Exercise1 - Intro to R tutorial

VISHAL BHASHYAAM

Table of contents

0.1 Introduction to R

0.1.1 Part I

Finding the version of R

```
R.version
```

```
               _
platform       x86_64-w64-mingw32
arch           x86_64
os             mingw32
crt            ucrt
system         x86_64, mingw32
status
major          4
minor          3.1
year           2023
month          06
day            16
svn rev        84548
language       R
version.string R version 4.3.1 (2023-06-16 ucrt)
nickname       Beagle Scouts
```

0.2 Packages

Installing “DMwR2” package to use Data mining in R, use `install.packages("package name")`

```
if(!require("DMwR2"))
install.packages("DMwR2",repos='http://cran.us.r-project.org')
```

```
Loading required package: DMwR2
```

```
Registered S3 method overwritten by 'quantmod':
  method            from
  as.zoo.data.frame zoo
```

Check what is available in the package, use `help()`, a window will appear with the documentation of the package.

```
help(package="DMwR2")
```

Now the package is installed in the computer or server(posit cloud). To use the function there are two ways:

(1) when function is called frequently, you need to load it to the current session by using `library()`

```
library(DMwR2)
```

Now you can use any function or dataset provided in `DMwR2` by referencing its name directly.

```
data (algae)
algae
```

```
# A tibble: 200 x 18
   season size  speed   mxPH  mnO2    Cl    NO3   NH4  oPO4   PO4  Chla    a1
   <fct>  <fct> <fct>  <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 winter small medium  8      9.8  60.8  6.24  578   105   170   50      0
 2 spring small medium  8.35   8    57.8  1.29  370   429.  559.   1.3    1.4
 3 autumn small medium  8.1   11.4  40.0  5.33  347.  126.  187.  15.6    3.3
 4 spring small medium  8.07   4.8  77.4  2.30   98.2  61.2 139.    1.4    3.1
 5 autumn small medium  8.06   9    55.4 10.4   234.   58.2  97.6 10.5    9.2
 6 winter small high    8.25  13.1  65.8  9.25  430    18.2  56.7 28.4   15.1
 7 summer small high    8.15  10.3  73.2  1.54  110    61.2 112.    3.2    2.4
 8 autumn small high    8.05  10.6  59.1  4.99  206.   44.7  77.4  6.9   18.2
 9 winter small medium  8.7    3.4  22.0  0.886 103.   36.3  71    5.54  25.4
10 winter small high    7.93   9.9   8    1.39    5.8  27.2  46.6  0.8   17
```

```
# i 190 more rows
# i 6 more variables: a2 <dbl>, a3 <dbl>, a4 <dbl>, a5 <dbl>, a6 <dbl>,
#   a7 <dbl>
```

```
manyNAs(algae)
```

```
[1]  62 199
```

`library()` without arguments, provides list of packages loaded in the computer.

```
library()
```

Show packages loaded in the current session:

```
(.packages())
```

```
[1] "DMwR2"     "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
```

If wrong package is loaded, we use `detach()` to remove from the session

```
if(!require("dbplyr"))
install.packages("dbplyr",repos='http://cran.us.r-project.org',ask=FALSE)
```

```
Loading required package: dbplyr
```

```
# installing an package so that it can removed later if not needed
library(dbplyr)
# loading the package to the session
```

```
(.packages())
```

```
[1] "dbplyr"    "DMwR2"     "stats"     "graphics"  "grDevices" "utils"
[7] "datasets"  "methods"   "base"
```

```r
# checks all the package in the current session

# now we dont want the package as it throws to much conflict, so we remove from the sessio
detach("package:dbplyr", unload=TRUE)
(.packages())
```

```
[1] "DMwR2"     "stats"     "graphics"  "grDevices" "utils"     "datasets" 
[7] "methods"   "base"     
```

```r
library(dplyr) #load the wanted library
```

```

Attaching package: 'dplyr'


The following objects are masked from 'package:stats':

    filter, lag


The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

Another way to see the installed packages in the system

```r
installed.packages()[1:5,]
```

```
          Package     LibPath                                         Version
backports "backports" "C:/Users/ASUS/AppData/Local/R/win-library/4.3" "1.4.1"
base64enc "base64enc" "C:/Users/ASUS/AppData/Local/R/win-library/4.3" "0.1-3"
bit       "bit"       "C:/Users/ASUS/AppData/Local/R/win-library/4.3" "4.0.5"
bit64     "bit64"     "C:/Users/ASUS/AppData/Local/R/win-library/4.3" "4.0.5"
blob      "blob"      "C:/Users/ASUS/AppData/Local/R/win-library/4.3" "1.2.4"
          Priority Depends                                              
backports NA       "R (>= 3.0.0)"                                       
base64enc NA       "R (>= 2.9.0)"                                       
bit       NA       "R (>= 2.9.2)"                                       
bit64     NA       "R (>= 3.0.1), bit (>= 4.0.0), utils, methods, stats"
blob      NA       NA                                                   
          Imports                            LinkingTo
```

```
backports NA                                  NA
base64enc NA                                  NA
bit       NA                                  NA
bit64     NA                                  NA
blob      "methods, rlang, vctrs (>= 0.2.1)" NA
          Suggests
backports NA
base64enc NA
bit       "testthat (>= 0.11.0), roxygen2, knitr, rmarkdown,\nmicrobenchmark, bit64 (>= 4.0.0
bit64     NA
blob      "covr, crayon, pillar (>= 1.2.1), testthat"
          Enhances License              License_is_FOSS License_restricts_use
backports NA       "GPL-2 | GPL-3"      NA              NA
base64enc "png"    "GPL-2 | GPL-3"      NA              NA
bit       NA       "GPL-2 | GPL-3"      NA              NA
bit64     NA       "GPL-2 | GPL-3"      NA              NA
blob      NA       "MIT + file LICENSE" NA              NA
          OS_type MD5sum NeedsCompilation Built
backports NA      NA     "yes"            "4.3.0"
base64enc NA      NA     "yes"            "4.3.0"
bit       NA      NA     "yes"            "4.3.1"
bit64     NA      NA     "yes"            "4.3.1"
blob      NA      NA     "no"             "4.3.1"
```

Check if there is any outdated packages installed

```
old.packages(repos='http://cran.us.r-project.org')
```

```
           Package      LibPath
dplyr      "dplyr"      "C:/Users/ASUS/AppData/Local/R/win-library/4.3"
KernSmooth "KernSmooth" "C:/Program Files/R/R-4.3.1/library"
Matrix     "Matrix"     "C:/Program Files/R/R-4.3.1/library"
mgcv       "mgcv"       "C:/Program Files/R/R-4.3.1/library"
nlme       "nlme"       "C:/Program Files/R/R-4.3.1/library"
spatial    "spatial"    "C:/Program Files/R/R-4.3.1/library"
survival   "survival"   "C:/Program Files/R/R-4.3.1/library"
           Installed Built   ReposVer
dplyr      "1.1.2"   "4.3.1" "1.1.3"
KernSmooth "2.23-21" "4.3.1" "2.23-22"
Matrix     "1.5-4.1" "4.3.1" "1.6-1"
mgcv       "1.8-42"  "4.3.1" "1.9-0"
nlme       "3.1-162" "4.3.1" "3.1-163"
```

