Introduction to Computing for Economics and Management

Midterm Summary
Data vectors

- The fundamental data type in R is the vector
- Data vectors are created with the construct `c`
  ```r
  > person.height <- c(1.70, 1.75, 1.62)
  > person.height <- c(person.height, 1.81)
  ```
- Vector elements must all have the same mode
- Available modes: integer, numeric, character, Boolean, complex
Data vectors

- Missing values are denoted with \texttt{NA}

- We can assign names to the elements of a data vector to make the vector more readable

\begin{verbatim}
> person.height <- c(Can=1.70, Cem=1.75, Hande=1.62)

> person.height
  Can   Cem Hande
1.70  1.75  1.62
\end{verbatim}
Data vector indexing

- We can access a single element of a vector by providing the index of the element in square brackets
  
  ```r
  > person.height[1]
  
  Can
  
  1.7
  ```

- We can select a subvector by providing a Boolean index vector
  
  ```r
  > person.height[c(T,F,T)]
  
  Can Hande
  
  1.70  1.62
  ```
Data vector indexing

- We can specify the element indices directly
  ```r
  > person.height[c(1,3)]
  Can  Hande
  1.70  1.62
  ```

- We exclude elements with negative indices
  ```r
  > person.height[c(-1, -3)]
  Cem
  1.75
  ```

- We can change the values of the selected elements
  ```r
  person.height[1] <- 1.72
  ```
Data vector filtering

- The idea behind filtering is to apply a Boolean evaluation function to each element of the vector

```r
> person.height > 1.65
  Can   Cem Hande
  TRUE  TRUE FALSE
```

- We use the results of the evaluation function for the filtering

```r
> person.height[person.height > 1.65]
  Can   Cem
  1.72  1.75
```
Data vector sorting

- We use the function `sort` for sorting a vector
  
  ```r
  > sort(person.height)
  Hande  Can  Cem
  1.62  1.70  1.75
  ```

- We can obtain a sorting in descending order
  
  ```r
  > sort(person.height, decreasing = TRUE)
  Cem  Can  Hande
  1.75  1.70  1.62
  ```

- We can sort a vector according to the values of some other vector
  
  ```r
  > person.weight[order(person.height)]
  Hande  Can  Cem
  61  65  66
  ```
Vector recycling

- When applying an operation to two vectors which requires them to be the same length, the shorter one will be repeated until it is long enough to match the longer one.

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

Warning message:
In `c(1, 2, 3) + c(1, 2, 3, 4)`:
  longer object length is not a multiple of shorter object length
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

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

- We can perform calculations with vectors just like ordinary numbers

- Element wise operations
  > person.height^2
  Can Cem Hande
  2.9584 3.0625 2.6244

- Vector addition, e.g. persons have gained/lost weight
  > person.weight + c(1.5, 1.75, -0.5)
  Can Cem Hande
  66.50 67.75 60.50
Data vector operations

- Operations on multiple vectors
  
  ```r
  > person.weight / person.height^2
  Can     Cem    Hande
  21.97134 21.55102 23.24341
  ```

- The result of a vector calculation can be assigned to a new data vector for further processing
  
  ```r
  > bmi <- person.weight / person.height^2
  > bmi
  Can     Cem    Hande
  21.97134 21.55102 23.24341
  ```
Rounding of numbers

- Round to the specified number of decimal places with function `round`
  
  ```
  > round(bmi, digits=1)
  Can   Cem Hande
  22.0  21.6  23.2
  ```

- Alternative functions for rounding of numbers are `ceiling`, `floor`, `trunc`, and `signif`
Data vector operations

- Often used functions that operates on vectors are `mean`, `length` and `sd`

```r
> bmi <- person.weight / person.height^2

> mean(bmi)
[1] 23.31768

> length(bmi)
[1] 3

> sd(bmi)
[1] 2.294295
```
Creating regular sequences

- The `from:to` syntax is a simple way to generate a sequence from `from` to `to` in steps of 1 or -1

```r
> 1:5
[1] 1 2 3 4 5

> 11:15
[1] 11 12 13 14 15

> 3:0
[1] 3 2 1 0

> seq_1_100 <- 1:100
```
Function `seq`

- `seq` function arguments
  - `from` starting value of the sequence
  - `to` end value of the sequence
  - `by` increment of the sequence
  - `length.out` desired length of the sequence
  - `along.with` take the length from the length of this argument

```r
> seq(5, 10)
[1] 5 6 7 8 9 10

> seq(5, 10, 2)
[1] 5 7 9
```
Creating repeated values with function `rep`

- **rep** function arguments and default values
  - **x** vector of factor that is repeated
  - **times = 1** number of times to repeat
  - **length.out = NA** desired length of the result
  - **each = 1** each element of x is repeated each times

```r
> rep(1, 3)
[1] 1 1 1

> rep(1:4, 2)
[1] 1 2 3 4 1 2 3 4

> rep(1:4, each = 2)
[1] 1 1 2 2 3 3 4 4
```
Matrix creation

- In R, a matrix is a vector with two additional attributes, the number of rows and number of columns.

