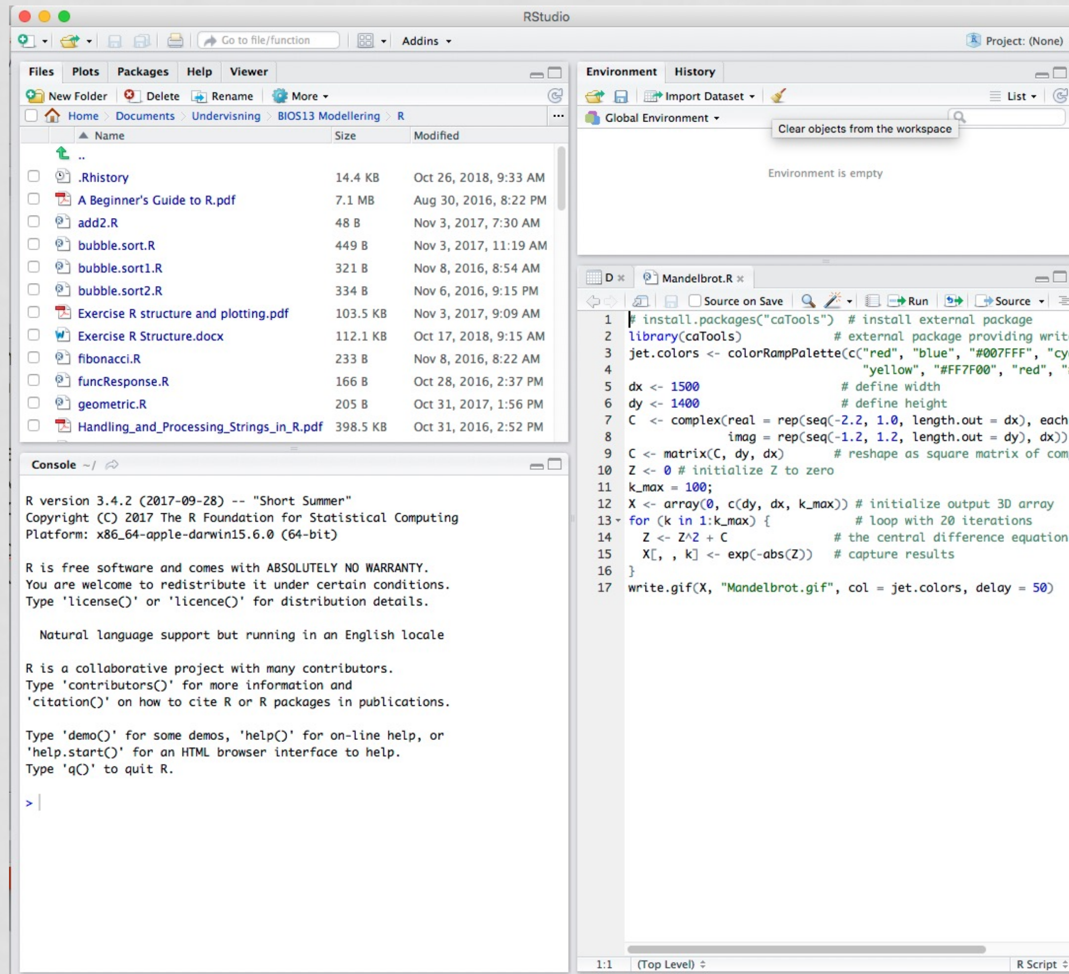
VARIABLES, VECTORS AND FLOW CONTROL IN R

MODELLING BIOLOGICAL SYSTEMS, BIOS13
PEDRO ROSERO

R

- **R** is a programming language
- **RStudio** is a convenient environment for working with R. There are others.
- We will use the RStudio environment throughout this course.



Variables in R

(<-, objects, ls, rm)

- Variables are created by assignment:

```
> x <- 3
> x
[1] 3
```

The value 3 is copied!

