Introduction to Computing for Economics and Management

Summary

Short Summary 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)
### 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

- **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
Short Summary Matrices

- Creation
  
  ```r
  > 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
  ```r
  > z[c(1,2),]
  ```

- Assign new values to submatrices
  ```r
  > 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
  ```r
  > z[z[,2] >= 5,]
  ```
Short Summary 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)
  ```

Short Summary 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,]
  ```
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)
  ```

Looping

- The most frequently used looping construct is
  ```r
  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`:

```r
> 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-loops 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:

```r
> 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

```r
> 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

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

- Write our own function for computing the length of a vector

```r
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 Euclidean norm of a vector

```r
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)
}
```
Square elements of a vector

- We can change the elements of the input vector and return a new vector, e.g. square the elements of a vector

```r
# square elements of vector vec
square.vec <- function(vec)
{
    # initialize output vector vec.res
    vec.res <- vector()

    # fill vec.res with squared elements of vec
    for(x in vec) {vec.res <- c(vec.res, x^2)}

    return(vec.res)
}
```

Nested loops

- In nested loops, an inner loop is placed inside of another outer loop

```r
for(i in 1:2)
{
    for(j in 1:3)
    {
        print(paste("outer", i, "inner", j))
    }
}
```

```
[1] "outer 1 inner 1"
[1] "outer 1 inner 2"
[1] "outer 1 inner 3"
[1] "outer 2 inner 1"
[1] "outer 2 inner 2"
[1] "outer 2 inner 3"
```
While loop

- A frequently used looping construct is
  \[ \text{while(condition) \{expression\}} \]

- As long as the condition is satisfied, the expression is executed

- Example:
  \[
  \begin{align*}
  &> \text{i <- 1} \\
  &> \text{while(i<5) \{i <- i+1\}} \\
  &> \text{i} \\
  &\quad [1] 5
  \end{align*}
  \]

  In the example we observe that the while loop is executed 4 times

---

Break

- We can control when to exit the while-loop by using `break` in combination with an `if` statement

  \[
  \begin{align*}
  &\text{i <- 1} \\
  &\text{while(TRUE)} \\
  &\quad \{ \\
  &\quad \quad \text{i <- i + 1} \\
  &\quad \quad \text{if(i >= 10) \{break\}} \\
  &\quad \} \\
  &\text{i} \\
  &\quad [1] 10
  \end{align*}
  \]
Repeat loop

- Another looping construct is
  `repeat {expression}`

- Expression is executed until the loop is terminated with `break`

- In comparison to the while-loop there is no longer a condition test

- We can use it whenever we don’t have a condition to test

Example in which we use `repeat` instead of `while(TRUE)`

```r
i <- 1
repeat
{
  i <- i + 1
  if(i >= 10) {break}
}
i
[1] 10
```
Another useful statement is `next`, which skips the remainder of the current iteration of the loop and proceed directly to the next iteration.

We can use a `next` statement in while-loops, repeat-loops and for-loops as well.

```r
for(i in 1:3)
{
    print("a")
    next
    print("b")
}
```

```r
table()
```

---

**Scatter plots of data frames**

```r
> plot(person.data[,2:5])
```

---
**Histogram**

- A histogram is similar to a barplot since we visualize how many observations fall within specified divisions called “bins”

- In R we simply call the `hist` function

- `hist` creates the bins automatically, counts the number of observations that fall within the bins and plots the result

- A histogram of normal distributed random numbers with mean 0 is created with

```r
> rnorm1000 <- rnorm(1000)
> hist(rnorm1000)
```

**Histogram**

Histogram of `rnorm1000`

**Frequency**

```
0 50 100 150
```

```
-3 -2 -1 0 1 2 3
```

**morm1000**

```
0 50 100 150
```

```
-3 -2 -1 0 1 2 3
```
Figure array

- Sometimes we need to place several figures in the same plot

- For example, we would like to have the plot of the normal distributed random numbers and the corresponding histogram in one plot

- We use `par` to specify how several figures will be drawn in an number of rows by number of columns array
  - `par(mfrow=c(nr, nc))` specifies that figures will be drawn in an nr-by-nc array by rows
  - `par(mfcol=c(nr, nc))` specifies that figures will be drawn in an nr-by-nc array by columns

```
> rnorm1000 <- rnorm(1000)
> par(mfrow=c(1,2))
> plot(rnorm1000)
> hist(rnorm1000)
> par(mfrow=c(1,1))
```

- The last line resets `mfrow` to its default value