Informatics

The R language and system
with programming examples
Part 1

Claudio Sartori
Department of Computer Science and Engineering
claudio.sartori@unibo.it
https://www.unibo.it/sitoweb/claudio.sartori/

Learning Objectives

- The types of R
- Structure of an expression
- Operators and operands
- Priority of operators
- The logical values
- Logical expressions
  - De Morgan laws
- The programming constructs of R
- The structured variables of R
- I/O
- Last but not least: algorithms and translation in to R programs

Facts about R

- a system and language for statistic computing and high-quality graphics
- allows interactive usage for medium-complexity computations
- includes a programming language based on the functional paradigm
- open--source https://www.r-project.org
- available powerful open-source front-ends to help the programmer
  - e.g. Rstudio https://www.rstudio.com

R: style of use

- Interactive
  - in this way the user sends a command line to the system and receives an immediate answer or result
- Batch
  - in this way the user prepares a program that can contain any number of command lines and sends it for execution; the program will manage input, output and computations
- Rstudio allows a perfect integration of the two styles

Interactive computing

- At the prompt **>** you can type a *statement* and hit the **<return key>**
- The system will show some reaction, depending on the kind of statements
- The ↑↓ keys allow to roll by the history of the statements previously sent
- A statement can span over several lines

```
Examples
> 1.09 + 1.01
[1] 2.1
exp(0)
[1] 1
> exp(1)
[1] 2.718282
> log(0)
[1] -Inf
> log(1)
[1] 0
```

Batch computing

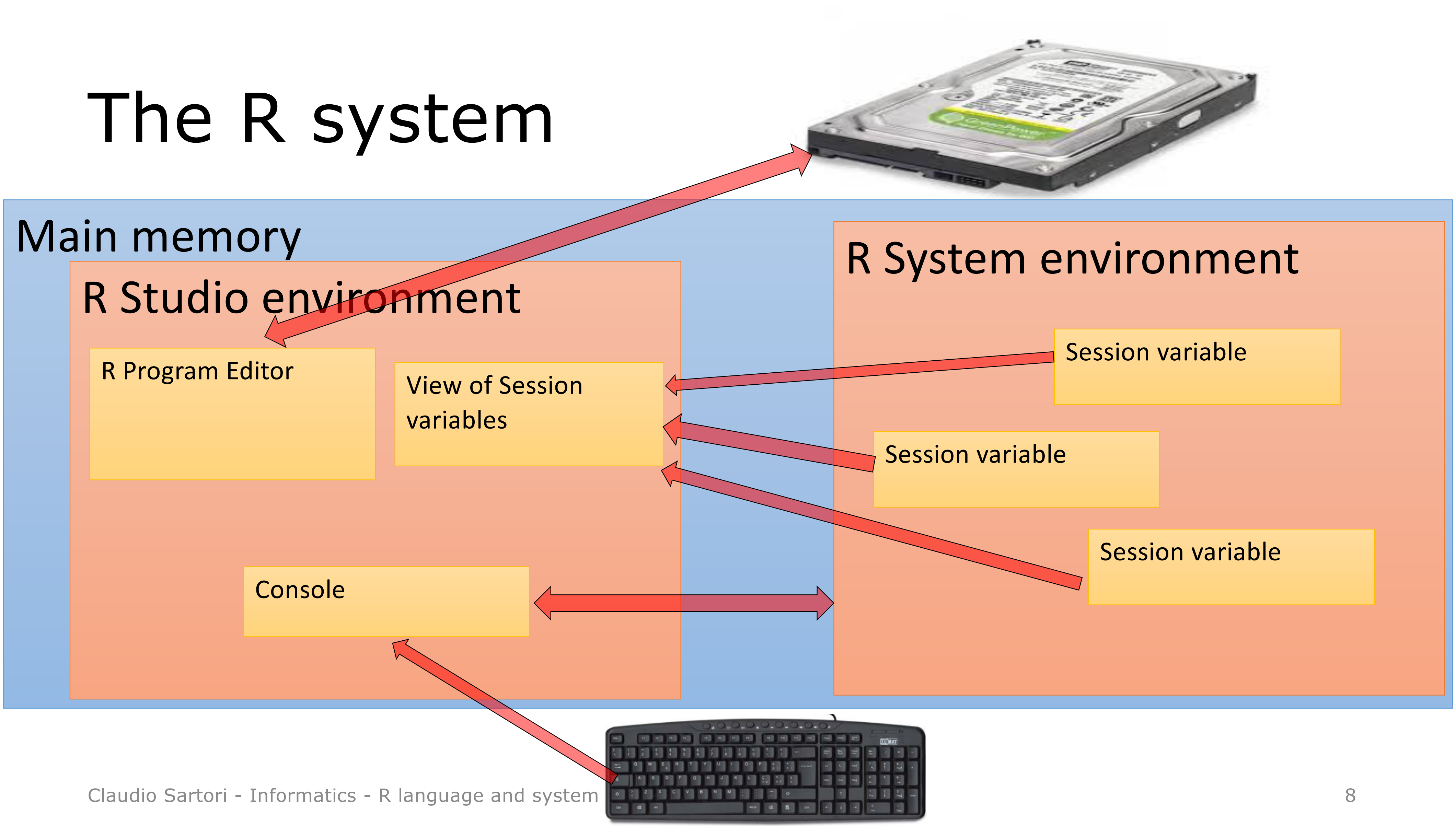
- A *program*, i.e. a *sequence of statements,* is written in a text file
  - e.g. using a text editor or a tool of the front-end
- The program is executed in the system
  - The program can/should contain input and output statemens

```
source('relative_or_absolute_path/file_name.R')
```

System environment

- It is the content of the R system memory
- It includes all the *variables* defined in the interactive actions or in the batch program
- Session
  - Starts when R is started
  - The environment, possibly changed during the session, can be saved at the end of the session, when R is closed
  - The saved environment can be restored in a subsequent session

The R system


Main memory
R Studio environment
R Program Editor
View of Session variables
Console
R System environment
Session variable
Session variable
Session variable

Types of statements

- Expression
  - A value computation
- Assignment
  - The name of a *variable*
  - The *assignment operator*
    `<-`
  - An expression

```
> 3 * 4^2 +12
[1] 60
> x <- 3 * 4^3 + 8
> x
[1] 200
```

Typing an expression

- The expression is computed, giving a value
- The value is shown in ouput

Typing an assignment

- The espression is computed
- If the variable name was not included in the environment then the variable is created in the environment
- The expression value is assigned to the variable

Expressions

Expression

- Defines a computation
- An alternate sequence of operands and operators
  - Associate two operands by means of an operator and compute the result
  - E.g. x + 2 uses adds two to the current value of x
- Priority
  - As in standard arithmetics, some operators must be executed before others
  - Each operator has a *priority value*, some operators have the same priority
    - Operators with the same priority are executed left to right
  - Parentheses increase the priority of the included operators

```
i * (j - 1) < k * j / n
```

Operand

- A constant
  - It has an immediate value, e.g. a number or a sequence of characters
- A variable
  - Its value is stored in the environment
    - If the variable is not included in the environment, an error is raised
- A function
  - Similar to matematical functions
  - Has *arguments*, that are expressions
  - It is evaluated and gives a value
  - It can have side effects
    - Modifications to the environment