```
spatial    "7.3-16"  "4.3.1" "7.3-17"
survival   "3.5-5"   "4.3.1" "3.5-7"
           Repository
dplyr      "http://cran.us.r-project.org/src/contrib"
KernSmooth "http://cran.us.r-project.org/src/contrib"
Matrix     "http://cran.us.r-project.org/src/contrib"
mgcv       "http://cran.us.r-project.org/src/contrib"
nlme       "http://cran.us.r-project.org/src/contrib"
spatial    "http://cran.us.r-project.org/src/contrib"
survival   "http://cran.us.r-project.org/src/contrib"
```

Update all the installed packages to its newer version on CRAN

```
# update.packages()
```

Update all the installed package without asking to confirm everytime(Still the process takes a long time)

```
#update.packages(ask=FALSE)
# all the packages are upto date, each time it takes a long time to run this function, so
```

Type a function name to see if it is included in the installed packages, e.g, **mean** is in base R:

Again use `help()` to find the documentation of the method

```
mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x00000279e8291500>
<environment: namespace:base>
```

```
help(mean)
```

```
starting httpd help server ... done
```

When you want to see if a package you need to use has already been made, search for it using some keywords inside the method `RSiteSearch()`

```
RSiteSearch('neural networks')
```

```
A search query has been submitted to https://search.r-project.org
The results page should open in your browser shortly
```

0.3 Project and Session Management

Use Project to manage Rscripts and data

`File>New Projects` to create a new folder for the project

`File>Open Project` to resume your current working workspace or project

Your project folder is the current working directory, where you save the `.R` and `.RData` files

`.R` can exist outside a project/ project folder

`Close a Project` to close the current project, but keep the current session

Quit Session closes the current Rstudio window

You can type all the commands in a text file and save it, then use [1] `source('path_to_mycode.R')` to execute the series of commands or [2] open `mycode.R` in RStudio script tab and execute your commands from there using `Run` or `Source` button.

`Run:` run the code line by line

`Source:` run the entire script

Often we need to sova data and functions for later use so we `save(), load()`

```r
# save(my.function, mydataset, file="path_to_mysession.RData")
# load("path_to_mysession.RData")
```

0.4 Save all objects

All objects are stored in `.RData` file, for us to load in the future .

```r
save.image()
```

Run `getwd()` and `setwd()` to show current working directory and set the current working directory.

```r
getwd()
```

```
[1] "C:/Users/ASUS/Desktop/R-python-exercise1-vishal bhashyaam/exercise1-R-python-tutorial-vi
```

```
#setwd(), im not using it, since im using RStudio in posit cloud
```

0.5 R Objects and Variables

Variables points out to memory location in a computer memory that holds some objects, they are like storage/container, it can hold numerical, character, strings to any complex model to associate an object to a variable, below is an example of an variable

```
vat <- 0.2
```

See what `vat` holds

```
vat
```

```
[1] 0.2
```

More examples of what a variable can do:

```
(vat <- 0.2)
```

```
[1] 0.2
```

```
x <- 5
y <- vat * x
y
```

```
[1] 1
```

```
z<-(y/2)^2
y
```

```
[1] 1
```

```
z
```

```
[1] 0.25
```

List the current available variables: `ls()` or `objects()`

```
ls()
```

```
[1] "algae"            "algae.sols"       "has_annotations" "test.algae"
[5] "vat"              "x"                "y"               "z"
```

```
objects()
```

```
[1] "algae"            "algae.sols"       "has_annotations" "test.algae"
[5] "vat"              "x"                "y"               "z"
```

Remove a variable to free memory use `rm`

```
rm(vat)
```

0.6 R Functions

Functions are a special type of R object designed to carry out some operation. Functions expects some input arguments and outputs results of it operation. R has many functions already, libraries you loaded contains functions you can use, you can also create new functions.

Examples of R functions:

```
max(4,5,6,12,-4)
```

```
[1] 12
```

```
mean(4,5,6,12,-4)
```

```
[1] 4
```

Same function different results, because we include another function inside the `max` function, `sample` generates a random sample of specied range of numerical value.

```
max(sample(1:100,30))
```

```
[1] 100
```

```r
mean(sample(1:100,30))
```

```
[1] 57.96667
```

We use `set.seed()`

Next time when we try to reproduce the same code it gives the same random numerical values, easy to debug the code values.

To create a new function, `se` (standard error of means), first test if `se` exists in our current environment.

```r
set.seed(1)

rnorm(1)
```

```
[1] -0.6264538
```

```r
rnorm(2)
```

```
[1]  0.1836433 -0.8356286
```

```r
set.seed(2)
rnorm(2)
```

```
[1] -0.8969145  0.1848492
```

```r
rnorm(2)
```

```
[1]  1.587845 -1.130376
```

```r
exists("se")
```

```
[1] FALSE
```

Until now no function `se` exists, creating one to calculate the standard error of sample

```
se <- function(x){
  variance <- var(x)
  n <- length(x)
  return(sqrt(variance/n))

  }
```

object `se` has been created :

```
exists("se")
```

```
[1] TRUE
```

Creating function with multiple arguments:

this function is used to convert inches to meters,feet,yard,miles

```
convInch <- function (x, to="meter"){
  factor = switch(to, meter=0.0254, foot=0.0833333, yard=0.0277778, mile=1.57828e-51, NA)
  if(is.na(factor)) stop ("unknown target unit")
  else return (x*factor)
}
convInch(23, "foot")
```

```
[1] 1.916666
```

If no argument is given it automatically converts to `meter`(IT IS DEFAULT)

```
convInch(40)
```

```
[1] 1.016
```

```
convInch(70,"yard")
```

```
[1] 1.944446
```

Order of the arguments can be shuffled, if required parameters match

```
convInch(to="meter",70)
```

```
[1] 1.778
```

0.7 Factors

Conceptually, factors are variables in R which take on a limited number of different values. A factor can be seen as a categorical (i.e., nominal) variable factor levels are the set of unique values the nominal variable could have. Factors are different from characters.

To create a factor, use `factor()`. Factors are represented internally as numeric vectors. This factor has two levels, `f` and `m`:

```
g <-c('f', 'm', 'f', 'f', 'f', 'm', 'm', 'f')

g <- factor(g)
```

More compact way to creating a factor with known levels, f and m:

```
other.g <-factor(c('m', 'm', 'm', 'm'), levels= c('f', 'm'))
other.g
```

```
[1] m m m m
Levels: f m
```

Comparing the above:

```
other.g <-factor(c('m', 'm', 'm', 'm'))
other.g
```

```
[1] m m m m
Levels: m
```

Factors are extremely useful for nominal values. Use factor to illustrate the concept of marginal frequencies or marginal distributions and `table()` function:

```
g <- factor(c('f', 'm', 'f', 'f', 'f', 'm', 'm', 'f'))
table(g)
```

```
g
f m
5 3
```

Add an age factor to the table (table can have more than two factors):

```
a <- factor(c('adult', 'juvenile','adult', 'juvenile','adult', 'juvenile','juvenile', 'juv
table(a, g)
```

```
          g
a          f m
  adult    3 0
  juvenile 2 3
```

R assumes the values at the same index in the two factors are associated with the same entity.
In our dataset, we have 3 female adult, 2 female juvenile, and 3 male juvenile.

What if the a factor is nto the same length as `g` factor?

It throws an error showing all arguments must have the same length

```
a <- factor(c('adult', 'juvenile','adult', 'juvenile','adult', 'juvenile','juvenile'))
table(a, g)
```

```
Error in table(a, g): all arguments must have the same length
```

Bring back the correct number of arguments for a and create a new table with factor `g`

```
a <- factor(c('adult', 'juvenile','adult', 'juvenile','adult', 'juvenile','juvenile', 'juv
t <- table(a, g)
t
```

```
          g
a          f m
  adult    3 0
  juvenile 2 3
```

Find marginal frequencies for a factor:

```
margin.table(t, 1)
```

```
a
   adult juvenile 
       3        5 
```

```
margin.table(t,2)
```

```
g
f m 
5 3 
```

```
t
```

```
          g
a          f m
  adult    3 0
  juvenile 2 3
```

We can also find relative frequencies (proportions) with respect to each margin and the overall:

```
prop.table(t,1)
```

```
          g
a            f   m
  adult    1.0 0.0
  juvenile 0.4 0.6
```

Adults are all female, and among the juveniles, 40% are female and 60% are male.

```
prop.table(t,2)
```

```
          g
a            f   m
  adult    0.6 0.0
  juvenile 0.4 1.0
```

```
prop.table(t)
```

```
          g
a              f     m
  adult    0.375 0.000
  juvenile 0.250 0.375
```

```
# overall
```

Percentage Conversion of the overall result

```
prop.table(t) * 100
```

```
          g
a             f    m
  adult    37.5  0.0
  juvenile 25.0 37.5
```

```
#converting to percentage
```

0.8 R data structures

0.8.1 Vectors

The most basic data object is a vector. One single number is a vector with a single element. All elements in one vector must be of one base data type.

Create a vector:

```
v <- c(1,2,3,4,5,5,6,67,7,7)
length(v)
```

```
[1] 10
```

Data type of elements in v:

```
mode(v)
```

```
[1] "numeric"
```

Making the data heterogeneous(mixing strings and numbers)

```
v <- c("me","him",12,445,56,6,67)
mode(v)
```

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

All values in the v have now become characters strings.

All vectors can contain a special value NA, often used to represent a missing value:

```
v
```

```
[1] "me"  "him" "12"  "445" "56"  "6"   "67"
```

```
v <- c(NA,12,2,4,54,66)

mode(v)
```

```
[1] "numeric"
```

```
v
```

```
[1] NA 12  2  4 54 66
```

A boolean vector (TRUE, FALSE):

```
b <- c(TRUE,FALSE,NA,FALSE)
```

```
mode(b)
```

```
[1] "logical"
```

```
b
```

```
[1]  TRUE FALSE    NA FALSE
```

Elements in vectors are indexed starting with 1:

```
b[4]
```

```
[1] FALSE
```

```
b[1] <- NA
```

Vectors are elastic; you can add values to any index position:

```
b[7] <- FALSE
```

Empty elements are filled with `NA`, as shown above

Create an empty vector:

```
e <- c()

mode(e)
```

```
[1] "NULL"
```

```
length(e)
```

```
[1] 0
```

Using vector elements to create another vector:

Vectorization performs an operation on each element of a vector. It is very powerful and used widely.

```
b1 <-c(b[4], b[7], b[5])
b1
```

```
[1] FALSE FALSE    NA
```

Finding the square root of all elements in `v` : `sqrt(v)`

```
sqrt(v)
```

```
[1]       NA 3.464102 1.414214 2.000000 7.348469 8.124038
```

0.9 Vector arithmetic

Vector addition

```
v1 <- c(1, 0, 1)
v2 <- c(0, 1, 0)

v1+v2
```

```
[1] 1 1 1
```

Dot product

```
v1*v2
```

```
[1] 0 0 0
```

Vector subtraction

```
v1-v2
```

```
[1]  1 -1  1
```

Vector division

```
v1/v2
```

```
[1] Inf   0 Inf
```

Warning: arithmetic with vectors of different sizes is allowed in R. R uses recycling rule to make the shorter vector the same length as the longer vector.

```
v3 <- c(1, 4)
v1+v3#the recycling rule makes v3 [1, 4, 1]
```

```
Warning in v1 + v3: longer object length is not a multiple of shorter object
length
```

```
[1] 2 4 2
```

A single value is also a vector

```
2 * v1
```

```
[1] 2 0 2
```

0.9.1 Vector summary:

Elements are of same data type, elastic, vectorization, arithmetic operations and the recycling rule.

Use vector to illustrate “for” loop:

```
mysum <- function (x){
  sum <- 0
  for(i in 1:length(x)){
    sum <- sum + x[i]
  }
  return (sum)
}

(mysum (c(1, 2, 3)))
```

```
[1] 6
```

1 PART II

1.1 Easy ways to generate vectors

With known distribution to test certain functions it is east to generate vector data.

Use () to print the result on the console

```
(x <- 1:10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
(x <- 10:1)
```

```
 [1] 10  9  8  7  6  5  4  3  2  1
```

Note the precedence of the operator `:` is higher than arithmetic operators.

```
10:15-1
```

```
[1]  9 10 11 12 13 14
```

Use `seq()` to generate sequence with real numbers:

```
(seq(from=1, to=5, length=4))
```

```
[1] 1.000000 2.333333 3.666667 5.000000
```

```
# 4 values between 1 and 5 inclusive, even intervals/steps
```

```
(seq(length=10, from=-2, by=0.5))
```

```
 [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5
```

```
#10 values, starting from 2, interval/step = 0.5
```

Use `rep(x, n)`: repeat `x` `n` times:

```
(rep(5, 10))
```

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

```
rep("hello", 10)
```

```
 [1] "hello" "hello" "hello" "hello" "hello" "hello" "hello" "hello" "hello"
[10] "hello"
```

```
(rep(0:1, 5))
```

```
 [1] 0 1 0 1 0 1 0 1 0 1
```

```
(rep(TRUE:FALSE, 3))
```

```
[1] 1 0 1 0 1 0
```

```
(rep(1:2, each=3))
```

```
[1] 1 1 1 2 2 2
```

`gl()` is for generating factor levels:

```
gl(3, 5) #three levels, each repeat 5 times
```

```
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3
```

```
gl(2, 5, labels= c('female', 'male'))#two levels, each level repeat 5 times
```

```
 [1] female female female female female male   male   male   male   male
Levels: female male
```

```
#first argument 2 says two levels.
#second argument 1 says repeat once
#third argment 20 says generate 20 values
gl(2, 1, 20, labels=c('female', 'male'))#10 alternating female and male pairs, a total of
```

```
 [1] female male   female male   female male   female male   female male
[11] female male   female male   female male   female male   female male
Levels: female male
```

Use `factor()` to convert number sequence to factor level labels. This is very useful for labeling a dataset:

```
n <- rep(1:2, each=3)
(n <- factor(n,
             levels = c(1, 2),
             labels = c('female','male')
             ))
```

```
[1] female female female male   male   male  
Levels: female male
```

```
n
```

```
[1] female female female male   male   male  
Levels: female male
```

Generate random data according to some probability density functions: the functions has a general signature of `rfunc(n, par1, par2, …)`

`r` for random,`func` is the name of the density function, `n` is the length of the data to be generated, `par1`, `par2`, … are the parameters needed for a density function

Generate 10 values following a `normal distribution` with `mean = 10` and `standard deviation = 3`:

```
(rnorm(10, mean=10, sd=3))
```

```
 [1]  9.759245 10.397261 12.123864  9.280906 15.953422  9.583639 11.252952
 [8] 12.945258  8.821914  6.880993
```

```
(rt(10, df=5)) #10 values following a Student T distribution with degree of freedom of 5
```

```
 [1]  0.8796238  4.2769289 -0.5210370  0.2567474  0.2583219 -0.4379602
 [7] -0.8103679 -0.6847279  0.5464090  1.7443301
```

Exercise:

(1) Generate a random sample of `normally distributed` data of `size 100`, with a `mean of 20` and `standard deviation 4`

(2) Compute the standard error of means of the dataset.

```
set.seed(1)
# exercise 1
```

```
(ex1 <- rnorm(100,mean=20,sd=4))
```

```
  [1] 17.49418 20.73457 16.65749 26.38112 21.31803 16.71813 21.94972 22.95330
  [9] 22.30313 18.77845 26.04712 21.55937 17.51504 11.14120 24.49972 19.82027
 [17] 19.93524 23.77534 23.28488 22.37561 23.67591 23.12855 20.29826 12.04259
 [25] 22.47930 19.77549 19.37682 14.11699 18.08740 21.67177 25.43472 19.58885
 [33] 21.55069 19.78478 14.49176 18.34002 18.42284 19.76275 24.40010 23.05270
 [41] 19.34191 18.98655 22.78785 22.22665 17.24498 17.17002 21.45833 23.07413
 [49] 19.55062 23.52443 21.59242 17.55189 21.36448 15.48255 25.73209 27.92160
 [57] 18.53111 15.82346 22.27888 19.45978 29.60647 19.84304 22.75896 20.11201
 [65] 17.02691 20.75517 12.78017 25.86222 20.61301 28.69045 21.90204 17.16021
 [73] 22.44291 16.26361 14.98547 21.16578 18.22683 20.00442 20.29737 17.64192
 [81] 17.72533 19.45929 24.71235 13.90573 22.37578 21.33180 24.25240 18.78326
 [89] 21.48008 21.06840 17.82992 24.83147 24.64161 22.80085 26.34733 22.23395
 [97] 14.89363 17.70694 15.10155 18.10640
```

```
#exercise 2

print(paste(se(ex1), "standard error of the above dataset"))
```

```
[1] "0.359279743864016 standard error of the above dataset"
```

1.2 Summary on vector generation:

range, seq, rep, gl, and distribution based random data:

```
sample <- rnorm(100, mean=20, sd=4)
se(sample)
```

```
[1] 0.3831516
```

1.3 Sub-setting

Flexible ways of select values from a vector.

Use boolean operators:

```
x <- c(0, -3,4,-1,45,90,-5)

(gtzero <- x[x>0])
```

```
[1]  4 45 90
```

Use | (or), and & (and) operators:

```
x <- c(0, -3, 4, -1, 45, 90, -5)
(x[x<=-2 | x>5])
```

```
[1] -3 45 90 -5
```

```
(x[x>40 & x<100])
```

```
[1] 45 90
```

Use a vector index:

```
x <- c(0, -3, 4, -1, 45, 90, -5)
(x[c(4, 6)])#select the 4th and 6th elements in the vector
```

```
[1] -1 90
```

```
(y<-c(4,6)) #same as above
```

```
[1] 4 6
```

```
(x[1:3])
```

```
[1]  0 -3  4
```

Usage of negative index:

select all but the first element

```
(x<-c(1,23,4,34,5,6,67))
```

```
[1]  1 23  4 34  5  6 67
```

```
(x[-1])
```

```
[1] 23  4 34  5  6 67
```

```
(x[-c(4, 6)])
```

```
[1]  1 23  4  5 67
```

```
(x[-(1:3)])
```

```
[1] 34  5  6 67
```

1.3.1 Named elements

Elements in a vector can have names.

Assign names to vector elements:

```
x <- c(0, -3, 4, -1, 45, 90, -5)
names(x) <- c('s1', 's2', 's3', 's4', 's5', 's6', 's7')
x
```

```
s1 s2 s3 s4 s5 s6 s7
 0 -3  4 -1 45 90 -5
```

Create a vector with named elements:

```
(pH <- c(area1=4.5, area2=5.7, area3=9.8, mud=7.2))
```

```
area1 area2 area3   mud
  4.5   5.7   9.8   7.2
```

Use individual names to reference/select elements:

```
pH['mud']
```

```
mud
7.2
```

```
pH[c('area1', 'mud')]
```

```
area1   mud
  4.5   7.2
```

Cannot use element names directly to exclude from select range of elements

```
x[-s1] #results in error
```

```
Error in eval(expr, envir, enclos): object 's1' not found
```

```
x[-"s1"] #results in error
```

```
Error in -"s1": invalid argument to unary operator
```

```
x[s1:s7] #results in error
```

```
Error in eval(expr, envir, enclos): object 's1' not found
```

```
x[c('s1':'s7')] #results in error
```

```
Warning: NAs introduced by coercion

Warning: NAs introduced by coercion
```

```
Error in "s1":"s7": NA/NaN argument
```

Empty index means to select all:

```
pH[]
```

```
area1 area2 area3   mud
  4.5   5.7   9.8   7.2
```

```
pH
```

```
area1 area2 area3   mud
  4.5   5.7   9.8   7.2
```

Use this method to reset a vector to 0:

```
pH[] <- 0
pH
```

```
area1 area2 area3   mud
    0     0     0     0
```

```
is.vector(pH)
```

```
[1] TRUE
```

```
pH<- 0
pH
```

```
[1] 0
```

```
is.vector(x)
```

```
[1] TRUE
```

`pH<- 0`this is different from `pH[]<-0` because we changing the whole function of that variable from a vector to a singular integer or a scalar. so both are different ways to assign a value.

1.3.2 Sub-setting summary:

boolean tests, index-based selection/exclusion, name-based selection

1.4 More R Data Structures

1.4.1 Matrices and Arrays

Arrays and matrices are essentially long vectors ***organized*** by dimensions.

Arrays can be multiple dimensions, while matrices are two dimensional, but they hold same type of values.

1.4.1.1 Matrices

To create a matrix:

```
m <- c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23)
is.vector(m)
```

```
[1] TRUE
```

```
is.matrix(m)
```

```
[1] FALSE
```

```
is.array(m)
```

```
[1] FALSE
```

```
dim(m)<- c(2,5)
```

```
is.vector(m)
```

```
[1] FALSE
```

```
is.matrix(m)
```

```
[1] TRUE
```

```
is.array(m)
```

```
[1] TRUE
```

By default, the elements are put in matrix by columns. Use `byrow=TRUE` to do it the other way:

```
(m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5, byrow = TRUE))
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   45   23   66   77   33
[2,]   44   56   12   78   23
```

Exercise:

Create a matrix with two columns:

First columns hold age data for a group of students `11, 11, 12, 13, 14, 9, 8`, and second columns hold grades `5, 5, 6, 7, 8, 4, 3`.

```
#exercise : create a matrix with 2 columns
(m<- matrix(c(11, 11, 12, 13, 14, 9, 8,5, 5, 6, 7, 8, 4, 3),7,2))
```

```
     [,1] [,2]
[1,]   11    5
[2,]   11    5
[3,]   12    6
[4,]   13    7
[5,]   14    8
[6,]    9    4
[7,]    8    3
```

Access matrix elements using position indexes (again, index starting from 1):

```
m <- c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23)
#then 'organize' the vector as a matrix
dim(m) <- c(2, 5)
#make the vector a 2 by 5 matrix, 2x5 must = lenght of the vector
m
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23
```

```
m[2, 3]#the element at row 2 and column 3
```

```
[1] 44
```

Sub-setting a matrix is similar to sub-setting on a vector.

The result is a value (a value is a vector), a vector, or a matrix:

```
(s<-m[2,1])
```

```
[1] 23
```

```
(m<- m [c(1,2), -c(3, 5)]) #select 1st row and 1st, 2nd, and 4th columns: result is a vect
```

```
     [,1] [,2] [,3]
[1,]   45   66   56
[2,]   23   77   12
```

```
(m [1, ]) #select complete row or column: 1st row, result is a vector
```

```
[1] 45 66 56
```

```
(v<-m [, 1]) # 1st column, result is a vector
```

```
[1] 45 23
```

```r
is.vector(m)
```

```
[1] FALSE
```

```r
is.matrix(m)
```

```
[1] TRUE
```

```r
is.vector(m)
```

```
[1] FALSE
```

```r
is.vector(v)
```

```
[1] TRUE
```

```r
is.matrix(v)
```

```
[1] FALSE
```

Use `drop = FALSE` to keep the results as a matrix (not vectors like shown above)

```r
m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5)
(m<-m [, 2, drop = FALSE])
```

```
     [,1]
[1,]   66
[2,]   77
```

```r
is.matrix(m)
```

```
[1] TRUE
```

```
is.vector(m)
```

```
[1] FALSE
```

`cbind()` and `rbind()`: join together two or more vectors or matrices, by column, or by row, respectively:

```
cbind(c(1,2,3), c(4,5,6)) #columnn bind
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
rbind(c(1,2,3), c(4,5,6)) #row bind
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```
m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5)
(a <- rbind (c(1,2,3,4,5), m))
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]   45   66   33   56   78
[3,]   23   77   44   12   23
```

```
is.array(a)
```

```
[1] TRUE
```

```
is.matrix(a)
```

```
[1] TRUE
```

m1 - m4 look like,

```
(m1 <- matrix(rep(10, 9), 3,3) )
```

```
     [,1] [,2] [,3]
[1,]   10   10   10
[2,]   10   10   10
[3,]   10   10   10
```

```
(m2 <- cbind (c(1,2,3), c(4, 5, 6)) )
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
(m3 <- cbind (m1[,1], m2[2,]))
```

```
Warning in cbind(m1[, 1], m2[2, ]): number of rows of result is not a multiple
of vector length (arg 2)
```

```
     [,1] [,2]
[1,]   10    2
[2,]   10    5
[3,]   10    2
```

```
(m4 <- cbind (m1[,1], m2[,2]))
```

```
     [,1] [,2]
[1,]   10    4
[2,]   10    5
[3,]   10    6
```

1.4.1.2 Named rows and columns:

```
sales <- matrix(c(10,30,40,50,43,56,21,30),2,4,byrow=TRUE)
colnames(sales)<- c('1qrt','2qrt','3qrt','4qrt')
rownames(sales)<-c('store1','store2')
```

```
sales
```

```
       1qrt 2qrt 3qrt 4qrt
store1   10   30   40   50
store2   43   56   21   30
```

Exercise:

Find `store1` `1qrt` sale. 2. List `store2`'s 1st and 4th quarter sales:

```
sales['store2','1qrt']
```

```
[1] 43
```

```
sales['store2',c('1qrt','4qrt')]
```

```
1qrt 4qrt 
  43   30 
```

1.4.1.3 Arrays

Arrays are similar to matrices, but arrays can have more than 2 dimensions

3-D array:

```
a <- array(1:48,dim= c(4,3,2))
a
```

```
, , 1

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

, , 2

     [,1] [,2] [,3]
```

```
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

Select array elements using indexes, results may be a value, a vector, a matrix or an array, depending on the use of `drop=FALSE`:

```
a[1,3,2]
```

```
[1] 21
```

```
a[1,,2]
```

```
[1] 13 17 21
```

```
a[1,,2,drop=FALSE]
```

```
, , 1

     [,1] [,2] [,3]
[1,]   13   17   21
```

```
a[4,3,]
```

```
[1] 12 24
```

```
a[c(2,3),,-2]
```

```
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
```

Assign names to dimensions of an array.

