

```
## [1] 7 3 2
```

This also works, without curly brackets, because there is only a single command after function:

```
sapply(list(run, species, sex), function(x) luniq(x))
```

```
## [1] 7 3 2
```

This also works because `luniq` is a function with a single argument:

```
sapply(list(run, species, sex), luniq )
```

```
## [1] 7 3 2
```

Would using `c(run, species, sex)` instead of `list(run, species, sex)` work? Try, and explain why not.

Now write a function that finds the length of overlap between two vectors:

```
overl = function(x1, x2) {
  # function to calculate no. of overlapping elements in 2 vectors
  length( intersect( x1, x2 ) )
}
```

```
overl
```

```
## function(x1, x2) {
##   # function to calculate no. of overlapping elements in 2 vectors
##   length( intersect( x1, x2 ) )
## }
```

```
overl(1:5, 4:8)
```

```
## [1] 2
```

By default the output of the last line of a function is returned, so you do not have to explicitly use `return`. So this works the same way:

```
overl2 = function(x1, x2) {
  x1  # this is not returned
  length( intersect( x1, x2 ) )
}
overl2(1:5, 4:8)
```

```
## [1] 2
```

```
overl3 = function(x1, x2) {
  return(x1)  # this is now returned
  length( intersect( x1, x2 ) )  # this is calculated but not returned
}
overl3(1:5, 4:8)
```

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

Exercise: Combination function

Now try to write a function that takes as input a number `n` and outputs the combinations of `n` choose `2`. In this case you will need to write a loop in a loop. Call the function `combn2`:

```
combn2 = function(n) {  # function to create combinations of n
  for (i in 1:(n-1)) {
    for (j in (i+1):n) {
      print( c(i,j) )
    }
  }
}
combn2(4)
```

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

```
# check using the combn function
t( combn(4, 2) )
```

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

We could also have a matrix as output. For this, if we are using a for loop, we will need to predefine an empty matrix, like:

```
matrix(,0,2)
```

```
##      [,1] [,2]
```

So the function would be as follows:

```
combn2b = function(n) {  # function to create combinations of n
  result = matrix(,0,2)  # create a matrix, with 0 rows and 2 columns
  for (i in 1:(n-1)) {
    for (j in (i+1):n) {
      result = rbind(result, c(i,j) )
    }
  }
  return(result)
}
combn2b(4)
```

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

`if` and `else` statements and `while` loops

`if` structures evaluate a logical statement and execute the following command only if the statement is `TRUE`:

```
if ( logical statement ) {
  a command
}

# some simple examples:
if (T) print("yes")
if (F) print("yes")
if (10) print("yes")
if (0) print("yes")
```

Some more examples:

```
i = 1
if (i > 5) {
  print ("BIG")
}

i = 10
if (i > 5) {
  print ("BIG")
}
```

```
## [1] "BIG"
```

You can also use `if` and `else` together:

```
if ( logical statement ) {
  a command
} else {
  another command
}

# or

if ( logical statement ) {
  a command
} else if ( another logical statement ) {
  another command
} else {
  a 3rd command
}
```

An example:

```
set.seed(2)
i = runif(1)
i
```

```
## [1] 0.1848823
```

```
if (i > 0.5) {
  print("big")
} else {
  print("small")
}
```

```
## [1] "small"
```

```
i = runif(1)
i
```

```
## [1] 0.702374
```

```
if (i > 0.5) {
  print("big")
} else {
  print("small")
}
```

```
## [1] "big"
```

Exercise: a p-value evaluation function

You could use these structures inside a function. Let's write a function that evaluates p-values and prints out "n.s" and "*","**":

```
evalp = function(p) {
    if ( p>0.05 ) {
      print ("n.s.")
    } else if ( p>0.01 & p<0.05 ) {
      print ("*")
    } else {
      print("**")
    }
}

evalp(0.001)
```

```
## [1] "**"
```

```
evalp(0.02)
```

```
## [1] "*"
```

```
evalp(0.3)
```

```
## [1] "n.s."
```

