Introduction to Spatial Data Programming - R

Lesson 04
Tables

Michael Dorman
Geography and Environmental Development
dorman@post.bgu.ac.il

Last updated: 2018-12-20 17:01:41

Ben-Gurion University of the Negev

Contents

Aims

- Learn to work with `data.frame`, the data structure used to represent tables in R
- Learn several automation methods for controlling code execution and automation in R -
  - Conditionals
  - Loops
  - The `apply` function
- Install and use packages beyond "base R"
- Join between tables

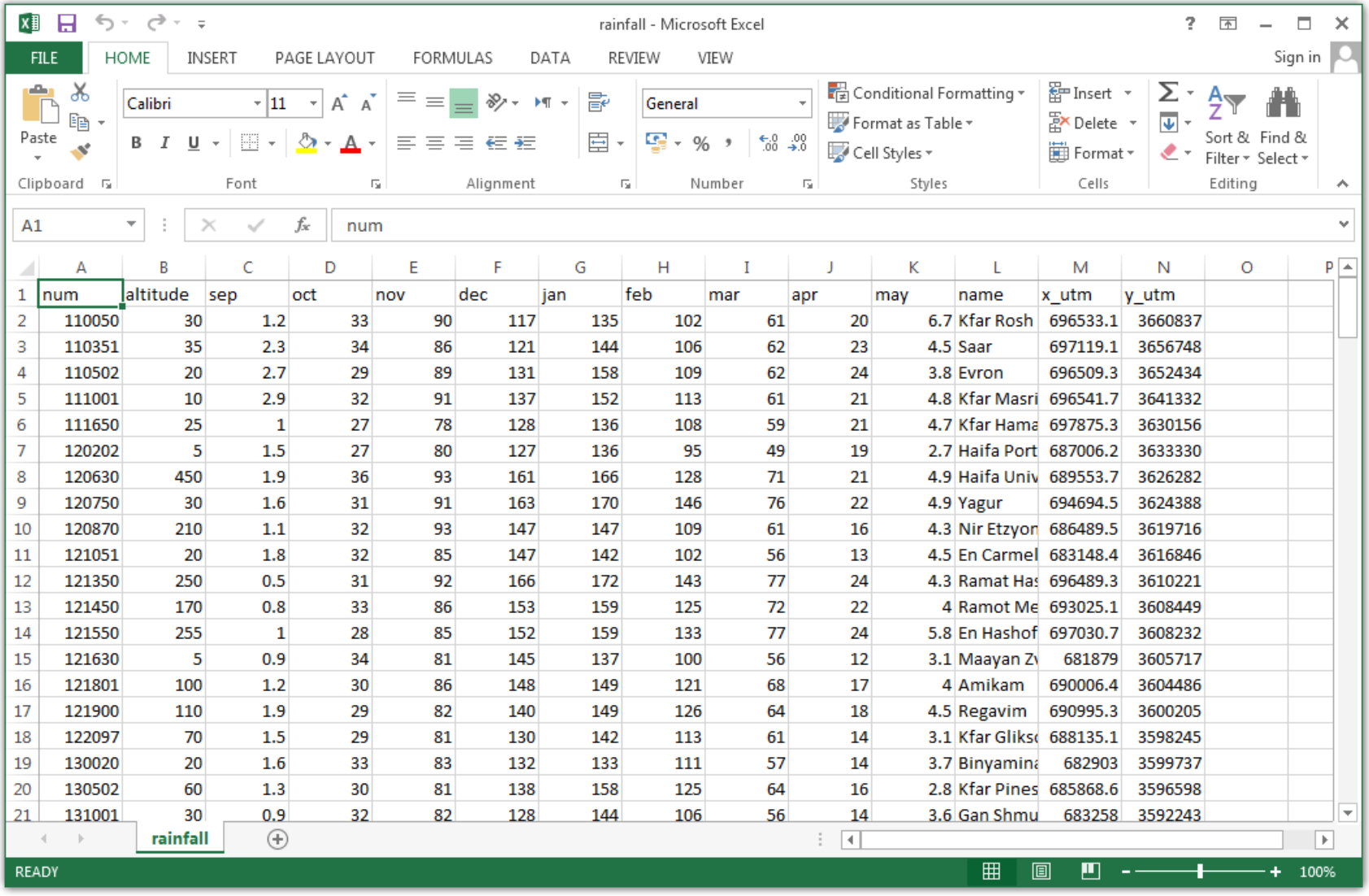
Preparation

```
library(dplyr)
```

Table

- A **table** in R is represented using the `data.frame` class
- A `data.frame` is basically a collection of **vectors** comprising columns, all of the **same size** but possibly of **different types**
- Conventionally -
  - Each row represents an **observation**, with values possibly of a **different** type for each variable
  - Each column represents a **variable**, with values of the **same** type

Table



num	altitude	sep	oct	nov	dec	jan	feb	mar	apr	may	name	x_utm	y_utm
110050	30	1.2	33	90	117	135	102	61	20	6.7	Kfar Rosh	696533.1	3660837
110351	35	2.3	34	86	121	144	106	62	23	4.5	Saar	697119.1	3656748
110502	20	2.7	29	89	131	158	109	62	24	3.8	Evron	696509.3	3652434
111001	10	2.9	32	91	137	152	113	61	21	4.8	Kfar Masri	696541.7	3641332
111650	25	1	27	78	128	136	108	59	21	4.7	Kfar Hama	697875.3	3630156
120202	5	1.5	27	80	127	136	95	49	19	2.7	Haifa Port	687006.2	3633330
120630	450	1.9	36	93	161	166	128	71	21	4.9	Haifa Univ	689553.7	3626282
120750	30	1.6	31	91	163	170	146	76	22	4.9	Yagur	694694.5	3624388
120870	210	1.1	32	93	147	147	109	61	16	4.3	Nir Etzyon	686489.5	3619716
121051	20	1.8	32	85	147	142	102	56	13	4.5	En Carmel	683148.4	3616846
121350	250	0.5	31	92	166	172	143	77	24	4.3	Ramat Has	696489.3	3610221
121450	170	0.8	33	86	153	159	125	72	22	4	Ramot Me	693025.1	3608449
121550	255	1	28	85	152	159	133	77	24	5.8	En Hashof	697030.7	3608232
121630	5	0.9	34	81	145	137	100	56	12	3.1	Maayan Zv	681879	3605717
121801	100	1.2	30	86	148	149	121	68	17	4	Amikam	690006.4	3604486
121900	110	1.9	29	82	140	149	126	64	18	4.5	Regavim	690995.3	3600205
122097	70	1.5	29	81	130	142	113	61	14	3.1	Kfar Gliks	688135.1	3598245
130020	20	1.6	33	83	132	133	111	57	14	3.7	Binyamina	682903	3599737
130502	60	1.3	30	81	138	158	125	64	16	2.8	Kfar Pines	685868.6	3596598
131001	30	0.9	32	82	128	144	106	56	14	3.6	Gan Shmu	683258	3592243

Figure 1: `rainfall.csv` opened in Excel

Creating a data.frame

- A data.frame can be **created** with the data.frame function, given one or more vectors which become columns
- The stringAsFactors=FALSE argument *prevents* the conversion of **text columns** to factor, which is what we usually want

```
d = data.frame(
  num = c(3, 7, 1),
  word = c("three", "seven", "one"),
  stringsAsFactors = FALSE
)
d
##   num  word
## 1   3 three
## 2   7 seven
## 3   1   one
```

Interactive view of a `data.frame`

- The `View` function opens an **interactive view** of a `data.frame`
- When using RStudio, the view also has **sort** and **filter** buttons
- Note: sorting or filtering the view have no effect on the object

