# STAT 210
# Applied Statistics with R
# Some useful functions

Joaquín Ortega

joaquin.ortegasanchez@kaust.edu.sa

## Some useful functions

## Some useful functions

We will review the following functions

| | |
|---|---|
| `subset` | `with` |
| `within` | `aggregate` |
| `order` | `apply` |
| `sweep` | `tapply` |
| `outer` | `split` |
| `expand.grid` | Statistical summaries |

`subset`

## subset

This function returns subsets that meet certain conditions. Syntax:

```
subset(x, subset, select, ...)
```

where

- `x` is the object we use to extract the subset,
- `subset` is a logical expression indicating elements or rows to keep,
- `select` is an expression indicating columns to select from a data frame.

## subset

```r
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1
```

```r
iris.ver <- subset(iris, Species == 'versicolor')
str(iris.ver, vec.len = 2)
```

```
## 'data.frame':    50 obs. of  5 variables:
##  $ Sepal.Length: num  7 6.4 6.9 5.5 6.5 ...
##  $ Sepal.Width : num  3.2 3.2 3.1 2.3 2.8 ...
##  $ Petal.Length: num  4.7 4.5 4.9 4 4.6 ...
##  $ Petal.Width : num  1.4 1.5 1.5 1.3 1.5 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 2 ...
```

The structure of the original data frame is preserved, but only the individuals belonging to this species are included in the subset.

## subset

In the second example we use the option ‘select’ to get only the variables ‘Sepal.Length’ and ‘Sepal.Width’.

```r
iris.ver2 <- subset(iris, Species == 'versicolor',
                 select = c(Sepal.Length,Sepal.Width))
str(iris.ver2, vec.len = 2)
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ Sepal.Length: num  7 6.4 6.9 5.5 6.5 ...
##  $ Sepal.Width : num  3.2 3.2 3.1 2.3 2.8 ...
```

`with`

## `with`

Even though the function `attach` is very useful, it can cause problems, especially if it is used several times in a script, and we forget to detach the objects we no longer need.

If several variables have the same name, it is easy to get confused about which version of the variable is available in the working directory.

For this reason, in many cases, it is more convenient to use the function `with`, which has format

```
with(data,expr,...)
```

where `data` is the data object that has the variables we want to use, and `expr` is the expression to evaluate.

## with

This function can be used to extract data from data objects, for instance

```r
with(iris, Sepal.Length[Petal.Length < 1.5
                        & Petal.Width >= 0.4])
```

```
## [1] 5.4
```

gives the same result as

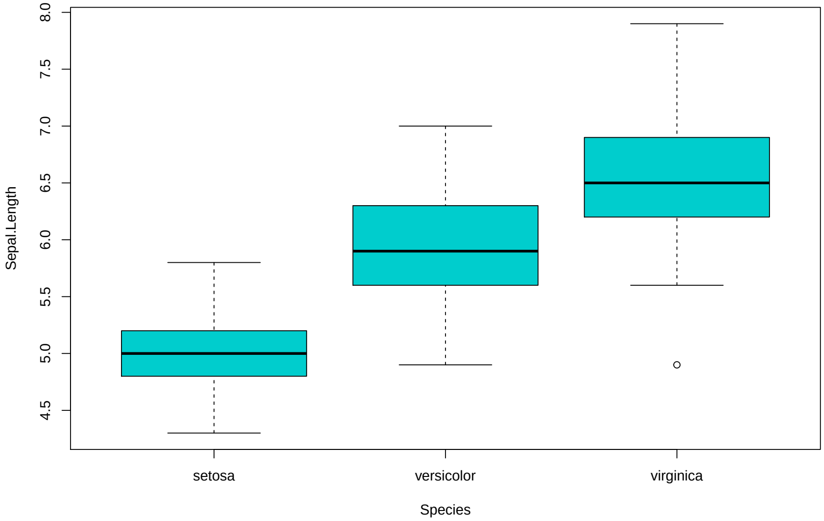
```r
iris$Sepal.Length[iris$Petal.Length < 1.5
                  & iris$Petal.Width >= 0.4]
```

```
## [1] 5.4
```

## with

Another example:

```r
with(iris, boxplot(Sepal.Length ~ Species, col = 'cyan3'))
```



## with

When this function is used, R builds an environment where the assignments within expr take place. They do not take place in the user's workspace.