Try this one 100 numbers from the uniform distribution:

```r
set.seed(1)
sapply(runif(100), function(i) c(i, evalp(i)))
```

```
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "*"
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "*"
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
```

```
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
## [1] "n.s."
```

```
##      [,1]              [,2]               [,3]
## [1,] "0.2655086631421" "0.37212389963679" "0.572853363351896"
## [2,] "n.s."            "n.s."             "n.s."
```

```
##      [,4]                [,5]                [,6]
## [1,] "0.908207789994776" "0.201681931037456" "0.898389684967697"
## [2,] "n.s."              "n.s."              "n.s."
##      [,7]                [,8]                [,9]
## [1,] "0.944675268605351" "0.660797792486846" "0.62911404389888"
## [2,] "n.s."              "n.s."              "n.s."
##      [,10]                [,11]               [,12]
## [1,] "0.061786270467564" "0.205974574899301" "0.176556752528995"
## [2,] "n.s."               "n.s."              "n.s."
##      [,13]               [,14]               [,15]
## [1,] "0.687022846657783" "0.384103718213737" "0.769841419998556"
## [2,] "n.s."              "n.s."              "n.s."
##      [,16]               [,17]               [,18]
## [1,] "0.497699242085218" "0.717618508264422" "0.991906094830483"
## [2,] "n.s."              "n.s."              "n.s."
##      [,19]               [,20]               [,21]
## [1,] "0.380035179434344" "0.777445221319795" "0.934705231105909"
## [2,] "n.s."              "n.s."              "n.s."
##      [,22]               [,23]               [,24]
## [1,] "0.212142521282658" "0.651673766085878" "0.125555095961317"
## [2,] "n.s."              "n.s."              "n.s."
##      [,25]               [,26]               [,27]
## [1,] "0.267220668727532" "0.386114092543721" "0.0133903331588954"
## [2,] "n.s."              "n.s."              "*"
##      [,28]               [,29]              [,30]
## [1,] "0.382387957070023" "0.86969084572047" "0.34034899668768"
## [2,] "n.s."              "n.s."             "n.s."
##      [,31]               [,32]               [,33]
## [1,] "0.482080115471035" "0.599565825425088" "0.493541307048872"
## [2,] "n.s."              "n.s."              "n.s."
##      [,34]               [,35]               [,36]
## [1,] "0.186217601411045" "0.827373318606988" "0.668466738192365"
## [2,] "n.s."              "n.s."              "n.s."
##      [,37]              [,38]               [,39]
## [1,] "0.79423986072652" "0.107943625887856" "0.723710946040228"
## [2,] "n.s."             "n.s."              "n.s."
##      [,40]               [,41]               [,42]
## [1,] "0.411274429643527" "0.820946294115856" "0.647060193819925"
## [2,] "n.s."              "n.s."              "n.s."
##      [,43]              [,44]               [,45]
## [1,] "0.78293276228942" "0.553036311641335" "0.529719580197707"
## [2,] "n.s."             "n.s."              "n.s."
##      [,46]               [,47]               [,48]
## [1,] "0.789356231689453" "0.023331202333793" "0.477230065036565"
## [2,] "n.s."              "*"                 "n.s."
##      [,49]             [,50]               [,51]
## [1,] "0.7323137386702" "0.692731556482613" "0.477619622135535"
## [2,] "n.s."            "n.s."              "n.s."
##      [,52]             [,53]               [,54]
## [1,] "0.8612094768323" "0.438097107224166" "0.244797277031466"
## [2,] "n.s."            "n.s."              "n.s."
##      [,55]                [,56]                [,57]
## [1,] "0.0706790471449494" "0.0994661601725966" "0.31627170718275"
## [2,] "n.s."               "n.s."               "n.s."
```

```
##      [,58]                [,59]                [,60]
## [1,] "0.518634263193235" "0.662005076417699" "0.406830187188461"
## [2,] "n.s."               "n.s."               "n.s."
##      [,61]                [,62]                [,63]
## [1,] "0.912875924259424" "0.293603372760117" "0.459065726259723"
## [2,] "n.s."               "n.s."               "n.s."
##      [,64]                [,65]               [,66]
## [1,] "0.332394674187526" "0.65087046707049" "0.258016780717298"
## [2,] "n.s."               "n.s."              "n.s."
##      [,67]                [,68]                [,69]
## [1,] "0.478545248275623" "0.766310670645908" "0.0842469143681228"
## [2,] "n.s."               "n.s."               "n.s."
##      [,70]                [,71]                [,72]
## [1,] "0.875321330036968" "0.339072937844321" "0.839440350187942"
## [2,] "n.s."               "n.s."               "n.s."
##      [,73]               [,74]                [,75]
## [1,] "0.34668348915875" "0.333774930797517" "0.476351245073602"
## [2,] "n.s."              "n.s."               "n.s."
##      [,76]                [,77]                [,78]
## [1,] "0.892198335845023" "0.864339470630512" "0.389989543473348"
## [2,] "n.s."               "n.s."               "n.s."
##      [,79]                [,80]                [,81]
## [1,] "0.777320698834956" "0.960617997217923" "0.434659484773874"
## [2,] "n.s."               "n.s."               "n.s."
##      [,82]                [,83]                [,84]
## [1,] "0.712514678714797" "0.399994368897751" "0.325352151878178"
## [2,] "n.s."               "n.s."               "n.s."
##      [,85]                [,86]                [,87]
## [1,] "0.757087148027495" "0.202692255144939" "0.711121222469956"
## [2,] "n.s."               "n.s."               "n.s."
##      [,88]                [,89]                [,90]
## [1,] "0.121691921027377" "0.245488513959572" "0.14330437942408"
## [2,] "n.s."               "n.s."               "n.s."
##      [,91]                [,92]                 [,93]
## [1,] "0.239629415096715" "0.0589343772735447" "0.642288258532062"
## [2,] "n.s."               "n.s."                "n.s."
##      [,94]                [,95]                [,96]
## [1,] "0.876269212691113" "0.778914677444845" "0.79730882588774"
## [2,] "n.s."               "n.s."               "n.s."
##      [,97]                [,98]                [,99]
## [1,] "0.455274453619495" "0.410084082046524" "0.810870242770761"
## [2,] "n.s."               "n.s."               "n.s."
##      [,100]
## [1,] "0.604933290276676"
## [2,] "n.s."
```