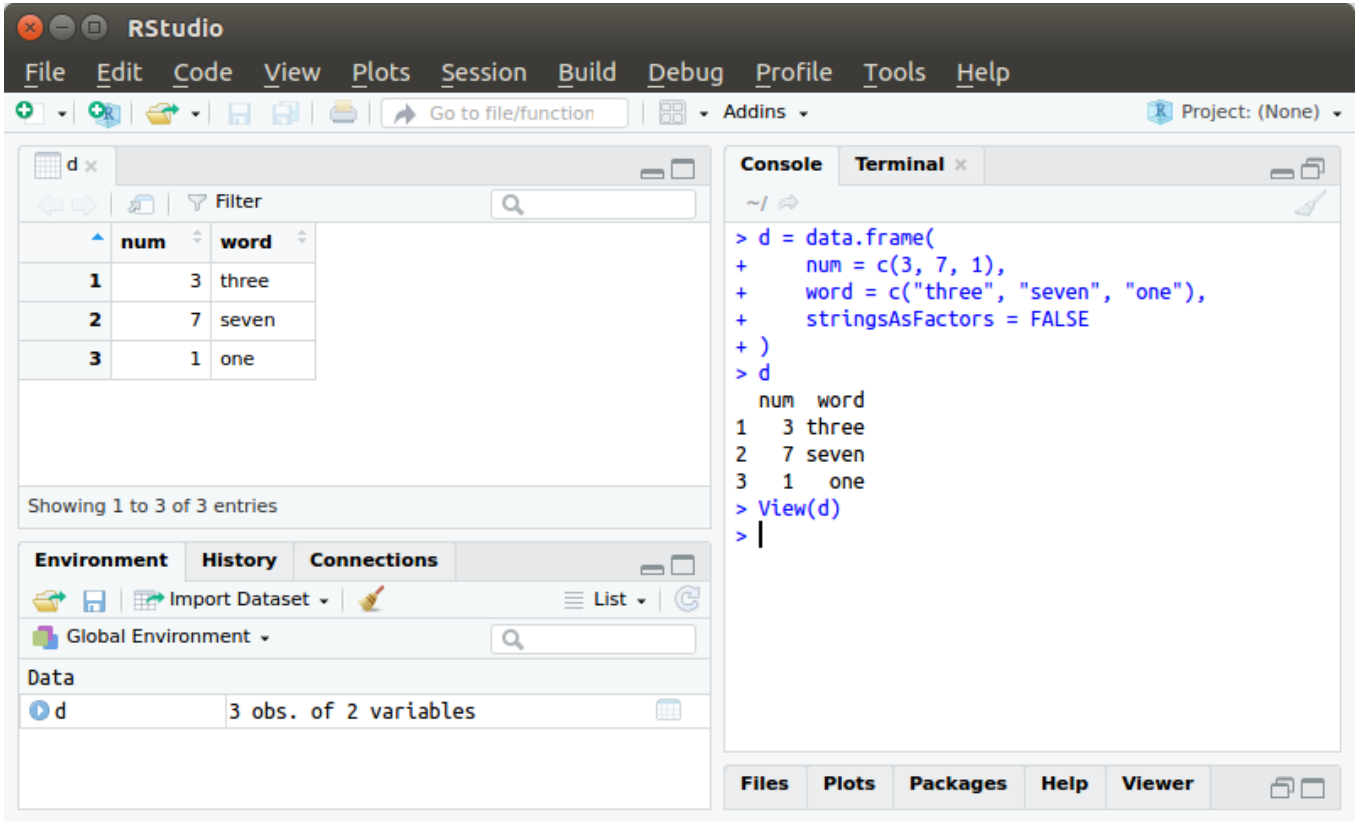


Figure 2: Table view in Rstudio

data.frame properties

- We can get the number of **rows** and number of **columns** in a `data.frame` with `nrow` and `ncol`, respectively -

```
nrow(d)
## [1] 3
```

```
ncol(d)
## [1] 2
```

- Or **both** of them as a vector of length 2 with `dim` -

```
dim(d)
## [1] 3 2
```

data.frame properties

- data.frames have row and column **names**, which we can get with rownames and colnames, respectively -

```
rownames(d)
## [1] "1" "2" "3"
```

```
colnames(d)
## [1] "num"  "word"
```

data.frame properties

- We can also **set** row or column names by assigning new values to these properties -

```
colnames(d)[1] = "Number"
d
##   Number  word
## 1      3 three
## 2      7 seven
## 3      1   one
```

```
colnames(d)[1] = "num"
d
##   num  word
## 1   3 three
## 2   7 seven
## 3   1   one
```

data.frame properties

- `str` gives a summary of the object **structure** -

```
str(d)
## 'data.frame':    3 obs. of  2 variables:
##  $ num : num  3 7 1
##  $ word: chr  "three" "seven" "one"
```

`data.frame` subsetting

- A `data.frame` **subset** can be obtained with the `[` operator
- A `data.frame` is a two-dimensional object, therefore the index is composed of **two vectors** -
  - The first vector refers to **rows**
  - The second vector refers to **columns**
- Each of these vectors can be one of the following **types** -
  - `numeric` - Specifying the **indices** of rows/columns to retain
  - `character` - Specifying the **names** of rows/columns to retain
  - `logical` - Specifying **whether** to retain each row/column
- The rows or column index can be **omitted**, in which case we get all rows or all columns, respectively

data.frame subsetting

```
d[1, 1]          # Row 1, column 1
## [1] 3

d[c(1, 3), 2]    # Rows 1 & 3, column 2
## [1] "three" "one"

d[2, ]           # Row 2
##   num  word
## 2   7 seven

d[, 2:1]         # Columns 2 & 1
##    word num
## 1 three   3
## 2 seven   7
## 3   one   1
```

data.frame subsetting

- A subset with a **single column** can be returned as -
  - A vector (`drop=TRUE`, the default)
  - A `data.frame` (`drop=FALSE`)
- For example -

```
d[1:2, 1]
## [1] 3 7
```

```
d[1:2, 1, drop = FALSE]
##   num
## 1   3
## 2   7
```

- Question: why do you think this applies to a single column, rather than a single row?

data.frame subsetting

- We can also use a `character` index to specify the **names** of rows and/or columns to retain in the subset -

```
d[, "num"]
## [1] 3 7 1
```

- The `$` operator is a shortcut for getting a **single column** from a `data.frame` -

```
d$num
## [1] 3 7 1
```

```
d$word
## [1] "three" "seven" "one"
```

Subset operators in R

Table 1: Subset operators in R

Syntax	Objects	Returns
`x[i]`	**vector**, **table**, **matrix**, **array**, **list**	Subset `i`
`x[[i]]`	vectors, **lists**	Single element `i`
`x$i`	**tables**, lists	Single element `i`
`x@n`	S4 objects	Slot `n`

data.frame subsetting

- The third option for a `data.frame` index is a `logical` vector, specifying **whether** to retain each row or column
- Most commonly it is used to filter `data.frame` **rows** based on the values of one or more **columns** -

```
d[d$num == 1, ]
##   num word
## 3   1  one
```

```
d[d$word == "seven" | d$word == "three", ]
##   num  word
## 1   3 three
## 2   7 seven
```

data.frame subsetting

- Let's go back to the Kinneret example from **Lesson 03** -

```
may = c(
  -211.92,-208.80,-208.84,-209.12,-209.01,-209.60,
  -210.24,-210.46,-211.76,-211.92,-213.13,-213.18,
  -209.74,-208.92,-209.73,-210.68,-211.10,-212.18,
  -213.26,-212.65,-212.37
)
nov = c(
  -212.79,-209.52,-209.72,-210.94,-210.85,-211.40,
  -212.01,-212.25,-213.00,-213.71,-214.78,-214.34,
  -210.93,-210.69,-211.64,-212.03,-212.60,-214.23,
  -214.33,-213.89,-213.68
)
```

data.frame subsetting

- Using these vectors we will construct the following `data.frame` -

```
kineret = data.frame(year = 1991:2011, may, nov)
```

```
head(kineret)
##   year     may     nov
## 1 1991 -211.92 -212.79
## 2 1992 -208.80 -209.52
## 3 1993 -208.84 -209.72
## 4 1994 -209.12 -210.94
## 5 1995 -209.01 -210.85
## 6 1996 -209.60 -211.40
```

data.frame subsetting

- Using a **logical index** we can get a subset of years when the Kinneret level in May was less than -213 -

```
kineret[kineret$may < -213, "year"]   # Method 1
## [1] 2001 2002 2009
```

```
kineret$year[kineret$may < -213]      # Method 2
## [1] 2001 2002 2009
```

