Introduction to Computing for Economics and Management

Lecture 4: Lists

Previous lecture

- Matrix creation
- Matrix operations
- Matrix indexing
- Matrix filtering
- Matrix function `apply()`
- Writing own functions
- Differences between vectors and matrices
- Higher-dimensional arrays
Previous lecture: 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
```

Previous lecture: matrix column and row notation

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

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

Notation for rows: `[1,]` means first row, `[2,]` second row, etc.
Previous lecture: 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
```

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

---

Previous lecture: 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
```

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

---
Previous lecture: 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
```

Previous lecture: 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
```
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
```

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
```
Previous lecture: 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
```

Previous lecture: matrix operations

- Matrix transposition `t(y)`
- Element by element product `y * y`
- Matrix multiplication `y %*% y`
- Matrix scalar multiplication `3 * y`
- Matrix addition `y + y`
Previous lecture: 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
  ```

Previous lecture: 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
  ```
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
```

We can assign new values to submatrices

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

> z
  [,1] [,2] [,3]
 [1,]  1  20  22
 [2,]  2  21  23
 [3,]  3   6   9
```
Previous lecture: matrix indexing

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

\[
> z = \begin{bmatrix}
[1,] & [2,] & [3,] \\
[1,] & 1 & 4 & 7 \\
[2,] & 2 & 5 & 8 \\
[3,] & 3 & 6 & 9 \\
\end{bmatrix}
\]

\[
> z <- z[\text{c}(1,2),] \\
> z = \begin{bmatrix}
[1,] & [2,] & [3,] \\
[1,] & 1 & 4 & 7 \\
[2,] & 2 & 5 & 8 \\
\end{bmatrix}
\]

---

Previous lecture: 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 \text{TRUE} in case of a positive evaluation and \text{FALSE} in case of a negative evaluation

\[
> z > 3 = \begin{bmatrix}
[1,] & [2,] & [3,] \\
[1,] & \text{FALSE} & \text{TRUE} & \text{TRUE} \\
[2,] & \text{FALSE} & \text{TRUE} & \text{TRUE} \\
[3,] & \text{FALSE} & \text{TRUE} & \text{TRUE} \\
\end{bmatrix}
\]
Previous lecture: matrix filtering

- In a second step, we use the results of the evaluation function for the filtering

- Example: obtain elements of $z$ that are larger than 3
  
  ```r
  > greater3 <- z > 3
  
  > greater3
  [,1] [,2] [,3]
  [1,] FALSE TRUE TRUE
  [2,] FALSE TRUE TRUE
  [3,] FALSE TRUE TRUE
  
  > z[greater3]
  [1] 4 5 6 7 8 9
  ```

---

Previous lecture: 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
Previous lecture: 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.

\[
> z = \begin{bmatrix}
1 & 4 & 7 \\
2 & 5 & 8 \\
3 & 6 & 9
\end{bmatrix}
\]

\[
> z[2,] = \begin{bmatrix}
2 & 5 & 8 \\
3 & 6 & 9
\end{bmatrix}
\]

Previous lecture: 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

\[
\text{apply}(m, \text{dimcode}, f, fargs)
\]

- \( m \) is the matrix
- \( \text{dimcode} \) equal to 1 means that the function is applied to rows, \( \text{dimcode} \) equal to 2 means that the function is applied to columns
- \( f \) is the function to be applied
- \( fargs \) is an optional set of arguments to be supplied to \( f \)
We write a simple function that adds 1 to its input and returns the result:

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

Instructions that take the inputs and use them to compute other values. The last computed value is returned by default.

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.
Previous lecture: 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

> 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

---

**Program today**

- 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
Shortcomings of vectors and matrices

- Vector elements must all have the same mode
  - 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")
  - ...

- 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

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

Lists

- Lists can combine objects of different types

- List plays a central role in R

- Lists forming the basis for data frames, object-oriented programming, and others

- Technically, a list is a vector
  - Ordinary vectors that we’ve been using so far are termed atomic vectors, since their components cannot be broken down into smaller components
  - Lists are referred to as recursive vectors
Creating lists

- Let’s consider again an employee database in which we need to store name, salary, and Boolean membership
  name="Joe", salary=55000, staff=T

- We have three different modes here: character, numeric, and Boolean

- We create a list to represent the data from Joe
  > joe <- list("Joe", 55000, T)

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

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]]
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

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

- The third example shows that although we have used name tags when creating the list, we still can access the list components with double brackets [[ ]] notation
List indexing

- If we use single brackets [] for indexing, we retrieve a sublist of the original list as result

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

```r
> joe[2]
$salary
[1] 55000
```

- By contrast, if we use double brackets [[ ]] for referencing a single component, the result has the type of that component

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

$salary
[1] 55000

List indexing

- We can use the single brackets [] notation for creating sublists from the original list

```r
> joe.sub <- joe[1:2]
```

```r
> joe.sub
$name
[1] "Joe"
```

$salary
[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)
```

After the previous adding operation we check the content of our list `joe`

We observe that `joe` consists of a mix of named tags and unnamed ones
Deleting list elements

- We can delete a list component by setting it to NULL

```r
> joe$salary <- NULL
> joe$staff <- NULL
```

- After deleting, the indices of subsequent elements automatically move up

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

$age
[1] 39

[[3]]
[1] 1976

[[4]]
[1] TRUE

[[5]]
[1] TRUE
```

