R for Data Analysis in Agrobiodiversity
6

Automation and advanced scripting

Automation and advanced scripting

- Now you know a decent amount of R tricks
- Time to develop tools to handle heavy computation
- Enter the control structures!

if/else

for

while

Control structures are used to control and automate the execution of your code

- The if/else statement is used to set conditions under which the code may be or may not be run
- The for statement is used to run a code routine multiple times
- The while statement is used to run a code routine depending on a condition

Formula 1 is a competition for the fastest car (and pilot)

Before each race, a qualifying session takes place: the car making the fastest lap gets to start the race from the first position in the grid

Consider a qualification session of two teams

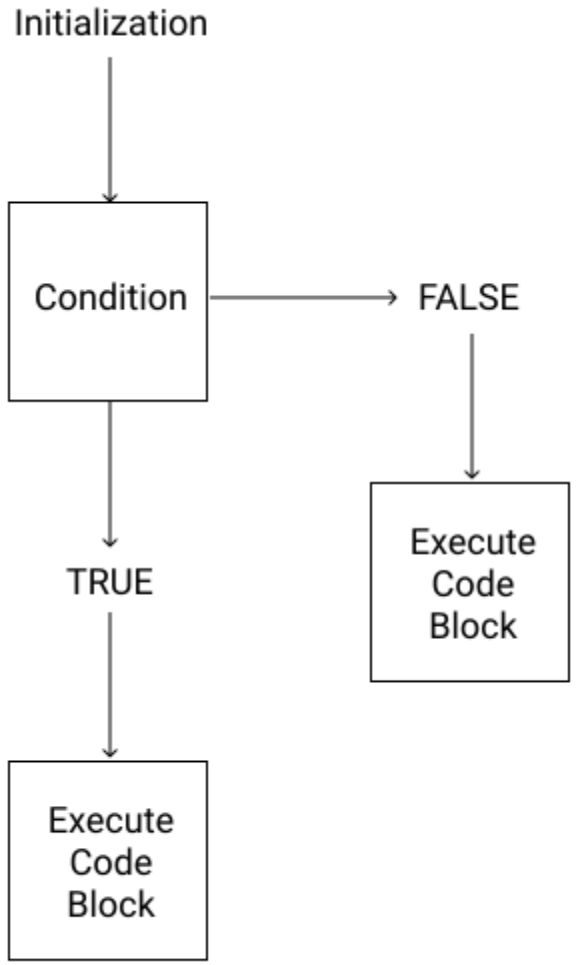
Mercedes VS Ferrari

Ferrari scores a shorter lap time → Ferrari gets the first position in the grid

Mercedes scores a shorter lap time → Mercedes gets the first position in the grid

If/else statement

```
if (condition){
    code block
} else{
    code block
}
```



```
mercedes<- 1.343
ferrari <-1.332

if (ferrari < mercedes){
        print ("Ferrari starts first")
    }

[1] "ferrari starts first"
```

The synthax of the if statement is the following:

if (condition TRUE) { do something }

- Note the multiple lines and indentation aiding readability
- No worries about spaces in the if statement
- When scripts get complex, it is a good idea to comment what the if statement is about after the closing curly bracket

What if Mercedes is fastest?

```
mercedes<- 1.330
ferrari <-1.332

if (ferrari < mercedes){
        print ("Ferrari starts first")
    } else {
        print ("Mercedes starts first")
    }

[1] "Mercedes starts first"
```

The synthax of the if/else statement is the following:

if (condition TRUE) { do something } else { do something else }

- The else condition will deal with **all** cases in which the initial if condition is FALSE
- Note that the if may also stand alone

What if we have more than two possible outcomes?

```
mercedes<- 1.330
ferrari <-1.332
redbull <- 1.334

if (ferrari < mercedes){
        print ("Ferrari starts first")
    } else if (redbull < mercedes) {
        print ("Redbull starts first")
    } else {
        print ("Mercedes starts first")
    }
[1] "Mercedes starts first"
```

