Lecture 3

Introduction to R:
Managing and Understanding Data

R History

- The S language has been developed in the late 1970s by John Chambers and his colleagues at Bell Labs as a language for programming with data.
- The language combines ideas from other languages, such as awk, Scheme, APL, etc. and it provides an environment for quantitative computations and visualization.
- S-Plus is a commercialization of the Bell Labs framework. “Plus” stands for “S Plus graphics”.
- R is an implementation of the S language combined with lexical scoping semantics inspired by Scheme. R was initially developed by Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand in the mid1990s.
- R is currently developed by the R Development Core Team, of which Chambers is a member.

Comparing R with other software

A growing number of libraries, currently more than 6,000, is the most noticeable feature of R, compared to other commercial software such as SAS, Stata, SPSS, and open source software such as Python and Octave. This feature enables R to have a huge number of tools for data management and statistical analysis.

R has no competitor that gets the most up-to-date packages for analysis in many areas such as machine learning, finance, mathematics, data mining, or even astronomy.

R has a more intuitive syntax structure than the previously mentioned software. Its object-oriented features make it more flexible than SPSS, Stata, SAS, and Octave. Python shares the object-oriented features too, but it is less flexible than R. Although it is easy to write a function in Python and Octave, writing functions in R is even easier.

R has one of the best graphics systems among all existing software.

Brief notes

- The R programming language is interpreted, i.e., it works through a command-line interpreter. The R command window is known as the R console, where commands are entered and results are produced.
- The R programming language is case sensitive, i.e., `Rose` and `rose` are different identifiers (names).
- Comments are preceded by a # (hash). Here is an example:
  ```
  # This is a comment line
  ```
- R does not have multiline or block comments. You must start each line of a multiline comment with #. For debugging purposes, you can surround code that you want the interpreter to ignore with a statement:

  if(FALSE) {…}

Installing R

R can be obtained by downloading it from the CRAN website (http://www.r-project.org/). The **Comprehensive R Archive Network** (**CRAN**) is a network of FTP and web servers around the world that store identical, up-to-date versions of code and documentation for R. You can select anyone of the CRAN sites to download the R software that is compatible with your computer platform, such as Windows, Mac, and Linux.

Getting help in R

R includes a help system to help you get different kinds of information. To get help on a function, for example ***c***, you would type:

```
> help(c)
```

or, equivalently:

```
>?c
```

If you'd like to try the examples in a help file, you can use the example function to automatically try them. For example, to see an example for ***c*** type:

```
> example(c)
```

You can search for help on a topic, for example "regression," using the help.search function:

```
> help.search("regression")
```

This can be very helpful if you can't remember the name of a function; R will return a list of relevant topics. There is a shorthand for this command as well:

```
> ??regression
```

Getting help in R *(cont.)*

You can also use the help option for the library command to get more complete information. For example, to get help on the *tm* library, you would use the following function:

```
> library(help="tm")
```

Some packages include at least one vignette. A *vignette* is a short document that describes how to use the package, complete with examples. You can view a vignette using the vignette command. For example, to view the vignette for the *tm* package (assuming that you have installed this package), you would use the following command:

```
> vignette("tm")
```

R packages

R comes with a number of default packages, a collection of previously programmed functions for specific tasks, and with datasets. There are two types of R packages:

- Default packages that come with the R executable
- Add-on packages that do not come during installation; we need to install them manually.

When you open the R console, it automatically loads its default packages with the associated functions, and you do not need to manually load those packages. A list of loaded packages can be obtained by typing *library()* in the R console. However, some packages need to load to execute functions. To load a specific package, the corresponding R command is *library(package)*, where package is the name of any package, such as ***tm***, provided that the package has already been installed on your computer.

If a package is not installed, you need to install it before you can load it into memory. The easiest way to install a package is via the install.packages() function. To install the ***tm*** package, at the R command prompt, simply type:

```
> install.packages("tm")
```

R packages

You will be prompted to select a CRAN mirror site from which to download the package. The same installation can be done manually from the main menu by selecting *Packages* > *Install package(s).* Then, you will get a long list of packages available for download.

You can get a list of all installed packages:

```
> installed.packages()
```

Once a package has been installed, you can load it into memory:

```
> library(tm)
```

To unload an R package, use the *detach()* function:

```
> detach("package:tm", unload = TRUE)
```

Basic examples of using the R console

When you enter an expression into the R console and press Enter, the R interpreter evaluates the expression and displays the result:

```
> "Hello world!"        # a character string evaluates to itself
[1] "Hello world!"
> 2/3 + 4*5             # arithmetic expression
[1] 20.66667
> log(2)                # log base e, the Euler's constant
[1] 0.6931472
> exp(10)               # exponent base e, the Euler's constant
[1] 22026.47
> log10(10)             # log base 10
[1] 1
> log(15, base=15)      # log base 15
[1] 1
> sqrt(10)              #square root
[1] 3.162278
> options(digits=16)    #see more digits
> sqrt(10)
[1] 3.162277660168379
```

