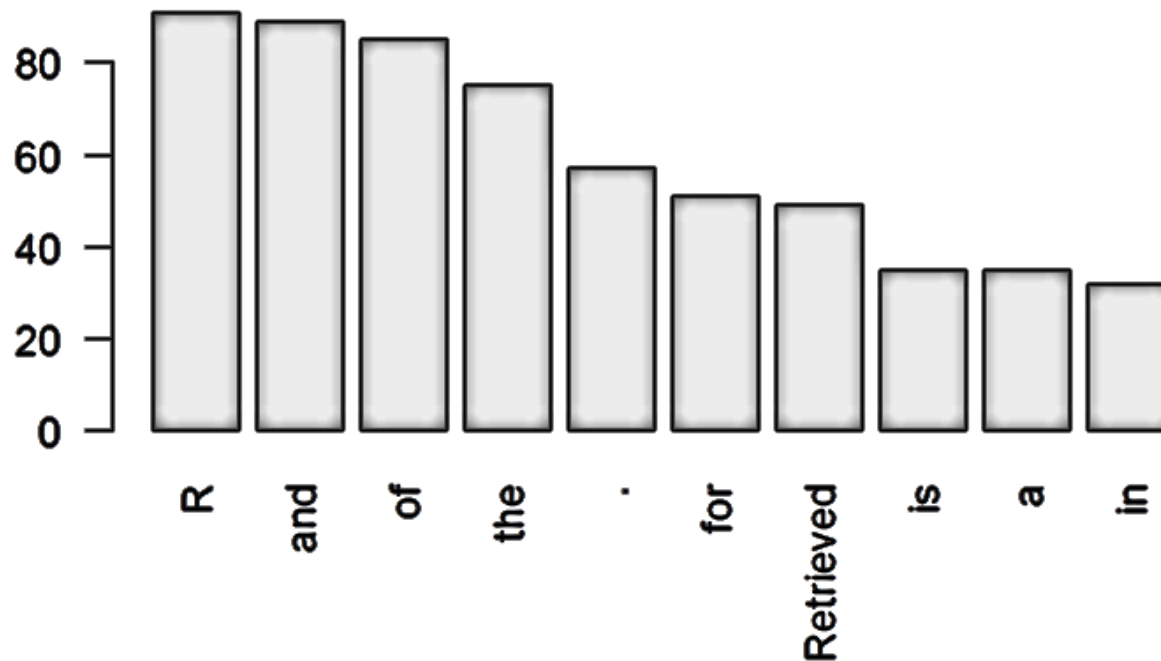


Introduction to Computing for Economics and Management

Lecture 6: Data Import & Loops



Acknowledgement

- These slides are adapted from Bert Arnrich's R lecture.



Previous lectures

- Vectors
- Matrices
- Lists
- Data Frames

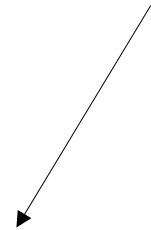
Data import

So far, we have entered our data into R

```
> person.height <- c(Can=1.70,  
Cem=1.75, Hande=1.62)
```

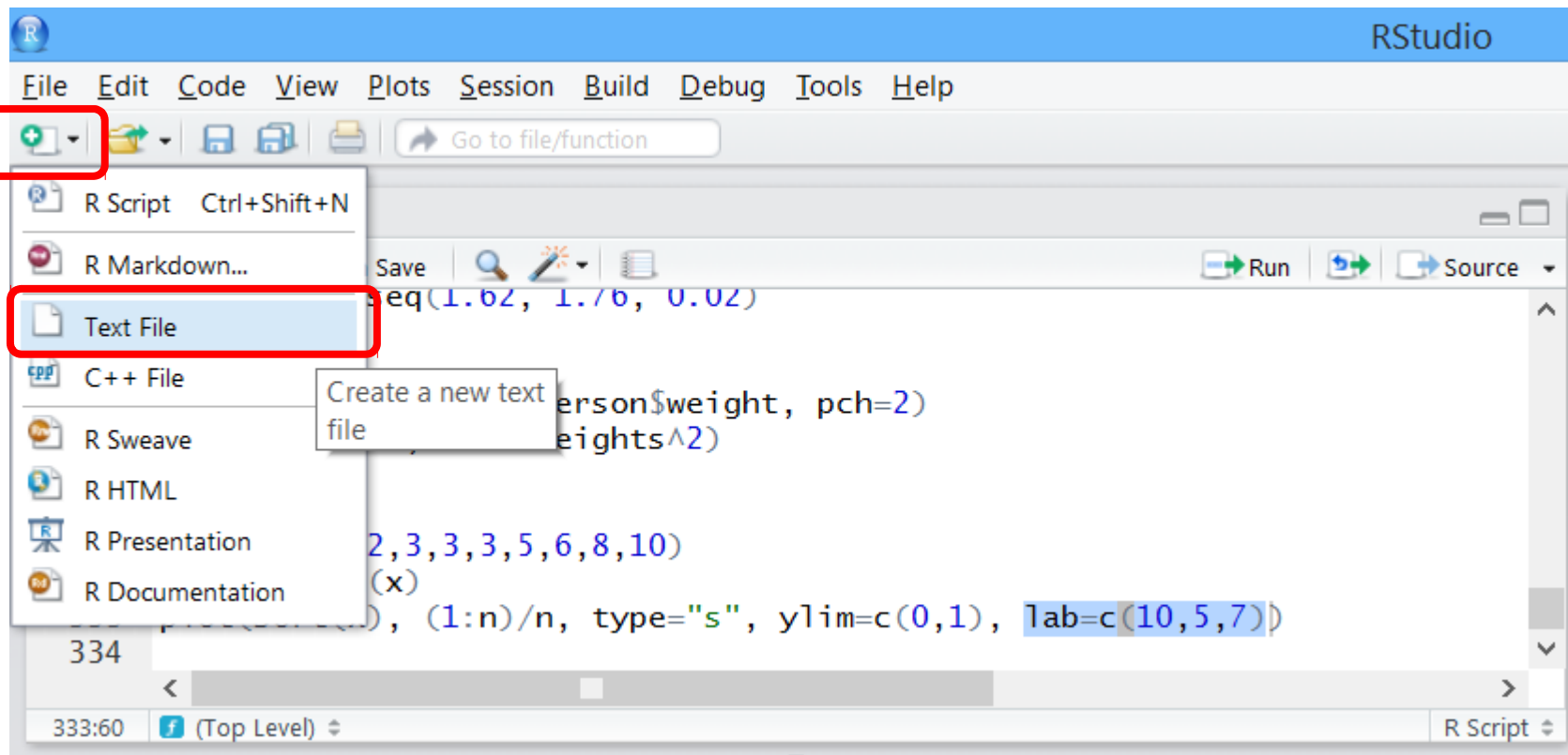
- In practice, data is usually stored in data bases or files and we import it from there
- In the following slides, we will prepare a file which contains our data and import the file content into R

