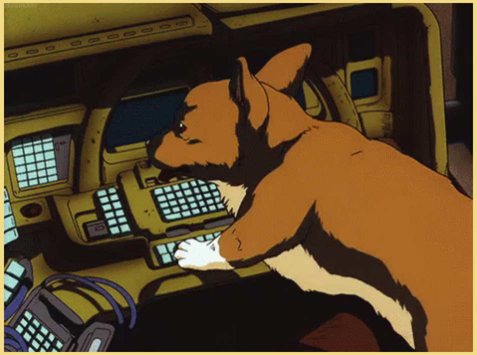
Introduction to R - part 2

🧉

Tiago Nardi

University of Pavia

Vectors



Let’s first create a **vector**, with 5 **elements**:

```r
x <- c(2, 5, 8, 1, 2)
```

Vectors

You can access elements inside a vector using their index position, using the square brackets *[ ]*

```
x[2] # show the second element
```

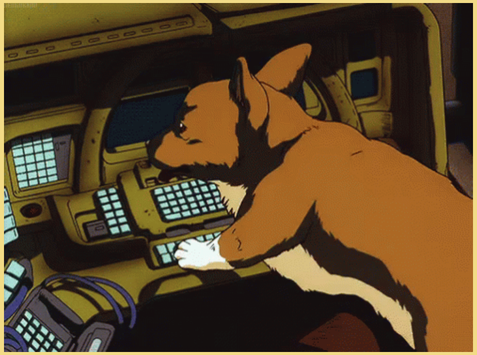
```
## [1] 5
```

To select multiple elements use the concatenate function

```
x[c(1,2)] # show the first and second element
```

```
## [1] 2 5
```

Indexing the vectors



Try these operations on your vector

```
vector[3:5]
vector[c(1,3:5)]
vector[-2]
vector[vector > 2]
```

Can you guess what you were extracting with these commands?

Indexing the vectors

You can select a slice of the vector using the colon *:*

```r
x[3:5] # show all elements from a starting index (3) to an ending index
```

```
## [1] 8 1 2
```

You can combine methods

```r
x[c(1,3:5)]
```

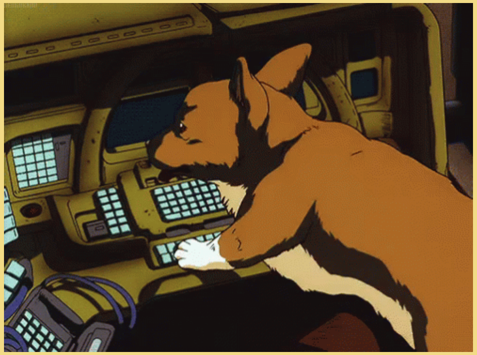
```
## [1] 2 8 1 2
```

You can get exclude elements putting a minus before them -

```r
x[-2]
```

```
## [1] 2 8 1 2
```

Indexing the vectors



Chose a numerical value and try these commands with you **vector**

```
vector[vector > value]
vector > value
```

Why these results are different? Can you guess it?

Write also a **vector** with only **boolean** values (TRUE/FALSE) with the same length of your first vector and use it to slice the first **vector**

```
vector2 <- c(FALSE,TRUE,TRUE,FALSE,FALSE)
vector[vector2]
```

Indexing the vectors

You can select elements in the vector using logical conditions

```
x[x > 2] # show all elements greater than 2
```

```
## [1] 5 8
```

To access the vector you have to specify the values *inside* the brackets.