`[[]]` selects one dimension:

```
dimnames(a)[[1]] <-c("1qrt", "2qrt", "3qrt", "4qrt")
dimnames(a)[[2]] <-c("store1", "store2", "store3")
dimnames(a)[[3]] <-c("2017", "2018")
a
```

```
, , 2017

     store1 store2 store3
1qrt      1      5      9
2qrt      2      6     10
3qrt      3      7     11
4qrt      4      8     12

, , 2018

     store1 store2 store3
1qrt     13     17     21
2qrt     14     18     22
3qrt     15     19     23
4qrt     16     20     24
```

Alternatively, use `list()` to specify names:

```
ar <- array(data     = 1:27,
            dim      = c(3, 3, 3),
            dimnames = list(c("a", "b", "c"),
ar
```

```
, , g

  d e f
a 1 4 7
b 2 5 8
c 3 6 9

, , h

   d  e  f
a 10 13 16
b 11 14 17
c 12 15 18
```

```
, , i

   d  e  f
a 19 22 25
b 20 23 26
c 21 24 27
```

1.4.1.4 Split array into matrices

Perform arithmetic operations on matrices, note the recycling rules apply:

```
matrix1 <- ar[,,g]
```

```
matrix1 <- ar[,,'g']
matrix1
```

```
  d e f
a 1 4 7
b 2 5 8
c 3 6 9
```

```
matrix1 <- ar[,,"g"]
matrix1
```

```
  d e f
a 1 4 7
b 2 5 8
c 3 6 9
```

```
matrix2<- ar[,,"h"]
matrix2
```

```
   d  e  f
a 10 13 16
b 11 14 17
c 12 15 18
```

```
sum<- matrix1+matrix2
sum
```

```
   d  e  f
a 11 17 23
b 13 19 25
c 15 21 27
```

```
matrix1*3
```

```
  d  e  f
a 3 12 21
b 6 15 24
c 9 18 27
```

A matrix is just a long vector organized into dimensions, note the recycling rules apply:

```
matrix1
```

```
  d e f
a 1 4 7
b 2 5 8
c 3 6 9
```

```
matrix1*c(2,3)
```

```
Warning in matrix1 * c(2, 3): longer object length is not a multiple of shorter
object length
```

```
  d  e  f
a 2 12 14
b 6 10 24
c 6 18 18
```

```
matrix1*c(2,3,4,4,5,5,35,333)
```

```
Warning in matrix1 * c(2, 3, 4, 4, 5, 5, 35, 333): longer object length is not
a multiple of shorter object length
```

```
   d  e    f
a  2 16  245
b  6 25 2664
c 12 30   18
```

```
matrix1*c(1,2,3)
```

```
  d  e  f
a 1  4  7
b 4 10 16
c 9 18 27
```

```
matrix1/c(1,2,3)
```

```
  d   e f
a 1 4.0 7
b 1 2.5 4
c 1 2.0 3
```

```
matrix1/c(1,2,3,4,45,5,6,7,7)
```

```
  d         e        f
a 1 1.0000000 1.166667
b 1 0.1111111 1.142857
c 1 1.2000000 1.285714
```

1.4.2 Lists

Lists are vectors too, but they are 'recursive' (as opposed to the 'atomic' vectors we learned before: vector, matrix, arrays), meaning they can hold other lists, meaning a list can hold data of different types. Lists consist of an ordered collection of objects known as their components ##list components do not need to be of the same type. ##list components are always numbered (with an index) and may also have a name attached to them.

Use `list$component_name` to access a component in a *list* can not be used on atomic vectors.

`[`, `[[`, and `$` - R accessors

```r
mylist<- list(stud.id=34453,
              stud.name="john",
              stud.marks=c(13,3,12,15,19)

          )
mylist$stud.id
```

```
[1] 34453
```

```r
mylist[1]
```

```
$stud.id
[1] 34453
```

```r
mylist[[1]]
```

```
[1] 34453
```

```r
mylist["stud.id"]
```

```
$stud.id
[1] 34453
```

```r
handle <- "stud.id"
mylist[handle]
```

```
$stud.id
[1] 34453
```

```r
mylist[["stud.id"]]
```

```
[1] 34453
```

1.4.3 Subset with [

Both indices and names can be used to extract the subset. In order to use names, object must have a name type attribute such as names, rownames, colnames, etc.

You can use negative integers to indicate exclusion.

Unquoted variables are interpolated within the brackets.

1.4.4 Extract one item with [[

The double square brackets are used to extract one element from potentially many. For vectors yield vectors with a single value; data frames give a column vector; for list, one element.

You can return only one item. The result is not (necessarily) the same type of object as the container. The dimension will be the dimension of the one item which is not necessarily 1. And, as before: Names or indices can both be used. #Variables are interpolated.

1.4.5 Interact with $

$ is a special case of [[in which you access a single item by actual name (but not used for atomic vectors). You cannot use integer indices.

The name will not be interpolated and returns only one item. If the name contains special characters, the name must be enclosed in back-ticks: "

```
mylist <- list(stud.id=34453,
               stud.name="John",
               stud.marks= c(13, 3, 12, 15, 19)
               )
mylist$stud.marks
```

```
[1] 13  3 12 15 19
```

```
mylist$stud.marks[2]
```

```
[1] 3
```

```
names(mylist)
```

```
[1] "stud.id"    "stud.name"  "stud.marks"
```

```
names(mylist) <- c('id',"name","marks")

names(mylist)
```

```
[1] "id"    "name"  "marks"
```

```
mylist
```

```
$id
[1] 34453

$name
[1] "John"

$marks
[1] 13  3 12 15 19
```

```
mylist$parents.name <- c('ana','mike')
mylist
```

```
$id
[1] 34453

$name
[1] "John"

$marks
[1] 13  3 12 15 19

$parents.name
[1] "ana"  "mike"
```

```
newlist <- list(age=19,sex="male")
expandedlist <- c(mylist,newlist )
expandedlist
```

```
$id
[1] 34453

$name
[1] "John"

$marks
[1] 13  3 12 15 19

$parents.name
[1] "ana"  "mike"

$age
[1] 19

$sex
[1] "male"
```

```
length(expandedlist)
```

```
[1] 6
```

1.4.6 Remove list components using negative index, or using NULL

Exercise:

Starting with the expanded list given above, what will be the result of the following statement? Consider the statement one by one.

```
expandedlist <- expandedlist[-5]
expandedlist <- expandedlist[c(-1,-5)]
expandedlist$parents.names <- NULL
expandedlist[['marks']] <- NULL
```

```
mylist
```

```
$id
[1] 34453

$name
```

```
[1] "John"

$marks
[1] 13  3 12 15 19

$parents.name
[1] "ana"  "mike"
```

unlist() coerces a list to a vector:

```r
unlist(mylist)
```

```
           id          name        marks1        marks2        marks3 
      "34453"        "John"          "13"           "3"          "12" 
       marks4        marks5 parents.name1 parents.name2 
         "15"          "19"         "ana"        "mike" 
```

```r
mode(mylist)
```

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

```r
mode(unlist(mylist))
```

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

```r
is.vector(unlist(mylist))
```

```
[1] TRUE
```

```r
is.vector(unlist(mylist))
```

```
[1] TRUE
```

```r
is.list(list)
```

```
[1] FALSE
```

```
is.atomic(mylist)
```

```
[1] FALSE
```

```
is.list(unlist(mylist))
```

```
[1] FALSE
```

1.5 Data Frames

The recommended data structure for tables (2-D), data frames are a special kind of list: each row is an observation, each column is an attribute.

The column names should be non-empty, and the row names should be unique.