- One of the ways to create a matrix is via the `matrix` function to obtain a matrix from a given data vector with `nrow` number of rows and `ncol` number of columns.

```r
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> y
     [,1] [,2]
[1,]   1   3
[2,]   2   4
```
Matrix column and row notation

\[
\begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix}
\]

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

Notation for columns: [,1] means first column, [,2] second column, etc.

Notation for rows: [1,] means first row, [2,] second row, etc.
Matrix column and row access

- We can access single columns and rows with the respective column/row notation

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

> y[,1]
[1] 1 2

> y[2,]
[1] 2 4
```
Matrix single element access

- We can access single elements of the matrix by providing the indices of row and column

```r
> y
[,1] [,2]
[1,] 1 3
[2,] 2 4

> y[1,1]
[1] 1

> y[2,1]
[1] 2
```
Matrix creation order

- Storage of a matrix is in column-major order: first all of column 1 is stored, then all of column 2, etc.

- We can change the column-major order by providing the additional argument `byrow = TRUE` for filling the matrix by rows

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

> y
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```
Matrix row names and column names

- We can provide names for the rows and columns of a matrix

```r
> y
   [,1] [,2]
[1,]  1  3
[2,]  2  4

> rownames(y) <- c("Row1", "Row2")
> colnames(y) <- c("Col1", "Col2")

> y
          Col1    Col2
Row1        1 3 Row2        2 4
```
Matrix creation with `cbind` and `rbind`

- We can “glue” vectors together, columnwise or rowwise, using the `cbind` and `rbind` functions

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

> rbind(c(1,2), c(3,4))
     [,1] [,2]
[1,]   1   2
[2,]   3   4
```
Matrix modification with `cbind` and `rbind`

- Add a column to an existing matrix

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

> y <- cbind(c(11, 12), y)
> y
   [,1] [,2] [,3]
[1,] 11  1  3
[2,] 12  2  4
```
Matrix recycling

- The automatic lengthening of vectors also works with matrices

```r
> z <- matrix(c(1:9), nrow=3)
> z

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

> cbind(10, z)

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

- Matrix transposition $\mathbb{t}(y)$
- Element by element product $y \times y$
- Matrix multiplication $y \odot \odot y$
- Matrix scalar multiplication $3 \times y$
- Matrix addition $y + y$
Matrix indexing

- We can access more than a single column/row/element at once

```r
> z <- matrix(c(1:9), nrow=3)
> z
[,1] [,2] [,3]
[1,] 1  4  7
[2,] 2  5  8
[3,] 3  6  9
```

- Select columns 2 and 3

```r
> z[, c(2,3)]
[,1] [,2]
[1,] 4  7
[2,] 5  8
[3,] 6  9
```
Matrix indexing

- Select first and second row
  ```r
  > z[c(1,2),]
  [,1] [,2] [,3]
  [1,] 1  4  7
  [2,] 2  5  8
  ```

- Select third column of first and second row
  ```r
  > z[c(1,2),3]
  [1] 7 8
  ```
Matrix indexing

- We use negative subscripts to exclude certain elements, e.g. request all rows except the second

```r
> z[, -2]
   [,1] [,2]
[1,]  1  7
[2,]  2  8
[3,]  3  9
```
Matrix indexing

- We can assign new values to submatrices

```r
> z
 [,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9

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

> z
 [,1] [,2] [,3]
[1,] 20 22
[2,] 21 23
[3,] 3 6 9
```
Matrix indexing

- We can delete rows or columns by reassignment, e.g. keep only first two rows and delete third row

```r
> z
[,1] [,2] [,3]
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9

> z <- z[c(1,2),]
> z
[,1] [,2] [,3]
[1,]  1  4  7
[2,]  2  5  8
```
Matrix filtering

- Similar to data vector filtering, the concept behind is to first apply a Boolean evaluation function.