Operand types: constants

- The type of an expression is inspected with the function **`typeof()`**

```
> typeof(2.35)
[1] "double"
> typeof(2)
[1] "double"
> typeof(2L)
[1] "integer"
> typeof("Hello world!")
[1] "character"
> typeof(TRUE)
[1] "logical"
```

The L suffix *forces* the integer type

Operand types: expressions

- The resulting type depends on the operand types and on the operators
- The interpreter checks the compatibility of types and operands

```
> typeof(2.3*5/8)
[1] "double"
> typeof(2L*6L)
[1] "integer"
> typeof(4L/2L)
[1] "double"
> typeof(2L*6)
[1] "double"
> typeof(2<3 & 10<7)
[1] "logical"
```

The *division* operator takes out of the integer domain

Mixed operands result in the more *complex* type

Operand types: variables

- The type of a variable *is* the type of the last assigned expression

```
> x <- 3/5
> typeof(x)
[1] "double"
> x <- "Hello world!"
> typeof(x)
[1] "character"
```

The interpreter checks the expression and executes it

- Evaluation of formal correctness
  - Alternance of operands and operators
  - Balanced parentheses
  - Well-formed constants
  - Variables already defined
- Preparation of the executable code to implement the operations

Operators priorities

operators	priority
() []	maximum
! + -	
^	
* / %% %/%	
+ -	
< <= > >=	
== !=	
&	
|	minimum

Negation

Exponent

Remainder of division

Integer division

Equality test

Inequality test

Logical AND

Logical OR

The list is not exhaustive

example

```
a * (b + c) / d < e – f + g * h & j > j * k
```

Some examples

- x * j < j + i * 3
- x == j
- x * y / j + i - 10

Data types and expressions

- you cannot apply numeric operators to non-numbers

```
> "a"+1
Error in "a" + 1
```

- TRUE and FALSE, besides being *logical* can also be used as *numbers*, and evaluate to 1 and 0, respectively

```
> TRUE == 1
[1] TRUE
```

Overflow in R

- **When an "integer operation" exceeds the limits type of the data is changed to double**
- **When a "double" operation exceeds the limits, the result is Inf**

Attention!

```
1<=x<=10          # is wrong

1<=x & x<=10      # is correct
```

Relational expression

- The possible results are *TRUE* and *FALSE*

operator

$$i < n$$

operands

Logical-relational expression

```
0 < i & i <= 9
i <= 0 | 9 < i
```

De Morgan equivalence laws

<table>
<tr><td>!(a==k & b==h)</td><td>a!=k | b!=h</td></tr>
<tr><td>!(a==k | b==h)</td><td>a!=k & b!=h</td></tr>
<tr><td>h<=x & x<=k</td><td>!(x<h | k<x)</td></tr>
</table>

In general

<table>
<tr><td>!(condition & condition)</td><td>!condition | !condition</td></tr>
<tr><td>!(condition | condition)</td><td>!condition & !condition</td></tr>
</table>

Logical operators for *conjunction* and *disjunction*

- & and | compute logical conjunction and disjunction (respectively) on both elementary values *and* vectors
  - we will study vectors later on
- && and || compute logical conjunction and disjunction (respectively) only on elementary values, when applied to vectors they consider only the first element

Input from *console*

- readline("prompt string")
  - Shows the prompt string and waits for input from the user
  - The function returns the string typed by the user
  - The returned value can be used in an expression and, therefore, stored in a variable
  - The returned value is of type *character*

```
> readline("Please, type something: ")
Please, type something: Hello world!
[1] "Hello world!"
> x <- readline("Please, type something: ")
Please, type something: 23
> typeof(x)
[1] "character"
```

`readline`

- `readline` is a function used to make an interaction from the *user* to the R system
- the argument is a message sent from the system to the user, to specify what is expected

Changing the type of an expression

- A wide set of functions to transform types
- Among others, dates are internally represented as numbers and can be added/subtracted

```
as.integer()
as.double()
as.character()
as.Date()
…
> x<-as.Date("2016/2/29")
> y<-as.Date("2016/2/28")
> x-y
Time difference of 1 days
```

Input from file

Generates as many elements as found in the file

```
> v <- scan("text1.txt")# by default reads double
Read 6 items
> print (v)
[1] 2 4 7 3 8 4
```

text1.txt

```
2
4
7
3
8
4
```

Input from file – more types

```
> cv <- scan("text2.txt", what = "character")
Read 4 items
> print(cv)
[1] "aa" "bb" "cc" "dd"
```

- Lots of scan options, see for full reference
  https://stat.ethz.ch/R-manual/R-devel/library/base/html/scan.html

Output

- Display the value of one or more expressions
- The **`cat()`** function
  - Concatenate and print
  - Similar to print, but with more detailed control on new lines
  - More suitable for batch programs
- The **`print()`** function
  - displays one expression
  - displays a complex object
    - examples later

Forces *new line*

```
> A <- 25
> cat("Value of A: ",A,"\n")
Value of A:  25
> print(paste("Value of A: ",A))
[1] "Value of A:  3"
```

Sequence

- statements written one after the other
- they will be executed one after the other, till the end
  - unless *special statements* are encoutered, such as `break` (explained later)
- a sequence can be grouped inside a pair of *braces* **{}**
- a sequence in braces can be included in *control statements*
  - *repetition*
  - *conditional*
  - *function definition*

Repetition

Repetition

- One of the base constructs necessary when writing algorithms is the *repetition*
- All the programming languages, including R, have several constructs for the repetition

`for`

`while`

`repeat`

while

- while is a reserved word of the language
- the logical expression can evaluate to TRUE or FALSE
- if the logical expression is true the body is executed and the logical expression is evaluated again
- if the logical expression is false in the beginning, the body is not executed

```
...instructions including assignment
...of values to the variables
...included in the logical expression
while (logical expression){
   ...sequence of statements
   ...including change of values
   ...of the variables included
   ...in the logical expression
}
```

for

Necessary to specify

- When to stop the repetition
- What is varying along with the repetition
- What instructions to repeat

```
for (var in seq){
    sequence of statements
    …
}
```

for (ii)

- *var* is a variable name
- *seq* is a sequence of values of any type
- *var* gets all the values in *seq* and for each of them the sequence of statements is executed
- After the last value the statement following the for is executed

```
for (var in seq){
    sequence of statements
    …
}
...
```

Sequences (i)

- A closed sequence of contiguous integers
- A sequence of numbers with a start value, an end value and a step
  - start, end, step are expressions
  - the end and can be reached or not, depending on the start and step values

```
> 1:8
[1] 1 2 3 4 5 6 7 8

> seq(1,2,0.3)
 [1] 1.0 1.3 1.6 1.9
> seq(0,2*pi,pi/2)
[1] 0.000000 1.570796 3.141593
4.712389 6.283185

for (x in seq(0,2*pi,pi/2)){
  cat(cos(x),"\n")
}
1
6.123234e-17
-1
-1.83697e-16
1
```

