2. Data Structures

2.1 Variables

Variables are essential in programming because they help store and manage data more efficiently.

- For example, using variables allows to *reuse* stored values and results multiple times.
- They increase the *readability* of the code when you use meaningful names for your variables.
- The code is also easier to *maintain* if you use variables, because if you need to change the value of a variable you just have to do it where it is declared on not everywhere it is used.

Overall, variables make code more efficient, organized, and scalable.

In R, variables are defined using by assigning a value to a name using the assignment operator `<-`. Your code is much easier to understand if you name your variables after what they contain (e.g. *names* for a collection of names).

In the following example, I do not have a very meaningful variable which is why I just call it `a`.

```r
a <- 44*467
```

Observe that executing these commands does not show the output but stores it as object `a`.

If I want to display `a` i have to write `a` after defining `a`, e.g.

```r
a # What is a?
```

```
## [1] 20548
```

We can also get more details on `a` using the `str` (structure) function. If there is any named object in the working environment, then

```
str(name_of_object)
```

gives information about it, e.g.

```r
str(a)
```

```
##  num 20548
```

Be careful, if you assign a new number or element to `a` the old `a` is replaced completely:

```r
a <- 44/467
a
```

```
## [1] 0.09421842
```

```r
str(a)
```

```
##  num 0.0942
```

```r
a <- NULL
a
```

```
## NULL
```

```r
str(a)
```

```
##  NULL
```

2.2.1 Side Note on Data types

The object `a` above contains a numerical (`num`) value. Other possible contents are:

- characters/strings (`chr`)

- logicals (logi)

```
b <- "Monday"
b
```

```
## [1] "Monday"
```

```
str(b)
```

```
##  chr "Monday"
```

```
c <- FALSE
c
```

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

```
str(c)
```

```
##  logi FALSE
```

```
d <- TRUE
d
```

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

```
str(d)
```

```
##  logi TRUE
```

2.3 More Complex Variables and Data Structures

Instead of simple objects as a, b, c above, variables can also contain more complex objects.

2.3.1 Vectors (one-dimensional arrays):

A vector is an object containing an (almost) arbitrary number of entries of the same type.

There are numerous possibilities on how to generate a vector in R.

(a) **Consecutive integers**

A Vector with consecutive integers from a lower bound a to an upper bound b can be defined using the notation $a : b$:

```
v <- 5:10
v
```

```
## [1]  5  6  7  8  9 10
```

```
w <- 17:3
w
```

```
##  [1] 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3
```

Warning: Pay attention to parentheses! Better use more parentheses than absolutely necessary…

```
- 1:10
```

```
##  [1] -1  0  1  2  3  4  5  6  7  8  9 10
```

```
-(1:10)
```

```
##  [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

```
2 * (1:5)
```

```
## [1]  2  4  6  8 10
```

```r
2 * 1:5
```

```
## [1]  2  4  6  8 10
```

```r
2 + 1:5
```

```
## [1] 3 4 5 6 7
```

```r
(2 + 1):5
```

```
## [1] 3 4 5
```

(b) Function c (combine)

A vector with specific entries $e_1, e_2, ..., e_n$ is generated with $c(e_1, e_2, ..., e_n)$, e.g.

```r
a <- c(1, 5, 11.7)
a
```

```
## [1]  1.0  5.0 11.7
```

```r
dow <- c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
dow
```

```
## [1] "Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"
```

(c) Function rep (repeat)

The function `rep` is mostly used if you what to fill n positions of a vector with the same entry e (`rep(e,n)`).

```r
a <- rep(1.999, 7)
a
```

```
## [1] 1.999 1.999 1.999 1.999 1.999 1.999 1.999
```

However there are different uses of `rep`.