textfile



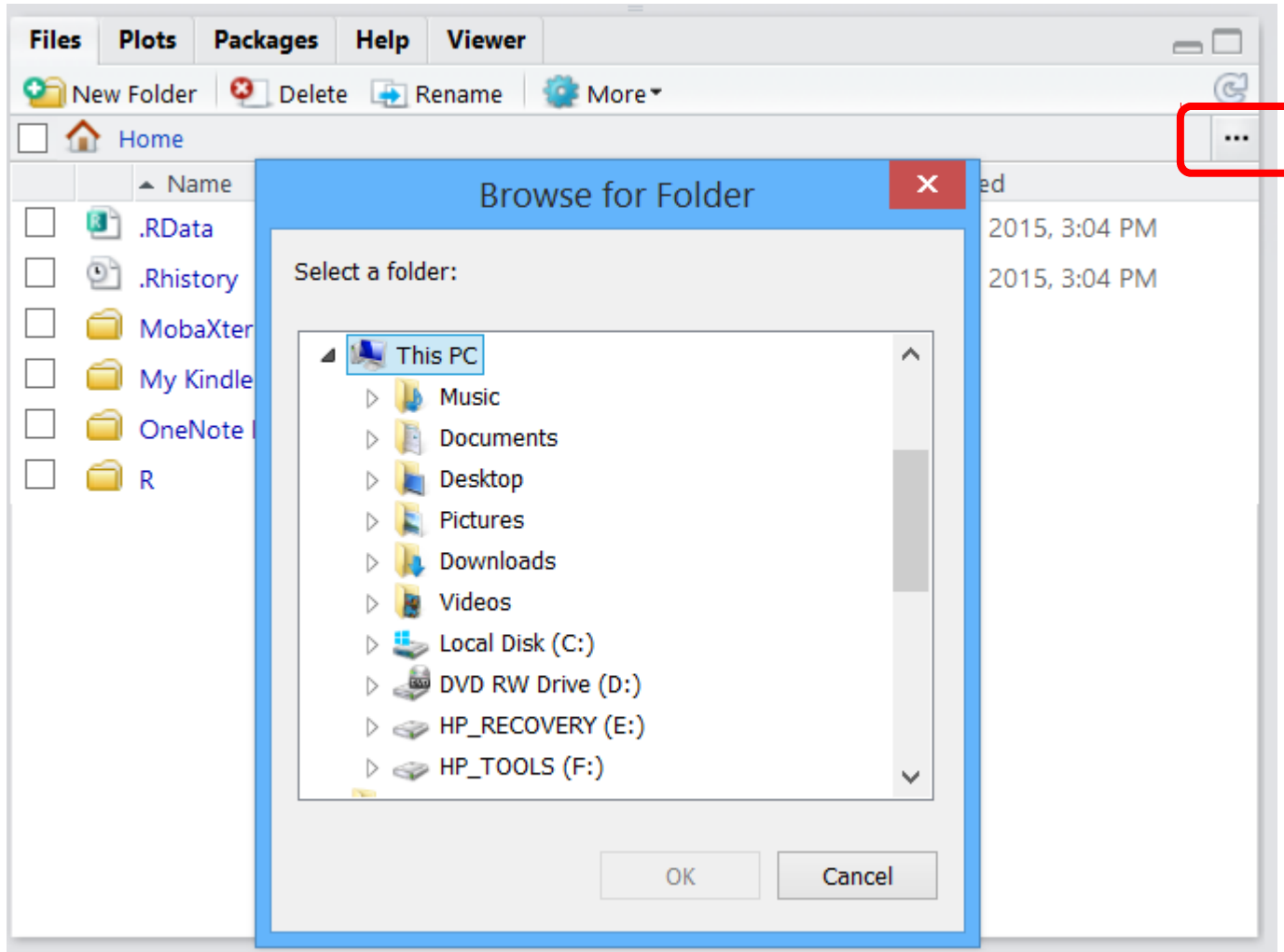
Name	Height	Weight
Can	1.7	65
Cem	1.75	66
Hande	1.62	61
Lale	1.76	64
Arda	1.78	63
Bilgin	1.77	84
Cem	1.69	75
Ozlem	1.75	65
Ali	1.73	75
Haluk	1.71	81

Data file



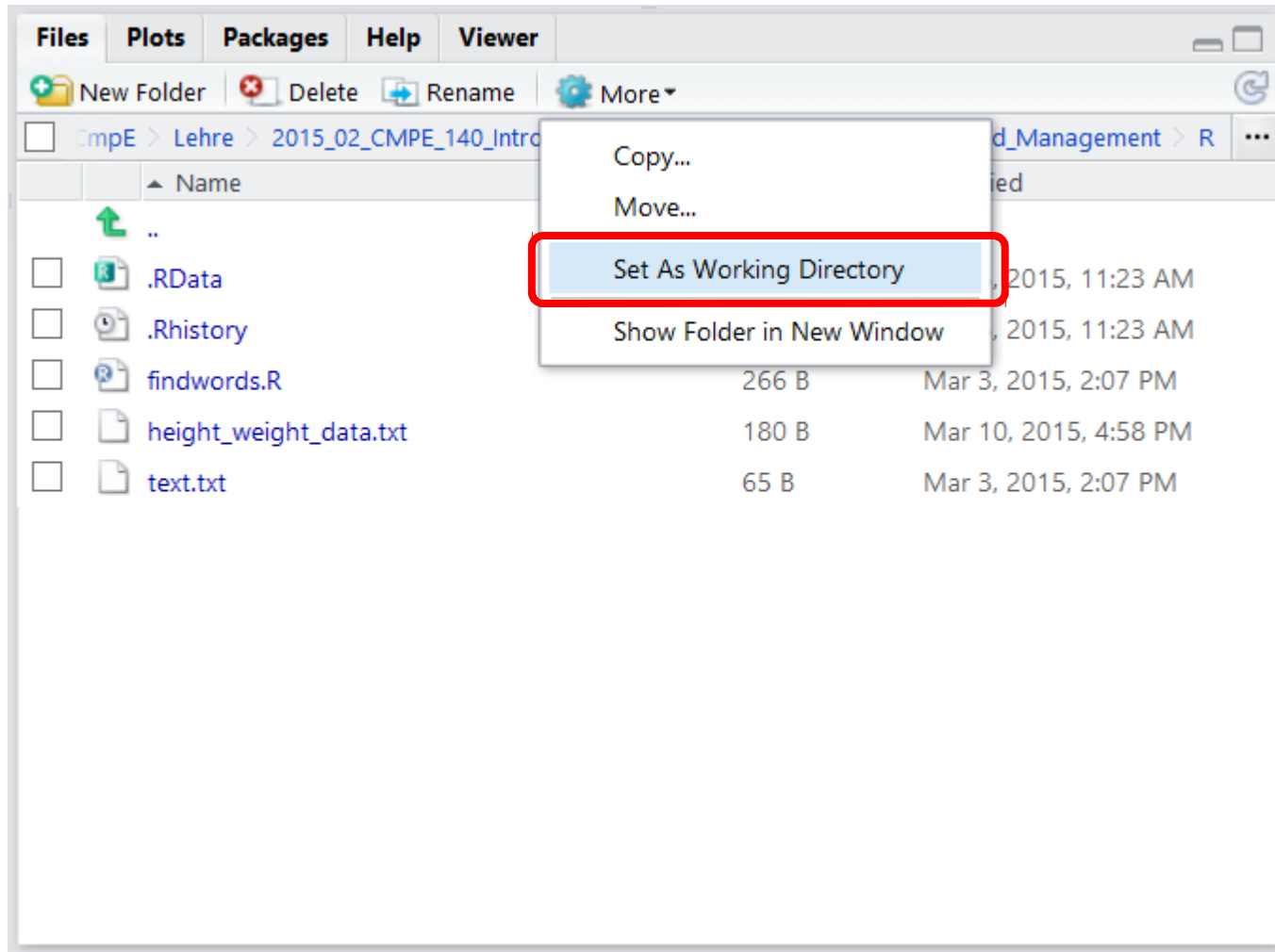
- As a first step we create a new text file using RStudio or an alternative text editor like Notepad++

Data file



- In the files tab we select the “...” item and browse to the folder in which we have stored the text file

Data file



- In the menu “More”, we select “Set As Working Directory”

Data import with `read.table`

The function `read.table` is the most convenient way to read in a rectangular grid of data from a text file

```
> help(read.table)
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE,
           fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           fileEncoding = "", encoding = "unknown", text)
```

Arguments of `read.table`

`file`

- Name of the file which the data are to be read from
- Each row of the table appears as one line of the file
- If it does not contain an absolute path, the file name is relative to the current working directory
- Can also be a complete URL

Arguments of read.table

header

- Logical value indicating whether the file contains the names of the variables as its first line
- If header information is available in the file, it will be used for variable names

Name, Height, Weight
Can, 1.7, 65
Cem, 1.75, 66
Hande, 1.62, 61
Lale, 1.76, 64
Arda, 1.78, 63
Bilgin, 1.77, 84
Cem, 1.69, 75
Ozlem, 1.75, 65
Ali, 1.73, 75
Haluk, 1.71, 81

Arguments of `read.table`

`sep`

- Field separator character
- Values on each line of the file are separated by this character
- Default value `sep = ""` means that the separator is 'white space': one or more spaces, tabs, newlines or carriage returns