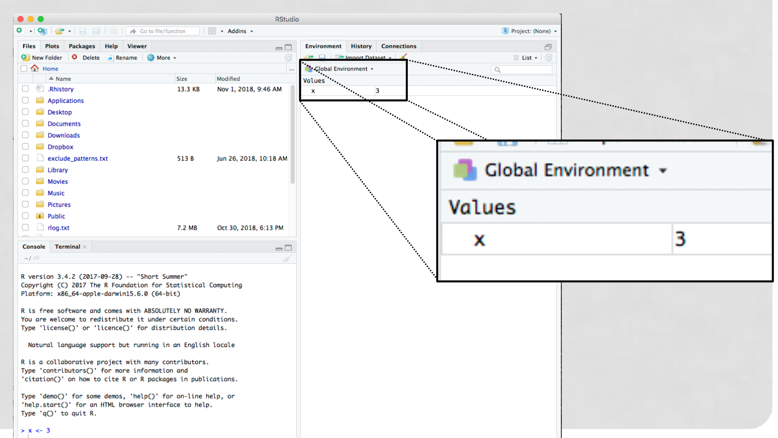
- Each variable is an object in the current workspace (= a part of computer memory):

```
> objects()
[1] "x"
```

Equivalent:

```
> ls()
[1] "x"
```

- Also check the "Environment" tab:



- Delete (remove) a variable:

```
> rm(x)
```

Mathematical Operators in R

- Mathematical operators: `+, -, *, /, ^`
- Examples:

```
> 23 + 13
[1] 36
> 4*5 + 7
[1] 27
> 4*(5+7)
[1] 48
> 4*(5+7)^2
[1] 576
```

- The operations are carried out in order of *precedence*, which is the reverse order of the list above. Precedence can be overriden with parantheses ().

Exercise (stolen from this afternoon)

Mathematical operators: `+, -, *, /, ^`

Calculate (you find the correct answer on the right):

a)	2^8	256
b)	$\frac{26}{7}$	3.714286
c)	$\frac{26+2^8}{7}$	40.28571
d)	$26 + 2^{8/7}$	28.20818
e)	$(26+2)^{8/7}$	45.07076

Mathematical Functions in R

- **Basic mathematical functions**

R	Standard math
`exp(x), log(x)`	e^x, $\ln(x)$ or $\log(x)$
`sin(x), cos(x), tan(x)`	$\sin(x), \cos(x), \tan(x)$
`asin(x), acos(x), atan(x)`	$\sin^{-1}(x), \cos^{-1}(x), \tan^{-1}(x)$
`log10(x)`	$\log_{10}(x)$
`sqrt(x)`	$\sqrt{x}$
`abs(x)`	$\|x\|$
`pi`	π (3.1415964 …)
`round(x), floor(x), ceiling(x)`	rounding (nearest, down, up)

- **Other operations**, not commonly used in math:

`%/%` integer division (remove decimals)

`%%` modulo (the remainder, after division)

Special values

`NA`	Missing Value (Not Available)
`NaN`	Undefined (Not A Number)
`NULL`	Empty set (Nothing)
`Inf, -Inf`	Infinity

```
> a <- NA
> a + 3
[1] NA
> 1/a
[1] NA
> 1/0
[1] Inf
> -1/0
[1] -Inf
> log(0)
[1] -Inf
> 0/0
[1] NaN
```

```
> 4 < 1/0
[1] TRUE
> 4 < -1/0
[1] FALSE
> c <- NULL
> c + 5
numeric(0)
```

A numeric vector with 0 elements

Logical operators in R

Logical operators generate logical values (TRUE or FALSE, T or F)

- *Comparisons*: <, <=, >, >=, ==, !=
- Operators:

`!`	negation
`&, &&`	And
`|, ||`	Or

These operators are 'lazy'. The second value is not calculated if the first is FALSE (&&) or TRUE (| |). Sometimes useful.

```
> 3 < 7
[1] TRUE
> 3 + 5 < 7
[1] FALSE
> 4 == 0
[1] FALSE
> a <- 6
> b <- 10
> a > 6
[1] FALSE
> a >= 6
[1] TRUE
> a != b
[1] TRUE
> a != a
[1] FALSE
```

```
> a > 8 & b > 1
[1] FALSE
> a > 8 && b > 1
[1] FALSE
> a > 8 | b > 1
[1] TRUE
> a > 8 || b > 1
[1] TRUE
> !(a > 8 || b > 1)
[1] FALSE
> c <- a > b
> c
[1] FALSE
> !c
[1] TRUE
```

Vectors in R
(c, seq, rep)

- Vectors are sequences of objects of the same type, usually numbers
- Vectors can be created with the `c()` (combine or concatenate) command

```
> c(3, 4.5, 6, 78)
[1]  3.0  4.5  6.0 78.0
```

Since the result is not stored in a variable, it is printed and then lost

- Other constructions:

```
> 1:5
[1] 1 2 3 4 5
> 5:1
[1] 5 4 3 2 1

> seq(1,5,by=2)
[1] 1 3 5
> seq(1,5,length=9)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

> rep(3,4)
[1] 3 3 3 3
> rep(3:7,2)
 [1] 3 4 5 6 7 3 4 5 6 7
```

Check "R quick guide" for more examples!

Some useful vector functions

`length(v)`	the number of elements
`max(v), min(v)`	the largest and smallest value
`range(v)`	equivalent to `c(min(x),max(v))`
`sort(v)`	creates a sorted vector (does not change v!)
`sum(v)`	the sum of all elements
`cumsum(v)`	the cumulative sum (a new vector)
`prod(v)`	the product of all elements
`cumprod(v)`	cumulative product (a new vector)
`mean(v), median(v)`	mean and median
`var(v), sd(v)`	variance and standard deviation
`all(v)`	logical testing if all elements of v == TRUE
`any(v)`	logical testing if any element of v == TRUE

Mathematical expressions with vectors

Basic mathematical expressions and functions are carried out element-by element in a vector.

```
> x <- 2:6
> x
[1] 2 3 4 5 6
> x + 4
[1]  6  7  8  9 10
> y <- 5:9
> y
[1] 5 6 7 8 9
> x + y
[1]  7  9 11 13 15
> x*y
[1] 10 18 28 40 54
> x^y
[1]       32      729    16384   390625 10077696
> exp(x)
[1]   7.389056  20.085537  54.598150 148.413159 403.428793
> sin(y)*cos(x)
[1]  0.3990533  0.2766192 -0.4294351  0.2806435  0.3957039
```

Indexing
([], length)

- Single elements of a vector are obtained through indexing within square brackets:

```
> v1 <- seq(1,3,by=0.5)    # create a vector
> v1                       # print out its values
[1] 1.0 1.5 2.0 2.5 3.0

> length(v1)               # check its size
[1] 5

> v1[3]                    # element number 3
[1] 2

> v1[4] <- 8               # assign a new value to element 4
> v1
[1] 1.0 1.5 2.0 8.0 3.0
```

More indexing
([])

- Extract a part of a vector: (v1 from previous slide)

```
> v1[2:4]
[1] 1.5 2.0 8.0

> v1[c(1,5,2)]
[1] 1.0 3.0 1.5

> v1[-2]   # everything but element 2
[1] 1 2 8 3

> v1[-(2:3)]
[1] 1 8 3
```

- Assigning values to a part of a vector:

```
> v1[4:5] <- -1
> v1
[1]  1.0  1.5  2.0 -1.0 -1.0
```

Indexing with logical vectors

- A logical vectors has elements that are either TRUE (T) or FALSE (F) (DON'T use T or F as variable names!)

```
> v2 <- c(T,T,T,F)
> v2
[1]  TRUE  TRUE  TRUE FALSE
```