The data stored in a data frame can be of numeric, factor or character type., and each column should contain same number of data items.

1.5.1 Create a data frame

Note: dataframe turns categorical values to a factor by default

```
my.dataframe <- data.frame(site=c('A', 'B', 'A','A', 'B'),
                           season=c('winter', 'summer', 'summer', 'spring', 'fall'),
my.dataframe
```

```
  site season  ph
1    A winter 7.4
2    B summer 6.3
3    A summer 8.6
4    A spring 7.2
5    B   fall 8.9
```

1.5.2 Indexes and names

Exercise:

Given ‘my.dataframes‘, what values will the following statements access,

```
my.dataframe <- data.frame(site=c('A', 'B', 'A','A', 'B'),
                           season=c('winter', 'summer', 'summer', 'spring', 'fall'),
my.dataframe[3, 2]
```

```
[1] "summer"
```

```
my.dataframe[['site']]
```

```
[1] "A" "B" "A" "A" "B"
```

```
my.dataframe[my.dataframe$ph>7,]
```

```
  site season  ph
1    A winter 7.4
3    A summer 8.6
4    A spring 7.2
5    B   fall 8.9
```

```
my.dataframe[my.dataframe$ph>7,"site"]
```

```
[1] "A" "A" "A" "B"
```

```
my.dataframe[my.dataframe$ph>7,c('site','ph')]
```

```
  site  ph
1    A 7.4
3    A 8.6
4    A 7.2
5    B 8.9
```

1.5.3 Use `subset()` to query a data frame

`subset()` can only query, it can not be used to change values in the data frame:

```
subset(my.dataframe,ph>7)
```

```
  site season  ph
1    A winter 7.4
3    A summer 8.6
4    A spring 7.2
5    B   fall 8.9
```

```
subset(my.dataframe,ph>7,c("site","ph"))
```

```
  site  ph
1    A 7.4
3    A 8.6
4    A 7.2
5    B 8.9
```

```
subset(my.dataframe[1:2,],ph>7,c(site,ph))
```

```
  site  ph
1    A 7.4
```

```
my.dataframe[my.dataframe$season=='summer', 'ph'] <-
  my.dataframe[my.dataframe$season=='summer', 'ph'] + 1

my.dataframe[my.dataframe$season=='summer', 'ph']
```

```
[1] 7.3 9.6
```

```
my.dataframe[my.dataframe$season=='summer' & my.dataframe$ph>8, 'ph'] <-
  my.dataframe[my.dataframe$season=='summer' & my.dataframe$ph>8, 'ph'] + 1

my.dataframe[my.dataframe$season=='summer', 'ph']
```

```
[1]  7.3 10.6
```

1.5.4 Add a column

```r
my.dataframe$NO3 <- c(234.5,123.4,456.7,567.8,789.0)
my.dataframe
```

```
  site season   ph   NO3
1    A winter  7.4 234.5
2    B summer  7.3 123.4
3    A summer 10.6 456.7
4    A spring  7.2 567.8
5    B   fall  8.9 789.0
```

1.5.5 Remove a column

```r
my.dataframe <- my.dataframe[,-4]
```

```r
my.dataframe
```

```
  site season   ph
1    A winter  7.4
2    B summer  7.3
3    A summer 10.6
4    A spring  7.2
5    B   fall  8.9
```

Check the structure of a data frame:

```r
str(my.dataframe)
```

```
'data.frame':   5 obs. of  3 variables:
 $ site  : chr  "A" "B" "A" "A" ...
 $ season: chr  "winter" "summer" "summer" "spring" ...
 $ ph    : num  7.4 7.3 10.6 7.2 8.9
```

```r
nrow(my.dataframe)
```

```
[1] 5
```

```
ncol(my.dataframe)
```

```
[1] 3
```

```
dim(my.dataframe)
```

```
[1] 5 3
```

Edit a data frame:

```
 edit(my.dataframe) #this brings up a data editor
```

```
  site season   ph
1    A winter  7.4
2    B summer  7.3
3    A summer 10.6
4    A spring  7.2
5    B   fall  8.9
```

```
 View(my.dataframe) #this brings up a uneditable tab that display the data for you to view
```

Update names of the columns:

```
names(my.dataframe)
```

```
[1] "site"   "season" "ph"
```

```
names(my.dataframe) <- c("area","season","P.h")
my.dataframe
```

```
  area season  P.h
1    A winter  7.4
2    B summer  7.3
3    A summer 10.6
4    A spring  7.2
5    B   fall  8.9
```

```r
names(my.dataframe)[3] <- 'ph'
my.dataframe
```

```
  area season   ph
1    A winter  7.4
2    B summer  7.3
3    A summer 10.6
4    A spring  7.2
5    B   fall  8.9
```

1.6 Tibbles

Tibbles are similar to data frame, but they are more convenient than data frame.

Columns can be defined based on other columns defined earlier. Tibbles cannot convert categorical valued attributes to factors and does not print an entire dataset (when it is large, it occupied all your screen and more).

```r
#tibble is already installed
if(!require("tibble"))
install.packages("tibble")
```

```
Loading required package: tibble
```

```r
library(tibble)
```

1.6.1 Create a tibble

```r
my.tibble <- tibble(TempCels = sample(-10:40, size=100, replace=TRUE),
                    TempFahr = TempCels*9/5+32,
                    Location = rep(letters[1:2], each=50))
my.tibble
```

```
# A tibble: 100 x 3
   TempCels TempFahr Location
      <int>    <dbl> <chr>
 1       34     93.2 a
 2       22     71.6 a
 3        8     46.4 a
 4       15     59   a
 5       12     53.6 a
 6       -7     19.4 a
 7        5     41   a
 8       38    100.  a
 9       16     60.8 a
10       28     82.4 a
# i 90 more rows
```

Use the penguins data frame from the palmerpenguins package:

```r
if(!require("palmerpenguins"))
install.packages("palmerpenguins")
```

```
Loading required package: palmerpenguins
```

```r
library(palmerpenguins)
data(penguins)
dim(penguins)
```

```
[1] 344   8
```

```r
class(penguins)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```r
penguins
```

```
# A tibble: 344 x 8
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

1.6.2 Convert a data frame to a tibble

```
pe <- as_tibble(penguins)
class(pe)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
pe
```

```
# A tibble: 344 x 8
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

`mode` is a mutually exclusive classification of objects according to their basic structure. The 'atomic' modes are numeric, complex, character and logical. Recursive objects have modes such as 'list' or 'function' or a few others. An object has one and only one mode.

`class` is a property assigned to an object that determines how generic functions operate with it. It is not a mutually exclusive classification. If an object has no specific class assigned to it, such as a simple numeric vector, it's class is usually the same as its mode, by convention.