The synthax of the if/else if/else statement is the following:

if (condition TRUE) { do something } else if (condition TRUE) { do something } else {do something else}

- You may add as many else if statements as you wish

```
mercedes<- 1.330
ferrari <-1.332
redbull <- 1.334

if (ferrari < mercedes){
        print ("Ferrari starts first")
    } else if (redbull < mercedes) {
        print ("Redbull starts first")
    } else {
        print ("Mercedes starts first")
    }
[1] "Mercedes starts first"
```

Remember: there are many different ways to do the same thing

```
mercedes<- 1.330
ferrari <-1.332
redbull <- 1.334
times<-c(mercedes, ferrari, redbull)
names(times)<-c("Mercedes", "Ferrari", "Redbull")
print(paste(names(times)[which(times==min(times))], "starts first"))

[1] "Mercedes starts first"
```

Let's try it

- Initialize an object that will hold your exam score
- Make an if statment so that:
  - if the vote is greater than 18, "passed" will be printed.
  - If the vote is less than 18, "try again" will be printed.
  - If the vote is outside the range [0,30], than "error" will appear.

Now let’s assume that we want to know the outcome of multiple qualification sessions

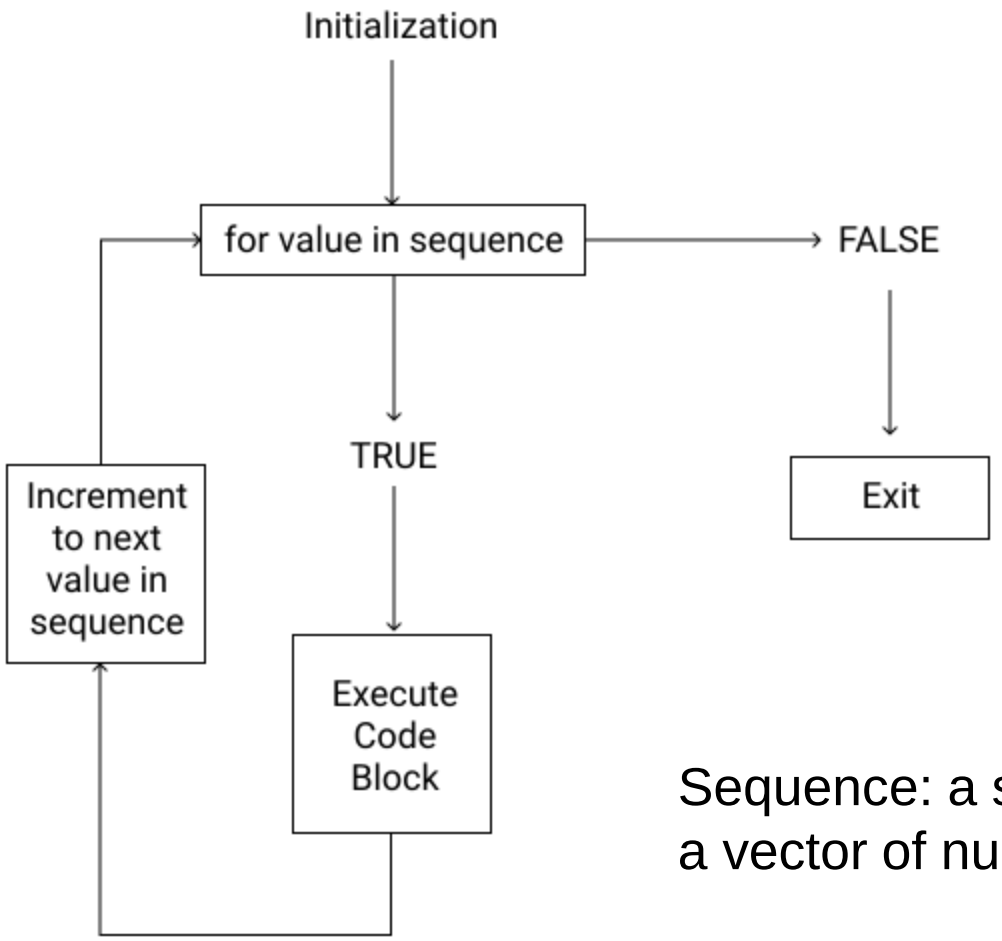
```
monza<-c(1.23, 1.42, 1.34)
suzuka <-c(1.56,1.50,1.55)
lemans <- c(2.31,2.33,2.43)
quals<- list(monza, suzuka, lemans)
quals
[[1]]
[1] 1.23 1.42 1.34

[[2]]
[1] 1.56 1.50 1.55

[[3]]
[1] 2.34 2.33 2.43
```