Basic examples of using the R console *(cont.)*

Sometimes, an R command does not fit on a single line. If you enter an incomplete command on one line, the R prompt will change to a plus sign, indicating a continuation from the previous line. Example:

```
> 2*3*4*
+ 5*6*7*
+ 8*9
[1] 362880
```

The symbol “[1]” that accompanies each return value means that the index of the first item displayed is 1. By default, R assumes that a vector (an ordered collection of elements) is going to be displayed. Even simple scalars, such as integers and doubles, are considered vectors of length 1. When the vector is long, displaying it can take many rows. To make the output more readable, each row starts with the index of the first element shown in the row. For example, the sequence operator 1:50 produces a sequence with all integers between 1 and 50:

```
> 1:50
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
[14] 14 15 16 17 18 19 20 21 22 23 24 25 26
[27] 27 28 29 30 31 32 33 34 35 36 37 38 39
[40] 40 41 42 43 44 45 46 47 48 49 50
```

Basic examples of using the R console *(cont.)*

As you can see, each new row starts with the index of the first element in the row: 1, 14, 27 and 40.

Even when there is only one element to be displayed and there is only one row, the row still starts with “[1]”.

You can scroll through commands by pressing the up or down arrow. The up arrow lets you look at previous commands, whereas down arrow lets you look at later commands. The function `history()` returns a list of previously executed commands.

R also provides an automatic completions for function names. Type the Tab key to see a list of possible completions for a function.

One of the most popular free IDEs for R is RStudio available at www.rstudio.org

Assignment

If an expression is entered, the result of the expression evaluation is printed. However, the result is lost and cannot be reused in further computations. For example:

```
> 1+2
> [1] 3      # prints the result, which cannot be reused
```

Like most other languages, R lets you assign values to variables and refer to them by a name. To assign a value to a variable, the assignment operator must be used:

```
> x <- 1 + 2  # assigns 3 to x in order to save and reuse
              # its value
> x           # evaluates x
[1] 3         # prints the value of x
```

The assignment operator usually reads as " x gets 3". Note that the assignment operator ('<-') consists of the two characters '<' ("less than") and '-' ("minus") occurring strictly side-by-side and it 'points' to the object receiving the value of the expression. The assignment operator assigns the object on the right to the name on the left.

A function in R is just another object that can be assigned to a name:

```
> f <- function(x){x+1}
> f(1)
[1] 2
```

Assignment *(cont.)*

Assignments can also be made in the other direction:

```
> 1 + 2/5 -> y
> y
[1] 1.4
```

In most contexts, the '=' operator can be used as an assignment. However, it has some restrictions.

Do not confuse the = operator with the relational == operator. For example:

```
> x <- 1
> y <- 2
> x = y     # this changes the value of x from 1 to 2
> x
[1] 2
```

On the other hand:

```
> x <- 1
> y <- 2
> x == y    # this is a logical comparison
[1] FALSE
```

Assignment *(cont.)*

After you assign a value to a variable, the R interpreter will substitute that value in place of the variable name when it evaluates an expression with that variable name, for example:

```
> x <- 1
> y <- 2
> z <- c(x,y)       # combines the values of x and y
> z
[1] 1 2
```

It is important to remember that the substitution is done at the time the values of x and y are assigned to z, not at the time z is used. Here is an example when the values of x and y x change before z is used for the first time:

```
> x <- 1
> y <- 2
> z <- c(x,y)       # z is a vector of 1 and 2
> x <- 5            # x and y change to 5 and 6, respectively
> y <- 6
> z                 # the change of x and y does not affect z
[1] 1 2
```

Assignment *(cont.)*

Assignment can also be made using the function assign():

```
> assign("x", 2*3+5)
> x
[1] 11
```

Note that x is enclosed in double quotation marks.
Sometimes, <- is called left assignment and -> is called right assignment.

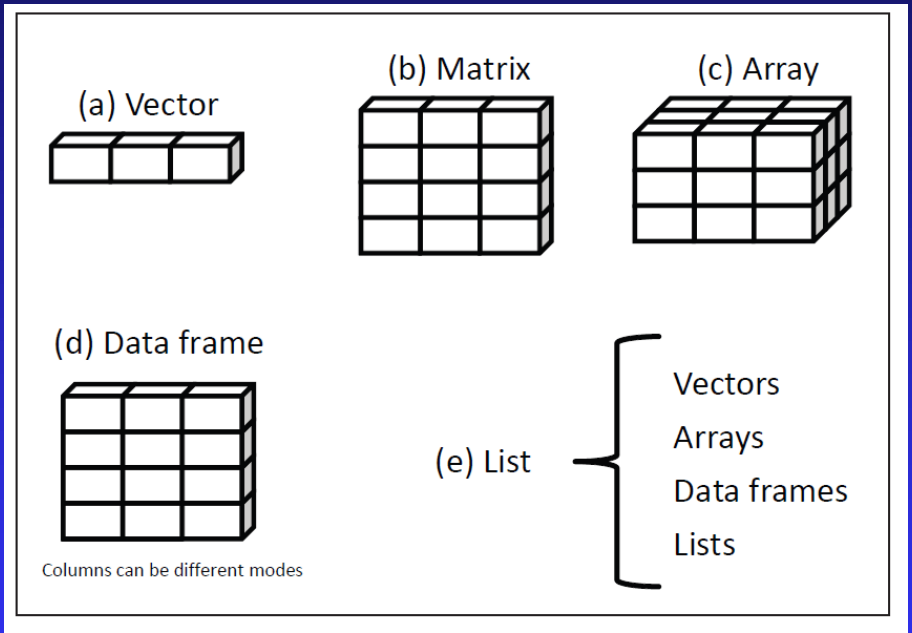
There could be a potential ambiguity if you get the spacing wrong. In the following example, we have a logical comparison, not an assignment:

```
> x < -5
[1] FALSE
```

