한양대학교
HANYANG UNIVERSITY

Introduction to the R language (2)

Seoncheol Park

2.5 Packages, libraries, and repositories

- We have already mentioned several *packages*, i.e. `base`, `knitr`, and `chron`.
- In R, a package is a module containing functions, data, and documentation.
    - R always contains the base packages (e.g. `base`, `stats`, `graphics`); these contain things that everyone will use.
    - There are also contributed packages (e.g. `knitr` and `chron`); these are modules written by others to use in R.
- When you start your R session, you will have some packages loaded and available for use, while others are stored on your computer in a `library`. To be sure a package is loaded, run code like

```
library(knitr)
```

To see which packages are loaded, run

```
search() # Your list will likely be different from ours.
```

```
 [1] ".GlobalEnv"        "package:knitr"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:datasets"
 [7] "renv:shims"        "package:utils"     "package:methods"
[10] "Autoloads"         "package:base"
```

- **(WARNING)** A package can only contain one function of any given name, but the same name may be used in another package.

- When you use that function, R will choose it from the first package in the search list.

- you want to force a function to be chosen from a particular package, prefix the name of the function with the name of the package and `::`, e.g.

```
stats::median(x)
```

- If you try to use a package which is not installed on your computer, you will receive an error message:

```
library(notInstalled)
```

```
Error in library(notInstalled): there is no package called 'notInstalled'
```

- The biggest **repository** of R packages is known as CRAN. To install a package from CRAN, you can run a command like

```
install.packages("knitr")
```

or, within RStudio, click on the `Packages` tab in the Output Pane, choose `Install`, and enter the name in the resulting dialog box.

2.6.1 Help pages

- If you know the name of the function that you need help with, the `help()` function is likely sufficient.
  - It may be called with a string or function name as an argument, or
  - you can simply put a question mark (`?`) in front of your query.
- For example, for help on the `q()` function, type

```
?q #or
help(q)
```

or just hit the F1 key while pointing at q in RStudio. Any of these will open a help page containing a description of the function for quitting R.

- `help(mean)` tells us that `mean()` will compute the ordinary arithmetic average.

```
help(mean)
```

- `help.search()` or '??" are often used, when you don't know the function name.

```
??optimization

#or

help.search("optimization")
```

- You may find pages describing functions that you do not have installed, because they are in user-contributed packages. You can usually install them by typing

```
install.packages("packagename")
```

2.6.2 Built-in examples

- A useful supplement to `help()` is the `example()` function, which runs examples from the end of the help page:

```
example(mean)
```