- Similarly, we can get a subset with **all data** for those years -

```
kineret[kineret$may < -213, ]
##    year     may     nov
## 11 2001 -213.13 -214.78
## 12 2002 -213.18 -214.34
## 19 2009 -213.26 -214.33
```

Creating new columns

- Assignment into a column which does not exist adds a **new column** -

```
kineret$d_nov = c(NA, diff(kineret$nov))
```

```
head(kineret)
##   year      may      nov d_nov
## 1 1991 -211.92 -212.79    NA
## 2 1992 -208.80 -209.52  3.27
## 3 1993 -208.84 -209.72 -0.20
## 4 1994 -209.12 -210.94 -1.22
## 5 1995 -209.01 -210.85  0.09
## 6 1996 -209.60 -211.40 -0.55
```

Flow control

- The default is to let the computer execute **all** expressions in the same **order** they are given in the code
- Flow control commands are a way to **modify** the sequence of code execution
- We learn two flow control operators, from two categories -
  - `if` and `else` - A **conditional**, conditioning the execution of code
  - `for` - A **loop**, executing code more than once

Conditionals

- The purpose of the **conditional** is to condition the execution of code
- An `if-else` conditional in R contains the following components -
  - The `if` keyword
  - A condition
  - Code to be executed if the condition is `TRUE`
  - The `else` keyword (optional)
  - Code to be executed if the condition is `FALSE` (optional)
- The condition needs to be evaluated to a **single logical value** (`TRUE` or `FALSE`); If it is `TRUE` then the code section after `if` is executed

Conditionals

- Conditional with `if` -

```
if(condition) {
  expressions
}
```

- Conditional with `if` and `else` -

```
if(condition) {
  trueExpressions
} else {
  falseExpressions
}
```

Conditionals

- For example -

```
x = 3
x > 2
## [1] TRUE

if(x > 2) print("x is large!")
## [1] "x is large!"
```

```
x = 0
x > 2
## [1] FALSE

if(x > 2) print("x is large!")
```

Conditionals

- Now also with `else`
- The first code section is executed when the condition is `TRUE` -

```
x = 3
if(x > 2) print("x is large!") else print("x is small!")
## [1] "x is large!"
```

- The second code section is executed when the condition is `FALSE` -

```
x = 1
if(x > 2) print("x is large!") else print("x is small!")
## [1] "x is small!"
```

Conditionals

- We can use a conditional to write our own version of the `abs` function -

```
abs2 = function(x) {
  if(x >= 0) return(x) else return(-x)
}
```

- For example -

```
abs2(-3)
## [1] 3
```

```
abs2(24)
## [1] 24
```

Conditionals

- Question 1: what happens when the argument of `abs2` is of length >1?
- Question 2: write a function `is_myname` that checks if the given text is your name, returning `TRUE` or `FALSE`

```
is_myname("Michael")
## [1] TRUE
is_myname("Yossi")
## [1] FALSE
```

Loops

- A **loop** is used to execute a given code section more than once
- The number of **times** the code is executed is determined in different ways in different types of loops
- In a `for` loop, the number of times the code is executed is determined in advance based on the length of a **vector** passed to the loop
- The code is executed once for **each element** of the vector
- In each "round", the current element is assigned to a **variable** which we can use in the loop code

Loops

- A `for` loop is composed of the following parts -
  - The `for` keyword
  - The variable name `symbol` getting the current vector value
  - The `in` keyword
  - The vector `sequence`
  - A code section `expressions`

```
for(symbol in sequence) {
  expressions
}
```

- Note: the constant keywords are just `for` and `in`

Loops

- For example -

```
for(i in 1:5) print(i)
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

- The expression `print(i)` is executed 5 times, according to the length of the vector `1:5`
- Each time, `i` gets the next value of `1:5` and the code section prints `i` on screen

Loops

- The vector defining the `for` loop does not necessarily need to be numeric -

```
for(b in c("Test", "One", "Two")) print(b)
## [1] "Test"
## [1] "One"
## [1] "Two"
```

- The expression `print(b)` is executed 3 times, according to the length of the vector `c("Test", "One", "Two")`
- Each time, `b` gets the next value of the vector and the code section prints `b` on screen

Loops

- In case the vector is numeric, it does not necessarily need to be composed of consecutive -

```
for(i in c(1,15,3)) print(i)
## [1] 1
## [1] 15
## [1] 3
```

- The expression `print(i)` is executed 3 times, according to the length of the vector `c(1,15,3)`
- Each time, `i` gets the next value of the vector and the code section prints `i` on screen

Loops

- The code section does not have to use the current value of the vector -

```
for(i in 1:5) print("A")
## [1] "A"
## [1] "A"
## [1] "A"
## [1] "A"
## [1] "A"
```

- The expression `print("A")` is executed 5 times, according to the length of the vector `1:5`
- Each time, the code section prints `"A"` on screen

Loops

- The following `for` loop prints each of the numbers from 1 to 10 multiplied by 5 -

```
for(i in 1:10) print(i * 5)
## [1] 5
## [1] 10
## [1] 15
## [1] 20
## [1] 25
## [1] 30
## [1] 35
## [1] 40
## [1] 45
## [1] 50
```

Loops

- Question: How can we prints a multiplication table for 1-10, using a `for` loop?

```
##  [1]  1  2  3  4  5  6  7  8  9 10
##  [1]  2  4  6  8 10 12 14 16 18 20
##  [1]  3  6  9 12 15 18 21 24 27 30
##  [1]  4  8 12 16 20 24 28 32 36 40
##  [1]  5 10 15 20 25 30 35 40 45 50
##  [1]  6 12 18 24 30 36 42 48 54 60
##  [1]  7 14 21 28 35 42 49 56 63 70
##  [1]  8 16 24 32 40 48 56 64 72 80
##  [1]  9 18 27 36 45 54 63 72 81 90
##  [1]  10  20  30  40  50  60  70  80  90 100
```

Loops

- As another example of using a `for` loop, we can write a function named `x_in_y`
- The function accepts **two vectors** `x` and `y`
- For each **element** in `x` the function checks if it is **found** in `y`
- It returns a `logical` vector of the same length as `x`
- For example -

```
x_in_y(c(1, 2, 3, 4, 5), c(2, 1, 5))
## [1]  TRUE  TRUE FALSE FALSE  TRUE
```

Loops

- We can use a `for` **loop** to check if each element in `x` is contained in `y` -

```
x = c(1, 2, 3, 4, 5)
y = c(2, 1, 5)
```

```
for(i in x) print(any(i == y))
## [1] TRUE
## [1] TRUE
## [1] FALSE
## [1] FALSE
## [1] TRUE
```

Loops

- In a function, we can **"collect"** the results into a vector instead of printing them on screen -

```
# Version 1
x_in_y = function(x, y) {
  result = NULL
  for(i in x) result = c(result, any(i == y))
  result
}
```

