Exercise 2

Object classes II

Lists

Lists can store any type of object including other lists. Think of them as the box in the kitchen where you put everything that doesn't fit elsewhere. It's usually more convenient to use more structured boxes, but sometimes you just need a place to put things.
The only thing new is that lists can be indexed in two ways:

- [**i**] returns a list of elements. i can be an integer or a vector [**i:j**], just like when indexing vectors.
- [[**i**]] returns a single element. i can only be a single value.

Because lists can take on complicated structures, function **str()** is a convenient tool to get an overview of a list's content. Try using it on the other objects, too.

```
x <- 10
y <- 10:15
z <- c("hello", "bye")
example <- function() print("example")

l <- list(x, y, z, example)
l
```

```
## [[1]]
## [1] 10
##
## [[2]]
## [1] 10 11 12 13 14 15
##
## [[3]]
## [1] "hello" "bye"
##
## [[4]]
## function ()
## print("example")
```

```
str(l) # Summarize the structure of a list
```

```
## List of 4
##  $ : num 10
##  $ : int [1:6] 10 11 12 13 14 15
##  $ : chr [1:2] "hello" "bye"
##  $ :function ()
##   ..- attr(*, "srcref")= 'srcref' int [1:8] 4 12 4 38 12 38 4 4
##   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x0000000017324b18>
```

```
# List indexing
# [ ] returns a list
l[1:2]
```

```
## [[1]]
## [1] 10
##
## [[2]]
## [1] 10 11 12 13 14 15
```

```
l[2]
```

```
## [[1]]
## [1] 10 11 12 13 14 15
```

```
# Given that the result of list[index] is of data type list,
# the following operation produces an error
#l[1]*2
# Arguably, this is a bit confusing, since we consider the first element of
# our list to be a number; i.e., the number 10.
# However, lists can represent complicated data structures. So we have to comply
# with programming rules how to use them properly.
# To extract the actual value stored in a position of the list, we need the
# double square bracket operator [[ ]]. Hence, to perform the above calculation:
l[[1]]*2
```

```
## [1] 20
```

```
# In general: [[ ]] returns a single list element
l[[2]]
```

```
## [1] 10 11 12 13 14 15
```

Data frames

Data frames are a special type of list, although they might not look like it. They are a list of vectors (so the vectors can have different types) with the same length. Think of them as the 'excel table' of objects.
Just like an excel table, we usually assign names to the columns and sometimes rows. These names are not part of the values inside the data frame (unlike, for example, in an excel table). The names can also be used to index by name rather than by position.
Data frames are lists, but they are also two (or more)-dimensional, so we can index them like a matrix to get specific rows or columns.

```
# All columns in the data.frame have to have the same length
# But they can have different classes
x <- 5:1
y <- seq(from = 1, to = 10, by = 2)
z <- c("a", "b", "c", "d", "e")

# Create a data.frame
df <- data.frame(x, y, z)
df
```

```
##   x y z
## 1 5 1 a
## 2 4 3 b
## 3 3 5 c
## 4 2 7 d
## 5 1 9 e
```

```
df <- data.frame("column1" = x, "column2" = y, z)
df
```

```
##   column1 column2 z
## 1       5       1 a
## 2       4       3 b
## 3       3       5 c
## 4       2       7 d
## 5       1       9 e
```

```
# We can reassign names, just like we reassigned dimensions
colnames(df) <- c("variable1", "variable2", "variable3")
rownames(df) <- c("obs1", "obs2", "obs3", "obs4", "obs5")
df
```

```
##      variable1 variable2 variable3
## obs1         5         1         a
## obs2         4         3         b
## obs3         3         5         c
## obs4         2         7         d
## obs5         1         9         e
```

```
# Indexing data frames
# By position
df[1:2,]
```

```
##      variable1 variable2 variable3
## obs1         5         1         a
## obs2         4         3         b
```

```
df[, c(2,3)]
```

```
##      variable2 variable3
## obs1         1         a
## obs2         3         b
## obs3         5         c
## obs4         7         d
## obs5         9         e
```

```
df[,2]
```

```
## [1] 1 3 5 7 9
```

```
# By name...
df[c("obs4", "obs5"), c("variable1", "variable2")]
```

```
##      variable1 variable2
## obs4         2         7
## obs5         1         9
```

```
# ...with a shortcut $ for selecting one column by name
df$variable2
```

```
## [1] 1 3 5 7 9
```

```
df["variable2"]
```

```
##      variable2
## obs1         1
## obs2         3
## obs3         5
## obs4         7
## obs5         9
```

```
# Calculating the mean of variable 2
mean(df$variable2)
```

```
## [1] 5
```

