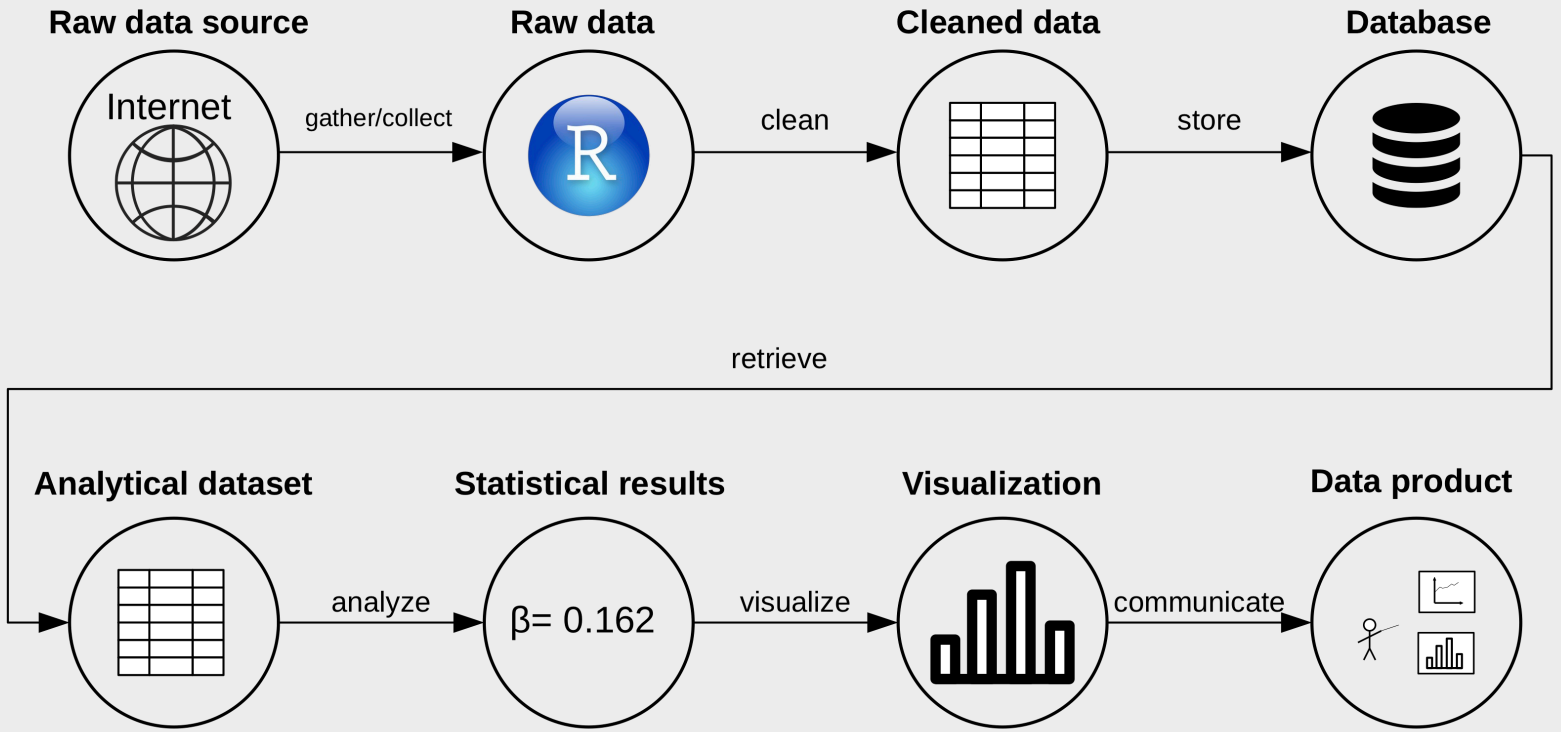
Data Handling: Import, Cleaning and Visualisation

Lecture 2: Programming with Data

Dr. Aurélien Sallin
26/09/2024

Recap

Data (science) pipeline



At the end of the course, you will be able to...

