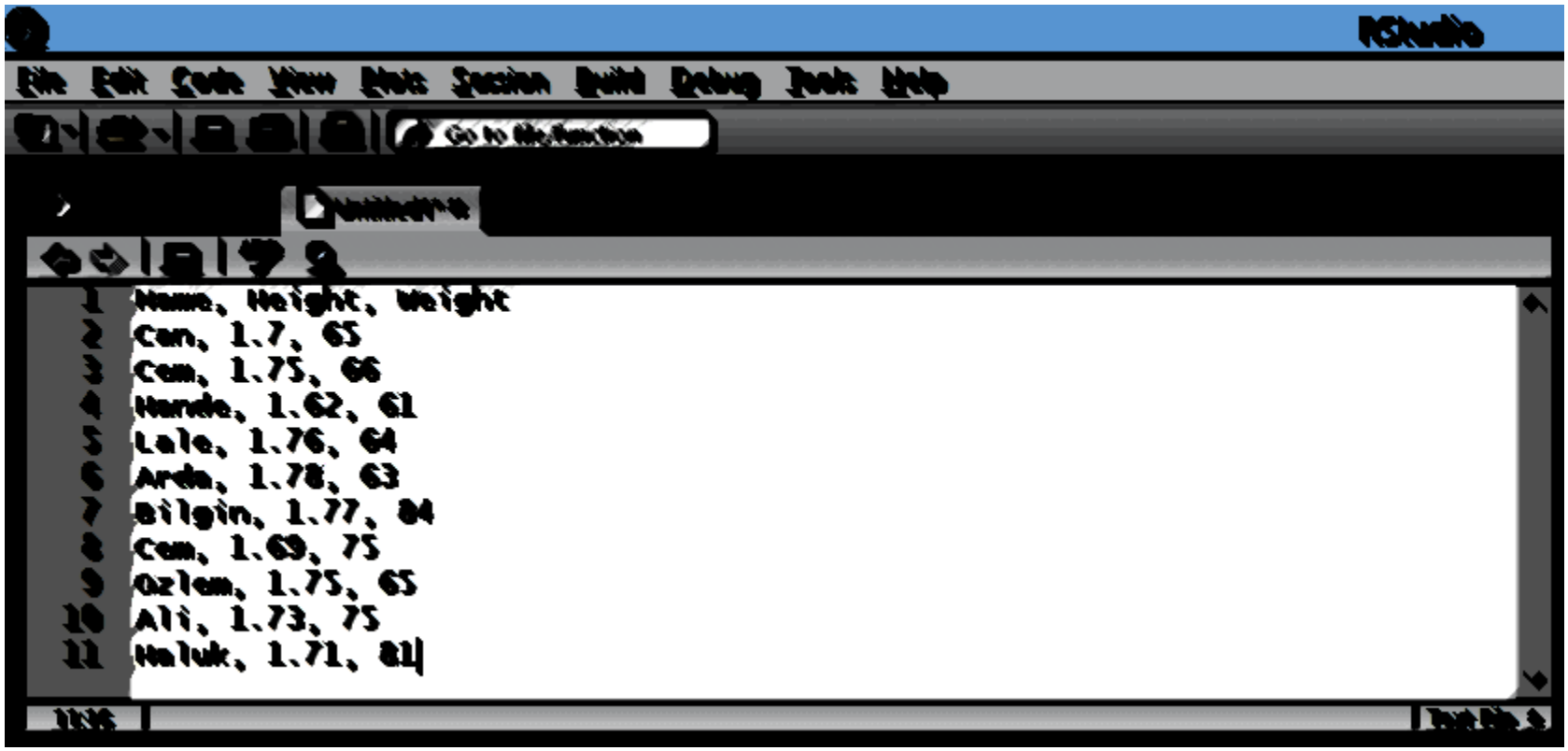
Introduction to Computing for Economics and Management

Lecture 5: Data Frames



Midterm on 21st

- Tricky questions
- Sections
- Additional office hours (BM33):
  - Today: 16:00 – 17:00
  - Thursday: 10:00 – 11:00

Acknowledgement

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

Previous lecture

- Shortcomings of vectors and matrices
- Creating lists
- List indexing
- Adding/deleting list elements
- Concatenate lists
- Vectors as list components
- Example: word list
- Accessing list components/values
- Example: sort word list alphabetically
- Applying functions to lists
- Example: sort word list by word frequency

Previous lecture: shortcomings of vectors and matrices

- Vector elements must all have the same mode

- Matrix elements must all have the same mode

However, in practice we often have to deal with mixed mode data sets, e.g. in an employee database we need to store name, salary, and Boolean membership

```
name="Joe", salary=55000, staff=TRUE
```

Previous lecture: lists

- Lists can combine objects of different types

We create a list to represent the data from Joe

```
> joe <- list("Joe", 55000, T)
```

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

Previous lecture: creating lists

Let's check our new list `joe`

```
> 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]]`

Previous lecture: creating lists

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

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

> joe
$name
[1] "Joe"

$salary
[1] 55000

$staff
[1] TRUE
```