This function returns the value of the last evaluated expression.

If we want to evaluate several expressions using the same data object, the expressions should be within braces separated by semicolons.

```
example.df <- data.frame(a = 1:5, b=-5:-1)
with(example.df, {a+b; 2*a})
```

```
## [1]  2  4  6  8 10
```

The two expressions have been executed, but R only returns the result of the last one.

within

## within

within is similar to with but it can modify the original data object. It returns the data object and not the evaluated expression.

Let's see a simple example to make the difference clear. We start by using with to calculate the sum of the vectors a and b in example.df

```r
with(example.df, a + b)
```

```
## [1] -4 -2  0  2  4
```

Now let's assign this sum to a new vector c. In this case, the command must go inside braces.

```r
with(example.df, {c = a + b})
```

There is no output!

The problem is that the assignment c = a + b was evaluated in a temporal environment and not in the workspace.

## within

Since we did not ask for c to be printed, we did not see any output. Let's include now a `print(c)` command

```
with(example.df, {c = a + b; print(c)})
```

```
## [1] -4 -2  0  2  4
```

However, the data frame example.df is not changed by these commands

```
with(example.df, {c = a + b; example.df})
```

```
##   a  b
## 1 1 -5
## 2 2 -4
## 3 3 -3
## 4 4 -2
## 5 5 -1
```

## within

In contrast, if we use within,

```
within(example.df, {c = a + b; example.df})
```

```
##   a  b  c
## 1 1 -5 -4
## 2 2 -4 -2
## 3 3 -3  0
## 4 4 -2  2
## 5 5 -1  4
```

we see that the data frame has been modified and now includes a new column for the variable `c = a + b`.

## within

However, the change in `example.df` is not permanent

```
example.df
```

```
##   a  b
## 1 1 -5
## 2 2 -4
## 3 3 -3
## 4 4 -2
## 5 5 -1
```

If you want to make it permanent, it is necessary to save it:

```
(example.df <- within(example.df, {c = a + b}))
```

```
##   a  b  c
## 1 1 -5 -4
## 2 2 -4 -2
## 3 3 -3  0
## 4 4 -2  2
## 5 5 -1  4
```

`aggregate`

## aggregate

The `aggregate` function splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

The syntax is

```
aggregate(x, by, FUN,...)
```

where `x` is the data frame, `by` is a list of the elements that determine the groups, and `FUN` is the function to be used.

## aggregate

```r
library(MASS)
str(crabs)
```

```
## 'data.frame':    200 obs. of  8 variables:
##  $ sp   : Factor w/ 2 levels "B","O": 1 1 1 1 1 1 1 1 1 1 ...
##  $ sex  : Factor w/ 2 levels "F","M": 2 2 2 2 2 2 2 2 2 2 ...
##  $ index: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ FL   : num  8.1 8.8 9.2 9.6 9.8 10.8 11.1 11.6 11.8 11.8 ...
##  $ RW   : num  6.7 7.7 7.8 7.9 8 9 9.9 9.1 9.6 10.5 ...
##  $ CL   : num  16.1 18.1 19 20.1 20.3 23 23.8 24.5 24.2 25.2 ...
##  $ CW   : num  19 20.8 22.4 23.1 23 26.5 27.1 28.4 27.8 29.3 ...
##  $ BD   : num  7 7.4 7.7 8.2 8.2 9.8 9.8 10.4 9.7 10.3 ...
```

```r
aggregate(crabs[, 4:8], list(sp=crabs$sp,
                             sex=crabs$sex), median)
```

```
##   sp sex    FL    RW    CL    CW    BD
## 1  B   F 13.15 12.20 27.90 32.35 11.60
## 2  O   F 18.00 14.65 34.70 39.55 15.65
## 3  B   M 15.10 11.70 32.45 37.10 13.60
## 4  O   M 16.70 12.10 33.35 36.30 15.00
```

order

## order

The function order is used for sorting complex structures, such as data frames. For vectors, there is also the function sort.