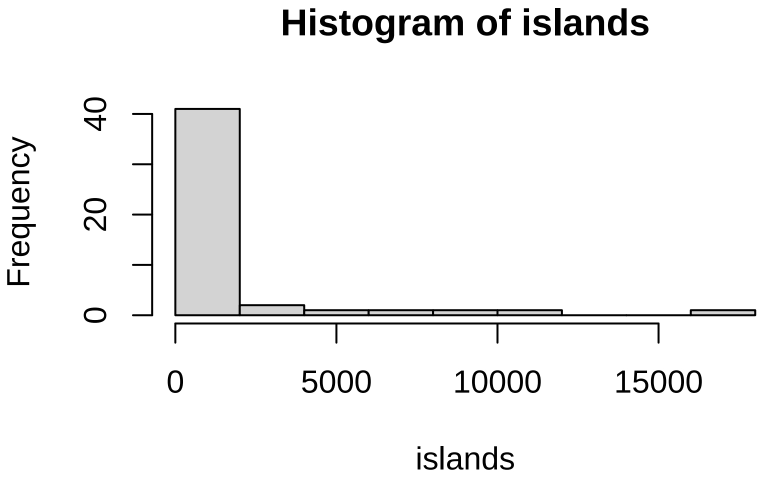
```
mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

- These examples show simple use of the `mean()` function as well as how to use the `trim` argument.
- When `trim = 0.1`, the highest 10% and lowest 10% of the data are deleted before the average is calculated.

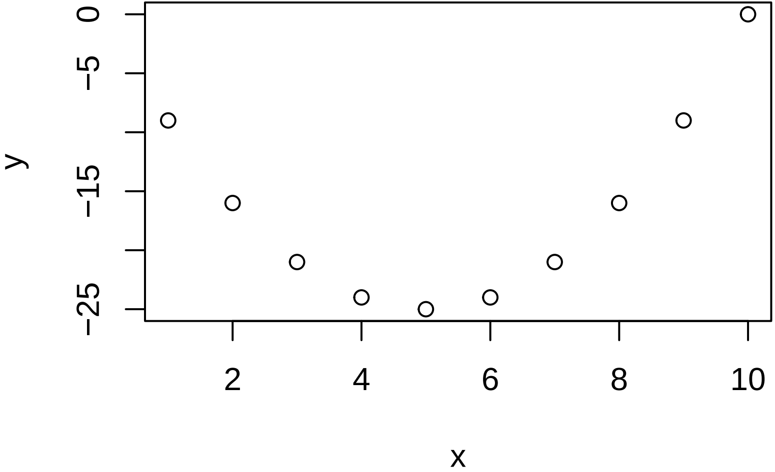
2.7.1 Some built-in graphics functions

- Two basic plots are the histogram and the scatterplot. The codes below were used to produce example graphs:

```
hist(islands)
```



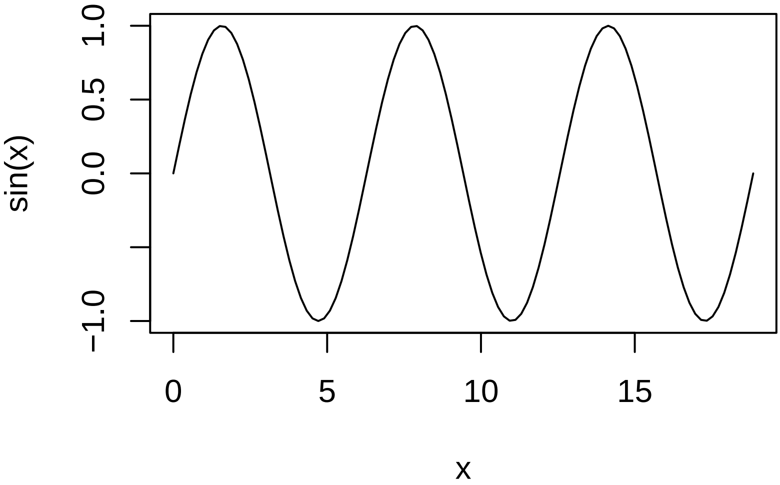
```r
x <- seq(1, 10)
y <- x^2 - 10 * x
plot(x, y)
```



Note that the `x` values are plotted along the horizontal axis.

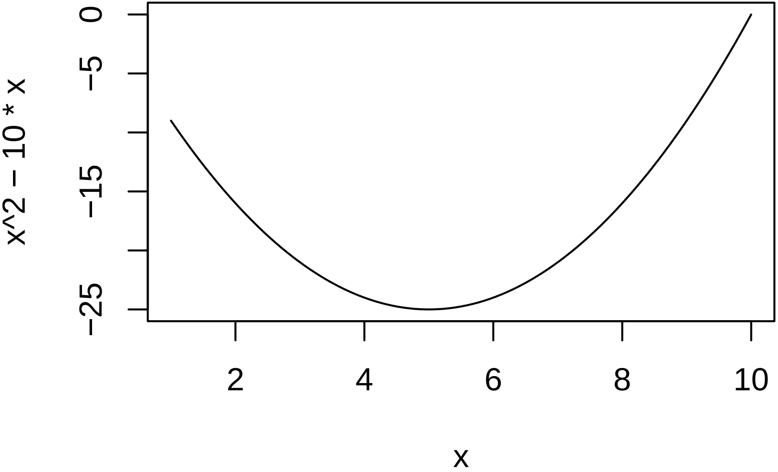
- Another useful plotting function is the `curve()` function for plotting the graph of a univariate mathematical function on an interval.

