INFO 523 Exercise

Weeks 1-2: Introduction to R Exercise

Table of contents

Part I

If you are seeing error messages, please review them and try to understand. It is a good practice for you debug.

If you are seeing warning messages, please review. Warning messages are typically not fatal. Might be obsolete/deprecated.

Universal things every R user should know:

Find which version of R you are using

```
R.version
```

```
               _
platform       aarch64-apple-darwin20
arch           aarch64
os             darwin20
system         aarch64, darwin20
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)
nickname       Beagle Scouts
```

Packages

R has many tools wrapped in packages, and we often use those tools in our work.

To use a tool, you need to install it.

The package used in Data Mining with R is `DMwR2`

In Windows 11, this shall run okay.

In Ubuntu 20.04, you might see error. one error requires run `sudo apt-get install libcurl4-openssl-dev` in your terminal.

```r
install.packages("DMwR2")
```

To see what is in a package, use `help()`. If you do not see documentation, there might be errors.

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

The above step takes some time and you need internet connection.

Now the packages are installed in your computer. To use a function in the package, either of the two ways works:

(1) when you need to use the function frequently, you would want to load it to the memory for your current session by using `library()` function (one RStuido window is one session, if you have multiple RStudio windows open, they are different sessions)

(2) when you only need to use the function one or twice, you can call the `function/dataset` through the notation `package::functionname`

```r
library(DMwR2)
```

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

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

```r
data(algae) # load algae dataset
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) # find rows with too many NAs
```

```
[1]  62 199
```

`library()` without arguments:

It will provide you the list of packages installed in different libraries on your computer.

```
library()
```

Show packages loaded in the current session:

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

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

Think of `library()` as a library of all installed packages. `library(packagename)` checks a package out.

`.packages()` shows all checked out packages in the current session.

If you loaded a package, say `dbplyr`, by mistake, you can detach it from your session using `detach`

```r
install.packages("dbplyr") # assuming you have dbplyr installed before

# now you try to check out dplyr, but typed dbplyr by accident
library(dbplyr)
(.packages())
# you realized the mistake, and you don't want this package to be live in this session due
# you can detach the package
detach("package:dbplyr", unload=TRUE)
(.packages())

library(dplyr)#load wanted library
```

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

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

```

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 what packages have been installed in your computer:

```r
installed.packages()
```

```
         Package
abind    "abind"
airports "airports"
ambient  "ambient"
         LibPath
abind    "/Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/library"
airports "/Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/library"
```

```
ambient  "/Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/library"
         Version Priority Depends        Imports
abind    "1.4-5" NA       "R (>= 1.5.0)" "methods, utils"
airports "0.1.0" NA       "R (>= 2.10)"  NA
ambient  "1.0.2" NA       "R (>= 3.0.2)" "rlang, grDevices, graphics, stats"
         LinkingTo          Suggests   Enhances License
abind    NA                 NA         NA       "LGPL (>= 2)"
airports NA                 "testthat" NA       "GPL-3"
ambient  "cpp11 (>= 0.4.2)" "covr"     NA       "MIT + file LICENSE"
         License_is_FOSS License_restricts_use OS_type MD5sum NeedsCompilation
abind    NA              NA                    NA      NA     "no"
airports NA              NA                    NA      NA     "no"
ambient  NA              NA                    NA      NA     "yes"
         Built
abind    "4.3.0"
airports "4.3.0"
ambient  "4.3.0"
```

Find out if your installed packages have a newer version on CRAN:

```
old.packages()
```

Update all your installed packages to the newest version – this may take a long time:

```
update.packages()
```

Update all your installed packages WITHOUT having to confirm for each package (Note: as this could take a long time, you don't have to practice this command. Do not worry too much if you see warning or failure messages)

```
update.packages(ask = FALSE)
```

Find out the namespace/package a function belongs in your installed packages, just type the function name - e.g., function `mean` is in base R:

```
mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x11102bbd8>
<environment: namespace:base>
```

Find help on a function in an installed package, say `mean()`. If you use R Studio, R documentation on the method `mean()` is display on lower right pane of the window:

```
help(mean)
```

If two packages provide a function with the same name and you need to use both functions, use `package::functionname` to differentiate the functions.

When you want to see if a package you need to use has already been made, search for it using some keywords:

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

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

```
#useful controls in R Studio#
#Ctrl+1  Move focus to the Source Editor.
#Ctrl+2  Move focus to the Console.
#Ctrl+l  Clear the Console.
#Esc     Interrupt R.
```

Project and Session Management

Use Project to manage your R scripts and data.

In RStudio, `File > New Project` to create a new folder on your computer for your project.

Multiple scripts can be created and saved in the project folder, along with data used

`File > Open Project` to resume your work in the workspace.

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

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

`Close a Project` in RStudio closes the current project, but still keep the session (RStudio interface is still up)

`Quit Session` closes the current RStudio window.

Typing long and complex commands in a console can be limiting.

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

You often need to save large data objects or function for later use

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

Save all objects

All objects are saved in `.RData` file in the current working directory for you to load in the future.