for loop

```
for (value in sequence){
     code block
}
```



Sequence: a set of objects. For example, a vector of numbers c(1,2,3,4,5).

Value: an **iterator variable** used to refer to each value in the sequence.

For each qualification session, report the shortest time

```
for (i in 1:3){
	gp<-quals[[i]]
	fastest<-min(gp)
	print(fastest)
}
```

- Note the multiple lines and indentation aiding readability
- No worries about spaces in the for statement
- When scripts get complex, it is a good idea to comment what the for loop is about after the closing curly bracket
- The iterator is an object that gets created at each loop: as such, it may be named as you wish («i» is a convention for «iterator»)

Lets put the data in a more useful format

```
monza<-c(1.23, 1.42, 1.34)
suzuka <-c(1.56,1.50,1.55)
lemans <- c(2.31,2.33,2.43)
quals<- data.frame(monza=monza, suzuka=suzuka, lemans=lemans)
rownames(quals)<-c(«Ferrari», «Mercedes», «Redbull»)
quals
```

	monza	suzuka	lemans
Ferrari	1.23	1.56	2.31
Mercedes	1.42	1.50	2.33
Redbull	1.34	1.55	2.43

A better for loop

```
for (i in 1:ncol(quals)) {
      fastest<-which(quals[,i]==min(quals[,i]))
      winner<-rownames(quals)[fastest]
      print(winner)
} # for i
```

for loop with if statement

```
for (i in 1:ncol(quals)) {
      fastest<-which(quals[,i]==min(quals[,i]))
      winner<-rownames(quals)[fastest]
      if (winner == "Ferrari"){
            print(paste(winner, "won! Yay!"))
      } else {
            print(paste(winner, "won. Sigh"))
      } #ifelse
} # for i
```

A nested for loop

```
for (i in 1:ncol(quals)) {
      tmp<-quals[,i]
      fastest<-which(tmp==min(tmp))
      winner<-rownames(quals)[fastest]
      for (j in 1:length(tmp)){
            print(paste(rownames(quals)[j],tmp[j], "min"))
      }#for j
      print(paste("therefore the winner is", winner))
} # for i
```

Adding the results of a for loop to an object

Initializing a list

```
winners<-list()
for (i in 1:ncol(quals)) {
      fastest<-which(quals[,i]==min(quals[,i]))
      winners[[i]]<-rownames(quals)[fastest]
}# for i
```

Using a vector

```
winners<-c()
for (i in 1:ncol(quals)) {
      fastest<-which(quals[,i]==min(quals[,i]))
      winners<-c(winners, rownames(quals)[fastest])
}# for i
```

Breaking a loop process or skipping a loop

The use of next

```
for (i in 1:ncol(quals)) {
      fastest<-which(quals[,i]==min(quals[,i]))
      winner<-rownames(quals)[fastest]
      if (winner != "Ferrari"){
            next
      } #if
      print(winner)
} # for i
```

The use of break

```
for (i in 1:ncol(quals)) {
      fastest<-which(quals[,i]==min(quals[,i]))
      winner<-rownames(quals)[fastest]
      if (winner != "Ferrari"){
            break
      } #if
      print(winner)
} # for i
```

Let's try

- Simulate a dice roll with the function *sample(1:6,1)*
- Then throw the dice 1K times and store results
- In the end, calculate how many times each face appeared using the function *table*

Bonus

- Make a chi squared goodness of fit test using *chisq.test(outcome, expected probabilities)*