```
#plotting a sine function on [0,6pi]
curve(expr = sin, from = 0, to = 6 * pi)
```



- Note that the `expr` parameter is either a function (whose output is a numeric vector when the input is a numeric vector) or an expression in terms of `x`. An example of the latter type of usage is:

```
curve(expr = x^2 - 10 * x, from = 1, to = 10)
```



2.7.2 Some elementary built-in functions

The sample median

- The sample median measures the middle value of a data set. If the ordered data are $x[1] \leq x[2] \leq \ldots \leq x[n]$,

$$\text{median}(x) = \begin{cases} x[(n+1)/2], & n \text{ is odd} \\ \{[x[n/2] + x[n/2+1]\}/2 & n \text{ is even} \end{cases}.$$

```
values_1 <- c(10, 10, 18, 30, 32)
median(values_1)
```

```
[1] 18
```

```
values_2 <- c(40, 10, 10, 18, 30, 32)
median(values_2) #average of 18 and 30
```

```
[1] 24
```

Other summary measures

- Summary statistics can be calculated for data stored in vectors. In particular, try

```
var(x) # computes the variance of the data in x
summary(x) # computes several summary statistics on the data in x length(x)
          # number of elements in x
min(x) # minimum value of x
max(x) # maximum value of x
pmin(x, y) # pairwise minima of corresponding elements of x and y pmax(x, y)
          # pairwise maxima of x and y
range(x) # difference between maximum and minimum of data in x
IQR(x) # interquartile range: difference between 1st and 3rd
        # quartiles of data in x
```

- For an example of the calculation of pairwise minima of two vectors, consider

```
x <- 1:5
y <- 7:3
pmin(x,y)
```

```
[1] 1 2 3 4 3
```

2.7.3 Presenting results using R Markdown

- **R Markdown** is one way to make presenting results easier.
  - It is a mixture of **Markdown**, a simple way to write a document in a plain text file, and **chunks** of code in R or another computer language.
  - When you **render** the input into a document, R runs the code, automatically collects printed output and graphics and inserts them into the final document.
- The simplest way to start is to ask RStudio to produce an initial template; then you delete the sample material, add your own, and render it. Using the menus in RStudio, choose `File|New File|R Markdown`....
- You may choose an HTML document (a web page) or a PDF document.
  - In this template, the first part (between the two `---` lines) is called the **YAML**. It contains information that will be used when rendering your document.
  - The actual document starts after the YAML. Headings are marked with an initial `##`, and text is written out in an essentially normal way.
  - Instructions within the template tell you how to include code chunks that will display results.
  - To do the rendering, click on `Knit` in the top of the pane. This will ask to save the file if you haven't already done that, then render it and display the result on the screen.

2.8 Logical vectors and relational operators

2.8.1 Boolean algebra

- The idea of **Boolean algebra** is to formalize a mathematical approach to logic. Logic deals with statements that are either true or false.
- For example, let A is the statement that the sky is clear, and B is the statement that it is raining. Depending on the weather where you are,
  - (1) those two statements may both be true (there is a *sunshower*),
  - (2) A may be true and B false (the usual clear day),
  - (3) A false and B true (the usual rainy day), or
  - (4) both may be false (a cloudy but dry day).

A	B	not A	not B	A and B	A or B
`A`	`B`	`!A`	`!B`	`A & B`	`A | B`
`TRUE`	`TRUE`	`FALSE`	`FALSE`	`TRUE`	`TRUE`
`TRUE`	`FALSE`	`FALSE`	`TRUE`	`FALSE`	`TRUE`
`FALSE`	`TRUE`	`TRUE`	`FALSE`	`FALSE`	`TRUE`
`FALSE`	`FALSE`	`TRUE`	`TRUE`	`FALSE`	`FALSE`

2.8.2 Logical operations in R

- A logical vector may be constructed as

```
a <- c(TRUE, FALSE, FALSE, TRUE)
```

- Logical vectors may be used as indices. The elements of `b` corresponding to `TRUE` are selected.