Should be 0, the computed value is not exact because of limited *precision*

Sequences (ii)

- The first n integers
  - if n is 0 the sequence is empty

- All the positions of a vector

```
> seq_len(4)
[1] 1 2 3 4
> seq_len(0)
Integer(0) # empty sequence

> a <- c(30,50,80)
> seq_along(a)
[1] 1 2 3
```

The compound interest

algo to R prog

```
# Compound interest for n years

# Use variables A, n, r, Af
# Read A, n, r
# Af <- A
# Repeat n times
#    Af <- Af * (1 + r)
# Write Af
#
```

```
A <- as.double(readline(
     "Insert the initial amount  "))
r <- as.double(readline(
     "Insert the interest rate   "))
n <- as.integer(readline(
     "Insert the number of years "))
Af <- A
for (i in 1:n){
  Af <- Af*(1+r)
}
cat("Starting value  ",A, "\n")
cat("Interest rate   ",r, "\n")
cat("Number of years ",n, "\n")
cat("Final amount    ",Af,"\n")
```

Exercises: writing algorithms

algorithms

a) Compute the product of two values **`m`** and **`n`** using only the *sum* operation
b) Compute the arithmetic mean of **`n`** numbers read from input
c) Compute the factorial of **`n`**
d) Compute the value of e^x with the MacLaurin series

Compute the product of two values **`m`** and **`n`** using only the *sum* operation

- Use m, n, p
- Input m, n
- p <- 0
- Repeat n times
  - p <- p + m
- Output m, n, p

- The solution is valid for any m numeric and for n positive integer

```
m <- as.double(readline(
    'insert a number'))
n <- as.integer(readline(
    'insert an integer '))
p <- 0
for (i in 1:n){
   p <- p + m
}
cat(m,' times ', n, ' gives ',p)
```

Limits of a solution

- when you write an algorithm (and then translate it into a program) you must be aware of the limits of the solution:
  - what are the *valid* input values?
    - i.e. the input values for which the program produces correct results

Compute the value of e^x with the MacLaurin series

Questions

algorithms

From mathematics we know that

$$e^x = 1 + x + x^2/2! + x^3/3! + \ldots$$

1. How can we compute the results of this sum?
    1. How can we compute all the terms to be added
2. How many terms should we add?
3. How many variables do we need?
4. Do we need to store all the terms?
5. Is there an easy way to compute one term, given the previous one?

algorithms

Computing the terms

$$e^x = \underbrace{1}_{t_0} + \underbrace{x}_{t_1} + \underbrace{x^2/2!}_{t_2} + \underbrace{x^3/3!}_{t_3} + \ldots$$

...

- Let's try to compute t_2/t_1

  x/2

- Let's try to compute t_3/t_2

  x/3

- In general, $t_i \leftarrow t_{i-1} * x / i$
- But we do not need the old value anymore, therefore

  $t \leftarrow t * x / i$

algorithms

Adding the terms

$$e^x = \underbrace{1}_{t_0} + \underbrace{x}_{t_1} + \underbrace{x^2/2!}_{t_2} + \underbrace{x^3/3!}_{t_3} + \underbrace{\ldots}_{\ldots}$$

- Each term will be added to the variable containing the sum

```
ex <- ex + t
```

- This operation will be repeated, according to the number of terms that will be used

```
Repeat n times varying i from 1 to n
     t ←t * x / i
     ex ← ex + t
```

algorithms

Initialization

$$e^x = 1 + x + x^2/2! + x^3/3! + \dots$$

t_0 t_1 t_2 t_3 ...

- Many iterative computations contain a step like this

t ← f(t)

- What should be the starting value?
- Rule of thumb
  - What should be the correct value in case of 0 repetitions?

t ← 1

ex ← t

algorithms

Putting things together

$$e^x = \underset{t_0}{1} + \underset{t_1}{x} + \underset{t_2}{x^2/2!} + \underset{t_3}{x^3/3!} + \underset{\ldots}{\ldots}$$

```
Use x, n, t, ex,
Input x and n
t ← 1
ex ← t
Repeat n times varying i from 1 to n
   t ← t * x / i
   ex ← ex + t
Output ex
```

algorithms

How many terms do we really need?

$$e^x = \underbrace{1}_{t_0} + \underbrace{x}_{t_1} + \underbrace{x^2/2!}_{t_2} + \underbrace{x^3/3!}_{t_3} + \underbrace{\ldots}_{\ldots}$$

- For x ≠ 0 the absolute value of terms is monotonically decreasing
- We could stop the computation when the term contribution is less than a given *precision*

t/ex < p

- We need a repetition *based on a condition* instead of one based on a sequence of values

Repeat *if condition is true*

algorithms

Repetition based on condition

$$e^x = 1 + x + x^2/2! + x^3/3! + \ldots$$

t_0 t_1 t_2 t_3 ...

Use x, t, ex, p

Input x and p

t ← 1

ex ← t

Repeat if the absolute value of t/ex >= p

    t ← t * x / i
    ex ← ex + t

Output ex

What's missing?

Repetition based on condition

Necessary to specify

- When to continue the repetition
  - Expressed as a logical expression
- What instructions to repeat
  - Must include some change that can make false the logical expression used to control the repetition

Need to manage i, no need of n

algorithms

$$e^x = \underbrace{1}_{t_0} + \underbrace{x}_{t_1} + \underbrace{x^2/2!}_{t_2} + \underbrace{x^3/3!}_{t_3} + \underbrace{\ldots}_{\ldots}$$

Use x, t, ex, p, i
Input x and p
t ← 1
ex ← t
i ← 1
Repeat if the absolute value of t/ex >= p
    t ← t * x / i
    ex ← ex + t
    i ← i + 1
Output ex, i

e^x with MacLaurin

algo to R prog

```
Use x, t, ex, p, i
Input x and p
t ← 1
ex ← t
i ← 1
Repeat if the absolute value of t/ex >= p
    t ← t * x / i
    ex ← ex + t
    i ← i + 1
Output ex, i
```

```
cat("Computation of exponential with
MacLaurin series\n")
x <- as.double(readline(
  "Insert x "))
p <- as.double(readline(
  "Insert the precision "))
t <- 1
ex <- t
i <- 1
while(abs(t/ex) >= p){
  t <- t * x / i
  ex <- ex + t
  i <- i + 1
}
cat("Exponential of ",x," with ",i,
    " MacLaurin terms: ",ex,"\n")
```

Recap – from algo to R – Repetition 1

algo to R prog

```
repeat varying i from a startV
       value to an endV value
  something
  something
something out of the repetition
```

i assumes all the consecutive values from startV to endV

```
for (i in startV:endV){
  something
  something
}
something out of the repetition
```

alternative solution

```
i <- startV
while (i <= endV){
  something
  something
  i <- i + 1
}
something out of the repetition
```

Recap – from algo to R – Repetition 2

algo to R prog

```
repeat varying i on all the
                positions of vector a
  something
  something
something out of the repetition
```

i assumes all the consecutive values from 1 to length of a

```
for (i in seq_along(a)){
  something
  something
}
something out of the repetition
```

alternative solution

```
i <- 1
while (i <= length(a)){
  something
  something
  i <- i + 1
}
something out of the repetition
```