Try it again

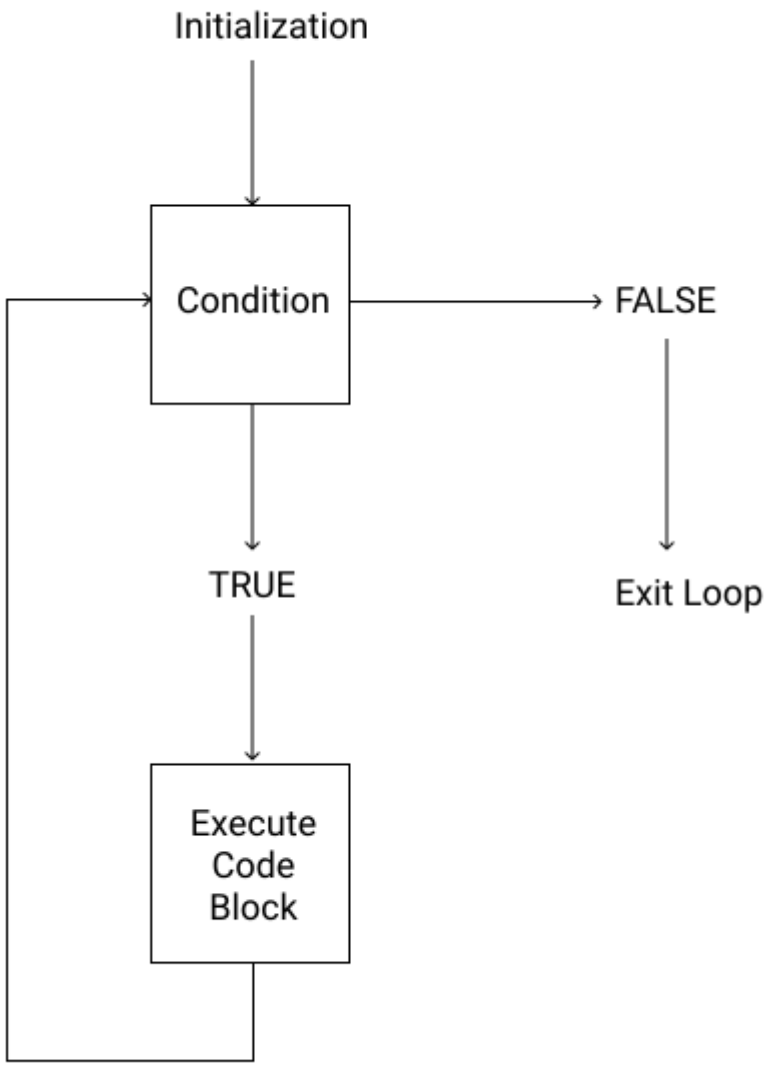
- Create a vector of 12 random numbers using *runif()*
- Arrange them in a matrix with 3 columns and 4 rows
- Create a nested for loop that gets and prints the value for each row in each column

Bonus

- Store the resulting values in a matrix with 4 columns and 3 rows, filling values by rows

while loop

```
while(condition){
    expression
}
```



A race goes for 10 laps

counters are very useful to keep track of loops

```
laps<-1
while (laps<=10){
	print(paste("lap number", laps))
	laps<-laps+1
}
```

Adding an if statement

```
laps<-1
while (laps<=10){
	if(laps<10){
		print(paste("lap number", laps))
	} else {
		print("End of race!")
	} #ifesle
	laps<-laps+1
} #while laps
```

- The while loop is very similar to the for loop, and indeed may be replaced by it
- The difference is that the while loop will check a logical condition, and keep running the loop as long as the condition is true

Try it

- You play a game flipping a coin with your friend, following these rules:
  - When you get heads, your money doubles
  - When you get tails, your money halves

- You start with 10 euros
- Create a while loop so that you continue playing until you have at least 0.5 euro
- In the end, print out the number of turns you played
- Optionally, print out also the maximum win you achieved