```
# Version 2
x_in_y = function(x, y) {
  result = rep(NA, length(x))
  for(i in 1:length(x)) result[i] = any(x[i] == y)
  result
}
```

Loops

- In fact, we don't need to write a function such as `x_in_y` ourselves; we can use the `%in%` **operator**
- The `%in%` operator, with an expression `x %in% y`, returns a `logical` vector indicating the **presence** of each element of `x` in `y` -

```
1:5 %in% c(1, 2, 5)
## [1]  TRUE  TRUE FALSE FALSE  TRUE
```

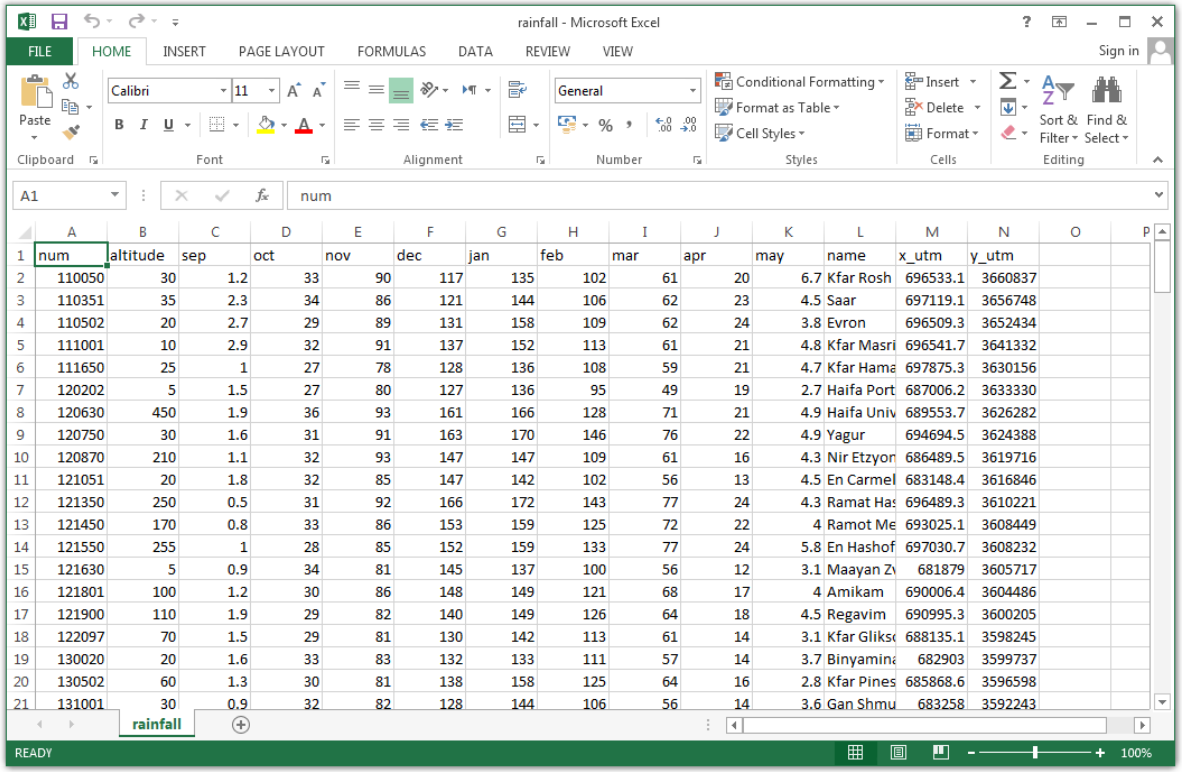
```
1:5 %in% c(1, 2, 3, 5)
## [1]  TRUE  TRUE  TRUE FALSE  TRUE
```

```
c("a", "B", "c", "ee") %in% letters
## [1]  TRUE FALSE  TRUE FALSE
```

```
c("a", "B", "c", "ee") %in% LETTERS
## [1] FALSE  TRUE FALSE FALSE
```

Reading table from a file

- The table `rainfall.csv` contains average **monthly rainfall** data (based on the period 1980-2010) for September-May, in 169 meteorological stations in Israel
- The table also contains station **name**, station **number**, **elevation** and **X-Y** coordinates



num	altitude	sep	oct	nov	dec	jan	feb	mar	apr	may	name	x_utm	y_utm
110050	30	1.2	33	90	117	135	102	61	20	6.7	Kfar Rosh	696533.1	3660837
110351	35	2.3	34	86	121	144	106	62	23	4.5	Saar	697119.1	3656748
110502	20	2.7	29	89	131	158	109	62	24	3.8	Evron	696509.3	3652434
111001	10	2.9	32	91	137	152	113	61	21	4.8	Kfar Masri	696541.7	3641332
111650	25	1	27	78	128	136	108	59	21	4.7	Kfar Hama	697875.3	3630156
120202	5	1.5	27	80	127	136	95	49	19	2.7	Haifa Port	687006.2	3633330
120630	450	1.9	36	93	161	166	128	71	21	4.9	Haifa Univ	689553.7	3626282
120750	30	1.6	31	91	163	170	146	76	22	4.9	Yagur	694694.5	3624388
120870	210	1.1	32	93	147	147	109	61	16	4.3	Nir Etzyon	686489.5	3619716
121051	20	1.8	32	85	147	142	102	56	13	4.5	En Carmel	683148.4	3616846
121350	250	0.5	31	92	166	172	143	77	24	4.3	Ramat Has	696489.3	3610221
121450	170	0.8	33	86	153	159	125	72	22	4	Ramot Me	693025.1	3608449
121550	255	1	28	85	152	159	133	77	24	5.8	En Hashof	697030.7	3608232
121630	5	0.9	34	81	145	137	100	56	12	3.1	Maayan Zv	681879	3605717
121801	100	1.2	30	86	148	149	121	68	17	4	Amikam	690006.4	3604486
121900	110	1.9	29	82	140	149	126	64	18	4.5	Regavim	690995.3	3600205
122097	70	1.5	29	81	130	142	113	61	14	3.1	Kfar Gliks	688135.1	3598245
130020	20	1.6	33	83	132	133	111	57	14	3.7	Binyamina	682903	3599737
130502	60	1.3	30	81	138	158	125	64	16	2.8	Kfar Pines	685868.6	3596598
131001	30	0.9	32	82	128	144	106	56	14	3.6	Gan Shmu	683258	3592243

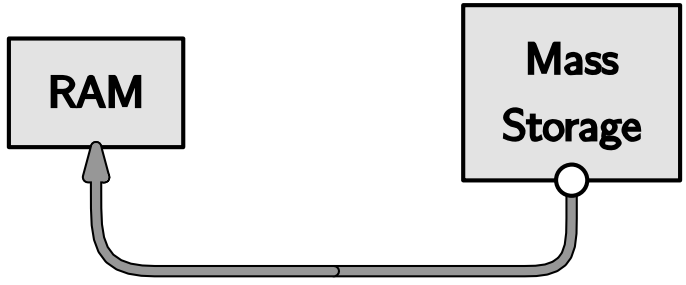
Figure 3: `rainfall.csv` opened in Excel

Reading table from a file

- In addition to creating a table with `data.frame`, we can read an existing table **from disk**, such as from a Comma-Separated Values (CSV) file
- We can read a CSV file with `read.csv` -

```
# Windows
read.csv("C:\\Data2\\rainfall.csv")
read.csv("C:/Data2/rainfall.csv")

# Mac / Linux
read.csv("~/Dropbox/Data2/rainfall.csv")
```



Reading table from a file

- Note: the separating character on Windows is / or \\, not the familiar \
- In case the file path uses the **incorrect separator** \ we get an error -

```
read.csv("C:\Data2\rainfall.csv")
## Error: '\D' is an unrecognized escape in character stri
```

- In case the file **does not exist** we get a different error -