Working directory

When interacting with files on your computer, you can always specify the full path to the file. If you don't, then R will search for the file at a default location called the *working directory*. You can set this to any location for convenience.
Note: RMarkdown sets the working directory to the directory of the document by default and avoids permanent changes.

```
# You can check which directory is the current working directory
getwd()
```

```
## [1] "C:/Users/AlisaK/Desktop/ex2"
```

```
# This is the location where the file was saved in the previous chunk...

# You can set the working directory to any location for convenience
#setwd(C:\\Project folder/Amazing project/Believe me)

# Aside: The directory is evaluated in the same way as in the command line.
# For example, ".." can be used to 'jump up' on level.
#setwd("..")
#getwd()
```

Saving/Loading data

Data is typically shared as a comma-separated list (.csv). Loading from and saving to this format can be done with the **write.csv()** and **read.csv()** functions. Aside from stating the file, several additional arguments can be used to correctly identify its formatting. The most important are define the separator and if the column names are given in the first row.
All R objects can also be saved as-they-are in .rds format using functions **saveRDS()** and **readRDS()**. This can be useful when saving less structured objects like lists.
Don't forget to check your data for consistency and correct formatting. Two practical functions for this are **str()**, which we know, and **summary()**.

```
# Loading data from a csv file
loans <- read.csv("loan_data.csv", sep = ";", header = TRUE)
# Check for consistency
str(loans)
```

```
## 'data.frame':    1225 obs. of  15 variables:
##  $ YOB    : num  19 41 66 51 65 42 59 43 52 65 ...
##  $ nKIDS  : num  4 2 0 2 0 2 0 1 0 0 ...
##  $ nDEP   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ PHON   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dINC_SP: num  0 0 0 0 0 10500 6500 13500 0 0 ...
##  $ EMPS_A : Factor w/ 11 levels "B","E","M","N",..: 6 5 4 5 5 2 1 2 2 5 ...
##  $ dINC_A : num  0 36000 30000 464 15000 48000 30000 9000 22500 19500 ...
##  $ RES    : Factor w/ 5 levels "F","N","O","P",..: 3 3 2 3 4 3 3 3 4 3 ...
##  $ dHVAL  : num  14464 0 0 24928 0 ...
##  $ dMBO   : num  4 0 0 8464 0 ...
##  $ dOUTM  : num  0 280 0 584 0 1120 520 0 0 540 ...
##  $ dOUTL  : num  0 664 0 320 0 0 0 200 200 0 ...
##  $ dOUTHP : num  0 0 0 0 0 0 96 0 0 0 ...
##  $ dOUTCC : num  0 80 0 60 0 0 0 0 80 0 ...
##  $ BAD    : num  0 0 0 0 0 0 0 0 0 0 ...
```

```
# Saving the data in .rds format.
saveRDS(loans, "loan_data.rds")
```

Code flow control

Humans tend to be lazy and make mistakes. When we want to do something more than one time or to several different objects, we therefore want to keep the code as simple as possible. Fortunately, there are several ways to control the flow of our code.

If/Else

Function **if()** checks if a condition is TRUE. If it is, the specified code is run. If not, either nothing happens or some other code, specified with **else**, is run.
if(condition){ operation if the condition is valid }else{ alternative operation}

Note that the curly brackets **{ }** are not part of the function, but are a convenient way to group statements. Here, we use them to tell R that we want it to run several lines of code.

```
a <- 2
if(a == 2) print("2. Great number.")
```

```
## [1] "2. Great number."
```

```
# Use an if-condition to check the size of a
if(a < 10){
  print("Value smaller than 10")
}else{
 print("Value larger than 10")
}
```

```
## [1] "Value smaller than 10"
```

Loops

We have seen how we can define several calls into a function and can run code if a condition is fulfilled with **if()**. The last missing tool, the *loop*, will help us to do one thing repeatedly, possibly with changing arguments.
For example, in the data that we loaded there are several numeric variables that we standardize for statistical reasons. We can do this by hand for each variable with our custom function **standardize()**, but will need to copy/paste and make typos. Plus, our mothers have raised us to be tidy with our code!
So we will automate the code to go over the categorical variables in turn and standardize them when appropriate. We cannot standardize character or factor variables, because they are not numbers. For the purpose of the exercise, we ignore that there are binary 0/1 variables in the data.

```
# Create a function standardize (see homework 1)
standardize <- function(x){
    # The actual calculations necessary to standardize the values
    mu <- mean(x)
    std <- sd(x)
    result <- (x - mu)/std
    # Return ends the function and outputs the result
    return(result)

    # A list can be returned to output more than one result
    #return(list("mean" = mu, "sd" = std, "output" = result))
}
```

```
# Create a vector with the names of the numeric variables in the data set loan_data.csv (see)
numericVars <- c("YOB", "nKIDS", "nDEP", "dINC_SP", "dINC_A", "dHVAL", "dMBO", "dOUTM", "dOUTL", "dOUTH

# Let's try looping:
# Think: For i taking on each value in the vector do ...
for(name in numericVars) print(name)
```

```
## [1] "YOB"
## [1] "nKIDS"
## [1] "nDEP"
## [1] "dINC_SP"
## [1] "dINC_A"
## [1] "dHVAL"
## [1] "dMBO"
## [1] "dOUTM"
## [1] "dOUTL"
## [1] "dOUTHP"
## [1] "dOUTCC"
```

```
# Let's try it with the data
for(name in numericVars) standardize(loans[, name])
mean(loans$dINC_SP) # This should now be zero! Nothing happened!
```

```
## [1] 1990.085
```

```
# We need to *save* the results during each iteration!
for(name in numericVars){
  loans[, name] <- standardize(loans[, name])
}
# Check the result
mean(loans$dINC_SP)
```

```
## [1] -2.793834e-17
```

```
sd(loans$dINC_SP)
```

```
## [1] 1
```

