Example Use cases

Michiel Noback

Example Use Cases

In this chapter, some example use cases will be presented demonstrating some concept or function. The topics for these use cases are selected because they appear to be harder to comprehend for my students, are a bit out of scope for the lectures, or because they are simply too extensive to fit into a few slides of a presentation.

Dataframe Selections

R offers a wealth of methods to make selection on dataframes by columns, rows, or both.

We'll explore the `iris` dataset, a dataframe holding morphological data on several species of plants from the genus *Iris*:

```
DT::datatable(iris)
```

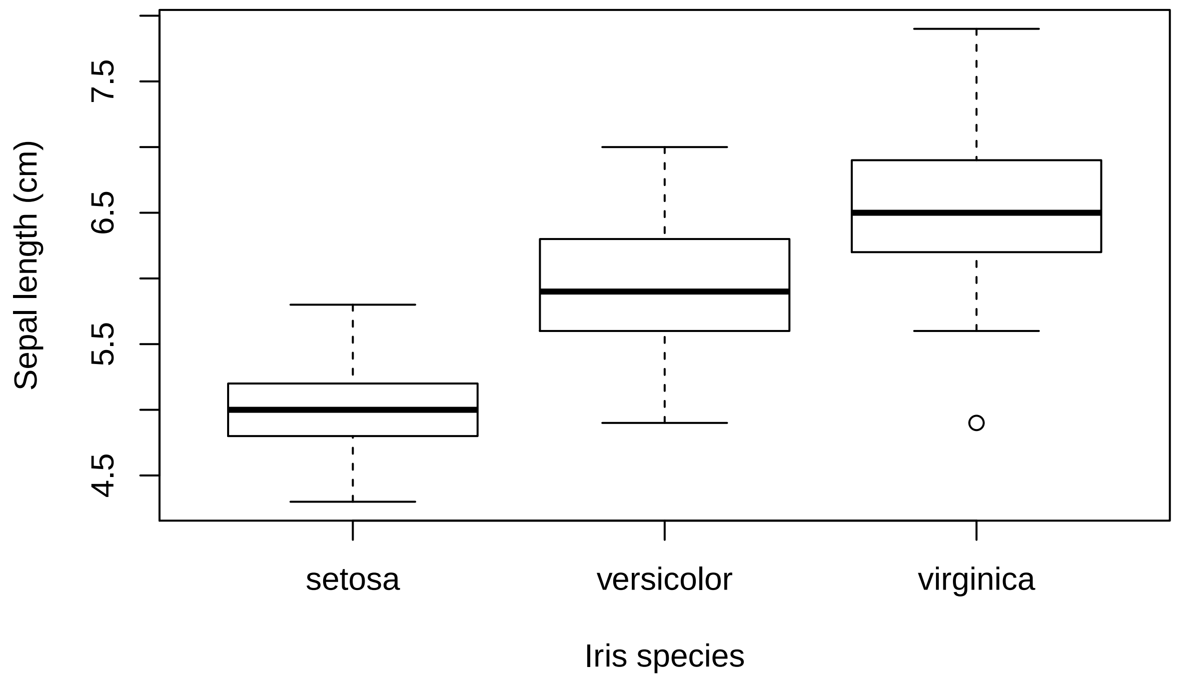
There are only three species in this dataset

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

but how do they relate to each other with repect to Sepal length?

```
with(iris, boxplot(Sepal.Length ~ Species,
                   ylab = "Sepal length (cm)",
                   xlab = "Iris species"))
```



Now suppose I want to get the data from *virginica* plants that have a Sepal length smaller than the largest Sepal length of *setosa* plants? First of course we'll need the maximum of the *setosa* plants:

```
max.setosa <- max(iris[iris$Species == "setosa", "Sepal.Length"])
max.setosa
```

```
## [1] 5.8
```

Which plant is it? Let's use the subset function to find out.

```
subset(x = iris,
       subset = (Species == "setosa" & Sepal.Length == max.setosa))
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 15          5.8           4          1.2         0.2  setosa
```

Now filter out the *virginica* plants that have a Sepal length smaller than this value. I'll show two approaches, one with logical indexing and one with `subset`

```
##get a logical for small plants
logi.small.sepal <- iris$Sepal.Length < max.setosa
logi.small.sepal
```

```
##   [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [12]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE
##  [56]  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE
##  [67]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
##  [89]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
## [100]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [122]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
##get a logical for virginica plants
logi.virginica <- iris$Species == "virginica"
logi.virginica
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [111]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [122]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [133]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [144]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
##combine the two via a boolean operation
logi.both <- logi.small.sepal & logi.virginica
logi.both
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [122]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
##use it as a selector on the rows of the iris DF
iris[logi.both, ]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 107          4.9         2.5          4.5         1.7 virginica
## 114          5.7         2.5          5.0         2.0 virginica
## 122          5.6         2.8          4.9         2.0 virginica
```