Now we can use `if` to create the combination function in an alternative way:

```
combn2c = function(n) {
  result = matrix(,0,2)
  for (i in 1:n) {
    for (j in 1:n) {
      if ((i != j) & (i < j))
      result = rbind(result, c(i,j) )
```

```
    }
  }
  return(result)
}
combn2c(4)
```

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

while

`while` is another type of loop structure which is open-ended (and thus, dangerous). It will run a loop infinitely as long as the statement remains `TRUE` or you externally cancel the process.

```
myi = c()
set.seed(1)
i = runif(1)
while (i <0.99) {   # runs the loop until i hits a value >=0.99
  i = runif(1)
  myi = c(myi, i)
}
myi
```

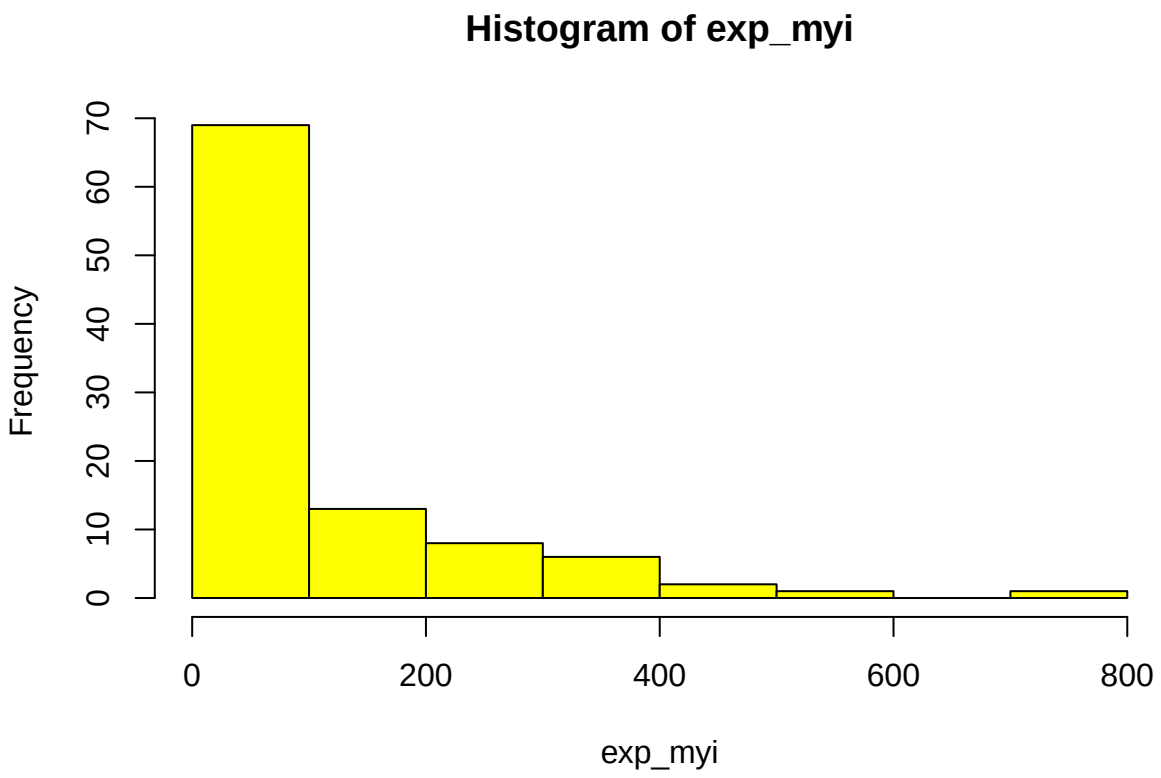
```
##  [1] 0.37212390 0.57285336 0.90820779 0.20168193 0.89838968 0.94467527
##  [7] 0.66079779 0.62911404 0.06178627 0.20597457 0.17655675 0.68702285
## [13] 0.38410372 0.76984142 0.49769924 0.71761851 0.99190609
```