- For each single element, the Boolean evaluation function returns **TRUE** in case of a positive evaluation and **FALSE** in case of a negative evaluation.

\[
> z > 3 \\
[1,] FALSE TRUE TRUE \\
[2,] FALSE TRUE TRUE \\
[3,] FALSE TRUE TRUE
\]
Matrix filtering

- Similar to data vector filtering, we can perform evaluation and filtering in one line

```R
> z[z > 3]
[1] 4 5 6 7 8 9
```

We provide the evaluation function directly in the square brackets for selecting those elements that fulfill the evaluation function.
Matrix filtering

- In contrast to data vector filtering, we can perform more complex filtering tasks with matrices, e.g. obtain those rows of matrix $z$ having elements in the second column which are at least equal to 5.

```r
> z
[,1] [,2] [,3]
[1,] 1  4  7
[2,] 2  5  8
[3,] 3  6  9

> z[z[,2] >= 5,]
[,1] [,2] [,3]
[1,] 2  5  8
[2,] 3  6  9
```
Matrix function \texttt{apply()}

- An often used generic function in R is \texttt{apply()}

- \texttt{apply()} executes a user-specified function on each of the rows or each of the columns of a matrix

- \texttt{apply(m, dimcode, f, fargs)}
  - \texttt{m} is the matrix
  - \texttt{dimcode} equal to 1 means that the function is applied to rows, \texttt{dimcode} equal to 2 means that the function is applied to columns
  - \texttt{f} is the function to be applied
  - \texttt{fargs} is an optional set of arguments to be supplied to \texttt{f}
Writing our own function

- We write a simple function that adds 1 to its input and returns the result

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

Function name | Function inputs | Instructions that take the inputs and use them to compute other values.
--- | --- | ---
AddOne | x | {x+1}

The last computed value is returned by default.
Writing our own function

- Let’s write another more sophisticated function that adds a user-specified value to its first input

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

In addition to the first input `x` we specify a second input `Addend` with default value 1.
Using our own function with `apply()`

- First we apply `AddValue` to the rows of `z`

  ```r
  > z
  [,1] [,2] [,3]
  [1,]  1  4  7
  [2,]  2  5  8
  [3,]  3  6  9
  ```

  ```r
  > apply(z, 1, AddValue)
  [,1] [,2] [,3]
  [1,]  2  3  4
  [2,]  5  6  7
  [3,]  8  9 10
  ```

  Resulting vector when adding 1 to the first row
Lists

- Lists can combine objects of different types

- We create a list to represent the data from Joe
  \[
  > \text{joe} <- \text{list}("\text{Joe}", 55000, T)
  \]

- An entire employee database might then be a list of lists
Creating lists

- Let's check our new list `joe`

  ```r
  > joe
  [[1]]
  [1] "Joe"

  [[2]]
  [1] 55000

  [[3]]
  [1] TRUE
  ```

- We observe that the three components name, salary and membership are indexed by `[[1]]`, `[[2]]`, and `[[3]]`
Creating lists

- We better provide name tags for our components when creating a list

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

> joe
$name
[1] "Joe"

$salary
[1] 55000

$staff
[1] TRUE
```
List indexing

- We can access list components in several different ways – each of them is useful in different contexts

```r
> joe$salary
[1] 55000

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

> joe[[2]]
[1] 55000
```
Adding list elements

- New components can be added after a list is created
- We can add new components in different ways

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

> joe$age <- 39

> joe[[5]] <- 1976

> joe[6:7] <- c(TRUE, TRUE)
```
Deleting list elements

- We can delete a list component by setting it to NULL
  
  > joe$salary <- NULL
  
  > joe$staff <- NULL

- After deleting, the indices of subsequent elements automatically move up
Vectors as list components

- Beside storing atomic entries like *Joe* or *55000* in a list, we can have vectors as list components

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

> my.list
$vec1
[1] 1 2

$vec2
[1] 3 4

$vec3
[1] 5 6 7
```
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
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
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)
Word list

- Let’s check our word list after the first iteration
  
  ```
  > word.list
  
  $a$
  
  [1] 1
  ```

- We interpret this intermediate result as word a has position 1

- We go on with a few other words
Word list

- When we check `word.list` again we obtain

```r
> word.list
$a
[1] 1 5

text
[1] 2

$consists
[1] 3

$of
[1] 4
```
Accessing list components

- If the components in a list do have tags, we can obtain them via `names()`

```r
> names(joe)
[1] "name" "salary" "staff"

> names(word.list)
[1] "a" "text" "consists" "of"
```
Sort word list alphabetically

- We can write all three steps in one line
  
  ```
  > word.list[sort(names(word.list))]
  $a
  [1] 1 5
  