Otherwise you apply the logical operation to all elements in the vector

```
x > 2
```

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

You can supply a logical vector to get only the elements in the positions with TRUE

```
mr_boole <- c(FALSE,TRUE,TRUE,FALSE,FALSE)
x[mr_boole]
```

```
## [1] 5 8
```

Vectors

As you can assign values to an **object**

You can assign values inside a vector

```
x[3]
```

```
## [1] 8
```

```
x[3] <- 11 # replace the third element with the number 11
x[3]
```

```
## [1] 11
```

Rows and columns

Let's load the usual table (remember to set up the **working directory**)

```
db <- read.csv("patric_redux.csv", header = T)
db1 <- db # let's make a copy of the data frame
head(db)
```

```
##            Species    ID Contigs Genome.Length GC.Content PATRIC.CDS
## 1 Yersinia pestis AS539     250       4572127       47.5       4398
## 2 Yersinia pestis AS147     277       4592682       47.5       4485
## 3 Yersinia pestis AS134     237       4572981       47.5       4378
## 4 Yersinia pestis AS509     263       4605070       47.5       4378
## 5 Yersinia pestis  A251     201       4593919       47.5       4507
## 6 Yersinia pestis   24H     140       4492822       47.6       4620
##   Isolation_location Source
## 1           Isengard Hobbit
## 2          The Shire Hobbit
## 3              Rohan  Human
## 4             Mordor  Human
## 5          The Shire Hobbit
## 6             Erebor Dragon
```

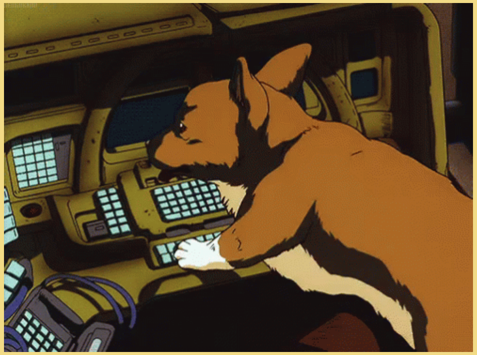
Rows and columns - Exercise

To get a column use the column number like a vector index

```
dataframe[n]
```

To change the name use the *colnames()* functions

```
colnames(dataframe)[n] <- "new column name"
```



Change the name of one column and look at the differences

Rows and columns - Exercise

```r
colnames(db1) # It shows column names
```

```
## [1] "Species"             "ID"                     "Contigs"
## [4] "Genome.Length"       "GC.Content"             "PATRIC.CDS"
## [7] "Isolation_location" "Source"
```

```r
colnames(db1)[4] <- "bp" # change the fourth column name
colnames(db1)
```

```
## [1] "Species"             "ID"                     "Contigs"
## [4] "bp"                  "GC.Content"             "PATRIC.CDS"
## [7] "Isolation_location" "Source"
```

```r
colnames(db1)[4] # Get only the name of the fourth column
```

```
## [1] "bp"
```

Rows and columns

To set row names:

```r
rownames(db1) # It shows row names
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [16] "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28" "29"
## [31] "31" "32" "33" "34" "35" "36" "37" "38" "39" "40" "41" "42" "43" "44"
## [46] "46" "47" "48" "49" "50" "51" "52" "53" "54" "55" "56" "57" "58"
```

```r
rownames(db1) <- db1$ID # We set as rownames all values from column
```

Rows and columns

```r
head(db) # read the first 6 rows
```

```
##             Species    ID Contigs Genome.Length GC.Content PATRIC.CDS
## 1 Yersinia pestis AS539     250       4572127       47.5       4398
## 2 Yersinia pestis AS147     277       4592682       47.5       4485
## 3 Yersinia pestis AS134     237       4572981       47.5       4378
## 4 Yersinia pestis AS509     263       4605070       47.5       4378
## 5 Yersinia pestis  A251     201       4593919       47.5       4507
## 6 Yersinia pestis   24H     140       4492822       47.6       4620
##   Isolation_location Source
## 1           Isengard Hobbit
## 2          The Shire Hobbit
## 3              Rohan  Human
## 4             Mordor  Human
## 5          The Shire Hobbit
## 6             Erebor Dragon
```