A few things to keep in mind

- Make sure to close all brackets
- Be careful in setting the operator range
- Make sure that the if statement is giving a logical value
- Unpack the loops to check for errors
- Add print statments in the loop to keep track of the process

```
for (i in 1:1e6) {
      if (i %% 1e5==0){
            print(i)
      }#if loop
      #do something
}#for
```

Check this link, especially the blocky game.
It is related to programming in general (not R), but very useful:
https://blockly-games.appspot.com/?lang=en

The power of automation: apply

When possibile, it is always better to vectorialize (faster!)

apply is a family of functions dealing with recurrent operations

apply() will apply a function to a matrix

apply(data, 1, mean)

The matrix on which to apply the function

The direction: 1 for rows, 2 for columns

The function: if it is a built in function the name is enough
Otherwise it may be stated as

function(x) "operation to be performed on x"
Where x is the vector resulting from the matrix

- apply() works on matrices
- lapply() works on lists, provides a list
- sapply() works on lists, provides a vector
- tapply() works on factor values in dataframes
- mapply() is the multivariate version of apply

Using a for loop

```
data(iris)
out<-vector()
for (i in 1:4){
    out[i]<-mean(iris[,i])
}
out
```

Time difference of 0.03868794 secs

Using apply

```
data(iris)
out<-apply(iris[,1:4], 2, mean)
out
```

Time difference of 0.01995778 secs

Using a dedicated function

```
data(iris)
out<-colMeans(iris[,1:4])
out
```

Time difference of 0.01797009 secs

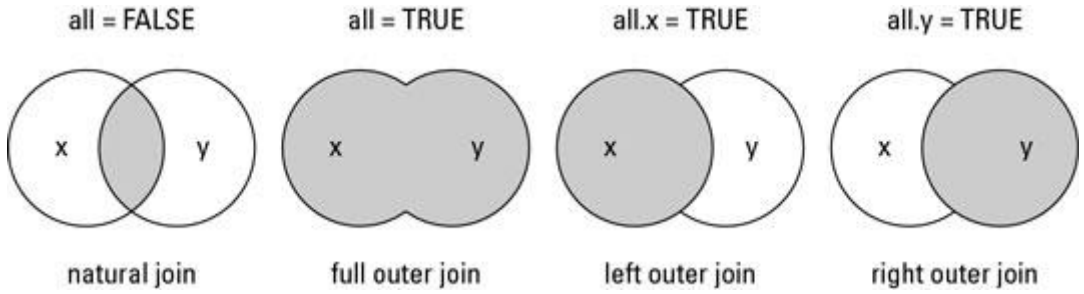
Merging datasets

A function to keep in mind is *merge()*

It will join two separate datasets into one, based on the values in specific columns or row names

Will automatically match entries; a big issue with datasets in my experience is the mismatch of lables caused by ill Excel sorting

merge(dataframe x, dataframe y, by.x="name of column", by.y="name of column", all=T)



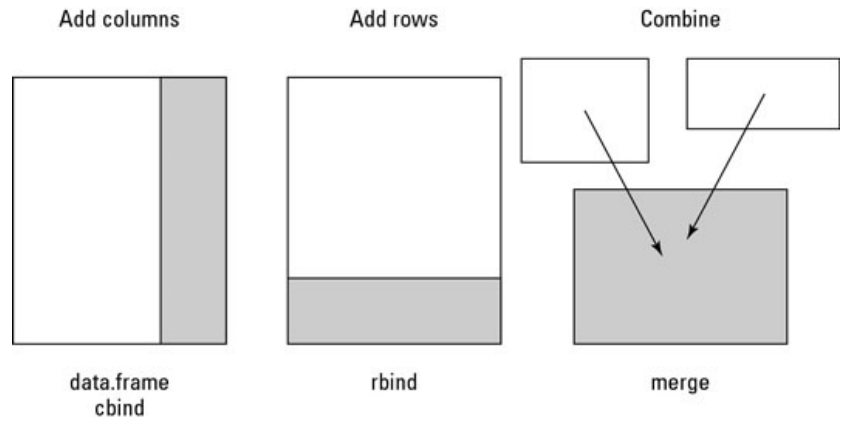
Simpler functions to join matrices by rows or columns are

rbind(dataframe x, dataframe y)
cbind(dataframe x, dataframe y)

provided that the dimensionality of dataframes x,y is identical

These functions may also be applied on lists with

do.call(rbind, your list)