What is the probability that `i>=0.99`? This should be `0.01`. If so, what will be the expected lengths of `myi` vectors (i.e. the number of events that will happen before `i` hits `>=0.99`?

```
exp_myi = sapply(1:100, function(j) {
  myi = c()
  i = runif(1)
  while (i <0.99) {   # runs the loop until i hits a value >=0.99
    i = runif(1)
    myi = c(myi, i)
  }
  length(myi)
})
summary( exp_myi )
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0    25.0    69.5   110.6   148.8   724.0
```

```
hist( exp_myi, col=7 )
```



The geometric and negative binomial distributions

This is `100`, which is the inverse of the rate `0.01`. The discrete probability distribution for the number of failures until a success is called the **geometric distribution**. The more general case, number of failures until the nth success, is called the **negative binomial distribution**:

https://onlinecourses.science.psu.edu/stat414/node/78.

ANOVA-Tukey HSD vs. pairwise t-tests

Remember that we were testing whether there is a species effect on total read counts in the transcriptome matrix. We'd first used ANOVA and then run a posthoc test:

```
summary( aov( totalcounts ~ species ) )
```

```
##             Df    Sum Sq   Mean Sq F value  Pr(>F)
## species      2 1.823e+12 9.113e+11   8.078 0.00139 **
## Residuals   33 3.723e+12 1.128e+11
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
TukeyHSD( aov( totalcounts ~ species ) )
```

```
##   Tukey multiple comparisons of means
```

```
##     95% family-wise confidence level
##
## Fit: aov(formula = totalcounts ~ species)
##
## $species
##           diff         lwr      upr     p adj
## PT-HS 199067.8 -137394.272 535529.9 0.3268430
## RM-HS 544617.3  208155.144 881079.4 0.0010361
## RM-PT 345549.4    9087.311 682011.5 0.0431198
```

```
# to obtain the p-values
TukeyHSD(aov(totalcounts ~ species))$species[,"p adj"]
```

```
##       PT-HS       RM-HS       RM-PT
## 0.326843031 0.001036065 0.043119759
```

So, it seems that macaques have higher read depth than other species.

We can also run t-tests. The function in R is called `t.test`. This is a one sample t-test example comparing the mean of the first argument, a vector, with the null hypothesis stated by `mu` - that the mean of the data `=2`:

```
t.test(1:10, mu = 2)
```

```
##
##  One Sample t-test
##
## data:  1:10
## t = 3.6556, df = 9, p-value = 0.005271
## alternative hypothesis: true mean is not equal to 2
## 95 percent confidence interval:
##  3.334149 7.665851
## sample estimates:
## mean of x
##       5.5
```

You can also run a one-sided test, testing whether the mean is `>2` (the null hypothesis is the opposite):

```
t.test(1:10, mu = 2, alternative = "greater")
```

```
##
##  One Sample t-test
##
## data:  1:10
## t = 3.6556, df = 9, p-value = 0.002636
## alternative hypothesis: true mean is greater than 2
## 95 percent confidence interval:
##  3.744928      Inf
## sample estimates:
## mean of x
##       5.5
```

Note that the p-value is different. The default is running the test two-sided.

You can also compare two vectors - a **two-sample t-test** (note that this is totally different from a paired t-test):

```
t.test(1:10, 3:8, alternative = "two.sided")
```

```
##
##  Welch Two Sample t-test
##
## data:  1:10 and 3:8
## t = 0, df = 13.939, p-value = 1
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.627899  2.627899
## sample estimates:
## mean of x mean of y
##       5.5       5.5
```

```
t.test(1:10, 3:8, alternative = "greater")
```

```
##
##  Welch Two Sample t-test
##
## data:  1:10 and 3:8
## t = 0, df = 13.939, p-value = 0.5
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  -2.157825       Inf
## sample estimates:
## mean of x mean of y
##       5.5       5.5
```

```
# same thing
t.test(1:10, 3:8, alt = "g")
```

```
##
##  Welch Two Sample t-test
##
## data:  1:10 and 3:8
## t = 0, df = 13.939, p-value = 0.5
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  -2.157825       Inf
## sample estimates:
## mean of x mean of y
##       5.5       5.5
```