- It can be the result of a logical expression:

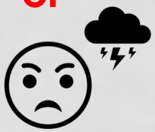
```
> x <- c(4,34,88,9)
> x < 10    # the comparison is done element-by-element
[1]  TRUE FALSE FALSE  TRUE
```

- Logical vectors can be used for indexing:

```
> x[c(T,T,F,T)] # pick out the elements corresponding to TRUE values
[1]  4 34  9
> x[x<10] <- 0 # assign value zero to the chosen elements
> x
[1]  0 34 88  0
```

Using scripts
(`source`)

- Scripts are a handy way of saving your commands, for later reference. It also simplifies using complicated commands tremendously.
- Workflow:
  - Edit the script, save
  - Run it (use `source` command in console window, or use keyboard shortcut (SHIFT+⌘+S on mac or SHIFT+CTRL+S on windows). **Avoid “run” or copy+paste in the console!**
  - Check results, edit again, a.s.o.



- Running a script is (almost) equivalent to typing the same sequence of commands in the console.

Why does "run" or copy+paste in the console is not appropriate if programming?

- It messes up output with code, making it difficult to track what has been done. What were the parameter values?
- The next day, do you know how you solved the problem?
- "Run the current selection" is using R as an advanced calculator, it is not programming.
  Programs are sequences of commands that together perform some well defined task.
  You will never learn programming unless you start writing programs.
- <u>From now on</u>: 1 task ⇔ 1 script (or more)

EXERCISE

(`cat`)

- Write a script that:
    - Creates a vector v1 with the values 1, 1.5, 2, 2.5, …,7
    - Creates another vector v2 with the 3rd, 4th and 11th elements of v1
    - Assigns zero to all elements of v2 larger than 2.
    - Prints out v2 in the console (use '`cat(v2)`')
- Run the script using '`source`' in the console
- If the output is *not*

  `2 0 0`

  , correct the script, save and run (source!!!) it again.

More on vectors
(`+`, `-`, `*`, `/`, `^`, `NA`)

- Assigning can extend a vector (`NA` = missing value)

```
> x <- 1:3
> x
[1] 1 2 3
> x[6] <- 99
> x
[1]  1  2  3 NA NA 99
```

- Vector arithmetics (element-by-element)

```
> x + c(3,5,1,1,0,1)
[1]   4   7   4  NA  NA 100
> x * c(3,5,1,1,0,1)
[1]  3 10  3 NA NA 99
```

- Vectors are *recycled* when necessary!

```
> x + c(1,2)
[1]   2   4   4  NA  NA 101
```

Strings = character objects
("", '', mode, paste, substr)

```
> s1 <- "Hi there" # or 'Hi there'
> s1
[1] "Hi there"

> mode(s1)
[1] "character"

> s1[2] <- "students" # adding an element will increase the length of s1
> s1
[1] "Hi there" "students" # s1 is now a character vector with two elements

> paste('Hi there', ', ', 'students', sep = '')
[1] "Hi there, students"
> # use paste to merge strings (in many, many, different ways)

> substr(s1[1],1,4) # use substr to access parts of a string
[1] "Hi t"
> substr(s1[1],1,4) <- 'Dude'
> s1
[1] "Dudehere" "students"

> nchar(s1[1]) # returns the number of characters in s1[1]
[1] 8
```

Lists

(list, $, [[]])

- Lists are like vectors, an indexed set of objects, *components*, but the components need not be of the same type.
- The components of lists are often named. It is easier to recall a name rather than an index. Names also makes the code easier to interpret.
- Use the '$' operator to access named components, or [[double brackets]] for indexing.

```
> mylist <- list(car="volvo", weight=9,
"unnamed")
> mylist
$car
[1] "volvo"

$weight
[1] 9

[[3]]
[1] "unnamed"

> mylist$car
[1] "volvo"
> mylist[[1]]
[1] "volvo"
> mylist[[3]]
[1] "unnamed"

> y <- list(1:5)
> y
[[1]]
[1] 1 2 3 4 5

> y[[1]]
[1] 1 2 3 4 5
```