Conditional execution

Conditional execution

- I need to do something only if some condition is true
- e.g.:
  - **a)** **if** there is no milk in the fridge, **then** buy milk
  - **b)** **if** there is milk in the fridge, **then** pour milk in a glass and drink milk, **else** go to the cornershop and buy milk
  - **c)** **if** you want to drink milk, **then if** there is no milk in the fridge **then** go to the cornershop, buy milk and come back home; pour milk in a glass and drink milk

Algorithm with conditional execution

```
a) if condition
         action
b) if condition
         action1
     else
         action2
c) if condition1
         if condition2
              action1
         action2
```

a	*execution*
condition true	action
condition false	-

b	*execution*
condition true	action 1
condition false	action 2

c	*condition 2 true*	*condition 2 false*
condition 1 true	action 1 action 2	action 2
condition 1 false	-	-

Example with conditionals: *what kind of triangle?*

- Given three numbers A, B, C in non-decreasing order
  - $A \leq B$ and $B \leq C$
- if the numbers are the measures of the sides of a triangle, what kind of triangle is?

- input A, B and C
- if A+B < C
  - Output "it isn't a triangle"
- else
  - if A equal to C
    - Output "it is equilateral"
  - else
    - if A = B or B = C
      - Output "it is isosceles"
    - else
      - Output "it is scalene"

Robust version

algo to R prog

```
input A, B and C
if A>B swap A and B
if C<A A<-C B<-A C<-B
else if (C<B) swap B and C
if A+B < C
    Output it isn't a triangle
else
    if A equal to C
        Output it is equilateral
    else
        if A = B or B = C
            Output it is isosceles
        else
            Output it is scalene
```

```
A <- as.double(readline("Insert number :"))
B <- as.double(readline("Insert number ")))
if (B<A){
   t<-A
   A<-B
   B<-t
}
# now A<=B
C <- as.double(readline("Insert number ")))
if (C<A){
   t<-C
   C<-B
   B<-A
   A<-t
} else {
   if (C<B){
      t<-C
      C<-B
      B<-t
}
#now A<=B B<=C
... as before
```

Conditional execution in R

```
if (logical expression){
   …
} else {
  ...
}
```

N.B.: the curly braces are optional if an optional path contains a single statement

What kind of triangle? R solution

```
A <- as.double(readline("Insert number :"))
B <- as.double(readline(paste(
          "Insert number not less than ",A," : ")))
C <- as.double(readline(paste(
          "Insert number not less than ",B," : ")))
if (A + B < C) print("It isn't a triangle") else
  if (A == C) print("It is equilateral") else
    if (A == B || B == C) print("It is isosceles") else
      print("It is scalene")
```

Pay attention to brackets

algo to R prog

```
x <- as.double(readline(
         "Insert number "))
if (x < 0)
  print("Changing sign of x")
  x <- -x
print(x)
```

Regardless the indentation, without braces the block ruled by if is a single statement
This is executed regardless the sign of x

The value shown is always the input with changed sign

```
x <- as.double(readline(
         "Insert number "))
if (x < 0){
  print("Changing sign of x")
  x <- -x
}
print(x)
```

In this case the value shown is always positive

See exercises

- Left triangle
- Right triangle
- Tree
- Check input range

Loops again: repeat

- the **repeat** construct introduces a block that is repeatedly executed
- the block must include an if statement to catch the ending condition and to execute, in this case, a break
- the break causes a *jump* out of the repetition

```
repeat {
  …
  if (end condition)
    break
}
```

An example with **`repeat`**

- repeat is useful when we need to execute the repetition at least once
- read a value checking its range, if the vaule is out of range, read again

```
N1 <- 10 # lower bound allowed
N2 <- 20 # upper bound allowed
repeat {
  n <- as.integer(
        readline(
          paste(
           "Insert a number between",
           N1," and",N2)))
  if (N1 <= n & n <= N2)
    break # if n is in range, loop ends
  cat("value out of range\n")
} # end of repeat
```

Vector

Several values on one variable

- A sequence of values of the same type
- A single variable name groups all the values
- The variable can be used *as a single object*
  - copy
  - manipulate
  - algebraic operations
- The single values can be accessed by an *index expression*
  - an integer ranging from 1 to the number of elements in the sequence
- The single values can also be accessed by a *name*, if names are defined

Operations with vectors

- vector creation/assignment
```
v <- c(1,4,12,7,5,2,9)
```
- assigning names
```
names(v) <- c("Monday","Tuesday","Wednesday","Thursday","Friday",
   "Saturday","Sunday")
```
- accessing one element
```
v[3]
v["Wednesday"]
```
- initialize an empty vector
```
v <- c()
```
- concatenate two vectors v1 and v2
```
v <- c(v1,v2)
```
- vector length
```
length(v)
```

Computations with vectors

- all arithmetic operations on vectors are intended element by element
- if lengths are different, the shortest one is replicated as necessary

```
> u <- c(3,6,2,7)
> v <- c(5,3,2,9)
> u+v
[1]  8  9  4 16
> w<-c(u,v)
> v+w #v=(5,3,2,9,5,3,2,9)
      #w=(3,6,2,7,5,3,2,9)
8 9 4 16 10 6 4 18
10*u # (10,10,10,10)*(3,6,2,7)
[1] 30 60 20 70
```

replicated

scalar

Looking for a value in a vector (i)

algorithms

- find what position of vector *v* has the same value as the variable *target*
- scan the components of *v* and stop when a value equal to *target* is found, returning the position where it has been found
- if the value is not found, return the position 0

Looking for a value in a vector (ii)

algo to R prog

```
use v:vector of values,
      target: value to be found
      found: integer, 0 if not found,
                between 1 and length of v
                if found
      i: index on v

input target

set found to 0

repeat varying i from 1 to length of v
    if the position i of v is equal to
    target
        set found to i and stop repeating

if found is zero display not found

else display the value of found
```

```
colors <- c("Red", "Blue", "Green", "Orange")
target <- readline("Insert target color ")
found <- 0
for (i in seq_along(colors)){
  if (colors[i] == target){
    found <- i
    break
  }
}
cat(target)
if (found == 0){
 cat(" not found")
} else {
cat(" found in position ",found)
}
```

Jumps out of the current repetition

from break jumps here

Looking for a value in a vector (iii)

also to R prog

alternate R implementation

```
use v:      vector of values,
    target: value to be found
    found:  integer, 0 if not found,
                between 1 and length of v
if found
      i: index on v

input target

set found to 0

repeat varying i from 1 to length of v
    if the position i of v is equal to
    target
        set found to i and stop repeating

if found is zero display not found

else display the value of found
```

```r
colors <- c("Red", "Blue", "Green", "Orange")
target <- readline("Insert target color ")
found <- 0
i <- 1
while (i <= length(colors) & found == 0){
  if (colors[i]==target){
    found <- i
  }
  i <- i + 1
}
cat(target)
if (found != 0){
  cat(" not found")
} else {
  cat(" found in position ",found)
}
```