```
b <- c(13, 7, 8, 2)
b[a]
```

```
[1] 13  2
```

- If we attempt arithmetic on a logical vector, then the operations are performed after converting `FALSE` to `0` and `TRUE` to `1`.

```
sum(a) #we count how many occurrences of TRUE are in the vector.
```

```
[1] 2
```

- There are two versions of the Boolean operators. The usual versions are `&`, `|` and `!`, as listed in the previous section. These are all vectorized.

```
!a
```

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

- If we attempt logical operations on a numerical vector, 0 is taken to be `FALSE`, and any non-zero value is taken to be `TRUE`:

```
a & (b - 2)
```

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

- The operators `&&` and `||` are similar to `&` and `|`, but behave differently in two respects.
  - First, they are not **vectorized**: only one calculation is done, and in newer versions of R, you'll get an error if you try to use them on longer vectors.
  - Second, they are guaranteed to be evaluated from left to right, with the right-hand operand only evaluated if necessary.

```
A <- FALSE; B <- TRUE
A && B
```

```
[1] FALSE
```

```
A <- FALSE; B <- FALSE
```

- This can save time if evaluating `B` would be very slow, and may make calculations easier, for example if evaluating `B` would cause an error when `A` was `FALSE`.

2.8.3 Relational operators

- R allows the relational operators: `<`, `>`, `==`, `>=`, `<=`, `!=`.

```
threeM <- c(3, 6, 9)
threeM > 4 # which elements are greater than 4
```

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

```
threeM == 4   # which elements are exactly equal to 4
```

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

```
threeM >= 4   # which elements are greater than or equal to 4
```

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

```
threeM != 4   # which elements are not equal to 4
```

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

```
threeM[threeM > 4] # elements of threeM which are greater than 4
```

```
[1] 6 9
```

```
four68 <- c(4, 6, 8)
four68 > threeM # four68 elements exceed corresponding threeM elements
```

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

```
four68[threeM < four68] # print them
```



```
[1] 4
```

2.9 Data frames, tibbles, and lists

- Data sets frequently consist of more than one column of data, where + each column represents measurements of a single variable, and
    - each row usually represents a single observation.
- This format is referred to as **case-by-variable** format.

- Most data sets are stored in R as `data.frame`. An example is `women` which contains the average `height`s (in inches) and `weight`s (in pounds) of American women aged 30 to 39:

```
head(women)
```

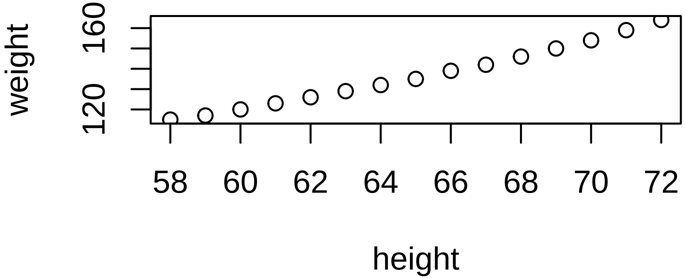
```
  height weight
1     58    115
2     59    117
3     60    120
4     61    123
5     62    126
6     63    129
```

- Other ways to view the data are through the use of the `summary()` function as shown below, or by constructing an appropriate graph:

```
summary(women)
```

```
     height         weight     
 Min.   :58.0   Min.   :115.0  
 1st Qu.:61.5   1st Qu.:124.5  
 Median :65.0   Median :135.0  
 Mean   :65.0   Mean   :136.7  
 3rd Qu.:68.5   3rd Qu.:148.0  
 Max.   :72.0   Max.   :164.0  
```