Changing the mode of an object is often called 'coercion'. The mode of an object can change without necessarily changing the class.

e.g., typeof or specific type testers: is.vector, is.atomic, is.data.frame, etc.

```r
x <- 1:16
mode(x)
```

```
[1] "numeric"
```

```r
dim(x) <- c(4,4)
class(x)
```

```
[1] "matrix" "array"
```

```r
is.numeric(x)
```

```
[1] TRUE
```

```r
mode(x) <- "character"
mode(x)
```

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

```r
class(x)
```

```
[1] "matrix" "array"
```

```r
x<- factor(x)
class(x)
```

```
[1] "factor"
```

```
mode(x)
```

```
[1] "numeric"
```

Class changed from 'matrix' to 'factor', even though the x is numeric and mode is numeric it's new class factor prohibits it from using arithmetic operation

```
is.array(x)
```

```
[1] FALSE
```

```
is.list(x)
```

```
[1] FALSE
```

```
is.data.frame(x)
```

```
[1] FALSE
```

```
is.matrix(x)
```

```
[1] FALSE
```

```
is_tibble
```

```
function (x)
{
    inherits(x, "tbl_df")
}
<bytecode: 0x00000279ec765b48>
<environment: namespace:tibble>
```

```
is.vector(x)
```

```
[1] FALSE
```

```
typeof(x)
```

```
[1] "integer"
```

Subsetting a tibble results in a smaller tibble

This is different from data frame - subsetting a data frame could result in a vector, when subsetting result in one series of values

```
class(pe[1:15, c("bill_length_mm", "bill_depth_mm")])
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
class(penguins[1:15, c("bill_length_mm", "bill_depth_mm")])
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
class(pe[1:15, c("bill_length_mm")])
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
class(penguins[1:15, c("bill_length_mm")])
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

1.7 dplyr

dplyr library is very useful for manipulate table-like data (Dataframes)

```
#install.packages("dplyr",repos='http://cran.us.r-project.org')
#it is already installed, throws error when trying to install again
library(dplyr)
```

1.7.1 filter() vs. select()

select() selects a subset of columns of the dataset.

filter() select a subset of rows.

```
select(filter(pe, species=="Adelie"), bill_length_mm, bill_depth_mm)
```

```
# A tibble: 152 x 2
   bill_length_mm bill_depth_mm
            <dbl>         <dbl>
 1           39.1          18.7
 2           39.5          17.4
 3           40.3          18  
 4           NA            NA  
 5           36.7          19.3
 6           39.3          20.6
 7           38.9          17.8
 8           39.2          19.6
 9           34.1          18.1
10           42            20.2
# i 142 more rows
```

```
filter(select(pe, bill_length_mm, bill_depth_mm, species), species=="Adelie")
```

```
# A tibble: 152 x 3
   bill_length_mm bill_depth_mm species
            <dbl>         <dbl> <fct>  
 1           39.1          18.7 Adelie 
 2           39.5          17.4 Adelie 
 3           40.3          18   Adelie 
 4           NA            NA   Adelie 
 5           36.7          19.3 Adelie 
 6           39.3          20.6 Adelie 
 7           38.9          17.8 Adelie 
 8           39.2          19.6 Adelie 
 9           34.1          18.1 Adelie 
10           42            20.2 Adelie 
# i 142 more rows
```

Exercise

How would you achieve the same result as the above but use tibble subsetting?

```
pe
```

```
# A tibble: 344 x 8
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

```
pe[pe$species=='Adelie', c("bill_length_mm", "bill_depth_mm")]
```

```
# A tibble: 152 x 2
   bill_length_mm bill_depth_mm
            <dbl>         <dbl>
 1           39.1          18.7
 2           39.5          17.4
 3           40.3          18  
 4           NA            NA  
 5           36.7          19.3
 6           39.3          20.6
 7           38.9          17.8
 8           39.2          19.6
 9           34.1          18.1
10           42            20.2
# i 142 more rows
```

```
subset(pe, pe$species=='Adelie', c("bill_length_mm", "bill_depth_mm"))
```

```
# A tibble: 152 x 2
   bill_length_mm bill_depth_mm
            <dbl>         <dbl>
 1           39.1          18.7
 2           39.5          17.4
 3           40.3          18  
 4           NA            NA  
 5           36.7          19.3
 6           39.3          20.6
 7           38.9          17.8
 8           39.2          19.6
 9           34.1          18.1
10           42            20.2
# i 142 more rows
```

Pipe `|>`, or the `magrittr` `%>%`, passes the output of a function to another function as its first argument. very useful for queries

```
select(pe, bill_length_mm, bill_depth_mm, species) |> filter(species=="Adelie")
```

```
# A tibble: 152 x 3
   bill_length_mm bill_depth_mm species
            <dbl>         <dbl> <fct>  
 1           39.1          18.7 Adelie 
 2           39.5          17.4 Adelie 
 3           40.3          18   Adelie 
 4           NA            NA   Adelie 
 5           36.7          19.3 Adelie 
 6           39.3          20.6 Adelie 
 7           38.9          17.8 Adelie 
 8           39.2          19.6 Adelie 
 9           34.1          18.1 Adelie 
10           42            20.2 Adelie 
# i 142 more rows
```

Exercise

Pass the result from the filter to the select function and achieve the same result as shown above.

```
filter(pe, species=="Adelie") |> select(bill_length_mm, bill_depth_mm, species)
```

```
# A tibble: 152 x 3
   bill_length_mm bill_depth_mm species
            <dbl>         <dbl> <fct>  
 1           39.1          18.7 Adelie 
 2           39.5          17.4 Adelie
 3           40.3          18   Adelie
 4           NA            NA   Adelie
 5           36.7          19.3 Adelie
 6           39.3          20.6 Adelie
 7           38.9          17.8 Adelie
 8           39.2          19.6 Adelie
 9           34.1          18.1 Adelie
10           42            20.2 Adelie
# i 142 more rows
```

Exercise

Create a data object to hold student names (`Judy`, `Max`, `Dan`) and their grades (‘78,85,99) `Convert number grades to letter grades:`90-100:A;80-89:B;70-79:C; \<70:F‘

```r
students <- list(names=c("Judy", "Max", "Dan"),
                 grades=c(78, 85, 99))
print ("before:")
```

```
[1] "before:"
```

```r
students
```

```
$names
[1] "Judy" "Max"  "Dan"

$grades
[1] 78 85 99
```

```r
gradeConvertor<- function (grade){
  grade = as.numeric(grade)
  if(grade > 100 | grade < 0) print ("grade out of the range")
  else if(grade >= 90 & grade <= 100) return ("A")
  else if(grade >= 80 & grade < 90) return ("B")
  else if(grade >= 70 & grade < 80) return ("C")
  else return ("F")
}

#students$grades <-sapply(students$grades, gradeConvertor)
```

```r
for(i in 1:length(students$grades)){
  students$grades[i] = gradeConvertor(students$grades[i])
}

print ("after:")
```

```
[1] "after:"
```

```r
students
```

```
$names
[1] "Judy" "Max"  "Dan"

$grades
[1] "C" "B" "A"
```