Create a list from a vector

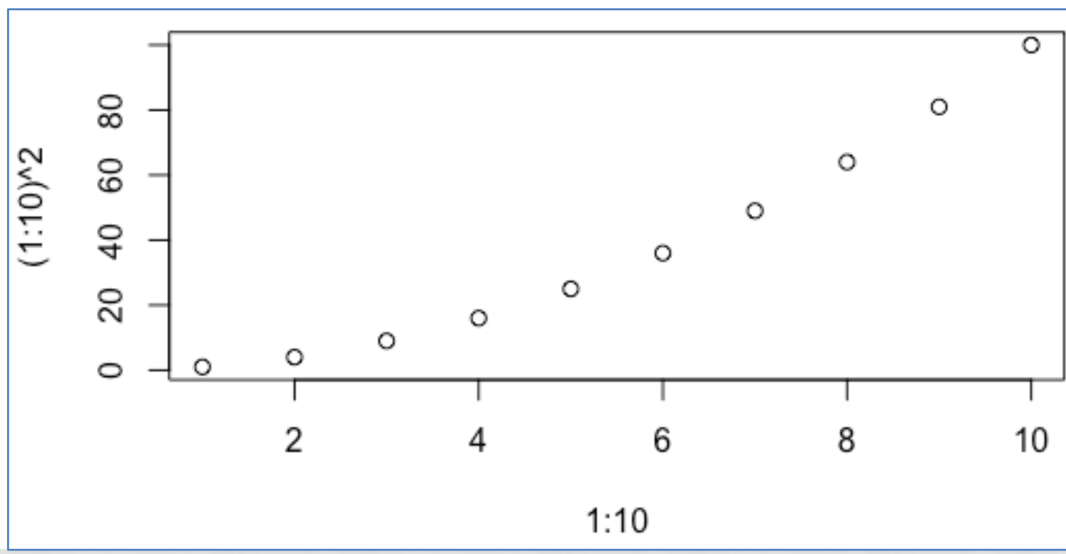
y has only one component

Conversions ('coercions') between types (modes)

```
> s <- "12"
> as.numeric(s)
[1] 12

> as.character(43)
[1] "43"

> as.character(pi)
[1] "3.14159265358979"

> as.integer(3.9)
[1] 3

> round(3.9)
[1] 4

> as.integer("549ppp")
[1] NA
Warning message:
NAs introduced by coercion
```

Simple input and output
(readline, scan, cat, plot, line, points)

- `x <- readline("Type x:")` `# For input of strings.`
  `# Convert to numbers if necessary.`
  `# 'scan()' is also useful for input.`
- `cat(x)` `# Outputs x in console window`
- `plot(x,y)` `# Plots y vs. x in plot window`
- `> plot(1:10, (1:10)^2)`
- `line` and `points` will add to an existing graph



Flow control

- If-statements
- While, repeat (and break)
- For-loops

The `if` statement
(`if, else`)

- The `if` statement is used for conditional expressions or commands

`if (` *`condition`* `)` *`expression`*

or

`if ( condition )` *`expression1`* `else` *`expression2`*

```
> x <- rnorm(1)
> if (x < 0) x <- -x
> x
[1] 0.3441755

> x <- runif(1)
> x
[1] 0.846157
> y <- runif(1)
> y
[1] 0.6319267
> if (x > y) z <- x else z <- y
> z
[1] 0.846157
```

The `if` statement, cont'd ({ })