Of course, you will usually perform this selection in one statement, but the operations carried out by R will be exactly the same (but without creating any variables of course):

```
iris[iris$Sepal.Length < max.setosa & iris$Species == "virginica", ]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 107          4.9         2.5          4.5         1.7 virginica
## 114          5.7         2.5          5.0         2.0 virginica
## 122          5.6         2.8          4.9         2.0 virginica
```

The function `subset` will do the same behind the scenes, but your code may be more to your liking:

```
subset(x = iris,
       subset = Sepal.Length < max.setosa & Species == "virginica")
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 107          4.9         2.5          4.5         1.7 virginica
## 114          5.7         2.5          5.0         2.0 virginica
## 122          5.6         2.8          4.9         2.0 virginica
```

By the way, **beware to use only one boolean and: &, not &&**. This will not give an error but only an empty result set

```
subset(x = iris,
       subset = Sepal.Length < max.setosa && Species == "virginica")
```

```
## [1] Sepal.Length Sepal.Width  Petal.Length Petal.Width  Species
## <0 rows> (or 0-length row.names)
```

> & and && indicate logical AND and | and || indicate logical OR. The shorter form performs elementwise comparisons in much the same way as arithmetic operators. The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined. The longer form is appropriate for programming control-flow and typically preferred in if clauses.

Can you figure out why using `&&` would give an empty set in the above case?

See The R manual for details.

Apply

Consider the `women` dataset, holding height and weight of a population sample of 15 women:

```
DT::datatable(women,
              options = list("pageLength" = 15),
              colnames = c("Woman", names(women)))
```

To calculate the average height and the average weight of this sample, one could of course simply do

```
with(women, {
    print(mean(height))
    print(mean(weight))
})
```

```
## [1] 65
## [1] 136.7333
```

However, when your dataset has (a lot) more columns, repeating this will be quite tedious… unless you use a `for` loop

```r
for (i in 1:length(women)) {
    print(mean(women[,i]))
}
```

```
## [1] 65
## [1] 136.7333
```

Enter `apply()`, a very nice function to do this in a handy one-liner

```r
apply(X = women, MARGIN = 2, FUN = mean)
```

```
##   height   weight
##  65.0000 136.7333
```

The arguments I supplied to `apply`have the following purpose:

1. `X = women` specifies the data to be processed
2. `MARGIN = 2` specifies wether columns or rows shoud be processed; 1 = rows and 2 = columns
3. `FUN = mean` speciefies the function to be applied to the given dataframe

Not only gives apply the the exact same result (of course, duh), but this approach has several advantages:

- `apply` returns a named vector where the elements are named the same as the corresponding columns of the original dataframe
- `apply` is computationally more efficient than the other approaches
- it requires less code; a good programmer types as little as possible - except for Java programmers of course :-)

If you really have strongh feelings about typing no more than strictly required, you can of course also omit the method parameters:

```r
apply(women, 2, mean)
```

```
##   height   weight
##  65.0000 136.7333
```

But if you are just starting out with R, I suggest you invest those few character strokes for readability later on.

The above example dealt with columns. For instance, if you want to calculate the BMI of these women, you'll need to target the rows. The BMI formula is

$$weight/height^2 * 703$$

where weight is in pounds and height is in inches.

This formula is implemented in the following function.

```r
bmi <- function(height, weight) {
    (weight / height^2) * 703
}
bmi(65, 150)
```

```
## [1] 24.95858
```

You can also apply the formula to the `women` dataset:

```r
women$bmi1 <- apply(
    X = women,
    MARGIN = 1,
    FUN = function(x){(x[2] / x[1]^2) * 703})
head(women, n = 4)
```

```
##   height weight     bmi1
## 1     58    115 24.03240
## 2     59    117 23.62856
## 3     60    120 23.43333
## 4     61    123 23.23811
```

if you like to use your own formula (it's always a good idea to write logic only once and reuse it in different places), you'll still need to wrap it inside an anonymous function call:

```
women$bmi2 <- apply(
    X = women,
    MARGIN = 1,
    FUN = function(x){bmi(x[1], x[2])})
head(women, n = 4)
```

```
##   height weight     bmi1     bmi2
## 1     58    115 24.03240 24.03240
## 2     59    117 23.62856 23.62856
## 3     60    120 23.43333 23.43333
## 4     61    123 23.23811 23.23811
```