```r
b <- 1:4
b
```

```
## [1] 1 2 3 4
```

```r
rep(b,times=3)
```

```
##  [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

```r
rep(b,each=3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

```r
rep(b,times=c(7,5,3,1))
```

```
##  [1] 1 1 1 1 1 1 1 2 2 2 2 2 3 3 3 4
```

And you could also combine `c()` and `rep()`, e.g.

```r
dow.is.wknd <- c(rep(FALSE,5),rep(TRUE,2))
dow.is.wknd
```

```
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

(d) Function seq (sequence)

The function `seq()` generates sequences of numbers, with customizable starting and ending points, step sizes, and lengths.

If we want a sequence of elements from a to b with step size d, we write `seq(from=a,to=b,by=d)`:

```r
seq(from=0,to=1,by=0.02)
```

```
##  [1] 0.00 0.02 0.04 0.06 0.08 0.10 0.12 0.14 0.16 0.18 0.20 0.22 0.24 0.26 0.28
## [16] 0.30 0.32 0.34 0.36 0.38 0.40 0.42 0.44 0.46 0.48 0.50 0.52 0.54 0.56 0.58
## [31] 0.60 0.62 0.64 0.66 0.68 0.70 0.72 0.74 0.76 0.78 0.80 0.82 0.84 0.86 0.88
## [46] 0.90 0.92 0.94 0.96 0.98 1.00
```

If we want a sequence of n elements from a to b, we write `seq(from=a,to=b,length.out=n)`:

```r
seq(0,1,length.out=21)
```

```
##  [1] 0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70
## [16] 0.75 0.80 0.85 0.90 0.95 1.00
```

Applying relations, operators or functions to vectors

If x, y are vectors with entries of a certain type, applying an operator, operation or function to them is done component-wise:

```r
x <- 0:4
x
```

```
## [1] 0 1 2 3 4
```

```r
x + 1
```

```
## [1] 1 2 3 4 5
```

```r
x^2
```

```
## [1]  0  1  4  9 16
```

```r
exp(x)  # exp(.) : exponential function
```

```
## [1]  1.000000  2.718282  7.389056 20.085537 54.598150
```

```r
log(x)  # log(.) : natural logarithm
```

```
## [1]      -Inf 0.0000000 0.6931472 1.0986123 1.3862944
```

Exercise:

Generate the following vectors with a few lines of code:

- v1 contains 1, 2, 4, 8, ..., 2048
- v2 contains 10, 20, 20, 30, 30, 30, 40, 40, 40, 40, ..., 1000, ..., 1000
- v3 contains 1, -1, 1, -1, 1, -1, ... (50 components)
- v4 contains 1, 3, 5, 7, ..., 99
- v5 contains 0, 0.15, 0.3, 0.45, ..., 3.

Subsetting vectors

Subsetting is crucial when using vectors because it allows for efficient extraction, modification, and analysis of specific elements within a vector. To subset from a vector, we use the square brackets `[]` and you specify the index or condition to extract the desired elements.

```r
x <- 10*(1:7)
x[1]          # 1st entry
```

```
## [1] 10
```

```r
x[2]          # 2nd entry
```

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

```r
x[2:7]        # entries 2 to 7
```

```
## [1] 20 30 40 50 60 70
```

```r
x[c(1,3,5)]   # entries 1, 3 and 5
```

```
## [1] 10 30 50
```

```r
x[-3]          # all but the 3rd entry
```

```
## [1] 10 20 40 50 60 70
```

```r
x[-(3:4)]      # all but the 3rd and 4th entry
```

```
## [1] 10 20 50 60 70
```

```r
x[-c(1,3,5)]  # all but the 1st, 3rd and 5th entry
```

```
## [1] 20 40 60 70
```

Often, one generates subvectors by means of logical conditions

```r
x[x < 30] # all entries less than 30
```

```
## [1] 10 20
```

```r
x[x >= 50 & x <= 100] # all entries between 50 AND 100
```

```
## [1] 50 60 70
```

```r
x[x < 30 | x > 50] # all entires that are either < 30 OR > 50
```

```
## [1] 10 20 60 70
```

Exercise:

Generate a vector with components 1, 2, 1, 3, 1, 4, 1, 5, ..., 1, 100.