Name, Height, Weight

Can, 1.7, 65

Cem, 1.75, 66

Hande, 1.62, 61

Lale, 1.76, 64

Arda, 1.78, 63

Bilgin, 1.77, 84

Cem, 1.69, 75

Ozlem, 1.75, 65

Ali, 1.73, 75

Haluk, 1.71, 81

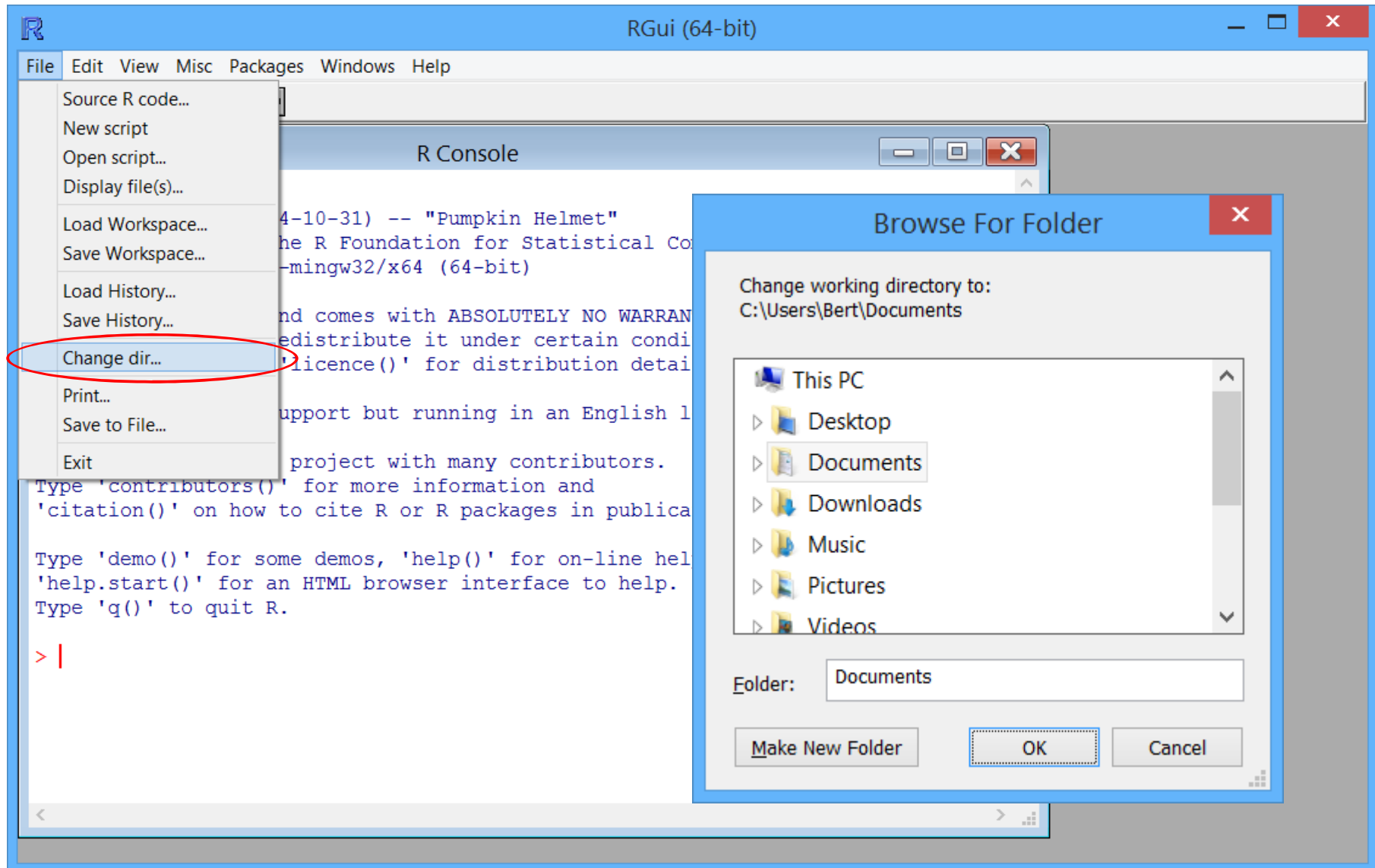
Working directory

Before the actual import, we need to check the current working directory to make sure which path to use when importing the data file

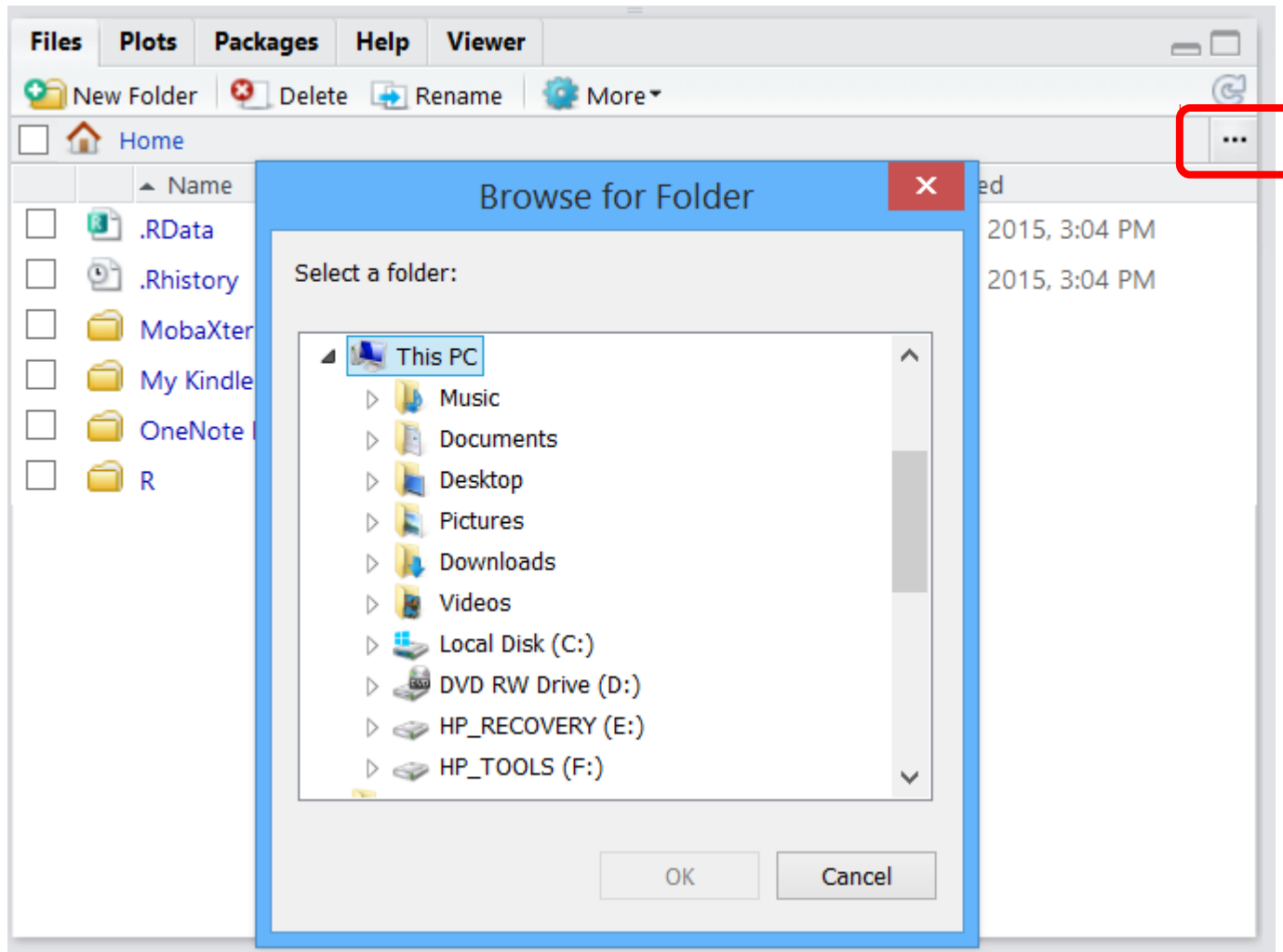
```
> getwd()  
[1] "/home/c7031082/R"
```

- We change working directory to the path where our data file is located in order to simplify data import