Introduction to R data structures

Introduction to R data structures

R data structures



Source: Kabacoff: R in Action

Vectors

In R the “base" type is a vector, not a scalar. A vector is an indexed set of elements that are all of the same type. R has six vector types: logical, integer, real, complex, character, and raw.

Vectors can be thought of as contiguous cells containing data.

```
> age <- c(10,20,30,40)
> age
[1] 10 20 30 40
```

This is an example of a vector. The age of four individuals is stored in the age vector. Here, *c()* is a function used to combine four numeric values into a single vector. If you want to see the third element of the vector, you need to type:

```
> age[3]
[1] 30
```

Unlike C-based languages, the lowest index is 1 not 0! In other words, age[1] refers to the first element of the vector age.

Individual cells or vector elements are accessed through indexing operations such as age[3].

Vectors *(cont.)*

One can select a slice of a vector. For example, age[2:4] selects all elements of vector age with indexes from 2 to 4:

```
> age[2:4]
[1] 20 30 40
```

Age[-2] selects all element of age excluding the second element:

```
> age[-2]
[1] 10 30 40
```

Scalars are considered vectors of length 1. In other words, single numbers, such as 5.5, and strings, such as “hello“, are vectors of length 1. For example,

```
> temperature <- 32
> temperature
[1] 32
```

Here, we have created a vector of length 1 and have stored a single number in it. We can store one additional value in the vector:

```
> temperature[2] <- 36.5
> temperature
[1] 32.0 36.5
```

Vectors *(cont.)*

Now, temperature has 2 elements. Note that the mode (the type) of the vector changed from integer to double. We can add a third value:

```
> temperature[3] <- "37"
> temperature
[1] "32"   "36.5" "37"
```

Since the third value is character, the mode of the whole vector changed to charcater.

An example of logical vector:

```
> logical <- c(TRUE,FALSE,TRUE,FALSE)
> logical
[1]  TRUE FALSE  TRUE FALSE
```

The vector logical contains four logical constants. A logical vector can be used as a row selector for another vector:

```
> age[logical]
[1] 10 30
```

Only elements with indexes corresponding to TRUE are selected. In this example, the first and the third elements of age are selected.

C: Combine Values into a Vector or List

c - a generic function which combines its parameters. For example, the following function call creates a vector of four integers and assigns it to variable age:

```
> age <- c(10,20,30,40)
```

“Generic” means that the function can take parameters of different types, for example,

```
c(1, "MALE", TRUE)
```

The function c combines its parameters to form a vector. All parameters are coerced to a common type, which is the highest type of the parameters in the hierarchy: NULL < raw < logical < integer < double < complex < character< list < expression.

Examples:

```
> x <- c(1, "MALE", TRUE)
> x
[1] "1"    "MALE" "TRUE"
```

Here, all types are coerced to character

```
> x <- c(1,2,3.2)
> x
[1] 1.0 2.0 3.2
```

Here, all types are coerced to double because the third value is double

Vectors *(cont.)*

The function c can be used to select particular elements from a vector. The following example creates a vector of six characters and selects the elements with indexes 1, 3, and 5:

```
> letters <- c("A","B","C","D","E","F")
> letters[c(1,3,5)]      # selects elements with indixes 1,
                         # 3, and 5
[1] "A" "C" "E"
```

The next example selects all elements of letters excluding the elements with indexes 1, 3, and 4:

```
> letters[-c(1,3,4)]
[1] "B" "E" "F "
```

In the next example, the range of indexes 3:4 is excluded:

```
> letters[-(3:4)]
[1] "A" "B" "E" "F"
```

In the next example, only the range of indexes 3:4 is included:

```
> letters[3:4]
[1] "C" "D"
```

Vector arithmetic

When arithmetic operations are performed on vectors, the operations are performed element by element:

```
> x <- c(2,3,4)
> y <- c(1,2,3)
> x + y
[1] 3 5 7
```

If vectors occurring in the same expression are not all of the same length, the value of the expression is a vector with the same length as the longest vector. Shorter vectors in the expression are recycled (repeated) as often as needed (perhaps fractionally) until they match the length of the longest vector. Constants are simply repeated:

```
> x + 1                 # this is equivalent to x + c(1,1,1)
[1] 3 4 5
> x*2                   # this is equivalent to x*c(2,2,2)
[1] 4 6 8
> z <- c(2,1,2,1,2,1)
> x + z                 # equivalent to c(2,3,4,2,3,4) + z
[1] 4 4 6 3 5 5
```

In the last example, x is recycled twice to produce a vector of 2 3 4 2 3 4, which is then added to 2 1 2 1 2 1.