Looking for the values of a vector satisfying a condition

- For example, given `v <- c(3,7,12,9)` look for the elements greater than 6

Tentative Algorithm

Consider all the elements of v and output those that satisfy the condition

Looking for the values of a vector satisfying a condition

- For example, given `v <- c(3,7,12,9)` look for the elements greater than 6

Tentative Algorithm

Consider all the elements of v and output those that satisfy the condition

Not enough specification

Looking for the values of a vector satisfying a condition

algo to R prog

```
Use
v:  vector of values,
condition: a logic expression with v
v_s: the vector of the elements of v
     satisfying the condition
i, is: indexes
Algorithm
create vs as empty vector of the same kind of v
set is to 0
repeat varying i from 1 to length of v
    if the position i of v satisfies the condition
        increment is by 1
        copy the i-th position of v to the
                 is-th position of vs
display vs
```

```
v <- c(3,7,12,9)
# condition v > 6
vs <- vector(mode = mode(v))
is <- 0
for (i in seq_along(v)){
  if (v[i] > 6){
    is <- is + 1
    vs[is] <- v[i]
  }
}
print(vs)
```

Accessing a vector **`v[<`*`index expression`*`>]`** (1)

- A positive integer **`i`**
  - If **`i`** is *not greater than* **`length(v)`**
    - **`v[i]`** is the i-th element of **`v`**
  - If **`i`** is *greater than* **`length(v)`**
    - **`v[i]`** is the **`NA`**
- A negative integer **`-i`**
  - If **`i`** is *not greater than* **`length(v)`**
    - **`v[-i]`** is all the elements of **`v`** excluded the i-th element
  - If **`i`** is *greater than* **`length(v)`**
    - **`v[-i]`** is the entire **`v`**

Accessing a vector **`v[<`*`index expression`*`>]`** (2)

- A vector of positive integers
  - The components of v corresponding to the integers are extracted
- A vector of booleans
  - **`v[c(T,F,F,T)]`** is the vector of the elements of **`v`** corresponding to True
  - If the length of the boolean vector is smaller than **`length(v)`**
    - The index expression elements are replicated to make the lengths equal
  - If the length of the index expression greater than **`length(v)`**
    - A number of **`NA`** values is appended to make the lengths equal

Abstraction

The essence of abstractions is preserving information that is relevant in a given context, forgetting information that is irrelevant in that context.

– John V. Guttag

Real world example (i)

How to cook pasta

- Put *enough* water in a pot
- Add salt
- Heat the water to boiling point
- Put *enough* pasta into the boiling water
- Wait for *enough time*
- Drain pasta and add *enough* seasoning

Real world example (ii)

- The previous example needs some specific values for a correct implementation
- The values are related to specific needs:
  - how many persons will eat
  - what kind of pasta
  - what kind of seasoning

Real world example (iii)

- The previous example can be abstracted as follows

Cook (pasta) spaghetti XYZ for 4 persons and add ragu

action

parameter 1

parameter 2

parameter 3

Real world example (iv)

function definition

```
cook_pasta(kind_of_pasta,n_persons,seasoning)
        …
        operating instructions
        …
```

Real world example (v)

function call

```
cook_pasta(kind_of_pasta=spaghetti_XYZ,
                 n_persons=4,
                 seasoning=ragu)
```

or, shortly

```
cook_pasta(spaghetti_XYZ, 4, ragu)
```

function

Function definition

- interface
  - name
  - formal parameters
- instructions
- the formal parameters are *variables* available *inside* the function

Function call

- name(… actual parameters...)
- the actual parameters are *expressions*

e^x with MacLaurin – function version

algo to R prog

```
function ex_mcLaurin
parameters v, prec
returns ex
use t, i
t ← 1
ex ← t
i ← 1
Repeat if the absolute value
               of t/ex >= prec
    t ← t * v / i
    ex ← ex + t
    i ← i + 1
return ex
```

```
# function ex_mcLaurin
ex_mcLaurin <- function(v, prec) {
    t <- 1
    ex <- t
    i <- 1
    while (abs(t / ex) >= prec) {
      t <- t * v / i
      ex <- ex + t
      i <- i + 1
    }
    return(ex) # computed value
               # is passed as result
  } # function ex_mcLaurin end
```

e^x with MacLaurin – function call

named parameters

algo to R prog

Input

function call with named parameters

output

```
cat("Computation of exponential with ",
    "MacLaurin series\n")
# input
x <- as.double(readline("Insert x "))
p <- as.double(readline(
    "Insert the precision "))
# function call,
# result stored in ex
ex <- ex_mcLaurin(v = x, prec = p)
cat("Exponential of ",x," : ",ex,"\n")
```

parameter assignment

e^x with MacLaurin – function call

positional parameters

algo to R prog

```
Input

function call with positional parameters

output
```

```
cat("Computation of exponential with ",
    "MacLaurin series\n")

# input

x <- as.double(readline("Insert x "))

p <- as.double(readline(
    "Insert the precision "))

# function call,
# x copied into the first parameter
# p copied into the second parameter
# result stored in ex

ex <- ex_mcLaurin(x, p)

cat("Exponential of ",x," : ",ex,"\n")
```

e^x with MacLaurin – function version

more flexibility

- define a *default value* for the prec parameter

```
# function ex_mcLaurin
ex_mcLaurin <-
           function(v, prec = 10E-15) {
    t <- 1
    ex <- t
    i <- 1
    while (abs(t / ex) >= prec) {
      t <- t * v / i
      ex <- ex + t
      i <- i + 1
    }
    return(ex) # computed value
               # is passed as result
  } # function ex_mcLaurin end
```

e^x with MacLaurin – function call

using default

- if the parameter with a default value is not specified, the default value is assumed

```
cat("Computation of exponential with ",
    "MacLaurin series\n")
# input
x <- as.double(readline("Insert x "))
p <- as.double(readline(
    "Insert the precision "))
# function call,
# x copied into the first parameter
# p not specified,
#       the default value is taken
# result stored in ex
ex <- ex_mcLaurin(x)
cat("Exponential of ",x," : ",ex,"\n")
```

Lists

List

- Sequence of elements *of any kind*
- are built with the constructor **`list()`**

```
> l <- list("a", 1, "b", 2)
```

- elements can have a name

```
> person <- list(first_name = "John",
                 phone = "335 234 5678",
                 birthdate = as.Date("1990/12/16"),
                 weight = 78)
```

- first_name, phone, birthdate, weight are the names of the elements of the variable person, that is a list

Using lists

- single elements can be accessed with an index (as vectors) but with [[]] (double square braces)

```
> person[[2]]
[1] "335 234 5678"
```

- single braces returns the pair <name,value>

```
> person[2]
$phone
[1] "335 234 5678"
> person$name
[1] "John"
```

Function returning more than one value

- The return value can be *a list*

```
ex_mcLaurin <- function(v,
                        prec = 10E-15) {
    t <- 1
    ex <- t
    i <- 1
    while (abs(t / ex) >= prec) {
      t <- t * x / i
      ex <- ex + t
      i <- i + 1
    }
    return(list(result = ex,
                iterations = i))
  } # function ex_mcLaurin end
```

computed value and number of iterations are passed as result into a list

Function returning more than one value
calling and showing the result

- The elements of the list are shown

```
# calling program
x <- as.double(readline("Insert x "))
# precision not passed, using default
r <- ex_mcLaurin(x) # the return
value is a list
cat("Exponential of ",x,
    " : ",r$result,
    " - Iterations : ",
    r$iterations,"\n")
```