- The expressions of an if statement are often *compound*, or *grouped*, i.e. a sequence of commands within {curly brackets}

```
print("Enter a number")
x <- scan(nmax=1)    # scan can be used for keyboard input of
numbers, but also input from files

cat("You entered:", x , fill=TRUE) # cat is useful for
combining and printing output as text

if (x >= 0) {
    cat("It is a positive number\n")
    cat("Its double value is", 2*x, "\n")
} else {
    cat("It is a negative number", fill=T)
    cat("Half its value is", x/2, fill=T)
}
```

Use one of these to get a new line

while and repeat
(while, repeat, break)

- while and repeat are used to repeat a command, or a sequence of commands, several times

Exercise: Generate a random number from a geometric distribution
It can be interpreted as the number of times you have to roll a die *before* you get a certain result, e.g. before you get a 6.

```
n <- 0               # starting value
x <- runif(1)        # roll the die!
while (x < 5/6) {    # while failure...
    n <- n+1         # increase the count
    x <- runif(1)    # roll again!
}
cat("The number is", n, fill=F)
```

```
n <- 0 # starting value
repeat {
    x <- runif(1)    # roll the die
    if (x < 5/6) {   # if failure
        n <- n+1     # increase the count
    } else {         # else
        break        # exit repeat
    }
}
cat("The number is", n, fill=F)
```

This is necessary to break the loop!

`for`-loops: if you know how many times something will be repeated

`for ( name in expression1 ) expression2`

If *expression1* is a vector, the loop-variable *name* is given its values, one at a time

```
> for (i in 1:3) print(1:i)
[1] 1
[1] 1 2
[1] 1 2 3
```

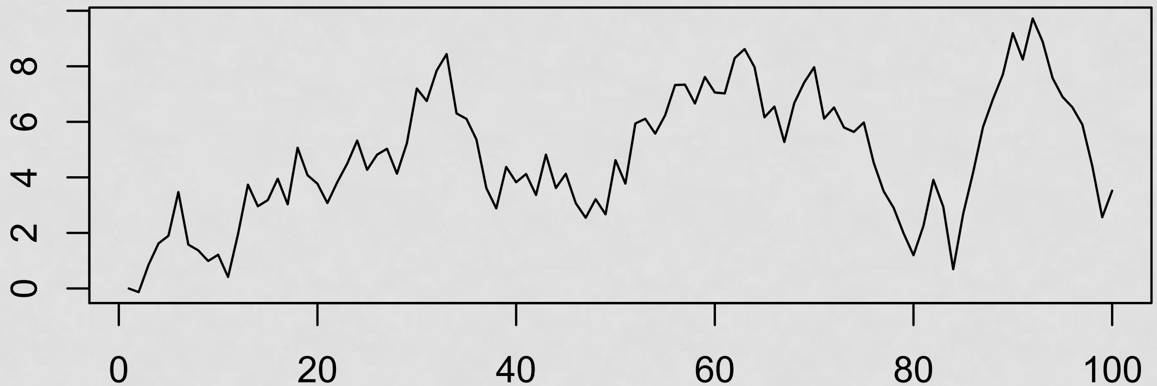
```
> str <- c('a','vector','of','strings')
> str
[1] "a"       "vector"  "of"      "strings"
> for (s in str) cat(substr(s,1,3),' ')
a  vec  of  str
```

for-loops, cont'd
(rnorm)

```
# a sample script to generate random numbers and plot local optima

x <- rnorm(20)              # generate 20 random numbers
plot(1:20, x, type='b') # 'b' stands for 'both', i.e. line + points
for (i in 2:19) {
    if (x[i-1] < x[i] && x[i+1]<x[i]) { # is x[i] a local max?
        cat('Found maximum at', i, fill=T)
        points(i, x[i], col='red') # mark the maximum
    }
}
```

Exercise: Generate a random walk and plot it

- Definition of a random walk:
  - x[1] = 0
  - x[i+1] = x[i] + rnorm(1)



- *Task:* Write a script that inputs the length of the walk, generates a sample time series, and plots it as a solid line (`plot(x,type='l')`)

- *Extra:* generate 10 sample time series and plot them all in the same graph