Change working directory in R

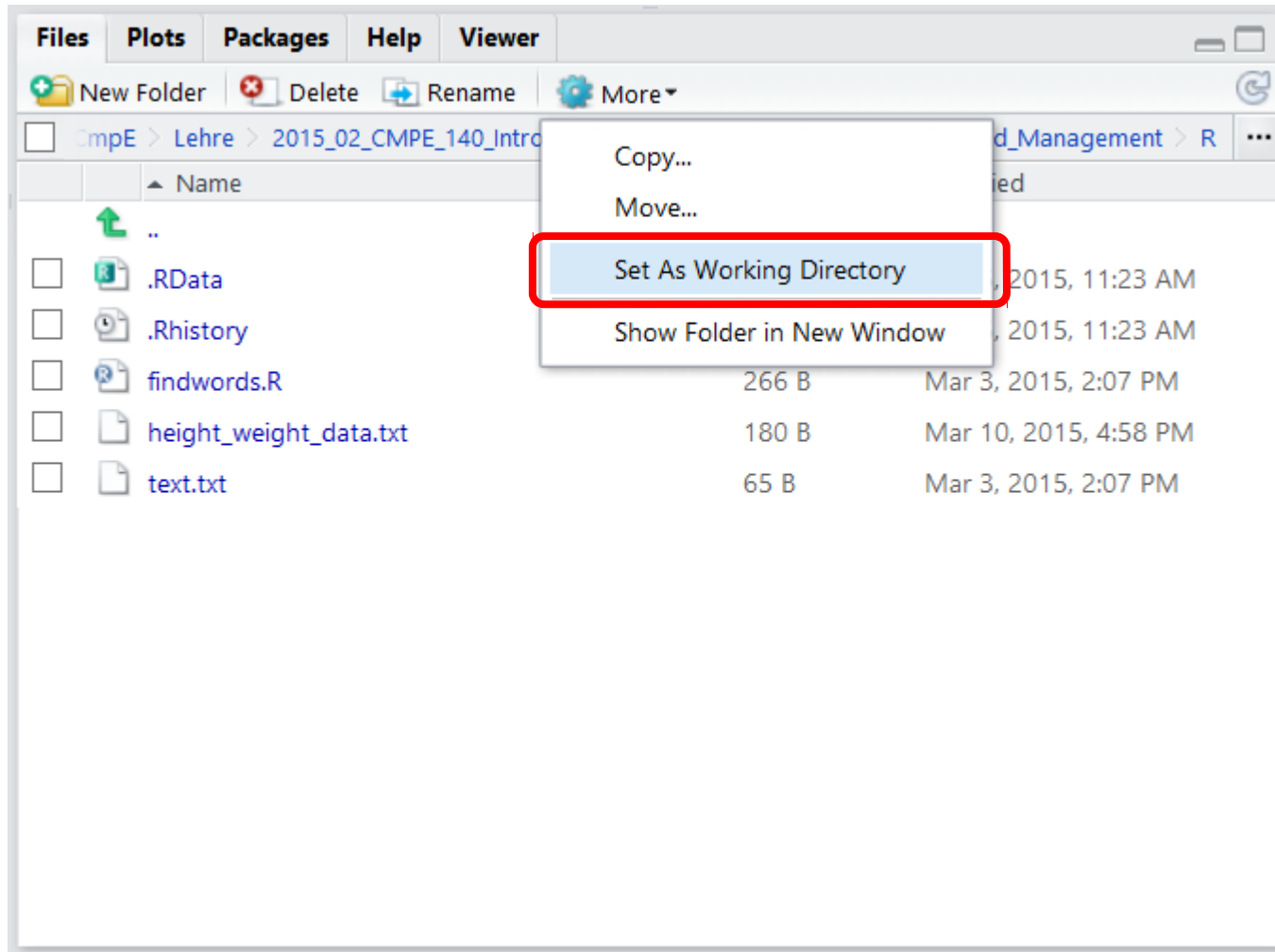


Change working directory in RStudio



- In the files tab we select the “...” item and browse to the folder in which we have stored the text file

Change working directory in RStudio



- In the menu “More”, we select “Set As Working Directory”

Data import

Now, we import the data into the data frame `person.data` by using the function `read.table`

```
> person.data <- read.table(header=TRUE,  
"height_weight_data.txt", sep=",")
```

What is the mode/type of `person.data`?

```
> person.data
```

	Name	Height	Weight
1	Can	1.70	65
2	Cem	1.75	66
3	Hande	1.62	61
4	Lale	1.76	64
5	Arda	1.78	63
6	Bilgin	1.77	84
7	Cem	1.69	75
8	Ozlem	1.75	65
9	Ali	1.73	75
10	Haluk	1.71	81

Data modifications

We add a new column BMI like we did before

```
> person.data$BMI <- person.data$Weight /  
person.data$Height^2
```

```
> person.data
```

	Name	Height	Weight	BMI
1	Can	1.70	65	22.49135
2	Cem	1.75	66	21.55102
3	Hande	1.62	61	23.24341
4	Lale	1.76	64	20.66116
5	Arda	1.78	63	19.88385
6	Bilgin	1.77	84	26.81222
7	Cem	1.69	75	26.25958
8	Ozlem	1.75	65	21.22449
9	Ali	1.73	75	25.05931
10	Haluk	1.71	81	27.70083

Data modifications

We can change the values of a column by reassigning the column with the new values, e.g. rounding BMI

```
> person.data$BMI <- round(person.data$BMI, 2)
```

```
> person.data
```

	Name	Height	Weight	BMI
1	Can	1.70	65	22.49
2	Cem	1.75	66	21.55
3	Hande	1.62	61	23.24
4	Lale	1.76	64	20.66
5	Arda	1.78	63	19.88
6	Bilgin	1.77	84	26.81
7	Cem	1.69	75	26.26
8	Ozlem	1.75	65	21.22
9	Ali	1.73	75	25.06
10	Haluk	1.71	81	27.70

Data modifications

- When creating new columns, we can make use of functions to compute the values of a new column
- Let's recapitulate the `ifelse()` function

`ifelse(test, yes, no)` returns a vector which is created from selected elements from the vectors `yes` and `no`: `yes[i]` is selected if `test[i]` is true and `no[i]` is selected if `test[i]` is false

```
> ifelse(person.height > 1.7, "tall", "small")
      Can      Cem      Hande
"small"  "tall"  "small"
```

Data frame modifications

Let's use `ifelse()` to create a new column which indicates whether BMI is above 22.5

```
> person.data$above22.5 <- ifelse(person.data$BMI>22.5, T  
, F)
```

```
> person.data
```

	Name	Height	Weight	BMI	above22.5
1	Can	1.70	65	22.49	FALSE
2	Cem	1.75	66	21.55	FALSE
3	Hande	1.62	61	23.24	TRUE
4	Lale	1.76	64	20.66	FALSE
5	Arda	1.78	63	19.88	FALSE
6	Bilgin	1.77	84	26.81	TRUE
7	Cem	1.69	75	26.26	TRUE
8	Ozlem	1.75	65	21.22	FALSE
9	Ali	1.73	75	25.06	TRUE
10	Haluk	1.71	81	27.70	TRUE

Data frame summary

`summary()` provides a summary statistic of a data frame

```
> summary(person.data)
```

Name		Height		Weight	
Cem	:2	Min.	:1.620	Min.	:61.00
Ali	:1	1st Qu.:	1.702	1st Qu.:	64.25
Arda	:1	Median	:1.740	Median	:65.50
Bilgin	:1	Mean	:1.726	Mean	:69.90
Can	:1	3rd Qu.:	1.758	3rd Qu.:	75.00
Haluk	:1	Max.	:1.780	Max.	:84.00
(Other) :3					
BMI		above22.5			
Min.	:19.88	Mode	:logical		
1st Qu.:	21.30	FALSE	:5		
Median	:22.86	TRUE	:5		
Mean	:23.49	NA's	:0		
3rd Qu.:	25.96				
Max.	:27.70				

Data frame summary

From the output of `summary()` we observe that the format of the summary statistic depends on the column mode

- For numeric modes like height we retrieve minimum, 1st quartile (25% quantile), median, mean, 3rd quartile (75% quantile), maximum
- For Boolean mode we retrieve number of True, False and missing values denoted with NA
- For character mode we retrieve the number of entries for each string