Saving a for later use

- A function definition can be saved in a separate file
  - from the editor
  - with an R command

```
dump("<function_name>",
     file ="<file_path_name>")
```

- The file can be loaded into the current program

```
source("<file_path_name>")
```

Exercise: Gini concentration index (aka Gini coefficient)

- given a vector of observations x_i in non decreasing order
  - e.g. the income of persons, $x_i \leq x_{i+1}$
- measures the inequality among values of the frequency distribution (for example, levels of income)
- Gini coefficient zero expresses *perfect equality*, where all values are the same
  - for example, where everyone has the same income
- Gini coefficient one (or 100%) expresses *maximal inequality* among values
  - e.g., for a large number of people, where only one person has all the income or consumption, and all others have none, the Gini coefficient will be very nearly one

(wikipedia)

Gini Concentration Index

X_1	100	100	10	10
X_2	100	101	20	100
X_3	100	102	30	1000
X_4	100	103	40	10000
X_5	100	104	400	100000
gini index	**0**	**0.009803922**	**0.8**	**0.9444694**

The cumulates (for different data sets)

	ds1	cum	ds2	cum	ds3	cum	ds4	cum
X_1	*100*	100	*100*	100	*10*	10	*10*	10
X_2	*100*	200	*101*	201	*20*	30	*100*	110
X_3	*100*	300	*102*	303	*30*	60	*1000*	1110
X_4	*100*	400	*103*	406	*40*	100	*10000*	11110
X_5	*100*	500	*104*	510	*400*	500	*100000*	111110

Exercise: Gini concentration index

- given a vector of observations x_i in non decreasing order
  - e.g. the income of persons, $x_i \leq x_{i+1}$
- compute the relative cumulates in case of uniform distribution P_i
- compute the relative cumulates Q_i

$$G = \frac{\sum_{i=1}^{n-1} P_i - \sum_{i=1}^{n-1} Q_i}{\sum_{i=1}^{n-1} P_i} = 1 - \frac{\sum_{i=1}^{n-1} Q_i}{\sum_{i=1}^{n-1} P_i}$$

Gini concentration index (ii)

- simplified formula, since

$$P_i = \frac{i}{n} \quad \Rightarrow \quad \sum_{i=1}^{n-1} \frac{i}{n} = \frac{1}{n} \sum_{i=1}^{n-1} i = \frac{1}{n} \frac{n(n-1)}{2} = \frac{n-1}{2}$$

- therefore

$$G = 1 - 2 \frac{\sum_{i=1}^{n-1} Q_i}{n-1}$$

See the implementations

https://github.com/bladerun16/Informatics-Stats-and-Maths/blob/main/R_programs/018a_gini_index.R

https://github.com/bladerun16/Informatics-Stats-and-Maths/blob/main/R_programs/018b_gini_index_compact.R

Algorithms:
Set Operations

Set operations

- given sets *v1* and *v2* compute
  - the intersection
  - the union
  - the difference
  - the test of inclusion: *target* included in *container*
  - given a vector produce the unique elements
- given a vector *v* generate the set of its *unique values*
- it is a good exercise to study their implementation in R
- sets will be represented as vectors
- see the R examples
  https://github.com/bladerun16/Informatics-Stats-and-Maths/tree/main/R_programs/020_set_functions

The original R set functions

```
union(x, y)
intersect(x, y)
setdiff(x, y)
setequal(x, y)
unique(v) # b.t.w. this is much more powerful
        # and has several additional parameters
duplicated(v) # returns a vector of booleans
     # false if element already present in
     # lower indexes
x %in% y
```

Exercise

write the function

`setequal_vec(v1,v2)`

- returns true if and only if v1 and v2 represent the same set
- must work also if any of the input vector is empty

More data structures

beyond vectors

matrix

- a two-dimensional vector
- all the elements are of the same type

```
> mat <-matrix(
                data=c(2,3,5,7,11,13,17,19,23,29,31,37),
                nrow=3,ncol=4)
> mat
     [,1] [,2] [,3] [,4]
[1,]    2    7   17   29
[2,]    3   11   19   31
[3,]    5   13   23   37
```

matrix – filling options

- the number of given values can be shorter than necessary, if with an *exact repetition* it is possible to fill the matrix
- in particular, if we give exactly nrows or ncols values, the matrix is *filled* as necessary

```
> mat <-matrix(data=c(2,3,5,7),nrow=3,ncol=4)
> mat
     [,1] [,2] [,3] [,4]
[1,]    2    7    5    3
[2,]    3    2    7    5
[3,]    5    3    2    7
```

Algorithm: matrix multiplication

- parameters
  - A, B, matrices to multiply
- use
  - m, p, n: relevant dimensions of matrices
  - i, j, k: indexes to drive loops
  - C: product matrix
- extract number of rows and columns from the two matrices passed as parameters
- if the dimensions are not compatible return NULL
- create C as an empty double vector of dimension m*p
- repeat for each row i of A
  - repeat for each column j of B
    - initialize to zero element i,j of C
    - repeat for each column k of A
      - add A[i,k]*B[k,j] to C[i,j]
- return C

```
matProd <- function (A, B) {
  m <- dim(A)[1] # extracts number of rows of A
  p <- dim(B)[2] # extracts number of columns of B
  n <- dim(A)[2] # extracts number of columns of A
  if (dim(B)[1] != n) # compare n with number of rows of B
    return(NULL) # if dimensions not compatible return NULL
  C <- vector(mode = "double", length = m * p)
  dim(C) <- c(m,p)
  for (i in seq_len(m)) # for each row of A
    for (j in seq_len(p)) { # for each column of B
      C[i,j] <- 0 # initialize the summation
      for (k in seq_len(n))
        # compute the summation
        C[i,j] <- C[i,j] + A[i,k] * B[k,j]
    }
  return(C)
}
```

array

- generalization of matrix: any number of *dimensions*
- all the elements are of the same type