We can now try to run t-tests for all pairwise comparisons of `species` on `totalcounts`, calculate the p-values, and report the results including the paired names of the compared species:

```
pvals = c()
namesx = c()
for (x in unique(species) ) {
  for (y in unique(species)) {
    if (x !=y & x > y) {
      p = t.test(totalcounts[species == x], totalcounts[species == y])$p.val
      pvals = c(pvals, p)
      namesx = c(namesx, paste(x, y, sep="-") )
    }
  }
}
names(pvals) = namesx
pvals
```

```
##        PT-HS        RM-HS        RM-PT
## 0.1588524559 0.0003596124 0.0272050808
```

```
# compare with Tukey's
TukeyHSD( aov( totalcounts ~ species ) )$species[,"p adj"]
```

```
##       PT-HS       RM-HS       RM-PT
## 0.326843031 0.001036065 0.043119759
```

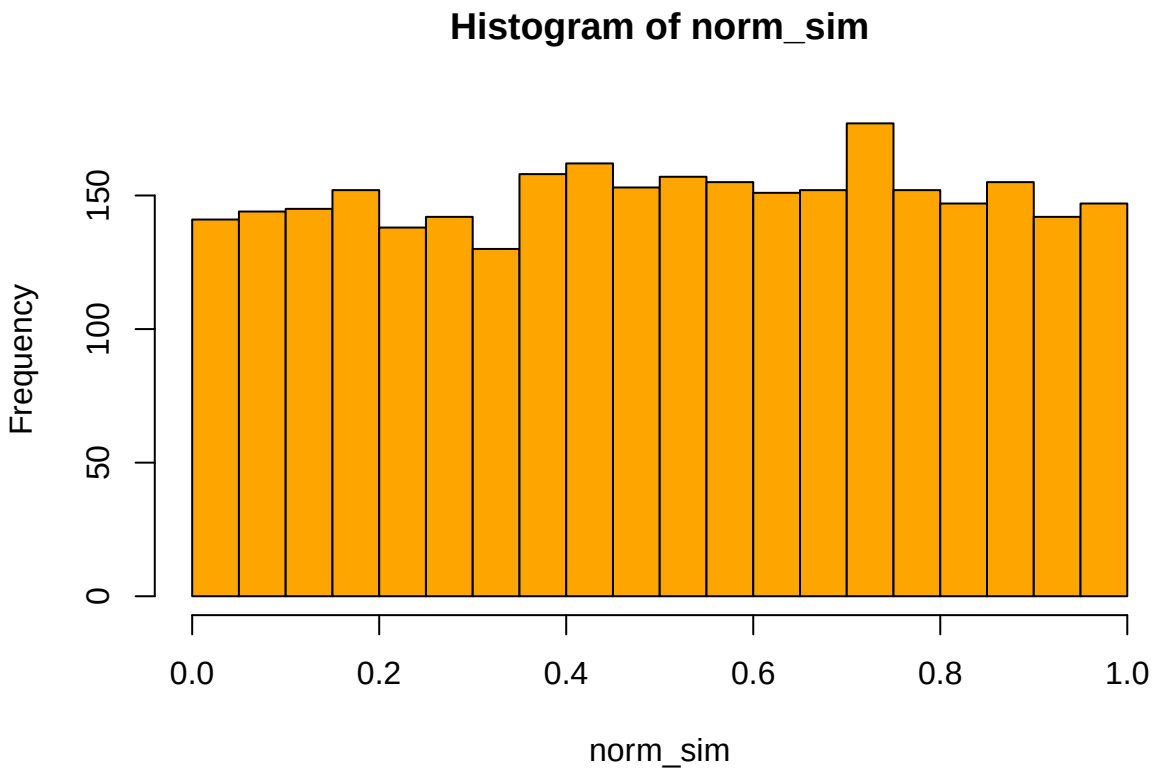
Note that the t-test p-values are lower, while the Tukey test p-values are higher, as they account for the fact that we are conducting 3 tests. Thus the Type I error rate (probability of rejecting a true null hypothesis of no difference) is inflated if we use pairwise t-tests.

