- You can also organize your work in projects within RStudio (an alternative way for project management).
  - I often use text-only terminals when I work on remote connections, so I don't use the GUI that much and I don't personally use the project functionality.

Additionally, you can also execute R files/ scripts in a `shell mode` (for instance on a remote terminal connection to utilize cloud computing or on a supercomputer).

- Any variable that you create is stored in the local environment in the local memory (local RAM).
- To see the variables that you defined:

```
ls() # lists all local variables
```

  - Or you can see it all in the RStudio's "Global Environment" window (upper right corner by default).

- To delete a single object in the memory use the `rm()` function.

```
rm(y)# deletes the variable y
```

To delete all variables in the memory use

```
rm(list=ls()) # deletes all objects in the
#memory. Warning you CANNOT undo!
```

***

- Terms "Folders" and "directories" can be used interchangeably although directory is an older term originally referring to a physical location on some physical storage device.
  - Directories are containers of files that point to the physical or logical location of each file.
    - The idea originated during the development of the original Unix (called Multics, Bell labs again) and now it is propagated in most computer based devices that we use.
- The directory structure is tree-like.

  - Most directories have a parent directory and possibly one or more child directories.

***

- Each time you start R it loads the `.RData` file from the current or the default directory (or it creates the file `.RData` if it is not already in the directory).
- When you end your session, you are asked if you want to save the data currently available in the memory.
  - This will overwrite the `.RData` file in the current directory. There is **no "undo"**.

- You can also save the environment to `.RData` when you want (in the current directory).
    - Again, there is **no "undo"**. This will overwrite the file in the current directory.

```
save.image() #save all variables in the
environment to .RData
```

- If you work in Linux `bash` or other shell, you can easily manage projects by simply changing the directory (folder) before you start your R session.
    - This does not work (automatically) in a Graphical User Interface (GUI) as R starts in the default directory.

- To see the directory that you are currently in R:

```
getwd() # prints the current directory
```

- To change the current directory (also called working directory) to an already existing directory use `setwd`.
- In Windows:
`setwd("C:/buff/Practicum")`
    - you might or might not need to provide the full path. For the scripts in my projects I use relative paths, e.g., "Vitiligo//GCTA" as opposed to the full path.
    - A relative path starts from the current directory, allowing you to easily move the branch to new locations if needed.

- In RStudio you could also hit < Ctrl + Shift + h> or < Cmnd + Shift + h> to see or set the working directory.
- Note that R still did load the `.RData` file with the environment <u>when it started</u> from the directory <u>it started</u> (the default directory when using GUI).
    - You can still manage projects by directories with the GUI.
        - A simple way to do so is to keep a clean (empty) `.RData` file in the default folder and `load()` the `.RData` after changing the directory to the working directory you desire.

```
setwd("C:/buff/Practicum"); load(".RData")
```

14. Importing Data (for the first time)

- The `read.table` function imports so called "flat files" into R as a data frame.
- Usage: `read.table(file, header = TRUE, sep = ",")`
- `file` is the filepath and name of the file you want to import into R
- If you don't know the full file path, use the dialog box in Rstudio or, in the function, type `file = file.choose()`. This will bring up a dialog box asking you to locate the file you want to import.

- `header` specifies whether the data file has a header (labels for each column of data in the first row of the data file).
  - If you don't specify this option in R or use `header=FALSE` or `header=F`, then R will assume the file doesn't have any headings.
  - `header=TRUE` or `header=T` tells R to read in the data as a data frame with column names taken from the first row of the data file.

- `sep` specifies the delimiter separating elements in the file.
  - If each column of data in the file is separated by a space, then use `sep = " "`
  - If each column of data in the file is separated by a comma, then use `sep = ","`
  - If each column of data in the file is separated by a tab, then use `sep = "\t"`.

Note that more information is available in the help file for `read.table()`, including usage examples.

- In a toy example, we will read 4 rows of data with header and tab separators.

- The file `example.txt` is available for download online,
  https://users.pfw.edu/yorgovd/IntroR/
  - save in your current R working directory.
  - if you are unsure what is it, use the `getwd()` function:

```
data <- read.table("example.txt",
     header = TRUE, sep = "\t")
data # the file is so small that you
#      don’t need to use the head command.
str(data) # 3 numeric variables
mean(data$Third) #mean of the variable Third

# hint: you could read.table directly from a
web location too. Let’s try this...
```

15. For Loops and Brute-Force

- For loops are a convenient way to cycle on a variable or index.
  - Note that for-loops are <u>not the most efficient</u> way to program in R!
    - If you have a large data set in a for loop, a copy is saved internally. You pay for this in terms of time and memory use.
    - R supports vectorized looping functions from the `apply()` family that avoid explicit use of loops.
      - If you do production coding for a computationally demanding project, you should do vectorizing.