```
save.image()
```

Run `getwd()` and `setwd()` in RStudio Console to show the current working directory and to set working directory respectively.

```
getwd()

setwd("/home/gchism/Documents/523") # setwd using what you get from getwd()

getwd()
```

R Objects and Variables

Variables are references to some storage locations in computer memory that holds some content (objects) that range from a simple number to an complex model to associate an object (e.g., the number `0.2`) to a variable.

```
vat <- 0.2
```

Now see what `vat` holds:

```
vat
```

```
[1] 0.2
```

Use `()` to enclose a statement to have the returned values print directly:

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

```
[1] 0.2
```

More examples:

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

```
[1] 1
```

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

```
[1] 1
```

```
z
```

```
[1] 0.25
```

All variables stay alive until you delete it or when your exit R without saving them to list variables currently alive: `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 space:

```r
rm(vat)
```

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:

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

```
[1] 12
```

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

```
[1] 4
```

```r
set.seed(1) #the seed determines the starting point used in generating a sequence of pseud
#there is a function to remove the seed:rm(.Random.seed, envir=.GlobalEnv)

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

```
[1] 97
```

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

```
[1] 48.4
```

Why the same function with same argument gave different results above? Use `help(sample)` to find out what function sample does.

What do you expect?

```r
set.seed(1)
rnorm(1) #give me one number from a normal distribution
```

```
[1] -0.6264538
```

```r
rnorm(1)
```

```
[1] 0.1836433
```

```r
set.seed(2)
rnorm(1)
```

```
[1] -0.8969145
```

```r
rnorm(1)
```

```
[1] 0.1848492
```

```r
set.seed(1)
rnorm(1) #give me one number from a normal distribution
```

```
[1] -0.6264538
```

```r
rnorm(1)
```

```
[1] 0.1836433
```

We use `set.seed()` to make sure multiple runs of a program involving random samples give the same result, for debugging purposes.

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

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

```
[1] FALSE
```

No object named `se` exists, now create the function that computes the standard error of a sample:

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

Object `se` has been created:

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

```
[1] TRUE
```

A side note: how is `se` different from `sd`? They are very different! See the following video.

Create another function with multiple arguments:

`convMeters` will convert meters to inches, feet, yards, and miles. `exists("convMeters")`

```r
convMeters <- function (x, to="inch"){
  factor = switch(to, inch=39.3701, foot=3.28084, yard=1.09361, mile=0.000621371, NA)
  if(is.na(factor)) stop ("unknown target unit")
  else return (x*factor)
}
convMeters(23, "foot")
```

```
[1] 75.45932
```

If no argument to is provided, the default value `'inch'` is used

```r
convMeters(40)
```

```
[1] 1574.804
```

Arguments for the function can be supplied in the order as in the function signature:

```r
convMeters(56.2, "yard")
```

```
[1] 61.46088
```

Arguments can also be supplied in other orders if sufficient arguments are named so R can un-ambiguously assign the arguments for a function.

```r
convMeters(to="yard", 56.2)
```

```
[1] 61.46088
```

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`:

```r
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:

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

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

Compare the above with the following:

```r
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:

```r
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 not the same length as `g` factor?

```
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 the good `a` back 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)#1 refers to the first factor, a (age)
```

```
a
   adult juvenile
       3        5
```

```
margin.table(t, 2)# now find the marginal freq of the second factor g
```

```
g
f m
5 3
```

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

```
t
```

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

```
prop.table(t, 1) #use the margin generated for the 1st factor a
```

```
          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) #overall
```

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

Show the same results in percentage:

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

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

R data structures

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(2, 5, 3, 4)
length(v)
```

```
[1] 4
```

Data type of elements in `v`:

```
mode(v)
```

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

If you mix strings and numbers:

```
v <- c(2, 5, 3, 4, 'me')
mode(v)
```

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

```
v
```

```
[1] "2"  "5"  "3"  "4"  "me"
```

See the difference? 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 <- c(2, 5, 3, 4, NA)
mode(v)
```

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

```
v
```

```
[1]  2  5  3  4 NA
```

A boolean vector (`TRUE`, `FALSE`)

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

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

```
b
```

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

Elements in vectors are indexed starting with 1:

```
b[3]
```

```
[1] NA
```

To update a value at a specific index:

```
b[3] <- TRUE
b
```

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

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

```
b[10] <- FALSE
b
```

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

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

Create an empty vector:

```
e <-vector()
mode(e)
```

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

```
e <- c()
mode(e)
```

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

```
length(e)
```

```
[1] 0
```

Use a vector elements to construct another vector:

```
b2 <-c(b[1], b[3], b[5])
b2
```

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

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

Find the square root of all elements in `v`:

```r
sqrt(v)
```

```
[1] 1.414214 2.236068 1.732051 2.000000       NA
```

Vector arithmetic

```r
v1 <- c(3, 6, 9)
v2 <- c(1, 4, 8)
v1+v2 #addition
```

```
[1]  4 10 17
```

```r
v1*v2 #dot product
```

```
[1]  3 24 72
```

```r
v1-v2 #subtraction
```

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

```r
v1/v2 #divsion
```

```
[1] 3.000 1.500 1.125
```

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.

```r
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]  4 10 10
```

Remember, a single value is a vector too?

```
2*v1
```

```
[1]  6 12 18
```

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
```

PART II

Easy ways to generate vectors

These are useful when you need to generate some data with known distribution to test certain functions.

Use `()` to print the result of a statement in the console `1 2 3 4 5 6 7 8 9 10`

```
(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. Observe the difference below, why they give different results?

```
10:15-1
```

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

```
10:(15-1)
```

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

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

```
(seq(from=1, to=5, length=4)) # 4 values between 1 and 5 inclusive, even intervals/steps
```

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

```
(seq(length=10, from=-2, by=0.5)) #10 values, starting from 2, interval/step = 0.5
```

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

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

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

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

```r
(rep("hi", 3))
```

```
[1] "hi" "hi" "hi"
```

1 2 1 2 1 2

```r
(rep(1:2, 3))
```

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

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

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

1 1 1 2 2 2

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

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

`gl()` is for generating factor levels:

```r
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
```

```r
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
```

```r
#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]  7.493114 14.785842 10.988523  7.538595 11.462287 12.214974 11.727344
 [8]  9.083835 14.535344 11.169530
```

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

```
 [1] -3.20099075 -0.42241451 -0.86409523 -1.50276529  0.85199410 -1.82436807
 [7] -0.06641194 -1.41288461 -0.32612422  0.44183505
```

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.

Summary on vector generation:

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

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

```
[1] 0.3654364
```

Sub-setting

Flexible ways of select values from a vector.

Use boolean operators:

```
x <- c(0, -3, 4, -1, 45, 90, -5)
#select all elements that is greater than 0
(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
```

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

```
[1] 45 90
```

Use a vector index:

```r
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
```

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

```
[1] 4 6
```

```r
(x[y])
```

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

```r
(x[1:3]) #select the 1st to the 3rd elements in the vector
```

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

Use negative index to exclude elements:

```r
x <- c(0, -3, 4, -1, 45, 90, -5)
(x[-1]) #select all but the first element
```

```
[1] -3  4 -1 45 90 -5
```

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

```
[1]  0 -3  4 45 -5
```

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

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

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
```

…but can not use directly element names to exclude or select a range of elements:

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

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

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

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

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

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

```r
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:

```r
pH[]
```

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

```r
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 
```

```
pH<- 0
pH
```

```
[1] 0
```

This is different from `pH <- 0`, why?

Sub-setting summary:

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

More R Data Structures

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.

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
```

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

```
[1] FALSE
```

```r
#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
```

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

```
[1] FALSE
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

```r
(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`.

```
test <-matrix(c(11, 11, 12, 13, 14, 9, 8, 5, 5, 6, 7, 8, 4, 3), 7, 2)
test
```

```
     [,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]) # select one value
```

```
[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
```

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

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

```r
(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(s)
```

```
[1] TRUE
```

```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
```

```
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))
```

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

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

```
     [,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
```

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

```
[1] TRUE
```

Exercise:

What will `m1-m4` look like?

```r
m1 <- matrix(rep(10, 9), 3, 3) m2 <- cbind (c(1,2,3), c(4, 5, 6)) m3 <- cbind (m1[,1], m2[
```

Named rows and columns

```r
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:

```r
sales['store1', '1qrt']
```

```
[1] 10
```

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

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

Arrays

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

3-D array:

```r
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`:

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

```
[1] 21
```

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

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

```r
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
```

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
```

```
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(2,3,2,3,2,3,2,3,2)
```

```
  d  e  f
a 2 12 14
b 6 10 24
c 6 18 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, 1, 2, 3, 1, 2, 3)
```

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

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 `$`: https://www.r-bloggers.com/r-accessors-explained/

```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
```

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.

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.

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: `"`

```r
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
```

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

```
[1] 3
```

Change names:

```r
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
```

Add new component:

```
mylist$parents.names <- c('Ana', "Mike")
mylist
```

```
$id
[1] 34453

$name
[1] "John"

$marks
[1] 13  3 12 15 19

$parents.names
[1] "Ana"  "Mike"
```

Use `c()` to concatenate two lists:

```r
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.names
[1] "Ana"  "Mike"

$age
[1] 19

$sex
[1] "male"
```

```r
length(expandedlist)
```

```
[1] 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.