Before deleting, the entries 1976 and the to Boolean entries TRUE had index positions 5, 6, and 7

After deleting two entries in the upper part of the list, their indices were shifted by 2 positions
### Concatenate lists

- Similar to atomic vectors, we can concatenate lists

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

> joe <- c(joe, list(age=39, 1976))

> joe
$name
 [1] "Joe"
$salary
 [1] 55000
$staff
 [1] TRUE
$age
 [1] 39

[[5]]
 [1] 1976
```

### 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
```
Example: word list

- Web search and other types of textual data mining are of great interest

- In the following we will learn how useful lists are when dealing with aspects that are important in text search functions

- Our first goal is to determine which words are in a text and at which location in the text each word occurs

- Later today we will learn how to sort word lists alphabetically and how to sort by word frequency

---

Example: 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
Example: word list

- How can we represent such word occurrences? Let’s first consider a matrix like representation:
  - Each unique word is represented by one (named) matrix row
  - The columns stand for positions 1, 2, 3, …

- For our exemplary text, the matrix would look like

<table>
<thead>
<tr>
<th></th>
<th>Position1</th>
<th>Position2</th>
<th>Position3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>text</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>consists</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>of</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>word</td>
<td>6</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>and</td>
<td>7</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>

Example: word list

- The matrix approach has a couple of major drawbacks:

  - We need to know the maximum frequency with which a word appears in our text in order to create the right number of matrix columns beforehand
  - The frequency of words in a text is not uniformly distributed: think about how often the word “the” appears in an English text in comparison to other words
  - Because of the non-uniform distribution of word occurrences, our matrix would consist of a lot of zeros which is a waste of memory
Example: word list

- Let's now consider how we can better represent the word occurrences with a 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

Example: word list

Before we go on with the algorithm for creating the word list, let's recapitulate two list characteristics that we need

1. We can access list components with named tags
   
   ```r
   > joe[["salary"]]
   [1] 55000
   ```

2. We can have vectors as list components

   ```r
   > my.list <- list(vec = c(1,2))
   
   > my.list
   $vec
   [1] 1 2
   ```
Example: word list

- Let's start with initializing our word list
  ```r
  > word.list <- list()
  ```

- Our first word `a` is stored in the variable `word`
  ```r
  word <- "a"
  ```

- Since it is our first word, our counter `i` has the value 1
  ```r
  > i <- 1
  ```

- Now we add our current word `a` to our word list
  ```r
  > 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 previously mentioned list characteristics
```r
> word.list[[word]] <- c(word.list[[word]], i)
```

1. We can access list components with named tags
   ```r
   > joe["salary"]
   [1] 55000
   ```

2. We can have vectors as list components
   ```r
   > my.list <- list(vec = c(1,2))
   > my.list
   $vec
   [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 a plain 1 since it is the first time that “a” was added to the list – think about what happens when an “a” is added again

- 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

- When we check `word.list` again we obtain
  > word.list
  $a
  [1] 1 5

  $text
  [1] 2

  $consists
  [1] 3

  $of
  [1] 4
Example: word list

- For list component “a” we observe what happens when the same letter is added more than ones to the list.

- Let’s fist remember how a word is added to the list.

```r
> word.list[[word]] <- c(word.list[[word]], 1)
```

- Let’s consider that the first “a” was already added to the list.

```r
> word.list

$a

[1] 1
```

- When “a” appears again, the new position is concatenated to the previous one and the value of the component “a” becomes a real vector.

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 realized with so called loops which we will learn in another lecture.

- Now, we will learn how to sort word lists alphabetically and how to sort by word frequency.
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"
  ```

Example: sort word list alphabetically

- Let's sort our `word.list` alphabetically by word

  ```r
  First, we obtain the words from `word.list` using `names()`
  > words <- names(word.list)
  ```

  ```r
  > words
  [1] "a"       "text"      "consists" "of"
  ```

  ```r
  Second, we sort the words alphabetically
  > words.sorted <- sort(words)
  ```

  ```r
  > words.sorted
  [1] "a"       "consists" "of"      "text"
  ```
Example: sort word list alphabetically

- Finally, we sort `word.list` using `words.sorted`

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

$consists
[1] 3

$of
[1] 4

$text
[1] 2
```

Example: sort word list alphabetically

- We can write all three steps in one line

```r
> word.list[sort(names(word.list))]
$s 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"
  ```

- 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.
  ```r
  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`
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
[1] 2 1 1 1
```
Example: sort word list by word frequency

- Let’s sort our word list by word frequency

- First, we determine the word frequency as shown in the previous slide

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

Next, we compute the order of the word frequency

```r
> word.freq.order <- order(word.freq)
> word.freq.order
[1] 2 3 4 1
```

The output indicates that word count of component 2 (“text”) is the smallest, word count of component 3 (“consists”) the second smallest, etc.
Example: sort word list by word frequency

- Finally, we use the obtained order of the word frequency for sorting our word list

```r
> word.list[order(word.freq.order)]
$text
[1] 2
$consists
[1] 3
$of
[1] 4
$a
[1] 1 5
```

Example: sort word list by word frequency

- 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
```
Example: sort word list by word frequency

- We can revert the sorting order by specifying the argument `decreasing = T` in the order function.

```r
> word.list[order(sapply(word.list, length), decreasing = T)]
$a
[1] 1 5
$text
[1] 2
$consists
[1] 3
$of
[1] 4
```

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.

4. 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.

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

March 2016