Vector arithmetic *(cont.)*

Functions *max()* and *min()* return the maximum and the minimum element of a vector:

```
> max(x)
[1] 4
> min(x)
[1] 2
```

sum() and *prod()* return the sum and the product of the vector elements, respectively.

```
> sum(x)
[1] 9
> prod(x)
[1] 24
```

length() returns the length and *sort()* sorts a vector:

```
> sort(z)
[1] 1 1 1 2 2 2
> length(z)
[1] 6
```

Sequences

We have already used the sequence operator 1:50 to generate a vector of all integers from 1 to 50. Sequences can be assigned to vectors:

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

Sequences can be used in expressions.:

```
> x <- 2*1:10-1
> x
 [1]  1  3  5  7  9 11 13 15 17 19
```

This is equivalent to first creating a vector x:

```
> x <- 1:10
```

And then using x in:

```
> 2*x-1
 [1]  1  3  5  7  9 11 13 15 17 19
```

In other words, the colon operator has higher precedence than arithmetic operators. Remember that multiplication and subtraction is done element by element.

Sequences *(cont.)*

Backward sequences:

```
> 11:3
[1] 11 10  9  8  7  6  5  4  3
```

1:10 is equivalent to *seq(1,10):*

```
> seq(1,10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Other examples of function seq():

```
> seq(-1,1,by=.2)
[1] -1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2  0.4  0.6  0.8  1.0
> seq(length=10, from=-1, by=.5)
[1] -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5
```

The following function repeats x three times

```
> x <- c(1,2)
> rep(x,times=3)
[1] 1 2 1 2 1 2
```

The next call of *rep()* repeats each element of x (preserving the order of elements) three times:

```
> rep(x,each=3)
[1] 1 1 1 2 2 2
```

Logical vectors

The elements of a logical vector can have the values TRUE, FALSE, and NA (for “not available”).

```
>log <- c(TRUE,TRUE,FALSE)
```

TRUE and FALSE can be abbreviated as T and F, respectively.

```
>log <- c(T,T,F)
```

The use of T and F is prone to errors because T and F are just predefined variables. They are not reserved words as TRUE and FALSE and can be redefined by the user:

```
> T                 # T is predefined as TRUE
[1] TRUE
> T <- FALSE        # we can redefine it as FALSE
> T                 # now, T is FALSE
[1] FALSE
```

Logical vectors *(cont.)*

Logical vectors can be generated by logical conditions. For example:

```
> x <- c(10,8,11)
> log <- x>9   # equivalent to log <- c(10,8,11) > c(9,9,9)
> log
[1]  TRUE FALSE  TRUE
```

sets log as a vector of the same length as x with values FALSE corresponding to elements of x for which the logical condition x>9 is not met and TRUE where it is. The logical condition is checked element by element.

In integer context, logical vectors are coerced to integers:

```
> FALSE+1
[1] 1
> TRUE+1
[1] 2
> FALSE:TRUE
[1] 0 1
```

The logical operators are <, <=, >, >=, == for exact equality and != for inequality.

Logical vectors *(cont.)*

Another example of how logical vectors are generated by logical conditions:

```
> x <- c(4,2,7)
> y <- c(1,3,5)
> log <- x<y
> log
[1] FALSE  TRUE FALSE
```

Example of vector conjunction and disjunction:

```
> log1 <- c(TRUE,FALSE,TRUE)
> log2 <- c(FALSE,FALSE,TRUE)
> log1 & log2   # & stands for logical conjunction
[1] FALSE FALSE TRUE
> log1 | log2   # | stands for logical disjunction
[1]  TRUE FALSE TRUE
```

Special values (NA, NaN, Inf)

Inf stands for infinity, i.e., a number that is too big. **Inf** stands for positive infinity and **–Inf** stands for negative infinity:

```
> 1/0
[1] Inf
> -1/0
[1] -Inf
> 2 ^1024
[1] Inf
> -2 ^ 1024
[1] -Inf
```

NaN stands for “Not a Number“

```
> 0/0
[1] NaN
```

NA represents “ not available“ or “ missing value“. In other words, if the value of an elemet is missing, a place within a vector may be reserved for it by assigning it the special value NA. Any operation on an NA becomes an NA:

Special values (NA, NaN, Inf)

```
> x <- c(NA,1,2)
> 2*x
[1] NA  2 4
```

If you expand the size of a vector (including matrices and arrays) beyond the size where values are defined, the new spaces will have the value NA:

```
> v
[1] 1 2 3
> length(v) <- 6
> v
[1]  1  2  3 NA NA NA
```

The function is.na(x) returns a logical vector of the same size as x with value TRUE if and only if the corresponding element in x is NA.