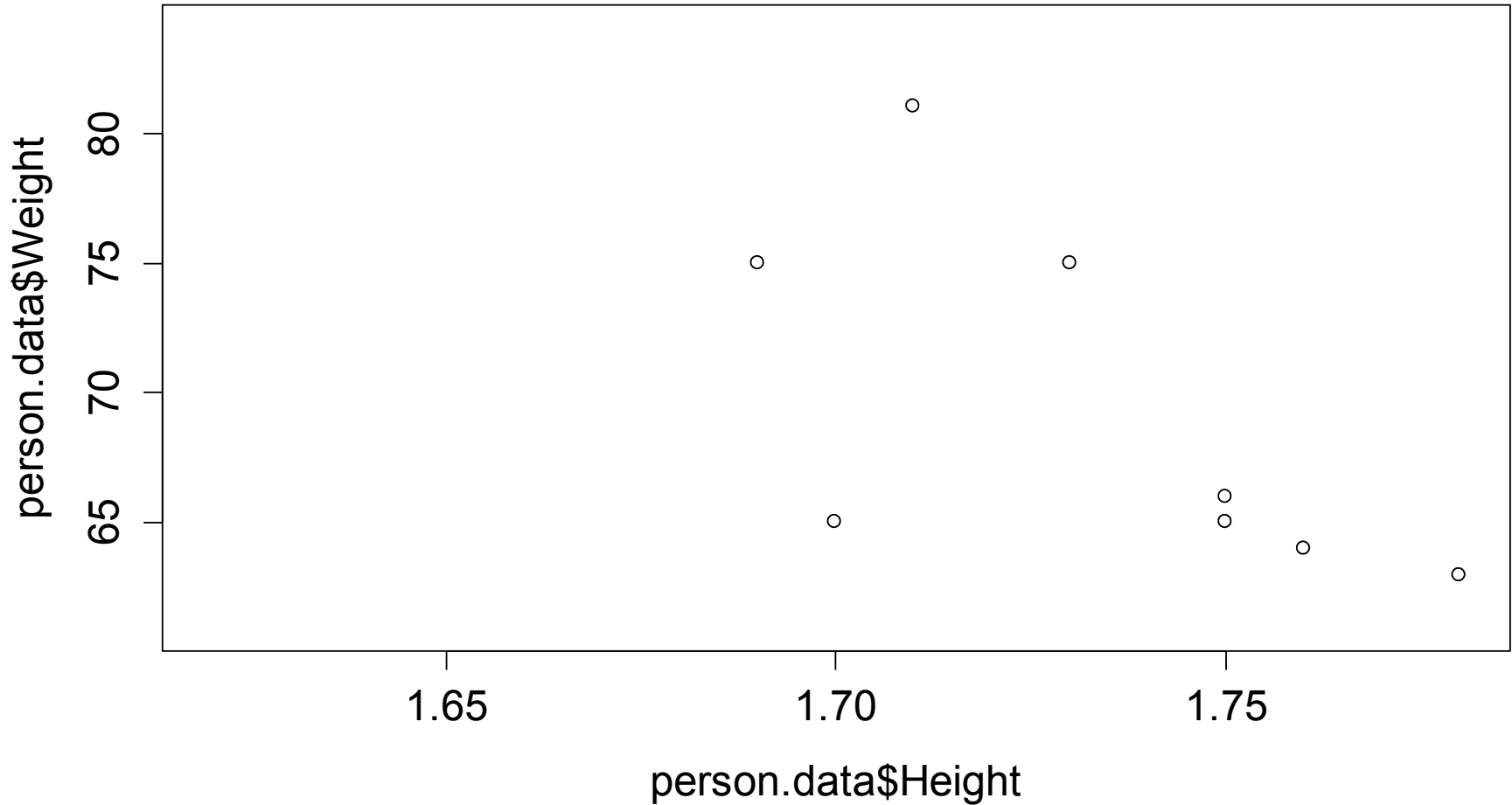
Scatter Plot

- Beside numeric summary statistics, a convenient way for data exploration is plotting
- R provides us many powerful tools for plotting
- We will learn more about plotting later

For now, we create a simple scatter plot by plotting height on the x-axis and weight on the y-axis

```
> plot(person.data$Height, person.data$Weight)
```

Scatter Plot



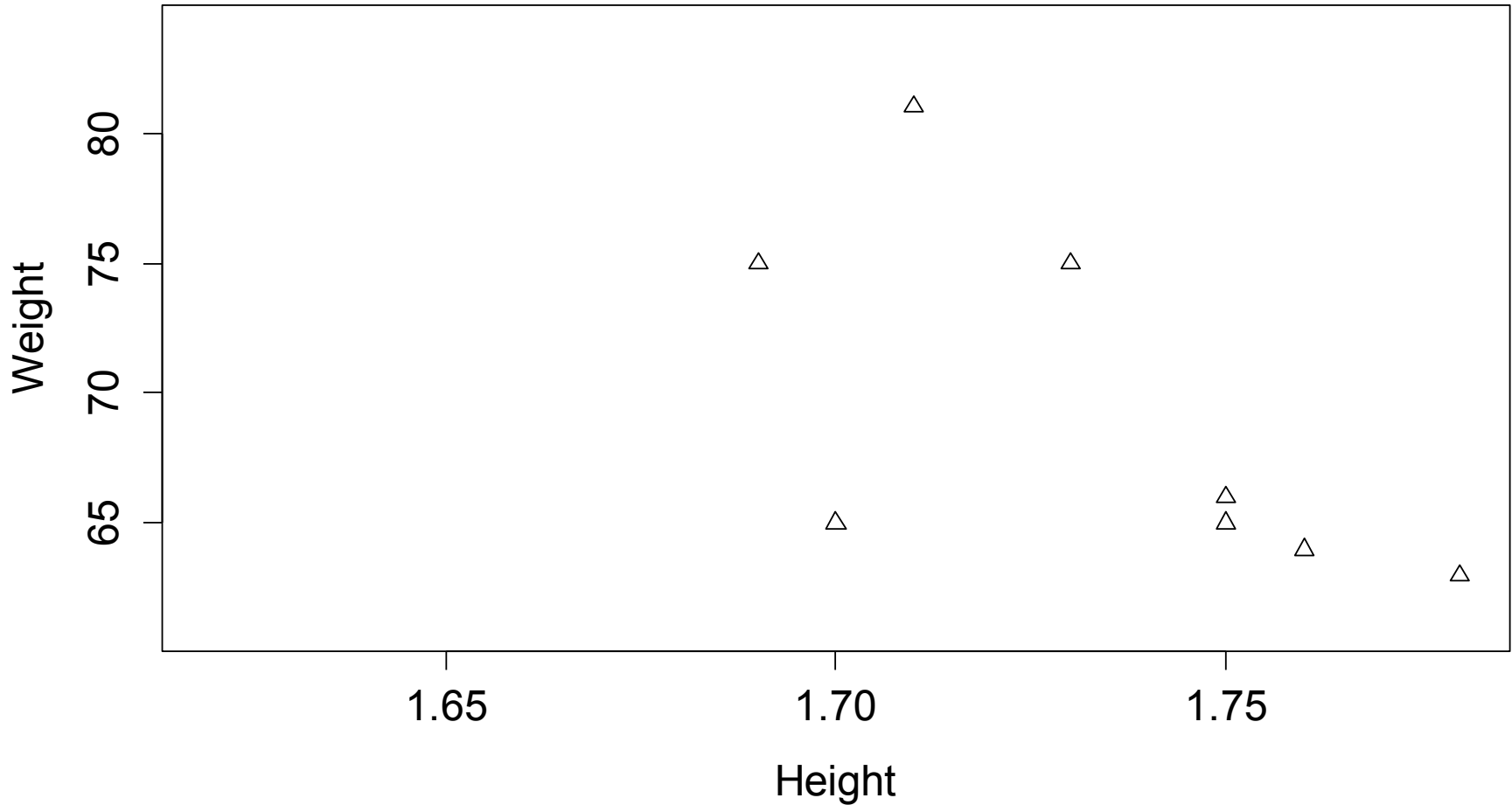
Scatter Plot

- From the previous plot we observe that the values of Height are plotted versus the corresponding values of Weight
- Similar to other functions, we can provide additional parameters to the plot function

With `pch=2` we change the point type (from circles to triangles) and beside that we provide axis labels

```
> plot(person.data$Height, person.data$Weight,  
pch=2, xlab="Height", ylab="Weight")
```