Rows and columns

```r
head(db1) # see the differences between this and the previous table
```

```
##                  Species    ID Contigs      bp GC.Content PATRIC.CDS
## AS539 Yersinia pestis AS539     250 4572127       47.5       4398
## AS147 Yersinia pestis AS147     277 4592682       47.5       4485
## AS134 Yersinia pestis AS134     237 4572981       47.5       4378
## AS509 Yersinia pestis AS509     263 4605070       47.5       4378
## A251  Yersinia pestis  A251     201 4593919       47.5       4507
## 24H   Yersinia pestis   24H     140 4492822       47.6       4620
##       Isolation_location Source
## AS539           Isengard Hobbit
## AS147          The Shire Hobbit
## AS134              Rohan  Human
## AS509             Mordor  Human
## A251           The Shire Hobbit
## 24H               Erebor Dragon
```

Cleaning up

To remove a column assign a **NULL** value to it

```
db1$Contigs <- NULL # delete the column
```

If you want to remove an **object** from R use

```
rm(x)
```

It will free your memory from that specified **object**

The **object** will disappear from the **environment tab**

Accessing the data frame

Extract columns and rows from a data frame (almost) like you access a **vector**

You can use the position indexes inside square brackets

First value is the row, the second the column

You can use position or names if you already set up row/column names

```
dataframe_name[1, 1]
dataframe_name["row_name", "column_name"]
```

If don't specify one of the two values you get all elements (All rows/All columns)

```
dataframe_name[,1] # all rows, first column
dataframe_name[1,] # first row, all column
```

Accessing the data frame

```
dataframe_name["row_name", "column_name"]
dataframe_name[row_index, column_index]
dataframe_name[ , "column_name"]
dataframe_name[ , column_index]
dataframe_name["row_name", ]
dataframe_name[row_index,]
```

Try to access:

the full row of sample *24H*

the *Source* of sample *EBD10-058*

the full column of *Isolation_location*

From the third to the 13th row the column *PATRIC.CDS"

All rows, excluding from the 4th row to the 10th, columns *bp GC.Content*

Accessing the data frame

```
db1["24H",]
```

```
##                Species  ID      bp GC.Content PATRIC.CDS Isolation_location So
## 24H Yersinia pestis 24H 4492822       47.6       4620             Erebor Dr
```

```
db1["EBD10-058","Source"]
```

```
## [1] "Hobbit"
```

```
db1[,"Isolation_location"]
```

```
##  [1] "Isengard"   "The Shire"  "Rohan"      "Mordor"     "The Shire" 
##  [6] "Erebor"     "Barad-dur"  "Mordor"     "Rohan"      "Rohan"     
## [11] "Lothlorien" "Rohan"      "Harad"      "Rohan"      "Gondor"    
## [16] "Gondor"     "Gondor"     "Gondor"     "Rohan"      "Gondor"    
## [21] "Gondor"     "The Shire"  "Gondor"     "Gondor"     "The Shire" 
## [26] "Harad"      "Rohan"      "Barad-dur"  "Barad-dur"  "Harad"     
## [31] "Gondor"     "The Shire"  "Lothlorien" "Lothlorien" "Erebor"    
## [36] "Erebor"     "Rivendell"  "Rivendell"  "Isengard"   "Lothlorien"
## [41] "Lothlorien" "Erebor"     "The Shire"  "Mirkwood"   "Gondor"    
## [46] "The Shire"  "Lothlorien" "Mordor"     "The Shire"  "Rivendell"
```

Accessing the data frame

```r
db1[1:13,"PATRIC.CDS"] #check the rows inspecting db1
```

```
##  [1] 4398 4485 4378 4378 4507 4620 4364 4570 4555 4605 4806 4720 4566
```

```r
db1[-(4:10),c("bp","GC.Content")]
```

```
##                     bp GC.Content
## AS539          4572127   47.50000
## AS147          4592682   47.50000
## AS134          4572981   47.50000
## M70            4889404   47.09967
## M71            4745596   47.10000
## M72            4743182   47.23000
## M74            4744232   47.16000
## M32            4692341   47.20000
## FORC_017       4723612   46.80000
## FORC_013       4749283   47.30000
## FORC_004       4695233   47.22671
## FDAARGOS_227 5073657   47.47000
## FORC_002       3924069   42.59000
## NCTC10460      4767810   46.91510
## NCTC10462      3840239   46.80498
```