2.3.2 Matrices (two-dimensional arrays):

A matrix is a table with an (almost) arbitrary number of rows and columns, each entry being of the same type.

There are numerous possibilities on how to create matrices in R.

(a) **cbind (column-wise binding)**

The `cbind()` function combines vectors or matrices by binding them column-wise to form a new matrix.

```
x <- seq(0,1,0.1)
y <- sin(2*pi*x)
z <- cbind(x,y)
z
```

```
##         x             y
##  [1,] 0.0  0.000000e+00
##  [2,] 0.1  5.877853e-01
##  [3,] 0.2  9.510565e-01
##  [4,] 0.3  9.510565e-01
##  [5,] 0.4  5.877853e-01
##  [6,] 0.5  1.224606e-16
##  [7,] 0.6 -5.877853e-01
##  [8,] 0.7 -9.510565e-01
##  [9,] 0.8 -9.510565e-01
## [10,] 0.9 -5.877853e-01
## [11,] 1.0 -2.449213e-16
```

This is a matrix (two-dimensional array) with numerical entries in 11 rows and 2 columns. Indeed,

```
str(z)
```

```
##  num [1:11, 1:2] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "x" "y"
```

The matrix z has an attribute "dimnames". This is a list of two character vectors, the row names (not specified) and the column names. Alternatively, one can use `dim()` to get the information on the dimension of our matrix.

```
dim(z)
```

```
## [1] 11  2
```

One could also specify the column names:

```
z <- cbind('x'=x,'sin(2*pi*x)'=y)
z
```

```
##         x   sin(2*pi*x)
##  [1,] 0.0  0.000000e+00
##  [2,] 0.1  5.877853e-01
##  [3,] 0.2  9.510565e-01
##  [4,] 0.3  9.510565e-01
##  [5,] 0.4  5.877853e-01
##  [6,] 0.5  1.224606e-16
##  [7,] 0.6 -5.877853e-01
##  [8,] 0.7 -9.510565e-01
##  [9,] 0.8 -9.510565e-01
## [10,] 0.9 -5.877853e-01
## [11,] 1.0 -2.449213e-16
```

And also glue another vector to our previously created matrix:

```
z <- cbind(z,'cos(2*pi*x)'=cos(x))
str(z)
```

```
##  num [1:11, 1:3] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:3] "x" "sin(2*pi*x)" "cos(2*pi*x)"
```

(b) rbind (row-wise binding)

The `rbind()` function combines vectors or matrices by binding them row-wise, creating a new matrix.

```
x <- seq(-2,2,0.5)
y <- exp(x)
z <- rbind('x'=x,'exp(x)'=y)
z
```

```
##              [,1]       [,2]       [,3]       [,4] [,5]     [,6]     [,7]
## x      -2.0000000 -1.5000000 -1.0000000 -0.5000000    0 0.500000 1.000000
## exp(x)  0.1353353  0.2231302  0.3678794  0.6065307    1 1.648721 2.718282
##            [,8]     [,9]
## x      1.500000 2.000000
## exp(x) 4.481689 7.389056
```

(c) matrix

The `matrix(v,nrow=m,ncol=n)` function creates generates a matrix with m rows and n columns whose components are filled column-wise with the components of v.

```
matrix(1:12,3,4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
matrix(1:3,3,4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    2    2    2    2
## [3,]    3    3    3    3
```

```
matrix(0,3,4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
```

```
matrix(NA,3,4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA
```

(d) array

The `array()` function creates multi-dimensional arrays by arranging data into specified dimensions (such as rows, columns, and additional layers).

```
array(1:12,dim=c(3,4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
array(1:3,dim=c(3,4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    2    2    2    2
## [3,]    3    3    3    3
```

In principle, there are three- (or higher-) dimensional arrays:

```
A <- array(1:24,dim=c(2,3,4))
A[,,1]
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
A[,,2]
```

```
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

```
A[1,,]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    3    9   15   21
## [3,]    5   11   17   23
```

Subsetting matrices

As for vectors substetting is also for matrices helpful.