ANOVA vs. Kruskal-Wallis

ANOVA is a parametric test and it assumes equal variance among groups, and normality. Both assumptions might be violated, as in this example:

```
# simulation 1: testing mean difference when the data derives from a normal distribution
# first define the groups
groups = factor( rep(c("a", "b"), each=10) )

# then run the ANOVA (with 2 groups, for simplicity)
norm_sim = sapply(1:3000, function(i) {
  x1 = rnorm(10, mean = 1, sd = 1)
  x2 = rnorm(10, mean = 1, sd = 1)
  pval = summary( aov(c(x1, x2) ~ groups ) )[[1]][1, "Pr(>F)"]
})
hist(norm_sim, col="orange", br=20)
```



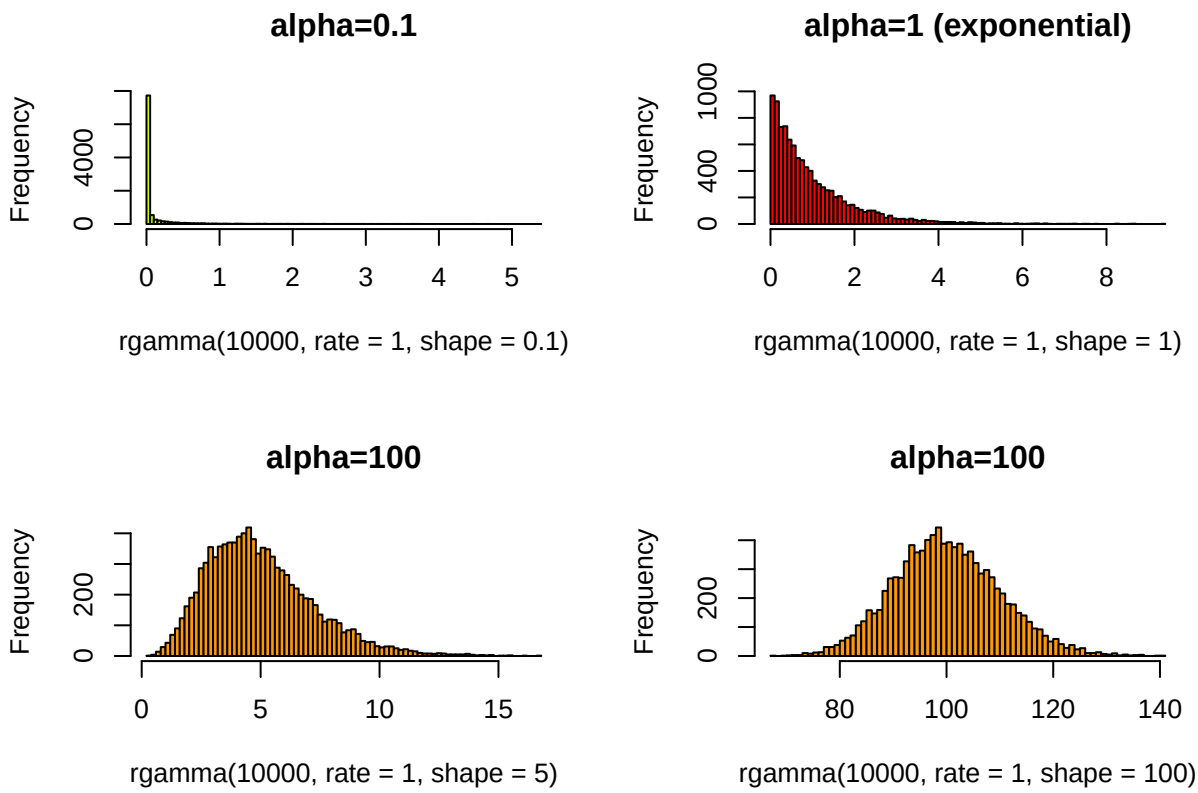
The distribution is uniform, as expected. What would happen if the data were chosen from a gamma distribution?

The gamma distribution

This is a family of continuous probability distributions frequently used to model waiting times (similar to the geometric), mutation effects, etc. It has two parameters, shape (alpha) and scale (beta). A gamma distribution with shape parameter alpha = 1 and is an **exponential distribution**:

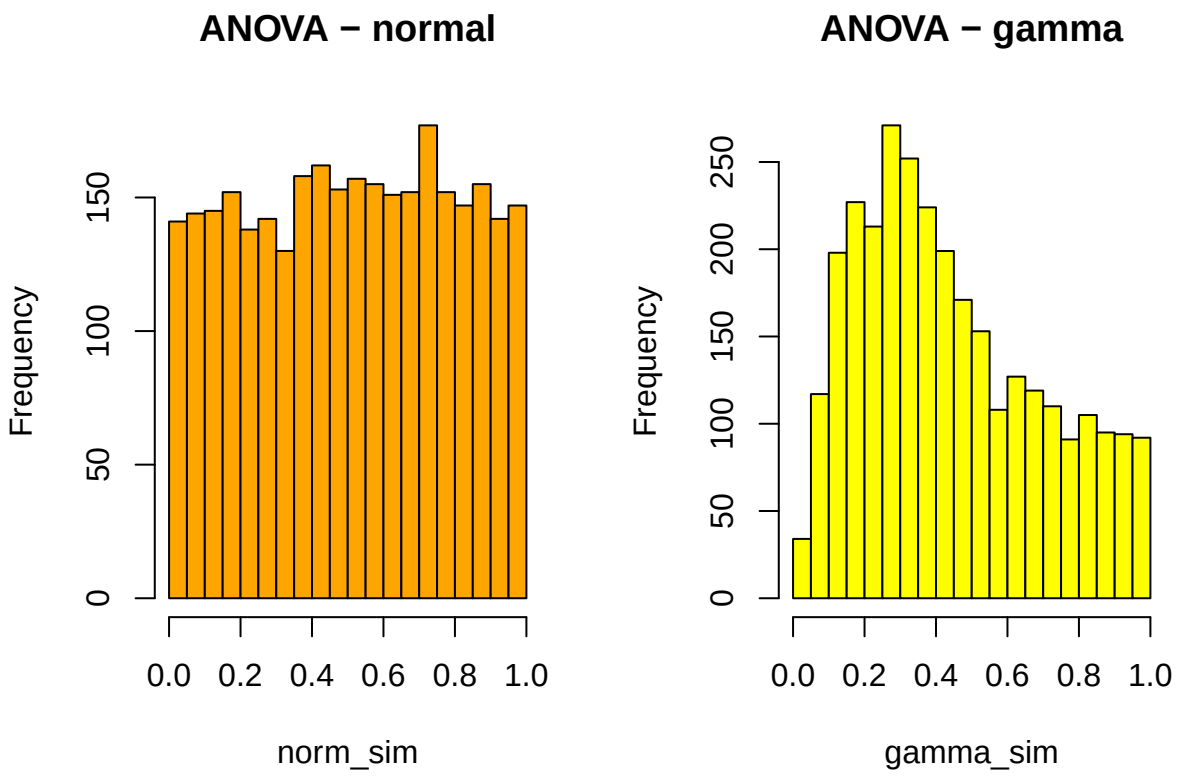
https://en.wikipedia.org/wiki/Gamma_distribution

```
par(mfrow=c(2,2))
hist(rgamma(10000, rate = 1, shape = 0.1), col=rainbow(10)[3], br=100, main="alpha=0.1")
hist(rgamma(10000, rate = 1, shape = 1), col=rainbow(10)[1], br=100, main="alpha=1 (exponential)")
hist(rgamma(10000, rate = 1, shape = 5), col=rainbow(10)[2], br=100, main="alpha=100")
hist(rgamma(10000, rate = 1, shape = 100), col=rainbow(10)[2], br=100, main="alpha=100")
```



Now we can run the same ANOVA test drawing data from the gamma:

```
# simulation 2: ANOVA when data derive from gamma
gamma_sim = sapply(1:3000, function(i) {
  x1 = rgamma(10, rate = 1, shape = 0.1)
  x2 = rgamma(10, rate = 1, shape = 0.1)
  pval = summary( aov(c(x1, x2) ~ groups ) )[[1]][1, "Pr(>F)"]
})
par(mfrow=c(1,2))
hist(norm_sim, col="orange", br=20, main="ANOVA - normal")
hist(gamma_sim, col="yellow", br=20, main="ANOVA - gamma")
```




```
mean(norm_sim < 0.05)
```

```
## [1] 0.047
```

```
mean(gamma_sim < 0.05)
```

```
## [1] 0.01133333
```