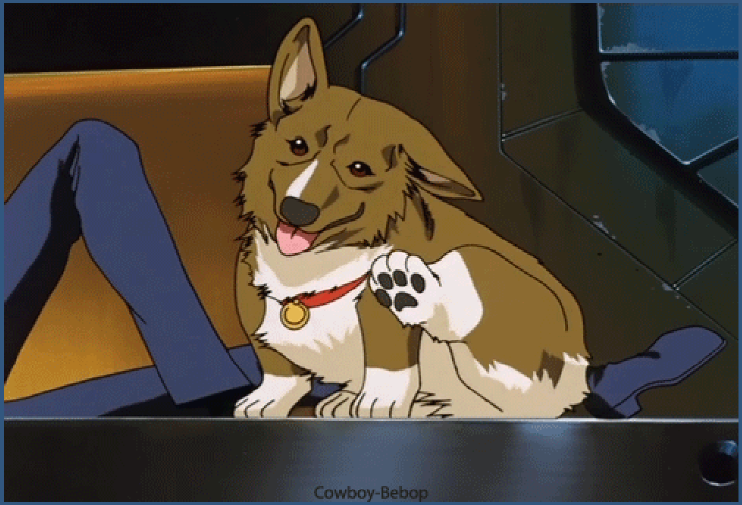
Accessing the data frame

```
dataframe_name[ , c("column_1_name", "column_2_name")]
dataframe_name[ , c(first_column_index, second_column_index)]
dataframe_name[c("row_1_name", "row_2_name"), ]
dataframe_name[c(first_row_index, second_row_index), ]
dataframe_name[ , c(first_column_index : last_column_index)]
dataframe_name[c(first_row_index : last_row_index),]
```

You can mix and match different ways to access the data

```
db[1,4] # First rown and 4th column
db[1,"Contigs"] #First row and column "Contigs"
db[35:78,] # From row 35 to 78 and all columns
db[,c(4,5)] #All rows and the 4th and 5th column
```

Remember the typeof() function?



Object type

With the various forms of extraction you get similar results but different data types

```
dataframe_name[column_index]
dataframe_name[[column_index]]
dataframe_name[, column_index] # Pay attention to the comma
dataframe_name$column_index
dataframe_name$column_index[row_index]
```

Choose a **column** and try the different options

Assign these values to a new **object**

Look at how they appear in the environment

What's the difference?

Object type

```r
typeof(db1[1])
```

```
## [1] "list"
```

```r
typeof(db1[, 1])
```

```
## [1] "character"
```

```r
typeof(db1[[1]])
```

```
## [1] "character"
```

```r
db1[1]
```

```
##                             Species
## AS539               Yersinia pestis
## AS147               Yersinia pestis
## AS134               Yersinia pestis
## AS509               Yersinia pestis
## A251                Yersinia pestis
## 24H                 Yersinia pestis
```

Not Available

In R, missing values are represented by the symbol **NA** (not available)

To test for missing values

```r
y <- c(1,2,3,NA)
is.na (y) #find which value is missing
```

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

To remove missing values from a data frame:

```r
db2 <- db1[complete.cases(db1), ] # remove all rows with NA values
```

Data frame functions

Table creates a tabular results of categorical variables

```
table(datafra$column)
```

```
table(db1$Isolation_location)
```