```
plot(weight ~ height, data = women)
```



- For larger data frames, a quick way of counting the number of rows and columns is important. The functions `nrow()` and `ncol()` play this role:

```
nrow(women)
```

```
[1] 15
```

```
ncol(women)
```

```
[1] 2
```

- We can get both at once using `dim()` (for dimension) and can get summary information using `str()` (for structure):

```
dim(women)
```

```
[1] 15  2
```

```
str(women)
```

```
'data.frame':   15 obs. of  2 variables:
 $ height: num  58 59 60 61 62 63 64 65 66 67 ...
 $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
```

- In fact, `str()` works with almost any R object, and is often a quick way to find what you are working with.

2.9.1 Extracting data frame elements and subsets

- We can extract elements from data frames using similar syntax to what was used with matrices.

```
women[7, 2]
```

```
[1] 132
```

```
#try at home
#women[3, ]; women[4:7, 1]
```

- Data frame columns can also be addressed using their names using the `$` operator. For example, the weight column can be extracted as follows:

```
women$weight
```

```
 [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164
```

- Thus, we can extract all heights for which the weights exceed 140 using

```
women$height[women$weight > 140]
```

```
[1] 67 68 69 70 71 72
```

- The `with()` function allows us to access columns of a `data.frame` directly without using the `$`.

```
with(women, weight/height)
```

```
 [1] 1.982759 1.983051 2.000000 2.016393 2.032258 2.047619 2.062500 2.076923
 [9] 2.106061 2.119403 2.147059 2.173913 2.200000 2.239437 2.277778
```

2.9.2 Taking random samples from populations

- The `sample()` function can be used to take samples (with or without replacement) from larger finite populations.
- Suppose that we have a data consisting of 15000 entries, and we would like to randomly select 8 entries (without replacement) for detailed study. It can be realized by selecting a random sample of indices:

```
sampleID <- sample(1:15000, size = 8, replace = FALSE)
sampleID
```

```
[1] 11759   904   776   654  2803 12814   713   497
```

2.9.3 Constructing data frames

- Use the `data.frame()` function to construct data frames from vectors that already exist in your workspace:

```
xy <- data.frame(x, y)
xy
```

```
  x y
1 1 7
2 2 6
3 3 5
4 4 4
5 5 3
```

- For another example, consider

```
xynew <- data.frame(x, y, new = 10:1)
```

2.9.4 Data frames can have non-numeric columns

- Columns of data frames can be of different types. For example, the built-in data frame `chickwts` has a numeric column and a factor. Again, the `summary()` function provides a quick peek at this data set.

```
summary(chickwts)
```

```
     weight             feed   
 Min.   :108.0   casein   :12  
 1st Qu.:204.5   horsebean:10  
 Median :258.0   linseed  :12  
 Mean   :261.3   meatmeal :11  
 3rd Qu.:323.5   soybean  :14  
 Max.   :423.0   sunflower:12  
```

- Here, displaying the entire data frame would have been a waste of space, as can be seen from:

```
nrow(chickwts)
```

```
[1] 71
```

- An important point to be aware of is that in older versions of R (before 4.0.0), the `data.frame()` function automatically converted character vectors to factors. As an example, consider the following data that might be used as a baseline in an obesity study:

```
gender <- c("M", "M", "F", "F", "F")
weight <- c(73, 68, 52, 69, 64)
obesityStudy <- data.frame(gender, weight)
```

- The vector `gender` is clearly a character vector, and in R 4.0.0 or later it will be left that way in the data frame:

```
obesityStudy$gender
```

```
[1] "M" "M" "F" "F" "F"
```

- If you want the older behavior, use the `stringsAsFactors = TRUE` argument when you create the data frame:

```
obesityStudy <- data.frame(gender, weight, stringsAsFactors = TRUE)
obesityStudy$gender
```