The apply family

To loop over the numeric variables, we looked at the structure of the data frame and made a list manually. That is another step that we could automate to have more break time. R has simplifying, optimized functions to loop over the values of an object and return the results in a structured form called the *apply* family. They are used to apply one function to several elements (rows/columns for apply, vectors/lists for the others). Let's see if we can standardize all of the numeric variables in the data using the custom function that you build in the homework exercises.
In the first step, we want to check for every variable if it is numeric. To do so, we'd like to apply function *is.numeric()* to every column in the data frame. Then, we will apply function standardize to these variables.

```
# Since our standardize function can't handle factor variables, we
# have to select all numeric variables first
num_vars_list <- lapply(loans, is.numeric) # Lists cannot be used to index, so we'd like a vector in re
class(num_vars_list)
```

```
## [1] "list"
```

```
num_vars <- sapply(loans, is.numeric) # sapply simplifies the result list, if possible
num_vars
```

```
##      YOB   nKIDS    nDEP    PHON dINC_SP  EMPS_A  dINC_A     RES   dHVAL
##     TRUE    TRUE    TRUE    TRUE    TRUE   FALSE    TRUE   FALSE    TRUE
##     dMBO   dOUTM   dOUTL  dOUTHP  dOUTCC     BAD
##     TRUE    TRUE    TRUE    TRUE    TRUE    TRUE
```

```
# Let's test this first
test <- sapply(loans[,num_vars], FUN = standardize)
str(test)
```

```
##  num [1:1225, 1:13] -2.09371 -0.65597 0.97782 -0.00245 0.91247 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:13] "YOB" "nKIDS" "nDEP" "PHON" ...
```

```
# And use it to update the correct columns in our data frame
# The second argument in apply specifies a loop over the rows (1) or columns (2)
loans[,num_vars] <- apply(loans[,num_vars], 2, FUN = standardize)

# Let's look at the 'final' data set
summary(loans)
```

```
##       YOB               nKIDS               nDEP               PHON
##  Min.   :-3.1393   Min.   :-0.6138   Min.   :-0.1745   Min.   :-3.0616
##  1st Qu.:-0.5906   1st Qu.:-0.6138   1st Qu.:-0.1745   1st Qu.: 0.3264
##  Median : 0.2590   Median :-0.6138   Median :-0.1745   Median : 0.3264
##  Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000
##  3rd Qu.: 0.7818   3rd Qu.: 0.3704   3rd Qu.:-0.1745   3rd Qu.: 0.3264
##  Max.   : 3.1344   Max.   : 4.3074   Max.   : 8.9199   Max.   : 0.3264
##
##     dINC_SP            EMPS_A        dINC_A          RES
##  Min.   :-0.4144   P      :531   Min.   :-1.3364   F:129
##  1st Qu.:-0.4144   V      :231   1st Qu.:-0.7703   N: 66
##  Median :-0.4144   E      :124   Median :-0.1097   O:624
##  Mean   : 0.0000   T      :123   Mean   : 0.0000   P:252
##  3rd Qu.:-0.1978   R      :104   3rd Qu.: 0.5886   U:154
##  Max.   : 9.9972   W      : 37   Max.   : 2.7400
##                    (Other): 75
##      dHVAL              dMBO              dOUTM              dOUTL
##  Min.   :-0.7568   Min.   :-0.5943   Min.   :-0.7991   Min.   :-0.1452
##  1st Qu.:-0.7568   1st Qu.:-0.5943   1st Qu.:-0.7991   1st Qu.:-0.1452
##  Median :-0.7568   Median :-0.5943   Median :-0.2009   Median :-0.1452
##  Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000
##  3rd Qu.: 0.6382   3rd Qu.: 0.4645   3rd Qu.: 0.4346   3rd Qu.:-0.1452
##  Max.   : 2.3743   Max.   : 2.7939   Max.   : 8.0795   Max.   :33.2024
##
##      dOUTHP             dOUTCC               BAD
##  Min.   :-0.2407   Min.   :-0.2347   Min.   :-0.5982
##  1st Qu.:-0.2407   1st Qu.:-0.2347   1st Qu.:-0.5982
##  Median :-0.2407   Median :-0.2347   Median :-0.5982
##  Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000
##  3rd Qu.:-0.2407   3rd Qu.:-0.2347   3rd Qu.: 1.6704
##  Max.   :13.1682   Max.   :16.3631   Max.   : 1.6704
##
```