Scatter Plot



Scatter Plot

- Previously we used `ifelse()` to create a column which indicates whether BMI is above 22.5 or not
- In a graphical point of view, we can draw a line which represents whether BMI is above 22.5 or not
- Since $\text{Weight} = \text{BMI} \times \text{Height}^2$ we can draw a line $22.5 \times \text{Height}^2$
- For drawing the line we need two height values which we take from the summary statistic

Scatter Plot

From the summery statistic of Height we learn the minimum and maximum numbers

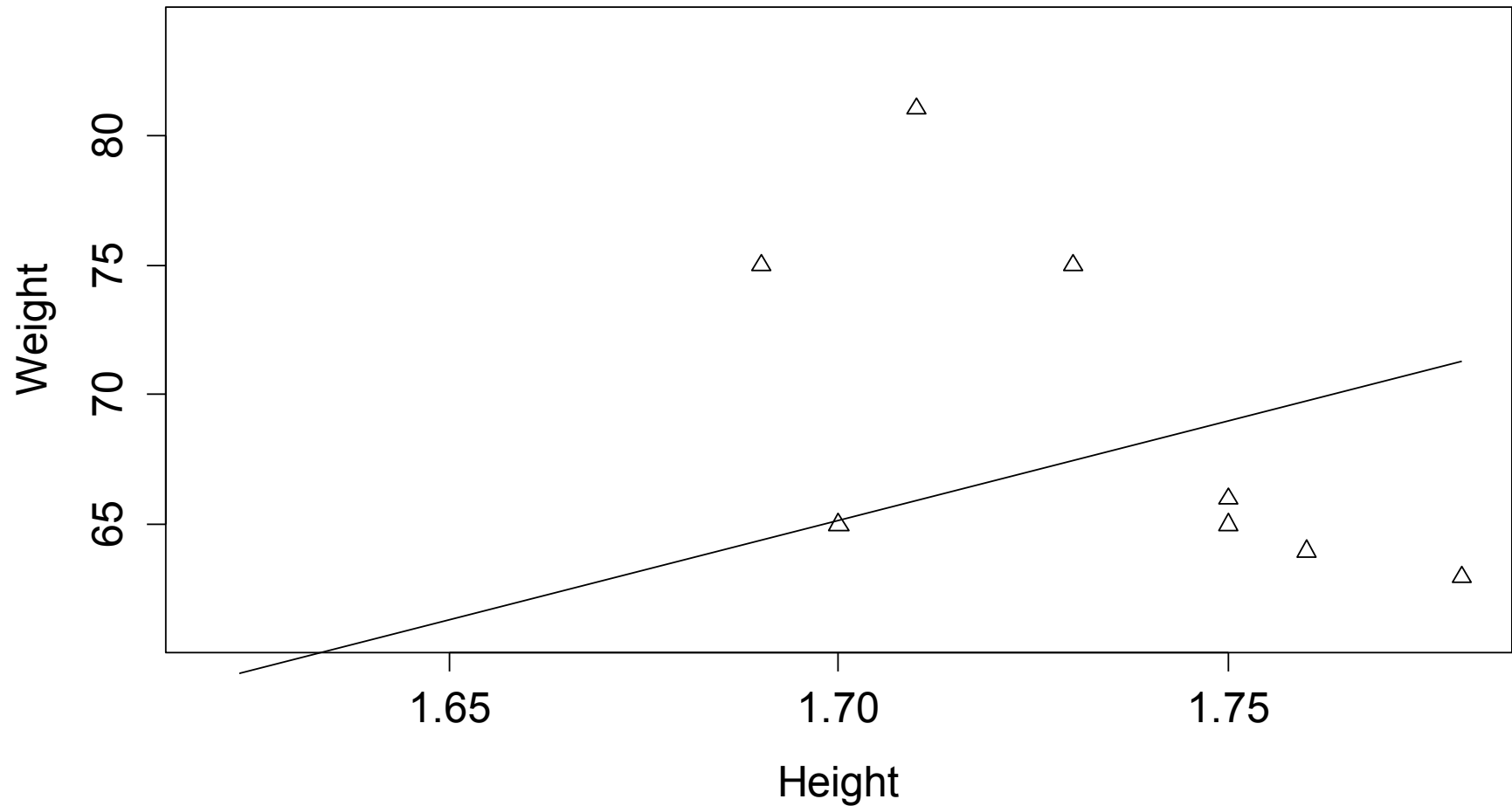
```
> summary(person.data$Height)
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
■	1.620	1.702	1.740	1.726	1.758	1.780

We draw the line from the minimum height (1.62) to the maximum height (1.78) using $22.5 \times \text{Height}^2$ for the y values of the line

```
> lines(c(1.62, 1.78), 22.5*c(1.62, 1.78)^2)
```

Scatter Plot



- Data points above the line represents persons having a BMI above 22.5

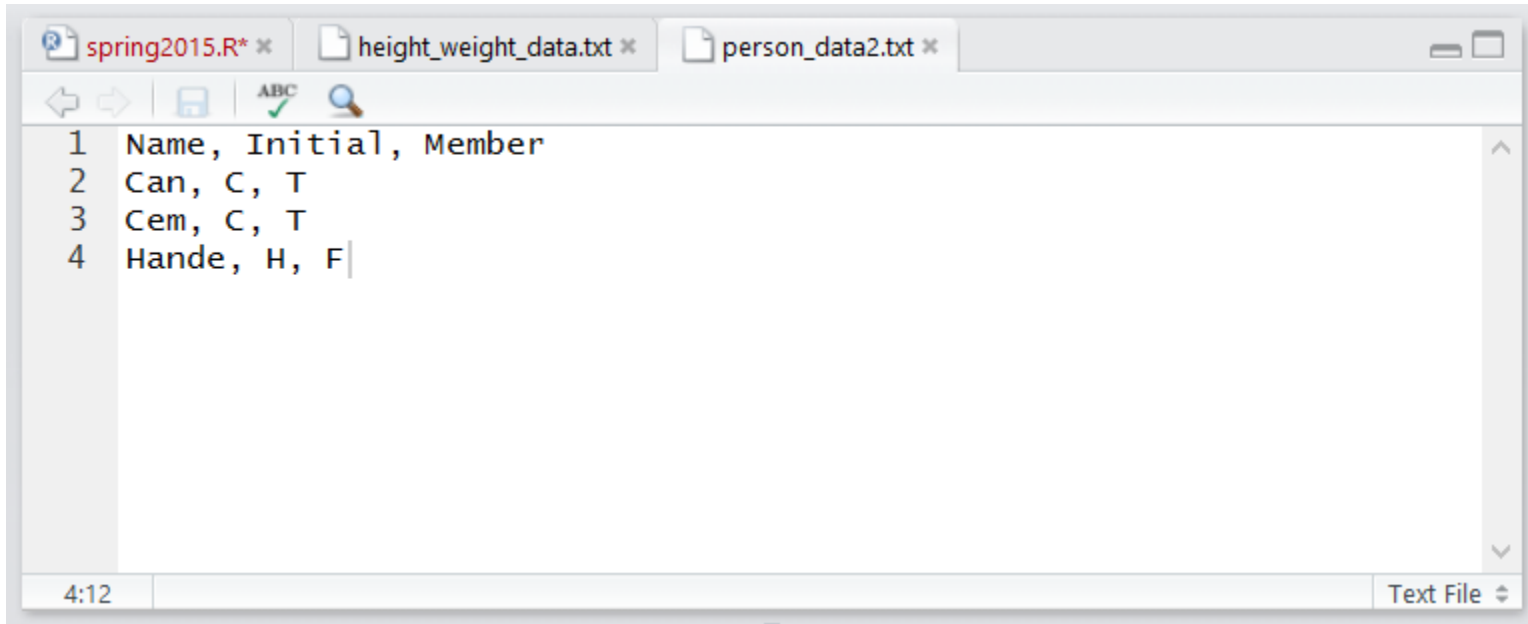
Some analysis with midterm results