```
read.csv("C:\\Data2\\rainfall.csv")
## Warning in file(file, "rt"): cannot open file 'C:
## \Data2\rainfall.csv': No such file or directory
## Error in file(file, "rt"): cannot open the connection
```

Reading table from a file

- The `stringsAsFactors` parameter of `read.csv` (and several other functions) determines whether **text columns** are converted to `factor` (default is `TRUE`)
- Usually, we want to **avoid** the conversion with `stringsAsFactors=FALSE`

```
read.csv(
  "~/Dropbox/Data2/rainfall.csv",
  stringsAsFactors = FALSE
)
```

Working directory

- R always points to a certain directory, knows as the **working directory**
- We can **get** the current working directory with `getwd` -

```
getwd()
## [1] "/home/michael/Dropbox/Data2"
```

- We can **set** a new working directory with `setwd` -

```
setwd("~/Dropbox/Data2")
```

- When **reading a file** from the working directory, we can specify just the **file name** instead of the full path -

```
read.csv("rainfall.csv")
```

data.frame structure

- Let's **read** the `rainfall.csv` file -

```
rainfall = read.csv(
  "rainfall.csv",
  stringsAsFactors = FALSE
)
```

data.frame structure

- We can also use the `head` or `tail` function which return a subset of the **first** or **last** several rows, respectively -

```
head(rainfall)
```

```
##      num altitude sep oct nov dec jan feb mar apr
## 1 110050       30 1.2  33  90 117 135 102  61  20
## 2 110351       35 2.3  34  86 121 144 106  62  23
## 3 110502       20 2.7  29  89 131 158 109  62  24
## 4 111001       10 2.9  32  91 137 152 113  61  21
## 5 111650       25 1.0  27  78 128 136 108  59  21
## 6 120202        5 1.5  27  80 127 136  95  49  19
##   may              name    x_utm   y_utm
## 1 6.7 Kfar Rosh Hanikra 696533.1 3660837
## 2 4.5              Saar 697119.1 3656748
## 3 3.8             Evron 696509.3 3652434
## 4 4.8       Kfar Masrik 696541.7 3641332
## 5 4.7     Kfar Hamakabi 697875.3 3630156
## 6 2.7        Haifa Port 687006.2 3633330
```

data.frame structure

- We can also use the `head` or `tail` function which return a subset of the **first** or **last** several rows, respectively -

```
tail(rainfall)
```

```
##         num altitude sep oct nov dec jan feb mar
## 164 321800     -180 0.2  12  37  55  65  59  36
## 165 321850     -220 0.2  13  33  53  64  56  35
## 166 330370     -375 0.1   6  10  20  22  19  11
## 167 337000     -390 0.0   5   3  10   7   7   7
## 168 345005       80 0.4   2   2   6   5   4   5
## 169 347702       11 0.0   4   2   5   4   3   3
##     apr may         name    x_utm   y_utm
## 164  11 5.0 Sde Eliyahu 736189.3 3591636
## 165  11 4.7   Tirat Zvi 737522.4 3590062
## 166   6 1.3       Kalya 733547.8 3515345
## 167   3 0.5        Sdom 728245.6 3435503
## 168   2 0.4     Yotveta 700626.3 3307819
## 169   2 1.0       Eilat 689139.3 3270290
```

data.frame structure

- We can check the table **structure** with `str` -

```
str(rainfall)
```

```
## 'data.frame':    169 obs. of  14 variables:
##  $ num     : int  110050 110351 110502 111001 111650 120202 120630 120750 12
##  $ altitude: int  30 35 20 10 25 5 450 30 210 20 ...
##  $ sep     : num  1.2 2.3 2.7 2.9 1 1.5 1.9 1.6 1.1 1.8 ...
##  $ oct     : int  33 34 29 32 27 27 36 31 32 32 ...
##  $ nov     : int  90 86 89 91 78 80 93 91 93 85 ...
##  $ dec     : int  117 121 131 137 128 127 161 163 147 147 ...
##  $ jan     : int  135 144 158 152 136 136 166 170 147 142 ...
##  $ feb     : int  102 106 109 113 108 95 128 146 109 102 ...
##  $ mar     : int  61 62 62 61 59 49 71 76 61 56 ...
##  $ apr     : int  20 23 24 21 21 19 21 22 16 13 ...
##  $ may     : num  6.7 4.5 3.8 4.8 4.7 2.7 4.9 4.9 4.3 4.5 ...
##  $ name    : chr  "Kfar Rosh Hanikra" "Saar" "Evron" "Kfar Masrik" ...
##  $ x_utm   : num  696533 697119 696509 696542 697875 ...
##  $ y_utm   : num  3660837 3656748 3652434 3641332 3630156 ...
```

`data.frame` structure

- Question: create a plot of rainfall in January (`jan`) as function of elevation (`altitude`) based on the `rainfall` table

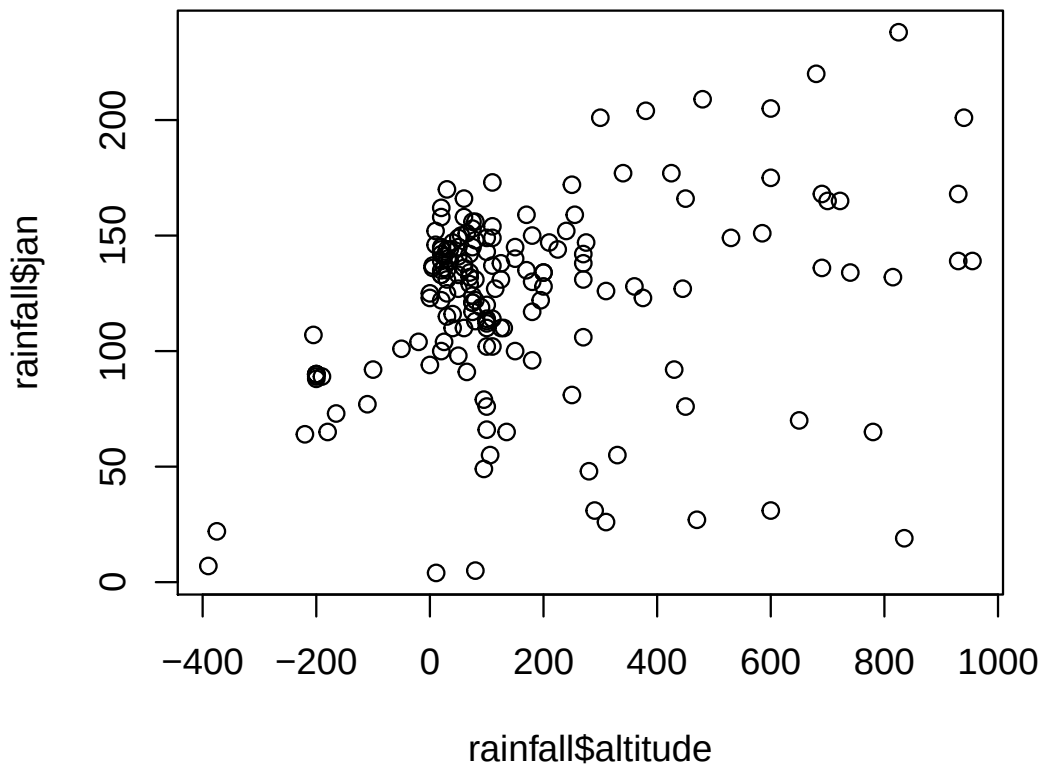


Figure 4: Rainfall in January as function of elevation

data.frame structure

- We can get specific information from the table trough **subsetting** and **summarizing**
- What is the elevation of the lowest and highest stations?

```
min(rainfall$altitude)
## [1] -390
```

```
max(rainfall$altitude)
## [1] 955
```

- What is the name of the lowest and highest station?