  $consists
  [1] 3
  
  $of
  [1] 4
  
  $text
  [1] 2
  ```
Accessing list values

- We can obtain list values by using `unlist()`
  ```r
  > unlist(joe)
  name  salary   staff
  "Joe" "55000" "TRUE"
  
  > unlist(word.list)
  a1       a2     text consists       of
  1        5        2        3        4
  ```

- We observe that in the first case we retrieve a vector of character strings and in the second case a numeric vector.

- The reason for the different result modes is that list components are coerced to a common mode during `unlist`
Applying functions to lists

- `apply()` executes a user-specified function on each of the rows or each of the columns of a matrix, e.g. `apply(z, 1, mean)` computes the row means of matrix `z`.

- The function `lapply()` works like the `apply()` function: the specified function is applied on each component of a list and another list is returned.

- `lapply(l, f, fargs)`
  - `l` is the list
  - `f` is the function
  - `fargs` is an optional set of arguments for function `f`
Applying functions to lists

- Example: count number of words from our `word.list`

```r
> lapply(word.list, length)
$a
[1] 2
$text
[1] 1
$consists
[1] 1
$of
[1] 1
```
Applying functions to lists

- `sapply()` works like `lapply()` but instead of a list it returns a vector or a matrix

- Previous example with `sapply()`

```r
> sapply(word.list, length)
a text consists of
2 1 1 1
```
We can write all three steps in one line

```r
> word.list[order(sapply(word.list, length))]
$text
[1] 2

$consists
[1] 3

$of
[1] 4

$a
[1] 1 5
Data frame

- A data frame is like a matrix, with a two-dimensional rows- and columns structure

- Each column may have a different mode, e.g. one column may consist of numbers, and another column might have character strings or Boolean entries

- On a technical level, a data frame is a list: each component of that list consists of equal-length vectors
Creating data frames

- One way to create a data frame is to combine available equal-length vectors

```r
> person <- data.frame(height=person.height, weight=person.weight, member=person.member, initial=c("C", "C", "H"))
> person

  height weight member initial
Can   1.70   65   TRUE       C
Cem   1.75   66   TRUE       C
Hande 1.62   61  FALSE       H
```

- We observe that the columns retain their original mode and that the vector element names are used to label the rows of the data frame
Accessing data frames

- Since a data frame is technically a list, we can access it via component index values or component names.

```r
> person[[1]]
[1] 1.70 1.75 1.62

> person[["height"]]
[1] 1.70 1.75 1.62

> person$height
[1] 1.70 1.75 1.62
```
Accessing data frames

- We can access it in a matrix-like fashion as well, e.g. view column 1

  ```r
  > person[,1]
  [1] 1.70 1.75 1.62
  ```

- Element in third row, second column

  ```r
  > person[3,2]
  [1] 61
  ```
Data frame indexing

- Since data frames can be accessed in a matrix-like fashion, we can select rows and columns in a matrix-like way

- First and second row
  
  ```
  > person[c(1,2),]
  height  weight  member
  Can     1.70     65   TRUE
  Cem     1.75     66   TRUE
  ```

- Third column of first and second row
  
  ```
  > person[c(1,2),3]
  [1] TRUE TRUE
Data frame indexing

- Like for matrices, we can use negative indices to exclude rows or columns

```r
> person[-3,]
  height  weight  member
Can 1.70     65   TRUE
Cem 1.75     66   TRUE

> person[, -3]
  height  weight
Can 1.70     65
Cem 1.75     66
Hande 1.62    61
```
Data frame filtering

- Similar to data vector and matrix filtering, the concept behind is to apply a Boolean evaluation function.

- Example: retrieve all observations for which person height is at least 1.7

> person[person$height >= 1.7,]
  height  weight member
Can  1.70    65    TRUE
Cem  1.75    66    TRUE
Data frame modifications

- Like for matrices, we can use `rbind()` and `cbind()` to add new rows or columns to a data frame.

- Usually, we add a new row in form of a list:

```r
> person <- rbind(person, Lale=list(1.76, 64, T))
```

```r
> person

<table>
<thead>
<tr>
<th>height</th>
<th>weight</th>
<th>member</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can</td>
<td>1.70</td>
<td>65</td>
</tr>
<tr>
<td>Cem</td>
<td>1.75</td>
<td>66</td>
</tr>
<tr>
<td>Hande</td>
<td>1.62</td>
<td>61</td>
</tr>
<tr>
<td>Lale</td>
<td>1.76</td>
<td>64</td>
</tr>
</tbody>
</table>
```
Data frame modifications

- We use `cbind()` for adding a new column.