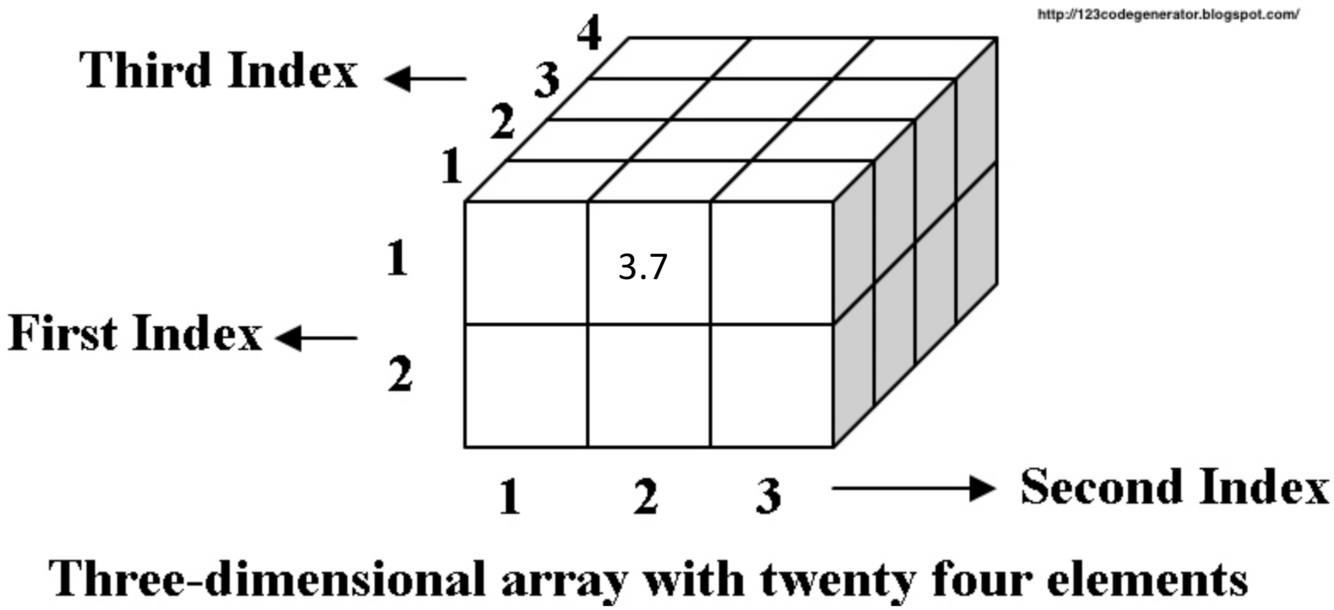
```
> arr <-array(
   data=c(2,3,5,7,11,13,17,19,23,29,31,37),
   dim=c(2,3,2))
> arr
, , 1

     [,1] [,2] [,3]
[1,]    2    5   11
[2,]    3    7   13

, , 2

     [,1] [,2] [,3]
[1,]   17   23   31
[2,]   19   29   37
```

Data structures and mathematics

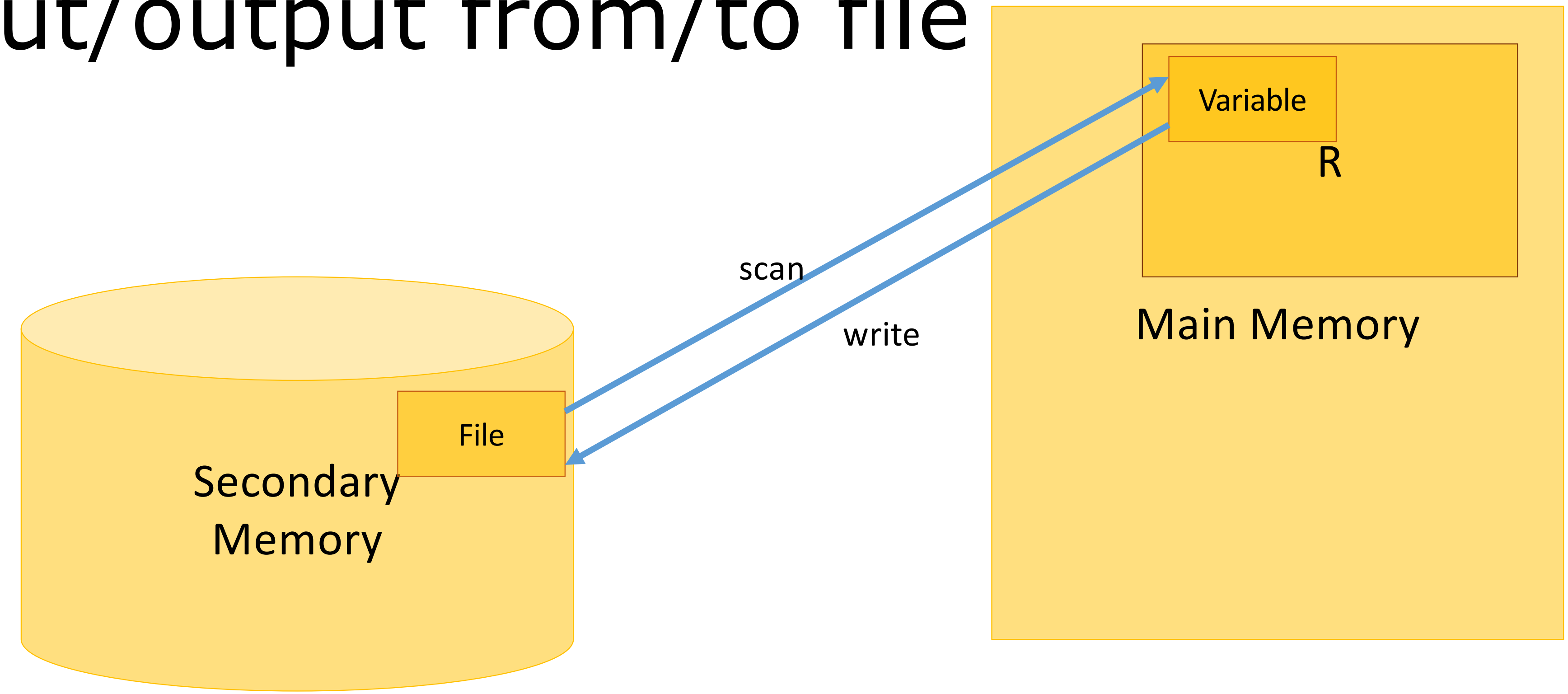
- A vector of n doubles can represent a point in an n-dimensional space
  - $x \in R^n$
- A matrix of doubles with m rows and n columns can represent m points in an n-dimensional space
  - $x_i \in R^n, i = 1, \dots m$
- An array of numbers with n-dimensions can represent values associated to points in an n-dimensional discrete space

Use of arrays

a[1,2,1]<- 3.7



Input/output from/to file



Input from file – need complex type

```
> v <- scan("text1.txt")# by default reads double
Read 6 items
> print (v)
[1] 2 4 7 3 8 4
> dim(v) <- c(2,3) # reshaping
> print (v)
     [,1] [,2] [,3]
[1,]    2    7    8
[2,]    4    3    4
```

text1.txt

```
2
4
7
3
8
4
```

data.frame

Example

- When doing a market research survey you often have questions such as:
  - 'Are your married?' or 'yes/no' questions (= Boolean data type)
  - 'How old are you?' (= numeric data type)
  - 'What is your opinion on this product?' or other 'open-ended' questions (= character data type)
  - …
- The respondents' answers to the questions formulated above, is a data set of different data types.
- A data frame has the variables of a data set as columns and the observations as rows.