```
rainfall$name[which.min(rainfall$altitude)]
## [1] "Sdom"
```

```
rainfall$name[which.max(rainfall$altitude)]
## [1] "Rosh Tzurim"
```

data.frame structure

- How much rainfall does the "Haifa University" station receive in April?

```
rainfall$apr[rainfall$name == "Haifa University"]
## [1] 21
```

Creating new table columns

- We can create a **new column** through assignment -

```
rainfall$sep_oct = rainfall$sep + rainfall$oct
```

- To accomodate more complex calculations, we can also create a new column inside a `for` **loop** going over table **rows** -

```
m = c(
  "sep", "oct", "nov", "dec", "jan",
  "feb", "mar", "apr", "may"
)
for(i in 1:nrow(rainfall)) {
  rainfall$annual[i] = sum(rainfall[i, m])
}
```

Creating new table columns

```
head(rainfall)
```

```
##      num altitude sep oct nov dec jan feb mar apr may
## 1 110050       30 1.2  33  90 117 135 102  61  20 6.7
## 2 110351       35 2.3  34  86 121 144 106  62  23 4.5
## 3 110502       20 2.7  29  89 131 158 109  62  24 3.8
## 4 111001       10 2.9  32  91 137 152 113  61  21 4.8
## 5 111650       25 1.0  27  78 128 136 108  59  21 4.7
## 6 120202        5 1.5  27  80 127 136  95  49  19 2.7
##                 name    x_utm   y_utm sep_oct annual
## 1 Kfar Rosh Hanikra 696533.1 3660837    34.2  565.9
## 2              Saar 697119.1 3656748    36.3  582.8
## 3             Evron 696509.3 3652434    31.7  608.5
## 4       Kfar Masrik 696541.7 3641332    34.9  614.7
## 5     Kfar Hamakabi 697875.3 3630156    28.0  562.7
## 6        Haifa Port 687006.2 3633330    28.5  537.2
```

The `apply` function

- The `apply` function can **replace** `for` loops, in situations when we are interested in applying the **same function** on all subsets of certain **dimension** of a `data.frame`, a `matrix` or an `array`
- In case of a table, there are two **dimensions** we can work on -
  - **Rows** = Dimension 1
  - **Columns** = Dimension 2
- The `apply` function needs three arguments -
  - `X` - The **object** we are working on: `data.frame`, `matrix` or `array`
  - `MARGIN` - The **dimension** we are working on
  - `FUN` - The **function** applied on that dimension

The apply function

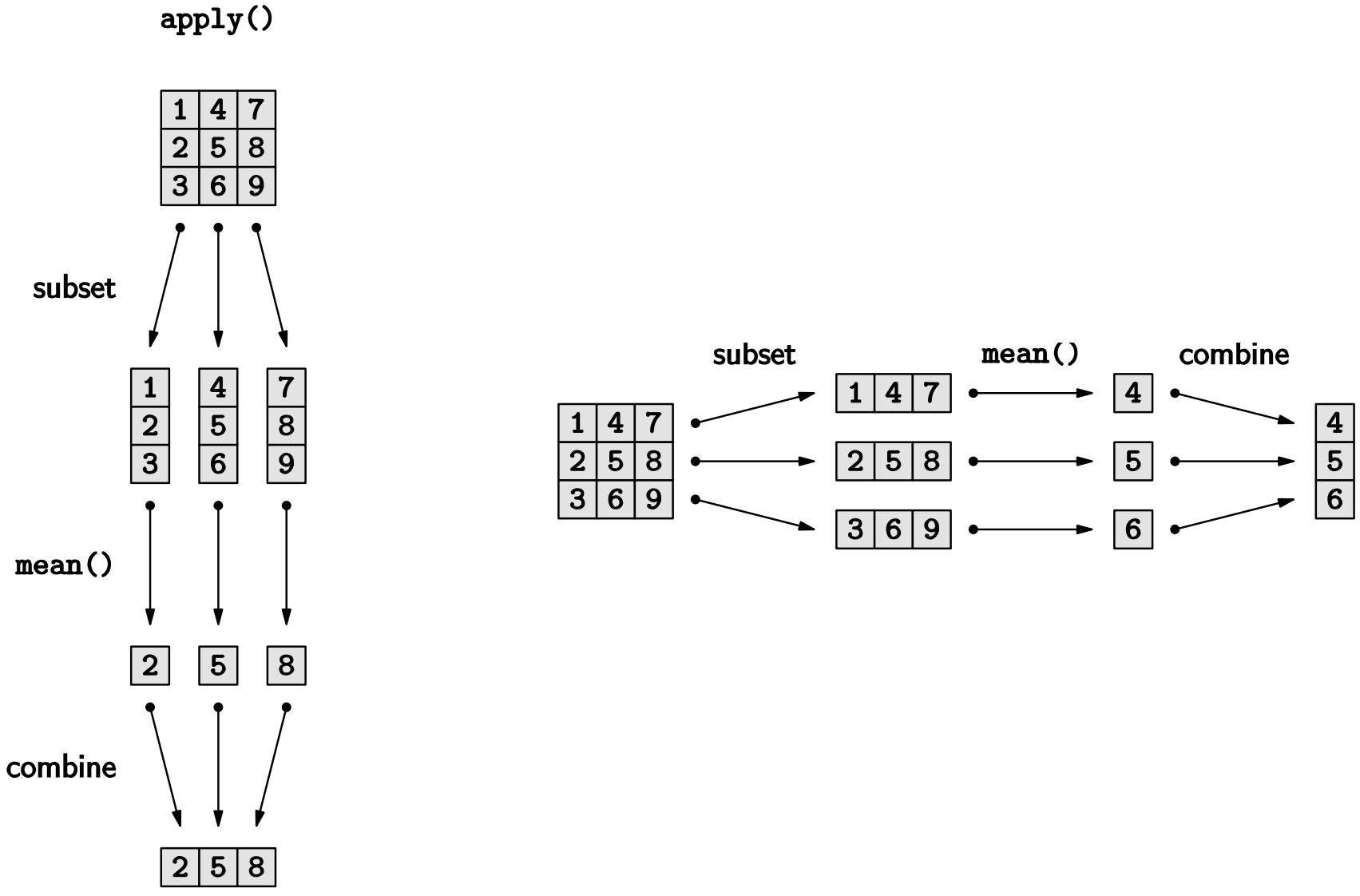


Figure 5: The apply function, operating on columns (left) or rows (rigth)

The apply function

- For example, the `apply` function can replace a `for` loop for calculating **average** annual rainfall **per station**
- The `sum` function is applied on the **rows** dimension -

```
rainfall$annual = apply(
  X = rainfall[, m],
  MARGIN = 1,
  FUN = sum
)
```

- As another example, we can calculate **average** monthly rainfall **per month**
- This time, the `mean` function is applied on the **columns** dimension -

```
avg_rain = apply(rainfall[, m], 2, mean)
```

The apply function

- The result `avg_rain` is a **named** `numeric` vector
- Element names correspond to **column** names of `X` -

```
avg_rain
##         sep         oct         nov         dec
##    1.025444   21.532544   64.852071 105.798817
##         jan         feb         mar         apr
## 123.053254 103.130178  58.366864  16.769231
##         may
##    3.968639
```

- We can quickly visualize the values with `barplot` -