Splitting dataframes

In some work routines it may be useful to split dataframes in chunks to conduct specific operations. You may do that using

split(dataframe, factor)

The object to split

The criteria to be used to split (e.g. value in column)

```
> data(mtcars)
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

From dataframe to list

```
> class(mtcars)
[1] "data.frame"
> class(bycyl)
[1] "list"
```

```
> bycyl<-split(mtcars, mtcars$cyl)
> head(bycyl)
$`4`
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2

$`6`
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
```

Splitting and looping to create summary stats

```
data(iris)
head(iris)
byspec<-split(iris,iris[,5])
means<-list()
for (i in 1:length(byspec)){
    tmp<-byspec[[i]]
    means[[i]]<-apply(tmp[,1:4],2,mean)
}
meandf<-do.call(rbind, means)
rownames(meandf)<-unique(iris[,5])
```

Operations with strings

In several istances when dealing with character entries (factors, strings, ...) it may be useful to asses the presence/absence of a certain patter and to manipulate it

Several functions, two mainly used:

sub("something", "by something else", somewhere), used to replace patterns

grep("something", somewhere), used to check the presence of patterns

```
> vec<-c("This", "course", "is", "R", "for", "ABD")
> vec
[1] "This"             "course"           "is"               "R"
[5] "for"              "ABD"
> sub("ABD", "agrobiodiversity", vec)
[1] "This"             "course"           "is"               "R"
[5] "for"              "agrobiodiversity"
```

sub will replace patterns recursively

```
> sub("s", "ZZZ", vec)
[1] "ThiZZZ"   "courZZZe" "iZZZ"     "R"        "for"      "ABD"
```

Unless you use regex expression to specify special cases

```
sub("s$", "ZZZ", vec)
[1] "ThiZZZ" "course" "iZZZ"   "R"
[5] "for"    "ABD"
```

```
> sub("^i.", "ZZZ", vec)
[1] "This"   "course" "ZZZs"     "R"
[5] "for"    "ABD"
```

\\d	Digit, 0,1,2 ... 9
\\D	Not Digit
\\s	Space
\\S	Not Space
\\w	Word
\\W	Not Word
\\t	Tab
\\n	New line
^	Beginning of the string
$	End of the string
\	Escape special characters, e.g. \\ is "\", \+ is "+"
|	Alternation match. e.g. /(e|d)n/ matches "en" and "dn"
•	Any character, except \n or line terminator
[ab]	a or b
[^ab]	Any character except a and b
[0-9]	All Digit
[A-Z]	All uppercase A to Z letters
[a-z]	All lowercase a to z letters
[A-z]	All Uppercase and lowercase a to z letters

grep will look for the occurence of a specific string and provide its position in the vector

```
> vec
[1] "This"            "course"          "is"              "R"
[5] "for"             "ABD"
> grep("is", vec)
[1] 1 3
```

Also in this case, regex expressions may be use to specify search options

```
> grep("\\bis\\b", vec)
[1] 3
```

```
> grep("^is$", vec)
[1] 3
```

Note: RegEx synthax is used across coding languages, there are plenty references on how to best use it, including https://regexr.com

Exercise

- Load NAM.data.toy
- Keep only columns starting with a "D" (phenology). Use grep
- Load NAM.data.disease.toy
- Merge the two dataframes; keep only entries appearing in both
- Divide the dataset by NAM family
- Make a loop calulating mean of each trait for each NAM family
- Make a plot showing mean values for each NAM family

For pros:

- Before entering the for loop use R/corrplot to produce a correlation plot among traits (save the image as pdf)
- In the for loop, add an if statement proding a tiff image of a boxplot of EtNAM family 1 traits