```
> z <- c(1:3,NA)
> is.na(z)
[1] FALSE FALSE FALSE  TRUE
```

Index vectors

Subsets of the elements of a vector may be selected by appending to the name of the vector an index vector in square brackets. Index vectors can be any of four distinct types: logical vector, vector of positive integers, vector of negative integers, vector of character strings.

If the index vector is logical vector, it is recycled to the same length as the vector from which elements are to be selected. Values corresponding to TRUE in the index vector are selected and those corresponding to FALSE are omitted. For example:

```
> y <- x[!is.na(x)]
```

creates (or re-creates) an object y which will contain the non-missing values of x. Another example:

```
> log <- c(TRUE,FALSE,TRUE)
> x <- 1:9
> x[log]
[1] 1 3 4 6 7 9
```

In this case, log is recycled three times

Index vectors *(cont.)*

If the index vector consists of positive integral quantities, its values must lie in the set {1, 2, . . . , length(x)}. The corresponding elements of vector x are selected and concatenated, in that order, in the result.

```
> x <- 10:1
> ind <- c(2,3,1)
> x[ind]
[1]  9  8 10  # the elements are selected out of order
```

If the index vector consists of negative integral quantities, then it specifies the values to be excluded rather than included:

```
> y <- x[-(1:5)]
```

stores in y all but the first five elements of x.

Index vectors *(cont.)*

An example of an index vector of character strings:

```
> fruit <- c(5, 10, 1, 20)
> names(fruit) <- c("orange", "banana", "apple", "peach")
> lunch <- fruit[c("apple","orange")]
```

First, vector fruit is created with four integer elements. The construct *names(fruit)* assigns a name to each element of the vector fruit. “orange” is assigned to 5, “banana” is assigned to 10, “apple” is assigned to 1, and “peach” is assigned to 20.

The index vector c("apple","orange") selects only the elements with names “apple” and “orange” in that order:

```
> lunch
 apple orange
     1      5
```

The *c()* function can also be used to name some elements of a vector and leave others blank:

```
> z <- c(apple = 1, banana = 2, "kiwi fruit" = 3, 4)
```

Index vectors *(cont.)*

A logical expression can be used as an index. Then, the expression is evaluated for each element of the vector, and the element is selected if and only if the expression evaluates to TRUE. The following example selects all elements of x which are multiples of 3:

```
> x <- 1:10
> x[x%%3 == 0]
[1] 3 6 9
```

In other words *x%%3 == 0* evaluates to a logical vector of length 9, which is used as an index vector:

```
> x%%3 == 0
 [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
FALSE
```

Factors

When analyzing data, it is quite common to come across nominal (a.k.a. categorical) values. Features that represent a characteristic with categories of values are known as **nominal**. In other words, a nominal variable takes a finite set of nominal values. For example, *gender* could be either Male of Female, *diabetes* could be Type1 or Type2.

Suppose we have a vector of data about eye colors of people:

eye.colors <- c(“brown”, “blue”, “blue”, “blue”, “green”, “brown”, “brown”, “brown”, “green”)

Although valid, this representation is ineffective if you are working with large vectors. R provides a better way to represent nominal values.

A **factor** is a vector object used to specify a discrete classification (grouping) of the components of other vectors.

```
> gender <- factor(c("MALE", "FEMALE", "MALE"))
> gender
[1] MALE   FEMALE MALE
Levels: FEMALE MALE
```

`c("MALE", "FEMALE", "MALE"))` is a standard vector of character strings. The function factor makes a factor out of it, i.e., all values are grouped into two categories called levels. The levels comprise the set of possible categories the factor could take, in this case: MALE or FEMALE.

Factors *(cont.)*

Internally, the function factor() stores the nominal values as a vector of integers in the range 1:2 because we have only two nominal values, FEMALE and MALE. In other words, gender is internally stored as (2 1 2) and associates 1 with FEMALE and 2 with MALE (the assignment is alphabetical). In the general case, factors are internally stored as vector of integers in the range [1..k], where k is the number of distinct unique values of the nominal variable, and an internal vector of character strings is mapped to these integers. As a result, factors take less space than equivalent vectors of character strings.

```
> blood <- factor(c("O", "AB", "A"),
  levels = c("A", "B", "AB", "O"))
  > blood[1:2]
  [1] O AB
  Levels: A B AB O
```

Here, the factor() function takes a level variable, which describes all possible levels the factor could take. Note that our sample of blood types is small and does not include all possible levels. Storing a complete description of all possible levels allows for the possibility of adding data with the other blood types. It also allows us to make informed conclusions based on all possible levels not on the levels present in the current sample.

Factors *(cont.)*

R provides both ordered and unordered factors.

The factors described so far have been unordered. Here is an example of an ordered factor:

```
> symptoms <- factor(c("SEVERE", "MILD", "MODERATE"),
levels = c("MILD", "MODERATE", "SEVERE"), ordered = TRUE)
```

Since the *ordered* variable is TRUE, levels are specified in an ascending order: MILD < MODERATE < SEVERE. This can be seen by running the following command:

```
> symptoms
[1] SEVERE MILD MODERATE
Levels: MILD < MODERATE < SEVERE
```

The benefit of having ordered levels is that we can compare them and include them in logical expressions. For example:

```
> symptoms > "MODERATE"
[1] TRUE FALSE FALSE
```

To remind you, the comparison is done element by element, i.e., each element of symptoms is compared with “MODERATE”.

Lists

A list is an ordered collection of objects known as its components. The components could be of different types. For example, a list may contain a combination of vectors, matrices, other lists, data frames, etc. A list can be created in two ways depending on whether the components are named or unnamed. For unnamed components, one can use the command:

myList <- list(object1, object2, ….)

The command for named components is:

myList <- list(name1=object1, name2=object2, ….).

You can create an unnamed list and name it later using the names() function:

```
> patient <- list("John Doe", 98.1, "MALE")
> names(patient) <- c("Name", "Temperature", "Gender")
> patient
$Name
[1] "John Doe"
$Temperature
[1] 98.1
$Gender
[1] "MALE"
```

Lists *(cont.)*

Components are always numbered and may always be referred to as such. In our case, patient is the name of a list of three components, which may be individually referred as patient[1], patient[2], and patient[3]. For example:

```
> patient[3]
$Gender
[1] "MALE"
```

The output shown includes both the name and the component itself. It is important to remember that when the components are accessed in this way, the result is a list of a single component, not the component itself. That is, patient[3] is a list of the third component. If you want to access the component itself, you need to use double brackets:

```
> patient[[3]]
[1] "MALE"
```

Components of named lists can also be referred to either by giving the component name as a character string in place of the number in double square brackets, or by giving an expression of the form list_name$component_name. For example:

```
> patient$Temperature
[1] 98.1
> patient[["Temperature"]]
[1] 98.1
```

Lists *(cont.)*

The latter is useful, when the name of the component to be extracted is stored in another variable as in:

```
> x<-"Temperature"
> patient[[x]]
[1] 98.1
```

The names of components may be abbreviated down to the minimum number of letters needed to identify them uniquely. Thus patient$Temperature may be specified as patient$Temp or even as patient@T:

```
> patient$Temp
[1] 98.1
> patient$T
[1] 98.1
```

Assigning a value to a nonexistent element of a list will expand the list to accommodate the new value. The same rule applies to a variety of data structures, such as vector, matrix, array, etc.

Lists can be concatenated using the c() function. For example:

```
> list.ABC <- c(list.A, list.B, list.C)
```

Matrices

A matrix is a two-dimensional. Matrices are created with the *matrix* function. The general format of the *matrix* function is:

mymatrix <- matrix(vector,nrow=number_of_rows, ncol=number_of_columns, byrow=logical_value, dimnames=list(char_vector_rownames, char_vector_colnames)

where *vector* contains the elements of the matrix, *nrow* and *ncol* specify the row and the column dimensions, and *dimnames* contains optional row and column labels stored in character vectors. The option *byrow* indicates whether the matrix should be filled in by row (*byrow=TRUE*) or by column (*byrow=FALSE*). The default is by column.

```
> y<-matrix(1:20,nrow=5, ncol=4) #defines a 2 by 2 matrix
> y
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

Matrices *(cont.)*

```
> cells <- c(1,26,24,68)
> rnames <- c("R1", "R2")
> cnames <- c("C1", "C2")
> myMatrix <- matrix(cells, nrow=2, ncol=2, byrow=TRUE,
dimnames=list(rnames,cnames))
> myMatrix
   C1 C2
R1  1 26
R2 24 68
```

You can refer to rows, columns, or elements of a matrix by subscripts in brackets. For example, *myMatrix[2,]* refers to the second row of myMatrix, *myMatrix[,1]* refers to the first column, and *myMatrix[1,2]* refers to the second element in the first row.

The subscripts could be numeric vectors in order to refer to specific elements. For example:

```
> x <-matrix(1:9, nrow=3)
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> x[1,c(1,3)]      # refers to the first and the third element
[1] 1 7            # in the first row
```

Matrices *(cont.)*

A matrix can be created from a list, too. Like a vector, a list has a dim attribute, which is initially NULL:

```
> B <- list(1,2,3,4,5,6)
> dim(B)
NULL
```

We can turn the list into a matrix by setting the dim attribute:

```
> dim(B) <- c(2,3)
> B
     [,1] [,2] [,3]
[1,] 1    3    5
[2,] 2    4    6
```

Arrays

Arrays may have more than two dimensions. Arrays are created with the array() function as follows:

myArray <- array(vector, dimensions, dimnames)

where vector is the data vector for the array, dimensions is a numeric vector of the maximum index for each dimension and dimnames is an optional list of dimension labels. For example,

```
> z <- array(1:24, c(2,3,4),
dimnames=list(c("A1","A2"),c("B1","B2","B3"),c("C1","C2","C3","C4")
))
> z
, , C1          # the first element in the fourth dimension
                # is a 2 by 3 matrix
   B1 B2 B3
A1  1  3  5
A2  2  4  6

, , C2           # the second element in the fourth dimension
```

Arrays *(cont.)*

```
   B1 B2 B3
A1  7  9 11
A2  8 10 12

, , C3          # the third element in the fourth dimension

   B1 B2 B3
A1 13 15 17
A2 14 16 18

, , C4          # the fifth element in the fourth dimension

   B1 B2 B3
A1 19 21 23
A2 20 22 24
```