```
barplot(avg_rain)
```

The apply function

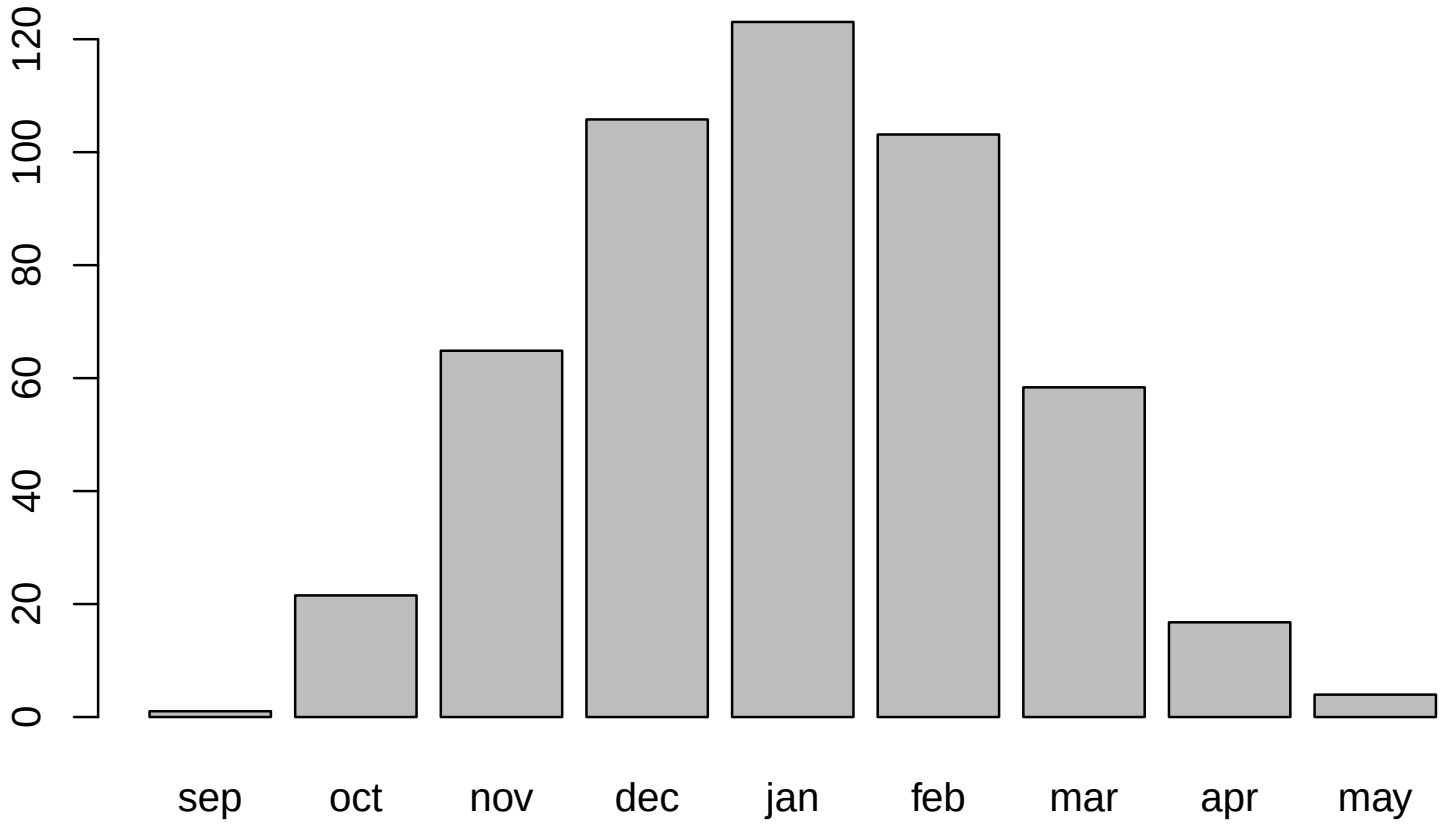


Figure 6: Average rainfall per month, among 169 stations in Israel

The apply function

- As another example, let's use apply to find the station **name** with the **highest** rainfall **per month**
- The following applies which.max on the columns, returning the row **indices** where the maximal rainfall values are located per column -

```
max_st = apply(rainfall[, m], 2, which.max)
max_st
## sep oct nov dec jan feb mar apr may
##  71  23  77  77  77  66  66  66  77
```

- We can get the corresponding station **names** by subsetting the name column using max_st -

```
rainfall$name[max_st]
## [1] "Eilon"       "Maabarot"    "Horashim"
## [4] "Horashim"    "Horashim"    "Golan Farm"
## [7] "Golan Farm" "Golan Farm" "Horashim"
```

The apply function

- It is convenient to **combine** the names and values in a table -

```
data.frame(
  month = m,
  name = rainfall$name[max_st],
  stringsAsFactors = FALSE
)
##   month         name
## 1   sep        Eilon
## 2   oct    Maabarot
## 3   nov    Horashim
## 4   dec    Horashim
## 5   jan    Horashim
## 6   feb Golan Farm
## 7   mar Golan Farm
## 8   apr Golan Farm
## 9   may    Horashim
```

Table join

- The `dates.csv` table contains the **dates** when each image in the raster `modis_south.tif`, which we will work with in **Lesson 05**, was taken -

```
dates = read.csv("dates.csv", stringsAsFactors = FALSE)
```

```
head(dates)
##   day month year
## 1  18     2 2000
## 2   5     3 2000
## 3  21     3 2000
## 4   6     4 2000
## 5  22     4 2000
## 6   8     5 2000
```

Table join

- For further analysis of the images we would like to be able to group them **by season**
- How can we calculate a new `season` **column**, specifying the season each date belongs to?

Table 2: Months and seasons

season	months
`"winter"`	12, 1, 2
`"spring"`	3, 4, 5
`"summer"`	6, 7, 8
`"fall"`	9, 10, 11

Table join

- One way is to **assign** each season name into a **subset** of a new `season` column -

```
dates$season[dates$month %in% c(12, 1:2)] = "winter"
dates$season[dates$month %in% 3:5] = "spring"
dates$season[dates$month %in% 6:8] = "summer"
dates$season[dates$month %in% 9:11] = "fall"
```

Table join

```
head(dates)
##   day month year season
## 1  18     2 2000 winter
## 2   5     3 2000 spring
## 3  21     3 2000 spring
## 4   6     4 2000 spring
## 5  22     4 2000 spring
## 6   8     5 2000 spring
```

- This method of classification may be inconvenient when we have many categories or complex criteria
- Another option is to use a **table join**

R packages

- All object definitions (including functions) in R are contained in **packages**
- To use a particular object, we need to -
  - **Install** the package with `install.packages` (once)
  - **Load** the package using the `library` function (in each new R session)
  - Loading a package basically means the code we downloaded is **executed**, loading all objects the package defined into the **RAM**
- All function calls we used until now did require this. Why? Because several packages are **installed** along with R and **loaded** on R start-up
- There are several more packages which are installed by default but **not loaded** on start-up (total of ~30)

R packages

Table 4-1. Packages included with R

Package name	Loaded by default	Description
base	✓	Basic functions of the R language, including arithmetic, I/O, programming support
boot		Bootstrap resampling
class		Classification algorithms, including nearest neighbors, self-organizing maps, and learning vector quantization
cluster		Clustering algorithms
codetools		Tools for analyzing R code
compiler		Byte code compiler for R
datasets	✓	Some famous data sets
foreign		Tools for reading data from other formats, including Stata, SAS, and SPSS files
graphics	✓	Functions for base graphics
grDevices	✓	Device support for base and grid graphics, including system-specific functions
grid		Tools for building more sophisticated graphics than the base graphics
KernSmooth		Functions for kernel smoothing
lattice		An implementation of Trellis graphics for R: prettier graphics than the default graphics
MASS		Functions and data used in the book *Modern Applied Statistics with S* by Venables and Ripley; contains a lot of useful statistics functions
methods	✓	Implementation of formal methods and classes introduced in S version 4 (called S4 methods and classes)
mgcv		Functions for generalized additive modeling and generalized additive mixed modeling
nlme		Linear and nonlinear mixed-effects models
nnet		Feed-forward neural networks and multinomial log linear models
parallel		Support for parallel computation, including random-number generation.
rpart		Tools for building recursive partitioning and regression tree models
spatial		Functions for Kriging and point pattern analysis
splines		Regression spline functions and classes
stats	✓	Functions for statistics calculations and random number generation; includes many common statistical tests, probability distributions, and modeling tools
stats4		Statistics functions as S4 methods and classes
survival		Survival analysis functions
tcltk		Interface to Tcl/Tk; used to create platform-independent UI tools
tools		Tools for developing packages
utils	✓	A variety of utility functions for R, including package management, file reading and writing, and editing

Figure 7: Packages included with R[1]

[1] R in a Nutshell, 2010

R packages

Table 4-1. Packages included with R

Package name	Loaded by default	Description
base	✓	Basic functions of the R language, including arithmetic, I/O, programming support
boot		Bootstrap resampling
class		Classification algorithms, including nearest neighbors, self-organizing maps, and learning vector quantization
cluster		Clustering algorithms
codetools		Tools for analyzing R code
compiler		Byte code compiler for R
datasets	✓	Some famous data sets
foreign		Tools for reading data from other formats, including Stata, SAS, and SPSS files
graphics	✓	Functions for base graphics
grDevices	✓	Device support for base and grid graphics, including system-specific functions
grid		Tools for building more sophisticated graphics than the base graphics
KernSmooth		Functions for kernel smoothing

R packages

- **Most** of the ~13,000 R packages (Sep 2018) are *not* installed by default
- To use one of these packages we first need to **install** it on the computer
- This is a **one-time** operation using the `install.packages` function
- After the package is installed, each time we want to use it we need to **load** it using `library`

R packages

- In the next example we use a package called `dplyr`
- This package is not installed with R, we need to **install** it ourselves -

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

- Note: if the package is already installed then `install.packages` **overwrites** the old installation
- Once the package is already installed, we need to use the `library` function to **load** it into memory -