```r
> person <- cbind(person, initial=c("C", "C", "H", "L"))
> person

    height weight member initial
Can  1.70    65  TRUE       C
Cem  1.75    66  TRUE       C
Hande 1.62    61  TRUE       H
Lale 1.76    64  TRUE       L
```
Data frame modifications

- As an alternative to `cbind()` we can use the `$` notation

```
> person$BMI <- person$weight / person$height^2

> person
   height weight member initial  BMI
Can   1.70    65   TRUE      C 22.49135
Cem   1.75    66   TRUE      C 21.55102
Hande 1.62    61   TRUE      H 23.24341
Lale  1.76    64   TRUE      L 20.66116
```
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.

```r
> 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)
```
Data import

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

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

> 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

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

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

```r
> 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

```r
> 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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Height</th>
<th>Weight</th>
<th>BMI</th>
<th>above22.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can</td>
<td>1.70</td>
<td>65</td>
<td>22.49</td>
<td>FALSE</td>
</tr>
<tr>
<td>Cem</td>
<td>1.75</td>
<td>66</td>
<td>21.55</td>
<td>FALSE</td>
</tr>
<tr>
<td>Hande</td>
<td>1.62</td>
<td>61</td>
<td>23.24</td>
<td>TRUE</td>
</tr>
<tr>
<td>Lale</td>
<td>1.76</td>
<td>64</td>
<td>20.66</td>
<td>FALSE</td>
</tr>
<tr>
<td>Arda</td>
<td>1.78</td>
<td>63</td>
<td>19.88</td>
<td>FALSE</td>
</tr>
<tr>
<td>Bilgin</td>
<td>1.77</td>
<td>84</td>
<td>26.81</td>
<td>TRUE</td>
</tr>
<tr>
<td>Cem</td>
<td>1.69</td>
<td>75</td>
<td>26.26</td>
<td>TRUE</td>
</tr>
<tr>
<td>Ozlem</td>
<td>1.75</td>
<td>65</td>
<td>21.22</td>
<td>FALSE</td>
</tr>
<tr>
<td>Ali</td>
<td>1.73</td>
<td>75</td>
<td>25.06</td>
<td>TRUE</td>
</tr>
<tr>
<td>Haluk</td>
<td>1.71</td>
<td>81</td>
<td>27.70</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
```
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

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

```
person.data$Height
```

```
person.data$Weight
```
Merging data frames

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

```r
> 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
```
Short Summary Vectors

- Integer mode
  \[ > \text{person.weight} \leftarrow \text{c}(65, 66, 61) \]

- Numeric (floating-point number)
  \[ > \text{person.height} \leftarrow \text{c}(1.70, 1.75, 1.62) \]

- Character (string)
  \[ > \text{person.name} \leftarrow \text{c}("Can", "Cem", "Hande") \]

- Logical (Boolean)
  \[ > \text{person.female} \leftarrow \text{c}(\text{FALSE}, \text{FALSE}, \text{TRUE}) \]

- Complex
  \[ > \text{complex.numbers} \leftarrow \text{c}(1+2i, -1+0i) \]
Short Summary Vectors

- Assign names to the elements of a data vector
  > person.height <- c(Can=1.70, Cem=1.75, Hande=1.62)

- Indexing
  > person.height[c(T,F,T)]
    Can     Hande
    1.70    1.62
  
  > person.height[c(1,3)]
    Can     Hande
    1.70    1.62
  
  > person.height[-1]
    Cem     Hande
    1.75    1.62
Short Summary Vectors

- Filtering
  \[
  \text{Can} \quad \text{Cem} \\
  1.72 \quad 1.75
  \]

- Recycling
  \[
  c(1, 2, 3) + c(1, 2, 3, 4) \\
  [1] \quad 2 \quad 4 \quad 6 \quad 5
  \]

- Vector operations
  \[
  \text{Can} \quad \text{Cem} \quad \text{Hande} \\
  21.97134 \quad 21.55102 \quad 23.24341
  \]
Short Summary 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),]`
Short Summary 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,]
Short Summary Lists

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

- **Indexing**
  
  ```r
  > joe$salary
  > joe[["salary"]]
  > joe[[2]]
  ```

- **Vectors as list components**
  
  ```r
  > my.list <- list(vec1 = c(1,2), vec2 = c(3,4), vec3 = 5:7)
  ```
Short Summary Data frames

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

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

- **Filtering**
  
  ```r
  > person[person$height >= 1.7,]
  ```
Short Summary Data frames

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

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

- Summary
  ```r
  > summary(person.data)
  ```

- Merging
  ```r
  > merge(person.data, person.data2)
  ```