Arrays *(cont.)*

Every array has an attribute *dim* (for dimensions). The second attribute of the *array()* function is the attribute *dim*. In the previous example, it was set implicitly. The next example sets *dim* explicitly:

```
> v <- array(1:24, dim=c(3,4,2)) #defines 3x4x2 array
> v
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

Arrays *(cont.)*

One can first turn a regular vector into an array by setting the *dim* attribute:

```
> v <- c(1:12)
> v
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> dim(v) <- c(3,2,2)  # turns the vector into 3 by 2 by 2 array
> v
, , 1

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

, , 2

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

Arrays

Arrays may be used in arithmetic expressions and the result is an array formed by element-by-element operations on the data vector. The dim attributes of operands generally need to be the same, and this becomes the dimension vector of the result. So if A, B and C are all similar arrays, then

```
> D <- 2*A*B + C + 1
```

makes D a similar array with its data vector being the result of the given element-by-element operations.

Data frames

A data frame is a list that contains multiple vectors or factors that are the same length. A data frame is a useful way to represent tabular data and it is similar to a spreadsheet or a database table.

```
> subject_name <- c("John Doe", "Jane Doe", "Steve Graves")
> temperature <- c(98.1, 98.6, 101.4)
> flu_status <- c(FALSE, FALSE, TRUE)
> gender <- factor(c("MALE", "FEMALE", "MALE"))
> blood <- factor(c("O", "AB", "A"), levels = c("A", "B", "AB", "O"))
> symptoms <- factor(c("SEVERE", "MILD", "MODERATE"),
                     levels = c("MILD", "MODERATE", "SEVERE"),
                     ordered = TRUE)
> pt_data <- data.frame(subject_name, temperature, flu_status,
gender,
                       blood, symptoms, stringsAsFactors = FALSE)
> pt_data
  subject_name temperature flu_status gender blood symptoms
1     John Doe        98.1      FALSE   MALE     O   SEVERE
2     Jane Doe        98.6      FALSE FEMALE    AB     MILD
3 Steve Graves       101.4       TRUE   MALE     A MODERATE
```

Data frames *(cont.)*

The data frame is displayed in matrix format. In this example, the data frame has one column for each vector of patient data and one row for each patient. Usually, the data frame's columns are the features or attributes and the rows are the examples. Each column may be a different type but each row must have the same length.

Since data frames are lists, we can refer to a single element of the data frame using list notation:

```
> pt_data$subject_name
[1] "John Doe"     "Jane Doe"     "Steve Graves"
```

A vector of names can be used to extract several columns from a data frame:

```
> pt_data[c("temperature", "flu_status")]
  temperature flu_status
1        98.1      FALSE
2        98.6      FALSE
3       101.4       TRUE
```

Data frames *(cont.)*

To select a particular element from the data frame you must specify its raw and column. For example, the following command selects the value in the first row and second column of the patient data frame (the temperature value for John Doe):

```
> pt_data[1, 2]
[1] 98.1
```

You can select a bigger portion of the data frame by specifying the rows and the columns to be selected:

```
> pt_data[c(1, 3), c(2, 4)]
  temperature gender
1        98.1   MALE
3       101.4   MALE
```

The standard matrix notation can be used to select a whole raw or a whole column:

```
> pt_data[, 1]
[1] "John Doe"     "Jane Doe"     "Steve Graves"
> pt_data[1, ]
  subject_name temperature flu_status gender blood symptoms
1     John Doe        98.1      FALSE   MALE     O   SEVERE
```

Data frames *(cont.)*

Columns can be selected by name rather than by position:

```
> pt_data[c(1, 3), c("temperature", "gender")]
  temperature gender
1        98.1   MALE
3       101.4   MALE
```

Similarly, some columns could be excluded by using negative signs for their positions. The previous command is equivalent to:

```
> pt_data[-2, c(-1, -3, -5, -6)]
```

Objects in R

All variables, data, functions, results etc. are stored in so called objects which have their own names and other attributes such as length, dimensions, etc. In R, objects are not declared. Instead, they are created at the moment of their first appearance and their type is determined by the type of the assigned value.

The simplest way to think about objects is as if they are “things” that are represented and are manipulated in R. Examples, of objects include numeric vectors, character vectors, lists, functions, data frames, etc.

By the mode of an object we mean the basic type of its fundamental constituents. For example, if x is a vector with three elements 1, 2 and 3, the mode of x is numeric and the length of x is three.

```
> x <- c(1,2,3)
> mode(x)
[1] "numeric"
> length(x)
[1] 3
```

The mode is a special “property” of an object. Another property of every object is its length. The functions *mode(object)* and *length(object)* can be used to find out the mode and length of a object.

Objects in R *(cont.)*

The type of an object determines how it is stored in R. Objects are also members of a class that defines what information the object contains and how it can be used. R provides some mechanism for object-oriented programming (not just simple manipulation of objects, the way we study them), which is beyond the scope of this course.

Objects in R can have many properties associated with them called attributes. These properties explain what an object represents and how it should be interpreted by R. Examples of attributes are *dim, dimnames, levels* of a factor. If an object has an attribute, say *a*, you can refer to the attribute through *a(x)*. In most cases, there are methods to get and set the current value of the attribute.