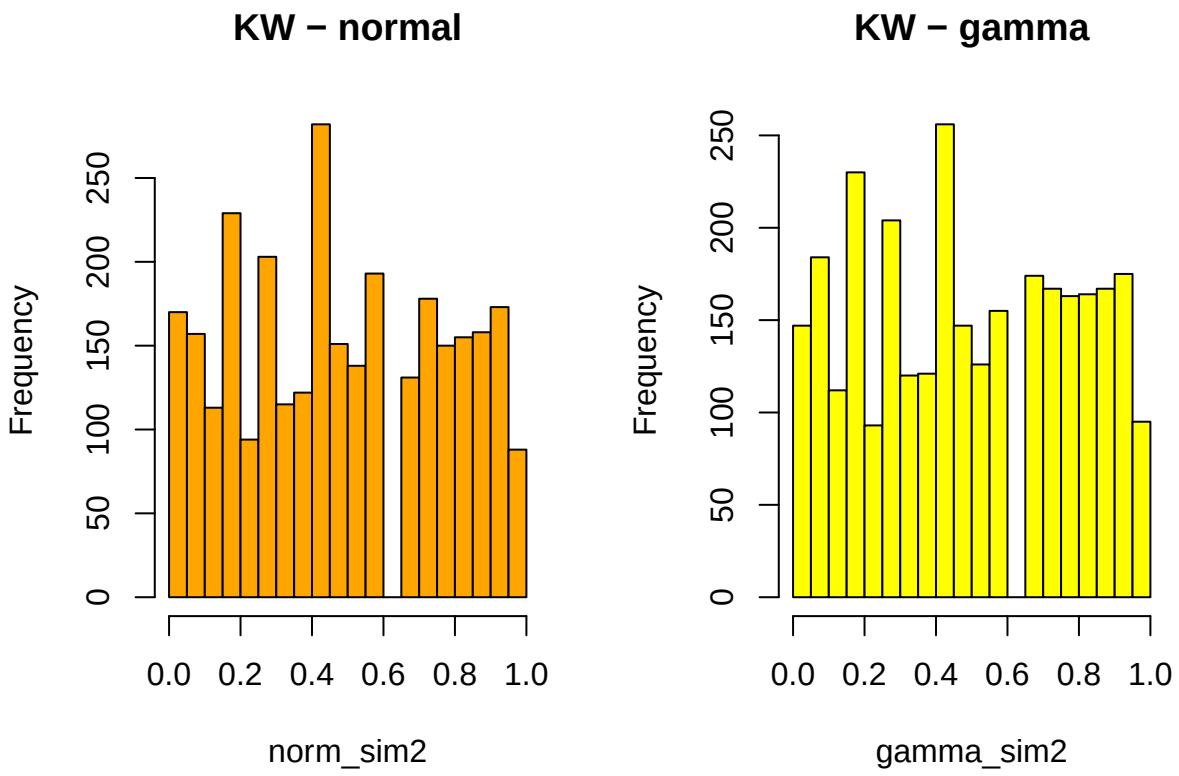
There is a deficiency of low p-values, which suggests the test becomes too conservative, in this case.

Non-parametric tests work by ranking the data, and therefore may be less sensitive to deviations from assumptions of normality and to outliers. One commonly used non-parametric equivalent of ANOVA is the Kruskal-Wallis test, which tests the equality of **medians** among groups. Note, however, that many non-parametric tests still assume similarity in shape between distributions (thus, they will also be sensitive to differences in variance).

We can apply the same simulation with KW:

```
norm_sim2 = sapply(1:3000, function(i) {
  x1 = rnorm(10, mean = 1, sd = 1)
  x2 = rnorm(10, mean = 1, sd = 1)
  pval = kruskal.test(c(x1, x2) ~ groups )$p.val
})
gamma_sim2 = sapply(1:3000, function(i) {
  x1 = rgamma(10, rate = 1, shape = 0.1)
  x2 = rgamma(10, rate = 1, shape = 0.1)
  pval = kruskal.test(c(x1, x2) ~ groups )$p.val
})
```

```
par(mfrow=c(1,2))
hist(norm_sim2, col="orange", br=20, main="KW - normal")
hist(gamma_sim2, col="yellow", br=20, main="KW - gamma")
```




```
mean(norm_sim2 < 0.05)
```

```
## [1] 0.05666667
```

```
mean(gamma_sim2 < 0.05)
```

```
## [1] 0.049
```

We see that the KW test p-values are not perfectly uniform due to small sample size and ranking. Importantly, however, the KW is not sensitive to the shape of the distributions.

Now we can apply the KW test to the sum of columns in the liver transcriptome dataset, for all the variables:

```
load("liver_transcriptome_v1.Rdata")
pvals = sapply( list(species, sex, run), function(varx) {
  kruskal.test( totalcounts ~ varx )$p.val
} )
round( pvals, 3)
```

```
## [1] 0.003 0.849 0.192
```

Summary

Using different methods (2-sample t-tests, ANOVA and Tukey HSD, Kruskal-Wallist), we repeatedly found the same results: There is a species effect on total read counts, but sex and run have limited influence. Among the 3 species, macaques appear to have the most different total read counts.

What could be the reason? biological, or technical? Given that same amount of total RNA was used, the authors (like many others) have assumed that the differences are technical and have **normalized** by dividing by total reads per sample. We will also normalize columns in a slightly different way, but later.