```
[1] M M F F F
Levels: F M
```

- Now, suppose we wish to globally change `F` to `Female` in the data frame. An incorrect approach is

```
wrongWay <- obesityStudy
whereF <- wrongWay$gender == "F"
wrongWay$gender[whereF] <- "Female"
```

```
Warning in `[<-.factor`(`*tmp*`, whereF, value = structure(c(2L, 2L, NA, :
invalid factor level, NA generated
```

```
wrongWay$gender
```

```
[1] M    M    <NA> <NA> <NA>
Levels: F M
```

- The correct approach is through the levels of the `obesityStudy$gender` factor:

```
levels(obesityStudy$gender)[1] <- "Female" # F is the 1st level -- why?
obesityStudy$gender # check that F was really replaced by Female
```

```
[1] M      M      Female Female Female
Levels: Female M
```

2.9.5 Lists

- Data frames are actually a special kind of **list**, or structure. Lists in R can contain any other objects.
- The `list()` function is one way of organizing multiple pieces of output from functions. For example,

```r
x <- c(3, 2, 3)
y <- c(7, 7)
z <- list(x = x, y = y)
z
```

```
$x
[1] 3 2 3


$y
[1] 7 7
```

- You can see the names of the objects in a list using the `names()` function, and extract parts of it:

```
names(z) # Print names of objects in list z
```

```
[1] "x" "y"
```

```
z$x # Print the x component of z
```

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

- There are several functions which make working with lists easy. Two of them are `lapply()` and `vapply()`. The `lapply()` function **applies** another function to every element of a list and returns the results in a new list.

```
lapply(z, mean)
```

```
$x
[1] 2.666667

$y
[1] 7
```

- Sometimes it might be more convenient to have the results in a vector; the `vapply()` function does that.

```
vapply(z, mean, 1)
```

```
       x        y 
2.666667 7.000000 
```

- If `mean()` had returned a different kind of result, `vapply()` would have given an error. If we expect more than a single value, the results will be organized into a matrix, e.g.

```
vapply(z, summary, numeric(6))
```

```
                x y
Min.     2.000000 7
1st Qu.  2.500000 7
Median   3.000000 7
Mean     2.666667 7
3rd Qu.  3.000000 7
Max.     3.000000 7
```

2.10 Data input and output

2.10.1 Changing directories

- In the RStudio Files Pane you can navigate to the directory where you want to work, and choose Set `As Working Directory` from the `More` menu item.

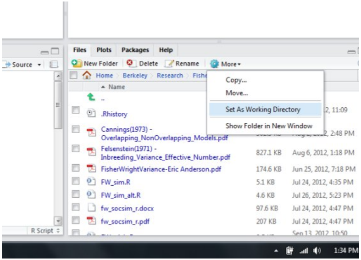


Figure 1: You can use File tab to change the working directory.

- Alternatively you can run the R function `setwd()`. For example, to work with data in the folder mydata on the C: drive, run

```
setwd("c:/mydata") # or setwd("c:\\mydata") #example: for WINDOWS
setwd("~/Desktops/mydata") #example: for MAC OS
```

2.10.2 `dump()` and `source()`

- Suppose you have constructed an R object called `usefuldata`. In order to save this object for a future session, type

```
dump("usefuldata", "useful.R")
```

This stores the command necessary to create the vector `usefuldata` into the file `useful.R` on your computer's hard drive. The choice of filename is up to you, as long as it conforms to the usual requirements for filenames on your computer.

- To retrieve the vector in a future session, type

```
dump(list = objects(), "all.R")
```

This produces a file called `all.R` on your computer's hard drive. Using `source("all.R")` at a later time will allow you to retrieve all of these objects.

Example 2.4

To save existing objects `humidity`, `temp` and `rain` to a file called `weather.R` on your hard drive, type

```
dump(c("humidity", "temp", "rain"), "weather.R")
```