The result of using order on a vector is a permutation that rearranges the vector in ascending or descending order, according to the decreasing option selected (that is false by default).

## order

Let's see an example

```r
(x <- c(14:16,12,11,13,17))
```

```
## [1] 14 15 16 12 11 13 17
```

```r
order(x)
```

```
## [1] 5 4 6 1 2 3 7
```

The result means that to sort the vector `x`, you have to place component 5 first, component 4 second, component 6 third, and so on.

To order `x`, type

```r
x[order(x)]
```

```
## [1] 11 12 13 14 15 16 17
```

## order

This function allows you to use the indices obtained to sort several vectors using the criteria corresponding to the vector x.

```
y <- 17:11
z <- 11:17
rbind(x,y,z)[,order(x)]
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## x   11   12   13   14   15   16   17
## y   13   14   12   17   16   15   11
## z   15   14   16   11   12   13   17
```

`apply`

## apply

This command successively applies a function to each row (first dimension), column (second dimension), or level of a higher-dimensional array.

The syntax is

```
apply(data, dim, function, ...)
```

where `data` is the name of the matrix or array and `function` is any function in `R`.

To see an example of the use of this function, we are going to work with the file `iris3` that has the information of the data set `iris` that we have used previously but in the format of an arrangement of dimensions $50 \times 4 \times 3$.

## apply

The third dimension corresponds to the species, and for each of the three species, there is a matrix of dimension 50 $\times$ 4 with the values of the four variables that we already know for the 50 plants of the corresponding species.

```
str(iris3)
```

```
##  num [1:50, 1:4, 1:3] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  - attr(*, "dimnames")=List of 3
##   ..$ : NULL
##   ..$ : chr [1:4] "Sepal L." "Sepal W." "Petal L." "Petal W."
##   ..$ : chr [1:3] "Setosa" "Versicolor" "Virginica"
```

To calculate the mean value for each variable in the data set, we write:

```
apply(iris3, 2, mean)
```

```
## Sepal L. Sepal W. Petal L. Petal W.
## 5.843333 3.057333 3.758000 1.199333
```

## apply

However, if we want the mean value for each species we have to use a vector as dimension, as follows

```
apply(iris3, c(2,3), mean)
```

```
##          Setosa Versicolor Virginica
## Sepal L.  5.006      5.936     6.588
## Sepal W.  3.428      2.770     2.974
## Petal L.  1.462      4.260     5.552
## Petal W.  0.246      1.326     2.026
```

If the function to be used requires additional parameters, these can be included after the function name.

## apply

As an example, we are going to calculate the trimmed averages for the same data above. Trimmed averages are a robust version of averages in which a percentage of the sample corresponding to the data furthest from the center is 'trimmed'.

```r
apply(iris3, c(2,3), mean, trim=0.1)
```

```
##          Setosa Versicolor Virginica
## Sepal L. 5.0025     5.9375    6.5725
## Sepal W. 3.4150     2.7800    2.9625
## Petal L. 1.4600     4.2925    5.5100
## Petal W. 0.2375     1.3250    2.0325
```

`sweep`

## sweep

This function returns an array obtained from an input array by sweeping out a summary statistic. It is often used in conjunction with the apply function.

the syntax is

```
sweep(x, MARGIN, STATS, FUN = "-")
```

where

- ‘x‘ is an array,
- ‘MARGIN‘ is a vector of indices, typically rows or columns if ‘x‘ is a matrix,
- ‘STATS‘ is the summary statistic which is to be swept out,
- ‘FUN‘ is the function to be applied.

The output is an array of the same shape as x.