data frame

one of the example data built-in in R

Row names

Column names

- in essence: something like a vector of homogeneous lists, or a list of homogeneous vectors

```
> mtcars
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
...
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
>
```

data.frame - accessing

- head()
  - shows the first (last) n rows of the data frame
- tail()
- row and column index
- row and/or column name
- an empty index means all the row/column

data.frame –accessing (ii)

```
> mtcars$disp
 [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8
[13] 275.8 275.8 472.0 460.0 440.0  78.7  75.7  71.1 120.1 318.0 304.0 350.0
[25] 400.0  79.0 120.3  95.1 351.0 145.0 301.0 121.0
> mtcars["Pontiac Firebird",]
                  mpg cyl disp  hp drat    wt  qsec vs am gear carb
Pontiac Firebird 19.2   8  400 175 3.08 3.845 17.05  0  0    3    2
> mtcars[1,1]
[1] 21
> mtcars[1,]
          mpg cyl disp  hp drat   wt  qsec vs am gear carb
Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
> mtcars[,1]
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
[15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
[29] 15.8 19.7 15.0 21.4
```

data.frame - creating

```
persons <- data.frame(name = c("John","Mary","Sue"),
               phone = c("335 234 5678",
                         "338 876 5432",
                         "340 1234 567"),
               birthdate = c(as.Date("1990/12/16"),
                             as.Date("1988/11/10"),
                             as.Date("1991/08/26")
                             ),
               weight = c(78,54,58)
)
```

data.frame - viewing

```
> print(persons)
  name         phone  birthdate weight
1 John 335 234 5678 1990-12-16     78
2 Mary 338 876 5432 1988-11-10     54
3  Sue 340 1234 567 1991-08-26     58
```

Input from file – data.frame

```
> gdp2014 <- read.csv("gdp2014.csv", sep=";",
                      stringsAsFactors = FALSE)
> head(gdp2014)
  country_code ranking        economy gdp_millions_usd
1          USA       1  United States         17419000
2          CHN       2          China         10354832
3          JPN       3          Japan          4601461
4          DEU       4        Germany          3868291
5          GBR       5 United Kingdom          2988893
6          FRA       6         France          2829192
```

Reshaping a data.frame

The values of first column become the names of the rows

Eliminate the first column

```
> gdp2014 <- read.csv("gdp2014.csv",sep=";")
> row.names(gdp2014) <- gdp2014[,1]
> gdp2014[,1] <- NULL
> head(gdp2014)
    ranking        economy gdp_millions_usd
USA       1  United States         17419000
CHN       2          China         10354832
JPN       3          Japan          4601461
DEU       4        Germany          3868291
GBR       5 United Kingdom          2988893
FRA       6         France          2829192
```

data frame – accessing (iii)

```
> gdp2014["USA",] # access by row name
    ranking          economy gdp_millions_usdUSA
1 United States               17419000
> gdp2014[1,] # access by row number
    ranking          economy gdp_millions_usdUSA
1 United States               17419000
```

Summary of data.frame functions (1)

- **`row.names / col.names`**
  - get or set the rows/columns names
- **`nrow / ncol`**
  - get number of rows / columns
- **`dim`**
  - get or set a vector with number of rows and columns
- **`str`**
  - report on the structure of the dataframe

Summary of data.frame functions (2)

- **`head / tail`**
  - preview on the beginning/ending rows
- **`summary`**
  - descriptive statistics on the columns
- **`View`**
  - opens a window with tabular view

Summary of data.frame functions (3)

- **`read.csv("text.txt")`**
  - read from a text file in secondary memory
  - header = True/False
    - if True the first row of the file is used as column names
  - sep [default is ","]
    - separator between values in a row
  - row.names/col.names
    - set the properties by passing an appropriate vector of names, otherwise defaults are assumed
    - default row names are numbers
    - default column names are V followed by column number

Summary of data.frame functions (4)

- **`write.csv(df, “text.txt”)`**
  - write into a text file in secondary memory the dataframe df
  - header = True/False
    - if True the first row of the file is used as column names
  - row.names = True/False
    - if True a column of row names is written
  - sep [default is ”,”]
    - separator between values in a row

Summary of data.frame functions (5)

- **`cbind / rbind`**
  - add columns / rows
- **`df$column_name <- NULL`**
  - delete column by name from **`df`**
- **`subset(df, <boolean_vector>)`**
  - keeps only the rows corresponding to TRUE

Examples

- given a data frame remove duplicated rows
- given two data frames with the same structure and without duplicates
  - produce the intersection
  - produce the union
- see **`example029_dataframe_set_ops.zip`**

Intersection

df1	i	df2	j	keep
G		G	*	T
W	*	S		F
F		W		T
H		F		T

Data Types: Factors

Factor variables in R

- *categorical variables* that can be either numeric or string
- a number of advantages to converting categorical variables to factors
- used in statistical modeling where they will be implemented correctly
    - assign the correct number of degrees of freedom
- storing string variables as factor variables is a more efficient use of memory
- **`factor`** function
- the only required argument is a vector of values that can be either string or numeric
- optional arguments include
    - levels
        - determines the categories of the factor variable
        - the default is the sorted list of all the distinct values of the data vector
    - ordered = FALSE
        - logical, if true the levels are assumed as ordered
    - labels
        - a vector of values that will be the labels of the categories in the levels argument
    - exclude
        - which levels will be classified as NA in any output using the factor variable

Factor

- symbolic representation of the values of a *categorical variable*
- the values are called *levels*
- Example: air quality, values
  - good
  - fair
  - sufficient
  - poor
  - bad
  - dangerous

Ordering of factor values

- when you create a factor you can decide
  ordered = FALSE or
  ordered = TRUE
- in the first case there not exists any order relationship among the values
- in the second case there exists an order relationship between any pair of values
- independently form the “ordered” property, we can request an ordered output, only for presentation purposes, not for computations

Pros and cons of factors

PRO

- when the number of distinct values in the column is much smaller than the number of rows
  - coding saves space (mildly important)
- when we need an ordering that is different from the alphabetic order given by the strings
- when we want to prevent ordering
- when we need to build cross tabulations based on that column

- CON
- when the number of distinct values is very high
- when it is unlikely to need cross tabulations based on that column
- when the alphabetic ordering is useful, but it is enough

R example

```
# without factor
air.quality.last.week <-
   c("good","fair","fair","sufficient",
     "fair","fair", "good")
print(air.quality.last.week)
[1] "good" "fair" "fair" "sufficient" "fair" "fair"
[7] "good"
```

R example

```
# with factor
air.quality.last.week.f <-
 factor(c("good","fair","fair","sufficient","fair",
          "fair", "good"),
        levels = c("good", "fair", "sufficient",
                   "poor", "bad"),
        ordered = TRUE)
print(air.quality.last.week.f)
[1] good fair fair sufficient fair fair good
Levels: good < fair < sufficient < poor < bad
```

Using factors with data frames

- When you create a data frame or read it from a file, there is a default value

  `stringsAsFactors = TRUE`
- this will convert all the columns of type **`string`** into **`factors`**
- this means that:
  - those columns will be coded and stored as integers, but the coded values will be decoded to the original value for any display purpose
  - the coded values are stored as integers, but they *aren't integers*, in the sense that arithmetic operations are not allowed

Operators allowed for factor values

Operators	Ordered Factor	Non Ordered Factor
+ - * / ^	no	no
== !=	yes	yes
>= > < <=	yes	no