You can get a list of all attributes of an object using the attributes() function:

```
>matrix(1:12,nrow=4,dimnames=list(c("r1","r2","r3","r4"),c("c1","c2","
+ c3")))
  > attributes(m)
  $dim
  [1] 4 3

  $dimnames
  $dimnames[[1]]
  [1] "r1" "r2" "r3" "r4"
```

Objects in R *(cont.)*

```
$dimnames[[2]]
[1] "c1" "c2" "c3"
```

The dim attributes shows the dimensions of an object, in this case four rows by three columns. The dimnames attribute is a two-elemnt list, consisting of the roes and column names. It is possible to access each attribute directly:

```
> dim(m)
[1] 4 3
> dimnames(m)
[[1]]
[1] "r1" "r2" "r3" "r4"

[[2]]
[1] "c1" "c2" "c3"

> colnames(m)
[1] "c1" "c2" "c3"
> rownames(m)
[1] "r1" "r2" "r3" "r4"
```

Functions

Functions are R objects, which can be cerates with the following syntax:

function(*arguments*) *body*

where arguments is a set of names and the body is an R expression. The body is enclosed in curly braces unless it is a single expression:

```
> f <- function(x,y) x+y
> f <- function(x,y) {x+y}
> f(1,2)
[1] 3
```

R allows function arguments to have default values:

```
> g <- function(x,y=10) {x+y}
> g(1)
[1] 11
> g(1,20)
[1] 21
```

R allows for variadic functions, i.e., functions with variable number of arguments. If you want a function to take extra arguments, but you do not know haw many, you specify an ellipsis:

```
> G <- function(x,…) {body}
```

In this case, x is the first argument and all remaining arguments can be collectively referred to as …

Functions *(cont.)*

In a function, you may use the return statement to specify the value returned by the function:

```
> g <- function(x){return(x^2+7)}
> g(2)
[1] 11
```

The first example on the previous slide shows that if the return statement is omitted, R returns the last evaluated expression as the result of a function:

```
> f <- function(x,y) {x+y}
```

Let's write a function that takes a variable number of integer arguments and returns their sum:

```
> addarg <- function(x,...){
+ args <- list(...)                # creates a list of arguments
+ for (a in args) x <- x + a       # a loop that adds up all args
+ x                                # returns x
+ }
> addarg(1,2,3,4,5)
[1] 15
```

Note once again that variables in R are not declared. They are created the first time they are used.

Functions *(cont.)*

A function can take another function as an argument. For example, the *sapply* function iterates through each element in a vector, applying another function to each element in the vector and returning the result:

```
  > a <- 1:15
> sapply(a,sqrt)                # applyes sqrt to each element of a
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
2.828427
   [9] 3.000000 3.162278 3.316625 3.464102 3.605551 3.741657 3.872983
```

When you specify a function, you assign a name to each argument . Inside the body of the function, you can access each argument by name:

```
  > addTwo <- function(first, second) { first + second^2}
```

The correspondence between formal and actual attributes could be positional:

```
   > addTwo(2,3)          # order the actual attributes is the same
                          # as the order of the formal attributes
[1] 11
```

The correspondence could be by name (the order is irrelevant)

```
   > addTwo(second=3,first=2)
[1] 11
```

The correspondence could be by partial name matching (it must be unambiguous)

```
  > addTwo(s=3,f=2)
  [1] 11
```

Control statements

Conditional statements take the form:

if (*condition*) *true_expression* else *false_expression*

or, alternatively:

if (*condition*) *expression*

For example:

```
> x <- 10
> y <- c(8, 10, 12, 3, 17)
> if (x < y) x else y
[1]  8 10 12  3 17
Warning message:
In if (x < y) x else y :
  the condition has length > 1 and only the first element
will be used
```

Control statements *(cont.)*

R also provides a *switch* statement. The most common usage of *switch* takes for its first argument an expression that returns a string, followed by several named arguments that provide results when the name matches the first argument. The names must match the first argument, and you can execute multiple expressions by enclosing them in curly braces:

```
> x <- "gamma"
> greek <- switch(
+   x,
+   alpha = 1,
+   beta = sqrt(4),
+   gamma = {
+       a <- pi / 3
+       4 * a ^ 2
+   }
+ )
> greek
[1] 4.386491
```

Control statements *(cont.)*

There are three different looping constructs in R: *repeat*, *while* and *for*. The simplest is *repeat*, which just repeats the same expression:

repeat *expression*

To stop repeating the expression, you can use the keyword *break*. To skip to the next iteration in a loop, you can use the command *next*. As an example, the following R code prints out multiples of 5 up to 25:

```
> i <- 5
> repeat {if (i > 25) break else {print(i); i <- i + 5;}}
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

Control statements *(cont.)*

The *while* loop repeats an expression while a condition is true:

while (*condition*) *expression*

As a simple example, let's rewrite the example above using a while loop:

```
> i <- 5
> while (i <= 25) {print(i); i <- i + 5}
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

Control statements *(cont.)*

R also provides for loops, which iterate through each item in a vector (or a list):

for (*var* in *list*) *expression*

Let's use the same example for a for loop:

```
> for (i in seq(from=5, to=25, by=5)) print(i)
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```