```
z[2,7]    # entry in row 2 and column 7
```

```
##   exp(x)
## 2.718282
```

```
z[1,]     # row 1
```

```
## [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
```

```
z[,2]     # column 2
```

```
##          x     exp(x)
## -1.5000000  0.2231302
```

```
z[1,3:5]  # rows 1 and and column 3:5
```

```
## [1] -1.0 -0.5  0.0
```

Exercise:

Specify a minimal integer $n_{min} \geq 0$ and a maximal integer $n_{max} > n_{min}$. Then generate a matrix with three columns:

- `n`: containing the integers $n_{min}, \ldots, n_{max}$
- `n2`: containing the squares of these integers $n_{min}^2, \ldots, n_{max}^2$
- `n3`: containing the cubes of these integers $n_{min}^3, \ldots, n_{max}^3$

Four lines of code, including the specification of n_{min}, n_{max} should suffice.

2.3.3 Data frames:

A data frame is a table where each column n is a vector of arbitrary type (e.g., numeric, character, factor), aka "variable" or "feature", and each row corresponds to an "observation" or "case". Data frames can also be viewed as special lists (see below).

Theoretically, one can create a data frame by hand. However, you will typically encounter data frames in R when you work with data sets. Thus, let us read a data set to study this data structure:

```
ds <- read.table("DataSets/Trees.txt", header=TRUE)
head(ds) # display the first 6 rows
```

```
##   diam height volume
## 1  8.3     70   10.3
## 2  8.6     65   10.3
## 3  8.8     63   10.2
## 4 10.5     72   16.4
## 5 10.7     81   18.8
## 6 10.8     83   19.7
```

```
str(ds)
```

```
## 'data.frame':    31 obs. of  3 variables:
##  $ diam  : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ height: int  70 65 63 72 81 83 66 75 80 75 ...
##  $ volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

(a) Subsetting dataframes

Subsetting data frames is analoguous to subsetting matrices, i.e.

```
ds[3,] # 3rd row
```

```
##   diam height volume
## 3  8.8     63   10.2
```

```
ds[,2] # 2nd column
```

```
##  [1] 70 65 63 72 81 83 66 75 80 75 79 76 76 69 75 74 85 86 71 64 78 80 74 72 77
## [26] 81 82 80 80 80 87
```

```
ds[1:5,c(1,3)] # rows 1 to 5 of columns 1 and 3
```

```
##   diam volume
## 1  8.3   10.3
## 2  8.6   10.3
## 3  8.8   10.2
## 4 10.5   16.4
## 5 10.7   18.8
```

Subsetting like done above, however, does not make your code easily understandable. Thus, when working with data frames one often uses the colum names rather than the column index for the selection:

```r
ds$height # column height
```

```
##  [1] 70 65 63 72 81 83 66 75 80 75 79 76 76 69 75 74 85 86 71 64 78 80 74 72 77
## [26] 81 82 80 80 80 87
```

```r
ds[2:6,]$height # rows 2:6 of columns height
```

```
## [1] 65 63 72 81 83
```

```r
ds[1:5,c("diam","volume")] # rows 1 to 5 of columns diam and volume
```

```
##   diam volume
## 1  8.3   10.3
## 2  8.6   10.3
## 3  8.8   10.2
## 4 10.5   16.4
## 5 10.7   18.8
```

(b) cbind (column-wise binding)