Processing Embedded Dataframes

Suppose you have imported some data that has a structure like this

```
genes <- c("gene A", "gene B", "gene C", "gene D")
positions <- c("chr01:128757:129667",
               "chr01:366389:486990",
               "chr02:8986463:9100856",
               "chr03:53536:87201")
my.genome <- data.frame(gene = genes, position = positions)
my.genome
```

```
##     gene              position
## 1 gene A   chr01:128757:129667
## 2 gene B   chr01:366389:486990
## 3 gene C chr02:8986463:9100856
## 4 gene D     chr03:53536:87201
```

The problem here is that the second column, `positions`, of type `character`, actually holds three different variables: the chromosome identifyer, the start position and the stop position on the chromosome. To be able to perform analyses of chromosomal contents, or positional contexts, we will need to split this column into separate columns, each holding exactly one variable of the correct type (`factor`, `integer` and `integer`).

When I first encountered this type of problem (it is a *challenge* actually, some teachers would object, not a *problem*...), my first thought was "easy, simply apply a split and bind as three columns".

Let's have a look at how the `strsplit` function works in splitting strings

```
strsplit(x = positions[1:2], split = ":")
```

```
## [[1]]
## [1] "chr01"  "128757" "129667"
##
## [[2]]
## [1] "chr01"  "366389" "486990"
```

As you can see, strsplit generates a list of vectors, with each vector corresponding to the string at the same index of the original character vector. So, easy, I thought. Simply assign these elements to three new columns of the original dataframe (assuming every split character results in a vector of three). I first created the columns, defined my splitter function and then used apply to get the job done

```r
## create columns
my.genome[, c("chromosome", "start", "stop")] <- NA
## define splitter function
loc.splitter <- function(x) {
    ## strsplit returns a list!
    strsplit(x["position"], ":")[[1]]
}
## use apply to fill the columns
my.genome[, 3:5] <- apply(X = my.genome,
                          MARGIN = 1,
                          FUN = loc.splitter)
my.genome
```

```
##     gene              position chromosome   start    stop
## 1 gene A   chr01:128757:129667      chr01  366389 9100856
## 2 gene B   chr01:366389:486990     128757  486990   chr03
## 3 gene C chr02:8986463:9100856     129667   chr02   53536
## 4 gene D     chr03:53536:87201      chr01 8986463   87201
```

Whoa, what happened here?! This was not what I had in mind. Can you figure out what happened?

...

I did figure it out (eventually...). The applied function returned three elements at a time, and I had apply fill three columns of my dataframe. And that is exactly what R did, fill the three columns, but not by row but by column! Have a look at the output from apply and you can see:

```r
apply(X = my.genome,
      MARGIN = 1,
      FUN = loc.splitter)
```

```
##      [,1]     [,2]     [,3]      [,4]
## [1,] "chr01"  "chr01"  "chr02"   "chr03"
## [2,] "128757" "366389" "8986463" "53536"
## [3,] "129667" "486990" "9100856" "87201"
```

Fortunately, R has a function to transpose this kind of structure (a matrix actually): the `t()` function, so that is what I did:

```r
my.genome[, 3:5] <- t(apply(X = my.genome,
                            MARGIN = 1,
                            FUN = loc.splitter))
my.genome
```

```
##     gene              position chromosome   start    stop
## 1 gene A   chr01:128757:129667      chr01  128757  129667
## 2 gene B   chr01:366389:486990      chr01  366389  486990
## 3 gene C chr02:8986463:9100856      chr02 8986463 9100856
## 4 gene D     chr03:53536:87201      chr03   53536   87201
```

Yeah, that's what I'm talking about! (Feeling very happy with myself... until I googled this problem). I found out there are a gazillion solutions to this problem, but only one of them is very very simple, because it uses a function you know really well: `read.table`, but not with the `file =` argument but with `text =`:

```r
my.genome <- data.frame(gene = genes, position = positions)
my.genome <- cbind(
    my.genome,
    read.table(
        text = as.character(my.genome$position),
        sep = ":"))
colnames(my.genome) <- c(colnames(my.genome)[1:2], "chr", "start", "stop")
my.genome
```

```
##     gene              position   chr   start    stop
## 1 gene A   chr01:128757:129667 chr01  128757  129667
## 2 gene B   chr01:366389:486990 chr01  366389  486990
## 3 gene C chr02:8986463:9100856 chr02 8986463 9100856
## 4 gene D     chr03:53536:87201 chr03   53536   87201
```

That's it. The lessons learned here:

- Always know that GIYF (Google Is Your Friend)
- When reading tables, also those embedded within others, use `read.table`
- You really learn a lot by fiddling about with data