```
scores <- read.table("scores.txt",header=TRUE)
plot(c(1:97),scores$Score)
plot(c(1:97),sort(scores$Score))
hist(scores$Score)
scores.cut <- cut(scores$Score,breaks=5)
scores.table <- table(scores.cut)
pie (scores.table)
```

Merging data frames

- Data from an entity (e.g. a person) is often stored in multiple data sets
- One frequent operation is to join data sets: combine data sets according to the values of a common variable
- Example: two data sets contain different kind of data about some persons can be joined according to person's name or person's ID number
- In R, two data frames can be joined using the `merge()` function

Merging data frames



The screenshot shows an RStudio editor window with three tabs: 'spring2015.R*', 'height_weight_data.txt', and 'person_data2.txt'. The active tab is 'person_data2.txt', which contains a text file with the following content:

```
1 Name, Initial, Member
2 Can, C, T
3 Cem, C, T
4 Hande, H, F
```

The status bar at the bottom indicates the cursor is at line 4, column 12, and the file is a 'Text File'.

- In a first step, we create an additional data set which contains the initial and member information of some persons

Merging data frames

We import the second data set using `read.table`

```
> person.data2 <- read.table(header=TRUE,  
"person_data2.txt", sep=",")
```

```
> person.data2
```

	Name	Initial	Member
1	Can	C	T
2	Cem	C	T
3	Hande	H	F

Merging data frames

We merge the two data frames using the `merge()` function

```
> merge(person.data, person.data2)
```

	Name	Height	Weight	BMI	above22.5	Initial	Member
1	Can	1.70	65	22.49	FALSE	C	T
2	Cem	1.69	75	26.26	TRUE	C	T
3	Cem	1.75	66	21.55	FALSE	C	T
4	Hande	1.62	61	23.24	TRUE	H	F

- We know that both data frames have the variable `Name` in common and we observe that the rows with identical name values were joined
- Problem: the name `Cem` appears twice in `person.data` and thus it is joined twice with `person.data2`

Merging data frames

We better use unique IDs for each person

```
> person.data <- cbind(ID=c(1:10), person.data)
```

```
> person.data2 <- cbind(ID=c(1:3), person.data2)
```

■ We specify with by which column is used for merging

```
> merge(person.data, person.data2, by="ID")
```

	ID	Name.x	Height	Weight	BMI	above22.5	Name.y	Initial
Member								
1	1	Can	1.70	65	22.49	FALSE	Can	C
T								
2	2	Cem	1.75	66	21.55	FALSE	Cem	C
T								
3	3	Hande	1.62	61	23.24	TRUE	Hande	H
F								

Homework

1. Create a text file which consist of body height and weight from 10 of your friends
2. Import the text file into R
3. Add a new BMI column
4. Create a scatter plot of your data
5. Create a second text file which consists of age and home city of your friends
6. Import the second file and merge it with the first one

Vectors

Integer mode

```
> person.weight <- c(65, 66, 61)
```

▪ Numeric (floating-point number)

```
> person.height <- c(1.70, 1.75, 1.62)
```

▪ Character (string)

```
> person.name <- c("Can", "Cem", "Hande")
```

Logical (Boolean)

```
> person.female <- c(FALSE, FALSE, TRUE)
```

Complex

```
> complex.numbers <- c(1+2i, -1+0i)
```

Vectors

Filtering

```
> person.height[person.height > 1.65]  
  Can   Cem  
1.72 1.75
```

Recycling

```
> c(1, 2, 3) + c(1, 2, 3, 4)  
[1] 2 4 6 5
```

Vector operations

```
> person.weight / person.height^2  
      Can      Cem      Hande  
21.97134 21.55102 23.24341
```

Matrices

Creation

```
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
```

```
> cbind(c(1,2), c(3,4))
```

■ Matrix operations

- Transposition $t(y)$
- Element by element product $y * y$
- Matrix multiplication $y \%*\% y$
- Matrix scalar multiplication $3 * y$
- Matrix addition $y + y$

Indexing, e.g. select first and second row

```
> z[c(1,2),]
```

Matrices

Assign new values to submatrices

```
> z[c(1:2), c(2:3)] <- matrix(c(20,21,22,23), nrow=2)
```

- Filtering, e.g. obtain those rows of matrix `z` having elements in the second column which are at least equal to 5

```
> z[z[,2] >= 5, ]
```

Lists

Creation

```
> joe <- list(name="Joe", salary=55000, staff=T)
```

▪ Indexing

```
> joe$salary
```

```
> joe[["salary"]]
```

```
> joe[[2]]
```

Vectors as list components

```
> my.list <- list(vec1 = c(1,2), vec2 = c(3,4),  
vec3 = 5:7)
```

Data frames

Creation

```
> person <- data.frame(height=person.height,  
weight=person.weight)
```

Indexing

```
> person[[1]]  
> person[["height"]]  
> person$height  
> person[c(1,2),]  
> person[-3,]
```

▪ Filtering

```
> person[person$height >= 1.7,]
```

Data frames

Data import

```
> person.data <- read.table(header=TRUE,  
"height_weight_data.txt", sep=",")
```

▪ Data modifications

```
> person.data$BMI <- person.data$Weight /  
person.data$Height^2
```

▪ Summary

```
> summary(person.data)
```

▪ Merging

```
> merge(person.data, person.data2)
```

Program today

- In a previous lecture, the word list example showed us that iterating through a set of elements is an important operation
- Today we will learn how we can program such iterations with so called “for loops”
- We will start with a recap of the word list example and continue with programming for loops

Looping

The most frequently used looping construct is

```
for(x in vec) {expression}
```

- The `for` loop iterates through all elements of the vector `vec`
- For each element of the vector `vec` there will be one iteration of the loop and `expression` is executed
- At each iteration, the variable `x` takes the value of the current element of `vec`
 - First iteration: `x = vec[1]`
 - Second iteration: `x = vec[2]`
 - ...

Print variable when iterating

Let's print out the value of variable `x` when iterating through the vector `vec`

```
> vec <- c(1:5)
```

```
> vec
```

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

```
> for(x in vec) {print (x) }
```

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

Recap: word list

- Web search and other types of textual data mining are of great interest
- Let's assume we have a collection of text documents
- Whenever we search for some term, we would like to retrieve those documents in which our search term appears most often
- Our first goal is to determine **which words** are in a text and **at which location** in the text each word occurs

Recap: word list

Let's consider this sentence as our text example:

- a text consists of a word and another word and so on and so forth

- For each word we need to obtain the location in the text:

a	1	5
text	2	
consists	3	
of	4	
word	6	9
and	7	10 13
another	8	
so	11	14
on	12	
forth	15	

Recap: word list

- Let's assume that we iterate through our text in a word by word manner: `a, text, consists, of, a, ...`
- Let's further assume that the current word in our iteration is always stored in the variable `word`
- Let's further assume that we have a counter `i` which is increased by 1 for every word: the counter tells the current position in the text

Recap: word list

Let's start with initializing our word list

```
> word.list <- list()
```

Our first word `a` is stored in the variable `word`

```
word <- "a"
```

Since it is our first word, our counter `i` has the value 1

```
> i <- 1
```

Now we add our current word `a` to our word list

```
> word.list[[word]] <- c(word.list[[word]], i)
```

Example: word list

In the last step, when adding the current word to our word list we used the two list characteristics

```
> word.list[[word]] <- c(word.list[[word]], i)
```

We can access list components with named tags

```
> joe[["salary"]]  
[1] 55000
```

1. We can have vectors as list components

```
2.> my.list <- list(vec = c(1,2))
```

```
3.> my.list
```

```
4.$vec
```

```
5.[1] 1 2
```

Example: word list

Let's check our word list after the first iteration

```
> word.list
```

```
$a
```

```
[1] 1
```

- The value for list component “a” is 1 since it is the first time that “a” was added to the list
- We interpret this intermediate result as word `a` has position 1

Example: word list

We go on with the second word `text`

```
> word <- "text"  
> i <- 2  
> word.list[[word]] <- c(word.list[[word]], i)
```

Let's check again our word list after the second iteration

```
> word.list  
$a  
[1] 1  
  
$text  
[1] 2
```

We observe that we now have a second list entry `text` with position value 2

Example: word list

We proceed with the next 3 iterations

```
> word <- "consists"  
> i <- 3  
> word.list[[word]] <- c(word.list[[word]], i)  
  
> word <- "of"  
> i <- 4  
> word.list[[word]] <- c(word.list[[word]], i)  
  
> word <- "a"  
> i <- 5  
> word.list[[word]] <- c(word.list[[word]], i)
```

Example: word list

We check our word list again