```
library(dplyr)
```

- Note: package name can be passed to `library` **without** quotes

Table join

- The `left_join` function from `dplyr` does a **left join** between tables
- The first two parameters are the **tables** that need to be joined, `x` and `y`
- The third `by` parameter is the common **column name(s)** by which the tables need to be joined

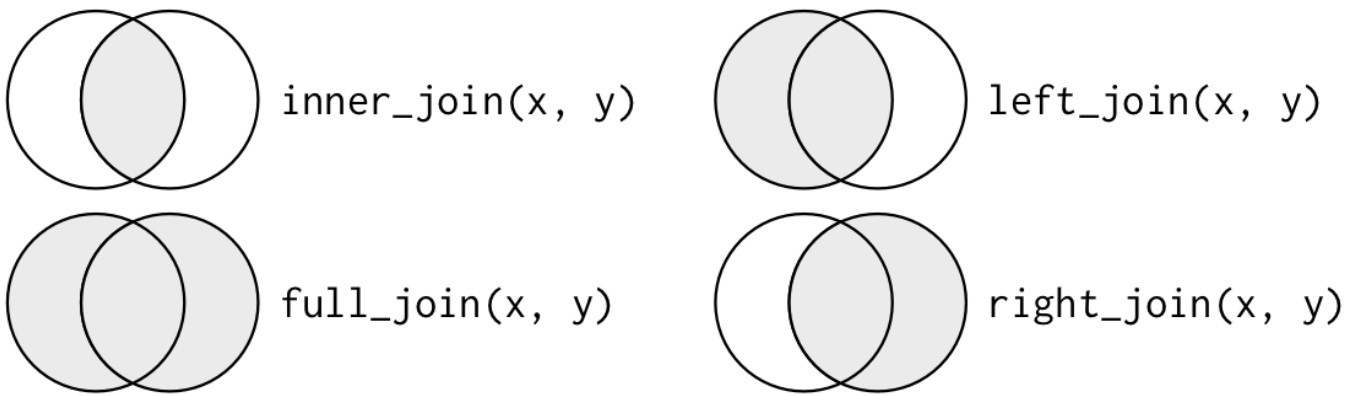


Figure 8: Join types[2]

[2] http://r4ds.had.co.nz/relational-data.html

Table join

- Next we prepare a table `tab` with the **classification** of months into seasons -

```
tab = data.frame(
  month = c(12, 1:11),
  season = rep(
    c("winter","spring","summer","fall"),
    each = 3
  ),
  stringsAsFactors = FALSE
)
```

Table join

- Next we prepare a table `tab` with the classification of months into seasons -

```
tab
```

```
##    month season
## 1     12 winter
## 2      1 winter
## 3      2 winter
## 4      3 spring
## 5      4 spring
## 6      5 spring
## 7      6 summer
## 8      7 summer
## 9      8 summer
## 10     9   fall
## 11    10   fall
## 12    11   fall
```

Table join

- Now we can **join** the `dates` and `tab` tables
- Before that, we **remove** the `season` column we manually created in the previous example -

```
dates$season = NULL
dates = left_join(dates, tab, by = "month")
```

```
head(dates)
##   day month year season
## 1  18     2 2000 winter
## 2   5     3 2000 spring
## 3  21     3 2000 spring
## 4   6     4 2000 spring
## 5  22     4 2000 spring
## 6   8     5 2000 spring
```

Dates

- To get a **date** column we can paste the `year`, `month` and `day` columns and apply `as.Date` -

```
dates$date = as.Date(
  paste(dates$year, dates$month, dates$day, sep = "-")
)
```

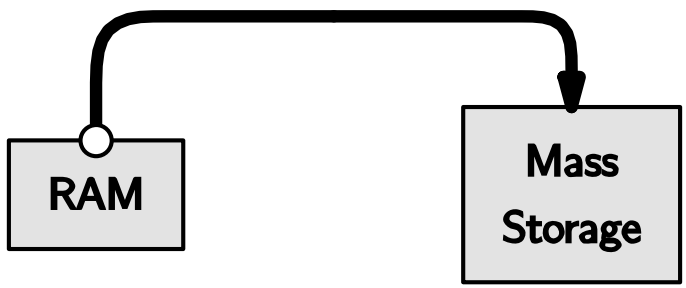
```
head(dates)
##   day month year season       date
## 1  18     2 2000 winter 2000-02-18
## 2   5     3 2000 spring 2000-03-05
## 3  21     3 2000 spring 2000-03-21
## 4   6     4 2000 spring 2000-04-06
## 5  22     4 2000 spring 2000-04-22
## 6   8     5 2000 spring 2000-05-08
```

Writing table to file

- Using `write.csv` we can **write** the contents of a `data.frame` to a **CSV file** -

```
write.csv(dates, "dates2.csv", row.names = FALSE)
```

- The `row.names` parameter determines whether the **row names** are saved
- Note: like in `read.csv`, we can either give a **full** file path or just the **file name**; if the specify just the file name then the file is written to the **working directory**



Exercise 2 - Submission date **2018-12-09**

Introduction to Spatial Data Programming

Execrise 2

Time series and function definition

& Tables, conditionals and loops

Last updated: 2018-10-03 08:46:52

Question 1

- Run the following four expressions to load two vectors named `year` and `co2` into memory

```
data(co2)
means = aggregate(co2, FUN = mean)
year = as.vector(time(means))
co2 = as.vector(means)
```

- The `co2` vector contains CO_2 measurements in the atmosphere, in *ppm* units, during the period 1959-1997. The `year` vector contains the corresponding years
- Assuming that the rate of CO_2 increase was constant and equal to the average rate during 1959-1997, calculate the predicted CO_2 concentration during each of the years 1998-2017
- Create a **plot** showing CO_2 concentration as function of time, with -
  - Observed values during 1959-1997, based on the `year` and `co2` vectors, in **black**
  - Predicted values during 1998-2017 in **blue**
- **Add** a point in **red** showing the true concentration in 2017, which was `406.53`