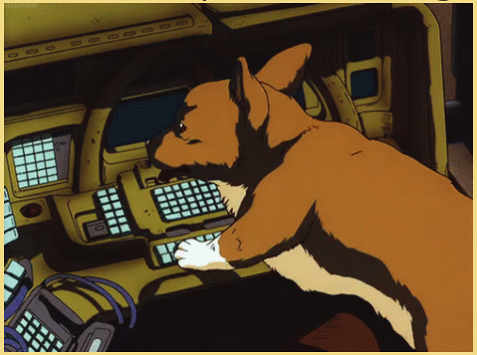
```
## 
##   Barad-dur      Erebor      Gondor       Harad   Isengard Lothlorien    Mirkwoo
##           3           4          11           4          2          6
##      Mordor   Rivendell       Rohan   The Shire
##           3           3           8          13
```

Data frame functions

gsub takes an input value and replace it with another one

```
vector <- gsub("Old string to replace","New string", vector )
```

You can replace the original vector, or assign it to a new vector



Try to replace something from the column *Isolation_location* and use *table* to inspect the results

```
db1$Isolation_location<-gsub("Barad-dur",
                             "Varallo Pombia",
                             db1$Isolation_location)
table(db1$Isolation_location)
```

```
##
##         Erebor          Gondor           Harad       Isengard     Lothlorien
##              4              11               4              2              6
##       Mirkwood          Mordor      Rivendell          Rohan      The Shire
##              1               3               3              8             13
## Varallo Pombia
##              3
```

Data frame functions

unique identify and eliminate duplicated values in an object. The object can be a vector or a data frame. On data frames the function removes duplicated rows

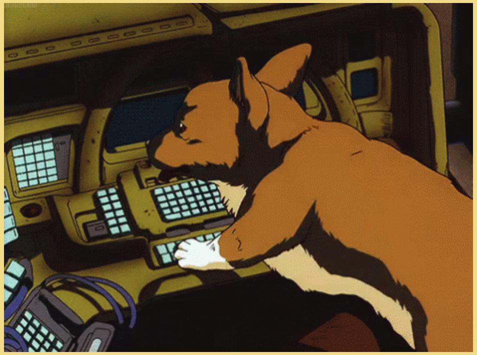
```
country <- db1$Isolation_location
unique_country <- unique(country)
length(country)
```

```
## [1] 58
```

```
length(unique_country)
```

```
## [1] 11
```

Functions on data frames



How to check if a value is present in a column or vector:

```
value %in% vector
```

To extract the first location that has the value of interest:

```
match("value", dataframe_name$column_name)
```

Use these functions to see if one value of your choice is present in the column *Isolation_location*, and see which row has the first occurrence

Functions on data frames

```r
"Varallo Pombia" %in% db1$Isolation_location
```

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

```r
match("Varallo Pombia", db1$Isolation_location)
```

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

```r
db1$Isolation_location[7]
```

```
## [1] "Varallo Pombia"
```

Subsetting and Binding

Cutting, pasting and modifying data frames

Subsetting

From a table, extract only a part of it, using a **pattern**

```
subset(dataframe_name, dataframe_name$column_name logical_condition)
```

For example, if we want to keep genomes not too fragmented we can select elements with less than 100 contigs

```
decent_qual <- subset(db, db$Contigs < 100)
```

You can use more than one logical condition using AND(&) OR (|), or just using two subsets

```
gondor <- subset(db,db$Contigs<100 &
                 db$Isolation_location=="Gondor")
```

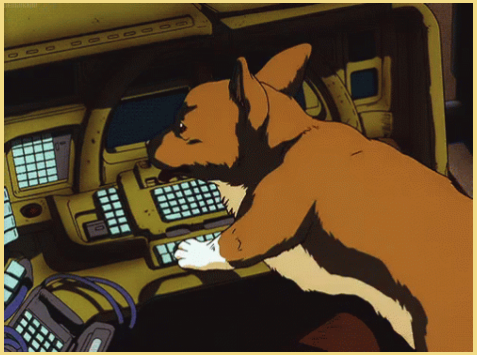
You can subset for columns:

```
red_table <- subset(db,select = c(1, 3, 4, 5))
```

Logical operators

Operator	Description
>	Greater than
>=	Greater than or equal tp
<	Lesser than
<=	Lesser than or equal to
==	Equal to
!=	Not equal to
&	AND
|	OR
!	NOT