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

`unlist()` coerces a list to a vector:

```r
mylist
```

```
$id
[1] 34453

$name
[1] "John"

$marks
[1] 13  3 12 15 19

$parents.names
[1] "Ana"  "Mike"
```

```r
unlist(mylist)
```

```
            id           name         marks1         marks2         marks3 
       "34453"         "John"           "13"            "3"           "12" 
        marks4         marks5 parents.names1 parents.names2 
          "15"           "19"          "Ana"         "Mike" 
```

```r
mode(mylist)
```

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

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

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

```r
is.vector(unlist(mylist)) #atomic list with names
```

```
[1] TRUE
```

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

```
[1] TRUE
```

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

```
[1] FALSE
```

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

```
[1] FALSE
```

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.

Create a data frame

Note: dataframe turns categorical values to a factor by default

```r
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
```

Different ways to access the elements in a dataframe (table): `[]`, `[[]]`, `$`,

Indexes and names

Exercise:

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

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

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

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

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

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

```
  site
1    A
2    B
3    A
4    A
5    B
```

```r
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
```

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

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

```r
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
```

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
```

To change values in data frame - add 1 to `summer` `ph`:

```
my.dataframe[my.dataframe$season=='summer', 'ph'] <- my.dataframe[my.dataframe$season=='su
                                                                 my.dataframe[my.datafram
```

```
[1] 7.3 9.6
```

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

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

```
[1]  7.3 10.6
```

Add a column

```
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
```

How do you remove a column?

```
#my.dataframe$NO3<-NULL
my.dataframe <- my.dataframe[, -4]
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:

```
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
```

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

```
[1] 5
```

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

```
[1] 3
```

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

```
[1] 5 3
```

Edit a data frame:

```r
 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
```

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

Update names of the columns:

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

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

```r
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
```

```
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
```

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).

```
install.packages("tibble")
                                                                  library(tibble)
```

Create a tibble

```
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       31     87.8 a
 2       35     95   a
 3       31     87.8 a
 4       13     55.4 a
 5       11     51.8 a
 6       27     80.6 a
 7       39    102.  a
 8        5     41   a
 9        19      66.2 a
10        18      64.4 a
# i 90 more rows
```

Use the penguins data frame from the `palmerpenguins` package:

```
install.packages("palmerpenguins")
library(palmerpenguins)
data(penguins)
dim(penguins)
class(penguins)
penguins
```

```
[1] 344   8
```

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

```
# 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>
```

Convert a data frame to a tibble

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

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

```r
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>
```

Note: you can use `print(pe, n=Inf, width=Inf)` to print the entire `pe` dataset.

`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
#mode changed from 'numeric' to 'character', but class stays 'matrix'
```

However:

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

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

```r
mode(x)
```

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

```r
#class changed from 'matrix' to 'factor', but mode stays 'numeric'
#At this stage, even though x has mode numeric again, its new class, 'factor', prohibits i
```

A set of ‘is.xxx()‘ functions can be used to check the data structure of an object

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

```
[1] FALSE
```

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

```
[1] FALSE
```

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

```
[1] FALSE
```

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

```
[1] FALSE
```

```r
is_tibble(x)
```

```
[1] FALSE
```

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

```
[1] FALSE
```

```r
typeof(x)
```

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

Subsetting a tibble results in a smaller tibble

Note: this is different from data frame – subsetting a data frame could result in a vector, when subsetting result in one or one series of values

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

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

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

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

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

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

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

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

R data structure summary:

http://adv-r.had.co.nz/Data-structures.html

`dplyr`

`dplyr` library is very useful for manipulate table-like data: https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html

Data wrangling cheatsheet

https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

`filter()` vs. `select()`

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

`filter()` select a subset of rows.

These two are often used in a nested fashion (like SQL SELECT/WHERE)

Other useful functions provided by `dplyr:mutate`, `summarise`, `arrange`, and joins (e.g,. `left_join()`, `right_join()`)

```
install.packages("dplyr")
library(dplyr)
```

Select bill lengths and widths of species `Adelie`:

```
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?

```r
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>
```

```r
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
```

```r
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 handy and widely used.

```
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‘

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

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

```
students
```

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

$grades
[1] 78 85 99
```

```
gradeConvertor<- function (grade){
  grade = as.numeric(grade)
  if(grade > 100 | grade < 0) print ("grade out of the range")
```

```r
  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)

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"
```