Previous lecture: list indexing

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

```
> joe$salary
[1] 55000

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

> joe[[2]]
[1] 55000
```

Previous lecture: adding list elements

- New components can be added after a list is created

We can add new components in different ways

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

> joe$age <- 39

> joe[[5]] <- 1976

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

Previous lecture: 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

Previous lecture: vectors as list components

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

```
> 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
```

Previous lecture: 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

Previous lecture: 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

Previous lecture: 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)
```

Previous lecture: 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

Previous lecture: word list

When we check `word.list` again we obtain

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

$text
[1] 2

$consists
[1] 3

$of
[1] 4
```

Previous lecture: accessing list components

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

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

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

Previous lecture: 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
```

Previous lecture: accessing list values

We can obtain list values by using `unlist()`

```
> 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

Previous lecture: 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)` compute 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`

Previous lecture: applying functions to lists

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

```
> lapply(word.list, length)
$a
[1] 2

$text
[1] 1

$consists
[1] 1

$of
[1] 1
```

Previous lecture: applying functions to lists

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

Previous example with `sapply()`

```
> sapply(word.list, length)
     a   text consists     of
     2      1        1      1
```

Previous lecture: sort word list by word frequency

We can write all three steps in one line

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

$consists
[1] 3

$of
[1] 4

$a
[1] 1 5
```

Previous lecture: Homework

1. Create a word list from the full text "`a text consists of a word and another word and so on and so forth`" using the helper variables `word` and `i` as shown in the lecture

2. Sort your word list alphabetically by word

3. Sort your word list by word frequency

Create another list which contains the vectors `(1.65,1.70,1.75,1.80,1.85, 1.90)` and `(1 1 2 3 3 4)`. Use the `seq` function to create the vectors first.

4. Compute the median of both vectors in the list using `sapply`

Program today

- Shortcomings of vectors and matrices
- Creating data frames
- Accessing data frames
- Data frame indexing
- Data frame modifications
- Data import from file
- Data frame summary
- Scatter plot
- Merging data frames

Shortcomings of vectors and matrices

- Let's come back to our person height and weight example

In a previous lecture we have seen how to manage height and weight observations with vectors

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

> person.weight <- c(Can=65, Cem=66, Hande=61)

> person.height
  Can   Cem Hande
 1.70  1.75  1.62
> person.weight
  Can   Cem Hande
   65    66    61
```

Shortcomings of vectors and matrices

Next, we stored height and weight in a matrix

```
> person.height.weight <- rbind(c(1.7,65),
c(1.75,66), c(1.62,61))

> rownames(person.height.weight) <- c("Can",
"Cem", "Hande")

> colnames(person.height.weight) <- c("Height",
"Weight")

> person.height.weight
       Height Weight
Can      1.70     65
Cem      1.75     66
Hande    1.62     61
```

Shortcomings of vectors and matrices

- Let's now assume we need to add a Boolean membership as a third dimension

When working with vectors, we need to create a new vector to handle the Boolean membership

```
> person.member <- c(Can=T, Cem=T, Hande=F)
```

- In general, we need a separate vector for each dimension of our data set

- Shortcomings
  - No single data structure but several vectors
  - When performing data modifications, like deleting/adding an entry, we have to modify many vectors

Shortcomings of vectors and matrices

- The main shortcoming of matrices is the fact that all entries must have the same mode

If we try to create a matrix with different modes, all entries will be coerced to one common mode, e.g. Boolean in combination with numbers will be coerced to numbers

```
> person.height.weight <- rbind(c(1.7,65,T),
c(1.75,66,T), c(1.62,61,F))

> person.height.weight
     [,1] [,2] [,3]
[1,] 1.70   65    1
[2,] 1.75   66    1
[3,] 1.62   61    0
```

Shortcomings of vectors and matrices

If we try to create a matrix with numbers and character strings, all entries will be coerced to strings

```
> person.height.weight <- rbind(c(1.7,65,T,"C"),
c(1.75,66,T,"C"), c(1.62,61,F,"H"))

> person.height.weight
     [,1]   [,2] [,3]    [,4]
[1,] "1.7"  "65" "TRUE"  "C"
[2,] "1.75" "66" "TRUE"  "C"
[3,] "1.62" "61" "FALSE" "H"
```

- Such a coercing is a serious disadvantage since we can not longer calculate with the numbers, e.g. BMI computation does not work any longer

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

```
> 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

Creating data frames

Data recycling works for data frames as well

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

> person
       height weight member
Can      1.70     65   TRUE
Cem      1.75     66   TRUE
Hande    1.62     61   TRUE
```

`TRUE` was repeated until it matched the length of the other vectors

Accessing data frames

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

```
> 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

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

Element in third row, second column

```
> 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

```
> 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

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

> person
      height weight member
Can     1.70     65   TRUE
Cem     1.75     66   TRUE
Hande   1.62     61   TRUE
Lale    1.76     64   TRUE
```

Data frame modifications

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

```
> 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

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 lecture, we will prepare a file which contains our data and import the file content into R