Exercise



Get a table with only genomes from Elf and 4500000 as genome length

```
subset(dataframe_name, dataframe_name$column_name logical_condition)
```

Exercise

```
elfs <- subset(db,db$Genome.Length > 4500000 &
                    db$Source=="Elf")
```

Species	ID	Contigs	Genome.Length	GC.Content	PATRIC.CDS	Isolation_location	Source
Yersinia kristensenii	M70	81	4889404	47.09967	4806	Lothlorien	Elf
Yersinia enterocolitica	IP05342	169	4722666	46.77000	4935	Lothlorien	Elf
Yersinia enterocolitica	IP06077	159	4594630	46.79000	4743	Lothlorien	Elf
Yersinia pestis	YP173	3	4569957	47.53000	4644	Rivendell	Elf
Yersinia pestis	YP1313	235	4598307	47.53702	5203	Lothlorien	Elf
Yersinia pestis	YP2514	212	4629574	47.50026	4766	Lothlorien	Elf

Dropping levels

When subsetting data frames there may be some levels (for factors) that are unused

Remove them because they may alter downstream analyses

```
table(data_frame$column)
```

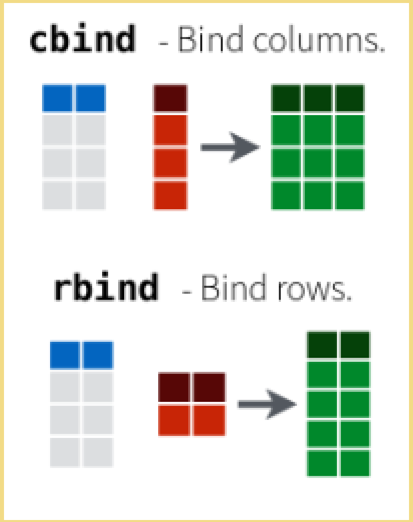
If you get a factor with a value of 0 you should drop the levels

```
dataframe_name <- droplevels (dataframe_name)
```

Binding

Combining vectors to a a dataframe:

- ***cbind*** **to bind a column to a dataframe**
- ***rbind*** **to bind a row to a dataframe**
- **vectors have to be of the same lenght**



Binding - Pasting

Paste combine strings to a single string, or two vectors of strings

```
new_vector<-paste(string_vector1,string_vector2,sep=" ")
```

In the new string/vector the original strings will be separated by the separator (*sep*=)

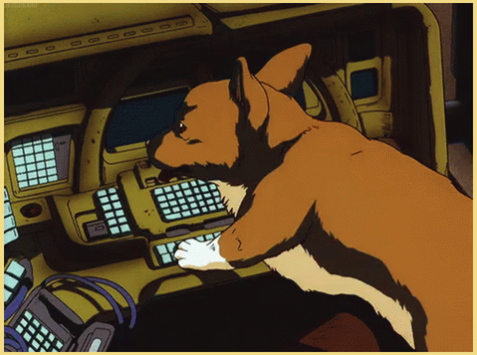
If you don't want any separation write *sep*=""

You can assign the new vector to a dataframe or use it to replace a column

You can use a fixed string

```
new_vector<-paste("fixed_string",string_vector2,sep="")
```

Exercise



Create 2 subsets:

- one with only samples from *Erebor*
- one with only samples from *Barad-dur*

Bind the two tables into a single dataframe, and add a column pasting *Isolation_location* and *Source* columns

```
df1 <- subset(dataframe,logical_vector)
new_df <- rbind(df1,df2) #new_r now is a new row of the data frame
new_vector<-paste(string_vector1,string_vector2,sep=" ")
```

Exercise

```r
Erebor <- subset(db, db$Isolation_location=="Erebor")
Barad_dur <- subset(db, db$Isolation_location=="Barad-dur")
db2 <- rbind(Erebor, Barad_dur)
db2$newco <- paste(db2$Isolation_location,db2$Source)
```