Sometimes during the analysis we would like to add columns (or rows) to our data frame. This can be achieved by using the previously introduced function `cbind()` (or `rowbind()`). For example, if I would like to add the area of the cross-section to the data frame, I could do it as follows.

```r
area <- (ds$diam/2)^2*pi
ds <- cbind(ds, area) # I need to replace the old data frame with the new one
head(ds)
```

```
##   diam height volume     area
## 1  8.3     70   10.3 54.10608
## 2  8.6     65   10.3 58.08805
## 3  8.8     63   10.2 60.82123
## 4 10.5     72   16.4 86.59015
## 5 10.7     81   18.8 89.92024
## 6 10.8     83   19.7 91.60884
```

Another option to accomplish this is

```r
ds$area2 <- (ds$diam/2)^2*pi
head(ds)
```

```
##   diam height volume     area    area2
## 1  8.3     70   10.3 54.10608 54.10608
## 2  8.6     65   10.3 58.08805 58.08805
## 3  8.8     63   10.2 60.82123 60.82123
## 4 10.5     72   16.4 86.59015 86.59015
## 5 10.7     81   18.8 89.92024 89.92024
## 6 10.8     83   19.7 91.60884 91.60884
```

2.3.4 Lists:

This is the most general data structure. It is an array of an (almost) arbitrary number of items/objects, where each item can be of arbitrary type itself. Arbitrary objects can be combined to lists:

```
Someone <- list(given.name='Dagobert',
                surname='Duck',
                age=57,
                residence='Duckburg',
                age.of.children=NA)
Someone
```

```
## $given.name
## [1] "Dagobert"
##
## $surname
## [1] "Duck"
##
## $age
## [1] 57
##
## $residence
## [1] "Duckburg"
##
## $age.of.children
## [1] NA
```

```
str(Someone)
```

```
## List of 5
##  $ given.name     : chr "Dagobert"
##  $ surname        : chr "Duck"
##  $ age            : num 57
##  $ residence      : chr "Duckburg"
##  $ age.of.children: logi NA
```

```
Someone.else <- list(given.name='Donald',
                     surname='Duck',ge=42,
                     residence='Duckburg',
                     age.of.children=c(7,9,11))
Someone.else
```

```
## $given.name
## [1] "Donald"
##
## $surname
## [1] "Duck"
##
## $ge
## [1] 42
##
## $residence
## [1] "Duckburg"
##
## $age.of.children
## [1]  7  9 11
```

```
str(Someone.else)
```

```
## List of 5
##  $ given.name     : chr "Donald"
##  $ surname        : chr "Duck"
##  $ ge             : num 42
##  $ residence      : chr "Duckburg"
##  $ age.of.children: num [1:3] 7 9 11
```

Access to items of a list:

```
Someone[2]
```

```
## $surname
## [1] "Duck"
```

```
Someone[[2]]
```

```
## [1] "Duck"
```

```
Someone$residence
```

```
## [1] "Duckburg"
```

```
Someone[[5]]
```

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

```
Someone.else[[5]]
```

```
## [1]  7  9 11
```

```
Someone$profession
```

```
## NULL
```

One can also start with an empty list, specifying only the number or names of items and fill it in later:

```
Me <- list(given.name=NULL,
           surname=NULL,age=NULL,residence=NULL,nationality=NULL)
str(Me)
```

```
## List of 5
##  $ given.name : NULL
##  $ surname    : NULL
##  $ age        : NULL
##  $ residence  : NULL
##  $ nationality: NULL
```

```
# Now specify the items in this list with your data:

Me$given.name <- "Anja"
Me$surname <- 'Muehlemann'
Me[3] <- 91
Me$residence <- 'Bern'
Me$nationality <- 'swiss'
Me
```

```
## $given.name
## [1] "Anja"
##
## $surname
## [1] "Muehlemann"
##
## $age
## [1] 91
```

```
##
## $residence
## [1] "Bern"
##
## $nationality
## [1] "swiss"
```

One could also start with an empty list:

```
You <- list()
You$given.name <- "Emily"
You
```

```
## $given.name
## [1] "Emily"
```