```
> word.list
```

```
$a
```

```
[1] 1 5
```

```
$text
```

```
[1] 2
```

```
$consists
```

```
[1] 3
```

```
$of
```

```
[1] 4
```

Example: word list

- So far we have learned how to represent which words are in a text and at which location in the text each word occurs
- We have iterated through our text in a word by word manner
- In practice, such iterations are automated with so called loops

Looping

The most frequently used looping construct is

```
for(x in vec) {expression}
```

- The `for` loop iterates through all elements of the vector `vec`
- For each element of the vector `vec` there will be one iteration of the loop and `expression` is executed
- At each iteration, the variable `x` takes the value of the current element of `vec`
 - First iteration: `x = vec[1]`
 - Second iteration: `x = vec[2]`
 - ...

Print variable when iterating

Let's print out the value of variable `x` when iterating through the vector `vec`

```
> vec <- c(1:5)
```

```
> vec
```

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

```
> for(x in vec) {print (x) }
```

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

Print variable when iterating

- The for loop works with other modes beside numeric as well

Example: like before we print the value of the variable when iterating through a vector of strings

```
> word.vector <- c("a", "text", "consists",  
"of")
```

```
> for(word in word.vector) {print (word)}  
[1] "a"  
[1] "text"  
[1] "consists"  
[1] "of"
```

Print variable when iterating

As an alternative we can create a new vector which ranges from 1 until the length of the vector, iterate through this vector and access the original vector via indexing

```
> vector.indices <- 1:length(word.vector)

> vector.indices
[1] 1 2 3 4

> for(i in vector.indices) {print(word.vector[i])}
[1] "a"
[1] "text"
[1] "consists"
[1] "of"
```

Print variable when iterating

We can write the alternative way in one line

```
> for(i in 1:length(word.vector)) {print  
(word.vector[i])}  
[1] "a"  
[1] "text"  
[1] "consists"  
[1] "of"
```

Print variable when iterating

- The advantage of the alternative way is that we have access to the iteration number and the vector elements

Example: print index `i` together with the vector element by using the `paste()` function for concatenating strings

```
> for(i in 1:length(word.vector)) {print  
(paste("Element", i, "is", word.vector[i]))}  
[1] "Element 1 is a"  
[1] "Element 2 is text"  
[1] "Element 3 is consists"  
[1] "Element 4 is of"
```

Compute length of a vector

- We can change the value of a variable during looping

For example, let's write our own function for computing the length of a vector

```
> vec <- c(1:10)

> counter <- 0

> for(x in vec) {   ??? }

> counter
[1] 10
```

Compute length of a vector

- We can change the value of a variable during looping

For example, let's write our own function for computing the length of a vector

```
> vec <- c(1:10)
```

```
> counter <- 0
```

```
> for(x in vec) {counter <- counter + 1}
```

```
> counter
```

```
[1] 10
```

Recap: writing our own function

A simple function that adds 1 to its input and returns the result

```
> AddOne <- function(x) { x+1 }
```

Function
name

Function
inputs

Instructions that take the
inputs and use them to
compute other values.

The last computed value
is returned by default.

Recap: writing our own function

After defining our function, we can work with it

```
> AddOne <- function(x) {x+1}
```

```
> AddOne(1)
[1] 2
```

```
> AddOne(-5)
[1] -4
```

```
> AddOne(c(1, 2, 3))
[1] 2 3 4
```

Recap: writing our own function

A more sophisticated function that adds a user-specified value to its first input

```
> AddValue <- function(x, Addend=1) {x+Addend}
```



In addition to the first input x we specify a second input `Addend` with default value 1.

Recap: writing our own function

After defining our new function, we can work with it

```
> AddValue <- function(x, Addend=1) {x+Addend}
```

```
> AddValue(1)
[1] 2
```

```
> AddValue(1,2)
[1] 3
```

```
> AddValue(c(1:3),2)
[1] 3 4 5
```

Compute length of a vector

Write our own function for computing the length of a vector

```
## function to compute length of vector vec
vec.length <- function(vec)
{
  # initialize counter
  counter <- 0

  # iterate through vec and increase counter
  for(x in vec) {counter <- counter + 1}

  # return counter
  return(counter)
}
```

Compute length of a vector

- In the previous function definition we've used # to comment our code
- It is a good practice to comment code in particular when sharing code

We test our function

```
> vec.length(c(1:10))  
[1] ?
```

```
> vec.length(c("Hande", "Cem", "Can"))  
[1] ?
```

Compute length of a vector

- We can change the value of a variable during looping

For example, let's write our own function for computing the length of a vector

```
> vec <- c(1:10)

> counter <- 0

> for(x in vec) {   ???? }

> counter
[1] 10
```

Compute norm of a vector

For a 3-dimensional vector, the Euclidean norm is defined as

-
- We will write two functions

(1) Compute the Euclidean norm of a n-dimensional vector

(2) Compute p-norm of a n-dimensional vector

-

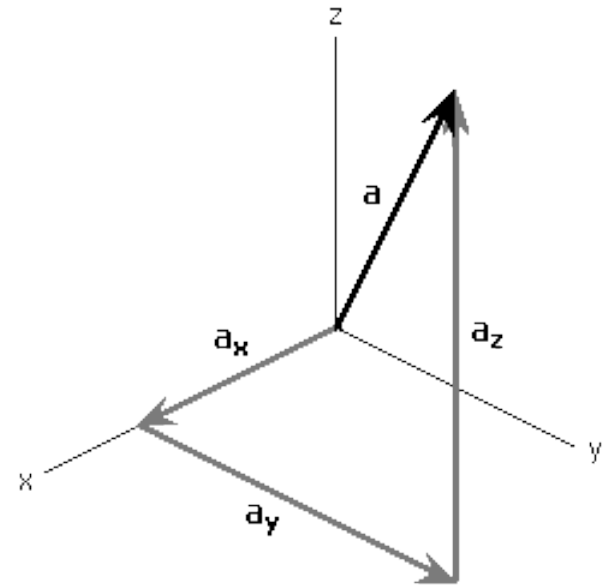


Figure 1

$\mathbf{x} = (x_1, x_2, \dots, x_n)$ is captured by the formula

$$\|\mathbf{x}\| := \sqrt{x_1^2 + \dots + x_n^2}.$$

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}$$

Compute Euclidean norm of a vector

```
## compute Euclidean norm of a vector vec
Euclid.norm <- function(vec)
{
  # initialize norm
  norm <- 0

  # compute sum of squared vector elements
  for(x in vec) {norm <- norm + x^2}

  # sqrt of sum
  norm <- sqrt(norm)

  return(norm)
}
```

Compute Euclidean norm of a vector

We test our function

```
> Euclid.norm(c(1, 2, 3))  
[1] 3.741657
```

```
> sqrt(1^2+2^2+3^2)  
[1] 3.741657
```

```
> Euclid.norm(c(sqrt(1), sqrt(3)))  
[1] 2
```

p-norm of a vector

```
## compute p-norm of a vector vec
p.norm <- function(vec, p=2)
{
  # initialize norm
  norm <- 0

  # compute sum of exponentiated vector elements
  for(x in vec) {norm <- norm + x^p}

  # p radical of sum
  norm <- (norm)^(1/p)

  return(norm)
}
```

p-norm of a vector

We test our function

```
> p.norm(c(1, 2, 3), p=1)  
[1] 6
```

```
> p.norm(c(1, 2, 3))  
[1] 3.741657
```

```
> p.norm(c(1, 2, 3), p=3)  
[1] 3.301927
```

Some analysis with midterm results

```
scores <- read.table("scores.txt",header=TRUE)
plot(c(1:97),scores$Score)
plot(c(1:97),sort(scores$Score))
hist(scores$Score)
scores.cut <- cut(scores$Score,breaks=5)
scores.table <- table(scores.cut)
pie (scores.table)
```

Homework

1. Create a text file which consist of body height and weight from 10 of your friends
2. Import the text file into R
3. Add a new BMI column
4. Create a scatter plot of your data
5. Create a second text file which consists of age and home city of your friends
6. Import the second file and merge it with the first one

Homework



1. Write a function that iterates through a vector and computes the sum of vector's elements
2. Write a function that iterates through all columns and rows of a matrix and computes the means of the columns and the rows