- For most datasets, even those with several hundred thousand observations, you could use a for-loop in R on a contemporary computer.
    - **Just get the job done**…
      especially if you are a novice programmer**!**

- *DataCamp: Intermediate R / Chapter 2. Loops* was assigned as part of a previous lab.
    - Briefly, "for loop" in R scans through the elements in a vector x (in their original order) and runs the same code on each element of x.
    - for loops provide an intuitive way to do more complicated tasks that are not readily available in R functions.

A for loop in R can be based on a vector of any type.

```
x<-1:10
for(i in x){
#  Do stuff with i: the current element of x
}
```

- Write a for loop that prints all strings in the input vector

```
x <- c("Dan", "Mary", "Maria", "Lisa", "Susan",
"Linda")

for(i in x){
# Do stuff with i: the current element of x
}
```

- Let's consider one example *combining simulations, a for loop, and an if statement*.

A derangement is a permutation of an ordered set (rearrangement of its elements) where not a single element ends up in its original position.
(1,2,3,4,5)-> (3,2,1,5,4) # is one possible derangement

What is the probability to have a derangement if permuting (rearranging) a 7-elements ordered set?

- One could find the exact answer, $\frac{!7}{7!} = \frac{\#\ of\ derangements}{\#\ of\ permunations}$
    - You can code your own recursive functions (factorial and !n , are not available in base R it seems).
    - Or you can find a library with both.

- Here, as an exercise, we will do a quick simulation, permuting the numbers from 1 to 7 and checking if each permutation is a derangement or not.
    - We will repeat many times counting the number of "hits".
    - A for loop will give us the estimated (experimental) probability.

```
# INIT
# set the number of elements
# set the number of trials
# set a counter for the # of derangements
# in the for loop we will increase
# this counter if a derangement is observed
# Let’s build the inside of the for loop!
# A counter inside of the loop will increase
# if you have a derangement.
  # == will return TRUEs for same positions
  # TRUE is converted to 1 if you sum
  # so you count the number of elements
  # in the same position
  # if sum == 0 (no same locations),
  # we have a derangement and so
  # Increase the counter d


# Outside of the loop, compute the
estimated, # experimental probability
# total derangements over total # of trials
```

- This was an experimental probability, approximating ! 7/7!.
  - If we want to obtain an answer closer to the exact, probability, we can **increase the number** of trials.
  - We can get **arbitrarily close**. That is, we can get as close as we want to the theoretical value by increasing the number of trials.

***

- Some questions are so hard that you can tackle them only with a <u>brute-force approach</u>, traversing all possibilities.
  - Exhaustive search is the “official name” for brute-force…

16. Functions

- A function is essentially a sequence of commands executed based on certain arguments supplied to the function.

In R, a function is defined using the general format:

```
myfunction <- function(arg1, arg2, arg3){
  code to execute
}
```

- The name of the function here is "*myfunction*", and to use this function, we need to supply 3 arguments.

- The body of function is between curly brackets { }.

- To call the function, you type the name and provide the arguments:

```
myfunction(10, 3.17, "PFW")
```

Default values for the input could be provided by the function so the arguments can be omitted when making a function call.

```
myfunction <- function(arg1, arg2=3,
arg3="Purdue Fort Wayne")
```

- R is very flexible. For instance, a function may or may not return something back that you can store for later use.
  - Typically, it does :)

- Why you might want to write your own functions?
    - You can recycle the same commands many times on different inputs.
    - You can produce cleaner to read code.
    - Easier debugging (modular programming).
    - Work on more complex projects that might be built into a package.

Example of a function that returns the density of a normal random variable with mean mu and standard deviation sigma for a vector x. R function is dnorm().

The arguments are:
- x, the vector of values at which I want to determine the density
- mean, the mean of the normal distribution
- sigma, the standard deviation of the normal distribution

```
normal.density <- function(x, mu = 0, sigma = 1)
{
return(exp(-(x-mu)^2/(2*sigma^2))/sqrt(2*pi*sigma^2))
}

# let’s compare to the built-in density function
```

Example of a function `stdev` *that returns the standard deviation of a vector x. R function is sd().*

The sole argument is `x,` the vector of values for which we want to determine the standard deviation. Assume sample data, i.e., divide by (n-1).

```
stdev <- function(x)
{
   s <- # compute in s
   s # return s
}
z <- rnorm(20) # 20 random values N(0,1)
stdev(z)
sd(z) # the stdev() and  sd() gives the same
result
```

Example: Create function that returns the mean and standard deviation of a vector x.

The sole argument is:
- x, the vector of values for which we want to determine the mean and standard deviation

```
ms <- function(x){
   m <- mean(x)
   s <- sd(x)
   return(list(m = m, s = s))
}
y <- 101:110
rslt <- ms(y)
rslt
rslt$m-mean(y) # should give 0
```