Species	ID	Contigs	Genome.Length	GC.Content	PATRIC.CDS	Isolation_location	Sourc
Yersinia pestis	24H	140	4492822	47.60000	4620	Erebor	Drag
Yersinia kristensenii	M53	90	4744530	46.90000	4797	Erebor	Dwar
Yersinia enterocolitica	IP00178	183	4609560	46.84000	4807	Erebor	Dwar
Yersinia pestis	YP1906	301	4573713	47.52974	4772	Erebor	Dwar
Yersinia pestis	EBD10-058	581	4542680	47.40000	4364	Barad-dur	Hobb
Yersinia enterocolitica	PS23	362	4639377	47.08801	4479	Barad-dur	Orc
Yersinia enterocolitica	FORC_003	202	4837317	47.04000	4580	Barad-dur	Orc

Sort

Sort a data frame in R (by default, sorting is ASCENDING)

```
dataframe_name <- dataframe_name[order(dataframe_name$column), ]
```

```
db2<- db2[order(db2$Contigs),] # sort the whole data frame by column “C
```

Vector functions

These can be applied in the same way on vectors

```
function(vector)
```

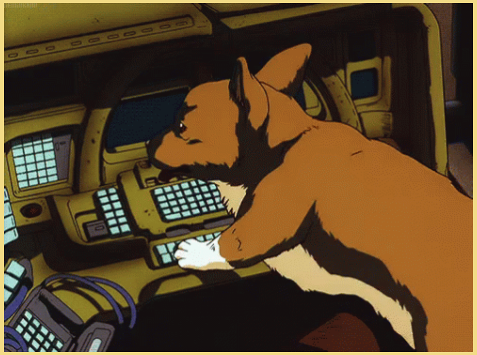
length returns the length of the vector (or the columns number in a dataframe)

max returns the maximum value of a numeric vector or column

min returns the minimum value of a numeric vector or column

range it outputs the smallest and largest values of the numeric vector or column

Exercise



- Get the range of the *GC.Content*
- Get the max value of contigs in only Hobbit samples
- Get the minimum value of *PATRIC.CDS* in samples with less than 100 contigs

Results

```
range(db$GC.Content)
```

```
## [1] 42.59000 48.33264
```

```
hobbit <- subset(db, db$Source == "Hobbit")
max(hobbit$Contigs)
```

```
## [1] 581
```

```
contig_100 <- subset(db, db$Contigs < 100)
min(contig_100$PATRIC.CDS)
```

```
## [1] 4124
```

Good things come in small packages

Install and load libraries

R, by default, only comes with a restricted number of functions

To extend the functionalities we can install external packages

To install a package in R:

```
install.packages("library_name")
```

after you have installed the package you can load it every time you need

```
library(library_name)
```

You have to load the packages every time your restart R

```
# There's a time and place for everything
# but do not run this code now
install.packages(“ggplot2”)
library(ggplot2)
```

```
install.packages("package")
```

library(package)

EXTRA

Libraries

- Arguably **tidyverse** is the most important set of packages, including:
  - *Dplyr*: makes data manipulation much more easier
  - *ggplot2*: extensive system to create graphics, extremely powerful
- *Bioconductor*: series of packages for biology and genomics
- *RColorBrewer*: useful tool to manage color palettes in R
- *Xaringan*: to use markdown to write slides (including these slide)

Dplyr

dplyr is a package for data manipulation, providing a consistent set of functions that help you solve the most common data manipulation challenges

```
install.packages (“dplyr”)
library (dplyr)
```

A common use is the *join* function

In order to combine two data frames, at least one column must be present in both data frames and have the same header

```
full <- full_join(db1,db2,by="Genome_ID") # all rows
inner <- inner_join(db1,db2,by="Genome_ID") # rows present in both
left <- left_join(db1,db2,by="Genome_ID") # all rows in db1
right <- right_join(db1,db2,by="Genome_ID") # all rows in db2
```