## sweep

For instance, suppose we want to subtract the mean we calculated in the previous example from all the components of the array `iris3`; that is, in each case, we want to subtract the mean corresponding to the variable and the species of each data. One way to do this is as follows:

```
iris.means <- apply(iris3, c(2,3), mean)
```

```
iris.ctd <- sweep(iris3, c(2,3), iris.means,'-')
head(iris.ctd)
```

```
## [1]  0.094 -0.106 -0.306 -0.406 -0.006
## [6]  0.394
```

## tapply

## tapply

tapply is used to apply a function to subsets of a data set, usually a vector.

The syntax is

```r
tapply(x, index, FUN = NULL, ...)
```

where

- ‘x‘ is an object -usually a vector- that can be divided into groups using a categorical variable (known as a ‘factor‘)
- ‘index‘ is a list of one or more factors with the same length as ‘x‘.
- ‘FUN‘ is the function to be applied.

## tapply

For the examples, we will use the data set `mtcars`

```r
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 1
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

## tapply

We first calculate the mean of mpg using the variable am to define the groups.

```r
tapply(mtcars$mpg, mtcars$am, mean)
```

```
##        0        1
## 17.14737 24.39231
```

## tapply

It is also possible to define the groups using more than one factor.

In this example, we use `am` and `cyl` to divide the set before calculating the mean for each group.

```
tapply(mtcars$mpg, list(mtcars$am,mtcars$cyl), mean)
```

```
##        4        6     8
## 0 22.900 19.12500 15.05
## 1 28.075 20.56667 15.40
```

outer

## `outer`

Another important function for matrices, data frames, and arrangements, in general, is the function `outer` or outer product.

If `A` and `B` are two arrays, their outer product is a new array whose dimension is the concatenation of the dimension vectors of the arrays $A$ and $B$, in the order of the product, and whose entries are obtained forming all possible products of the elements of $A$ with all the elements of $B$.

For example, the outer product of a vector of length 2 with another of length 3 produces an array (matrix) of dimensions $2 \times 3$; the product of an array of dimensions $m \times n$ times another array of dimensions $p \times q$ produces an array of dimensions $m \times n \times p \times q$. In this new array, the element in position $(i, j, k, l)$ is the product

$$A[i, j] \times B[k, l]$$

## outer

The notation for the outer product is %o%.

```
(aa <- 1:4)
```

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

```
(bb <- c(2,4,6))
```

```
## [1] 2 4 6
```

```
(ab <- aa %o% bb)
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    4    8   12
## [3,]    6   12   18
## [4,]    8   16   24
```

## outer

In terms of linear algebra, the outer product %o% of vectors $x$ and $y$ is the product $xy^t$, while the matrix product %*% represents $x^t y$, and requires the vectors to be of equal length.

```
(A  <- matrix(1:12, ncol=3))
```

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

```
(B <- 9:10)
```

```
## [1]  9 10
```

## outer

```r
(AB <- A %o% B)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    9   45   81
## [2,]   18   54   90
## [3,]   27   63   99
## [4,]   36   72  108
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   50   90
## [2,]   20   60  100
## [3,]   30   70  110
## [4,]   40   80  120
```

## outer

A more flexible way of writing this function is

```
outer( array1, array2, FUN='*')
```

where `array1, array2` are the arrays and `FUN` is the operation that is going to be used to get the new array, which by default is multiplication, but can be any other operation or any function of two variables. Let's see examples

## outer

Suppose we want to evaluate the function

$$f(x,y) = \frac{1}{2\pi}\exp\Big\{-\frac{x^2+y^2}{2}\Big\}$$

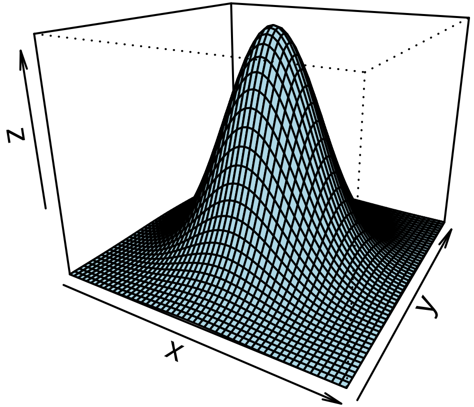
which is the standard bivariate normal distribution, in a square of sides $[-3,3]$. We use a mesh of 51 points in $[-3,3]$ and then define the function `f(x,y)` before doing the outer product:

```
x <- y <- seq(-3,3,length=51)
f <- function(x,y){
  exp(-(x^2+y^2)/2)/(2*pi)
  }
z <- outer(x,y,f)
str(z)
```

```
##  num [1:51, 1:51] 1.96e-05 2.80e-05 3.92e-05 5.42e-05 7.39e-05 ...
```

## outer

```r
persp(x,y,z,phi=20, theta=30,expand = 0.8,col='lightblue')
```



## outer

As a final example let's see how to produce a multiplication table:

```r
options(width = 70)
outer(1:9,1:9)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    1    2    3    4    5    6    7    8    9
##  [2,]    2    4    6    8   10   12   14   16   18
##  [3,]    3    6    9   12   15   18   21   24   27
##  [4,]    4    8   12   16   20   24   28   32   36
##  [5,]    5   10   15   20   25   30   35   40   45
##  [6,]    6   12   18   24   30   36   42   48   54
##  [7,]    7   14   21   28   35   42   49   56   63
##  [8,]    8   16   24   32   40   48   56   64   72
##  [9,]    9   18   27   36   45   54   63   72   81
```

`split`

## split

The function split divides the values of one variable according to the values of another. The result is a list that has as many elements as the second variable has different values. For example, the command

```
library(lattice)
barley.lst <- split(barley$yield, barley$site)
```

produces a list with six elements, since there are six experimental stations in the data. Each component of the list contains the yield for one of the experimental stations. The names of the components in the list correspond to the levels of the variable site.

## split

```r
library(lattice)
barley.lst <- split(barley$yield, barley$site)
str(barley.lst)
```

```
## List of 6
##  $ Grand Rapids    : num [1:20] 33 29.1 29.7 23 29.8 ...
##  $ Duluth          : num [1:20] 29 29.7 25.7 26.3 33.9 ...
##  $ University Farm: num [1:20] 27 43.1 35.1 39.9 36.6 ...
##  $ Morris          : num [1:20] 27.4 28.8 25.8 26.1 43.8 ...
##  $ Crookston       : num [1:20] 39.9 38.1 40.5 41.3 46.9 ...
##  $ Waseca          : num [1:20] 48.9 55.2 47.3 50.2 63.8 ...
```

```r
barley.lst[[2]]
```

```
##  [1] 28.96667 29.66667 25.70000 26.30000 33.93333 33.60000 28.10000
##  [8] 32.00000 33.06666 31.60000 22.56667 25.86667 22.23333 22.46667
## [15] 30.60000 22.70000 22.50000 31.36667 27.36667 29.33333
```

## `expand.grid`

## expand.grid

The function `expand.grid` creates a data frame with all possible combinations of vectors or factors that appear as arguments for the function.

```
expand.grid(age=c(20,40),weight=c(50,70),
            sex=c('Male','Female'))
```

```
##   age weight    sex
## 1  20     50   Male
## 2  40     50   Male
## 3  20     70   Male
## 4  40     70   Male
## 5  20     50 Female
## 6  40     50 Female
## 7  20     70 Female
## 8  40     70 Female
```

## Summary statistics

## Summary statistics

R has functions for calculating several statistics from a sample. For numerical data, we have, among others, the following

| Function | Operation |
|---|---|
| `mean()` | Average |
| `median()` | Median |
| `fivenum()` | 5 Number summary |
| `summary()` | Numerical summary |
| `min(), max()` | Smallest and largest value in the sample |
| `quantile()` | Sample quantiles |
| `var(), sd()` | Variance and standard deviation |
| `cov(), cor()` | Covariance and correlation |