- Understand the tools you need when working with data
- Work independently with data
- Ask the right questions to a dataset
- Learn to communicate about data

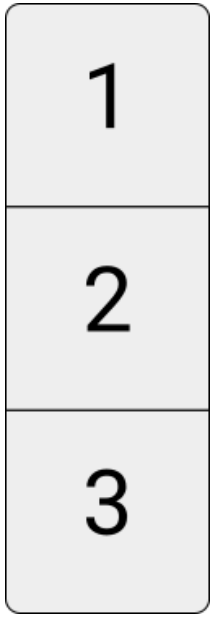
A shoemaker and his apprentice c.1914, Emile Adan

Basic Programming Concepts

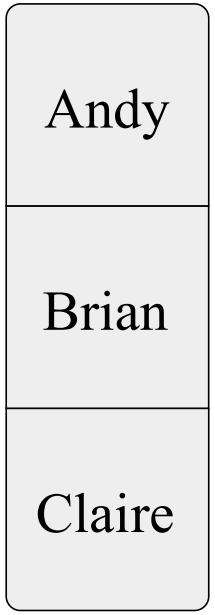
Values and variables

Vectors

Vectors



Vectors



Matrices

Matrices

Matrices are combinations of vectors

```r
cbind(c(1,2,3), c(4,5,6), c(7,8,9))
rbind(c(1,4,7), c(2,5,8), c(3,6,9))
matrix(nrow=3, ncol = 3, 1:9)
```

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

```r
cbind(c(1,2,3), c(4,5,6), c(7,8,9))
rbind(c(1,4,7), c(2,5,8), c(3,6,9))
matrix(nrow=3, ncol = 3, 1:9)
```

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

```r
cbind(c(1,2,3), c(4,5,6), c(7,8,9))
rbind(c(1,4,7), c(2,5,8), c(3,6,9))
matrix(nrow=3, ncol = 3, 1:9)
```

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

Math operators

Math operators: basic arithmetic

```
# basic arithmetic
2+2
sum_result <- 2+2
sum_result
sum_result -2
4*5
20/5

# Modulo (remainder)
5 %% 3

# Integral division
5 %/% 3
```

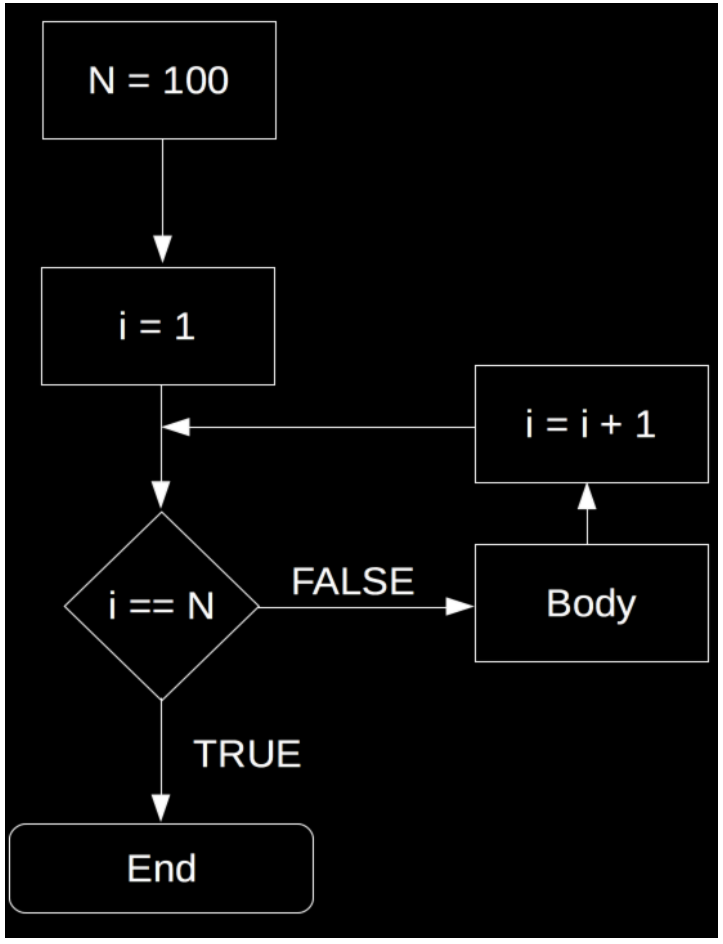
```
## [1] 4

## [1] 4

## [1] 2

## [1] 20

## [1] 4

## [1] 2

## [1] 1
```

Math operators: other operators

```
# other common math operators and functions
4^2
sqrt(4^2)
log(2)
exp(10)
log(exp(10))
```

Loops

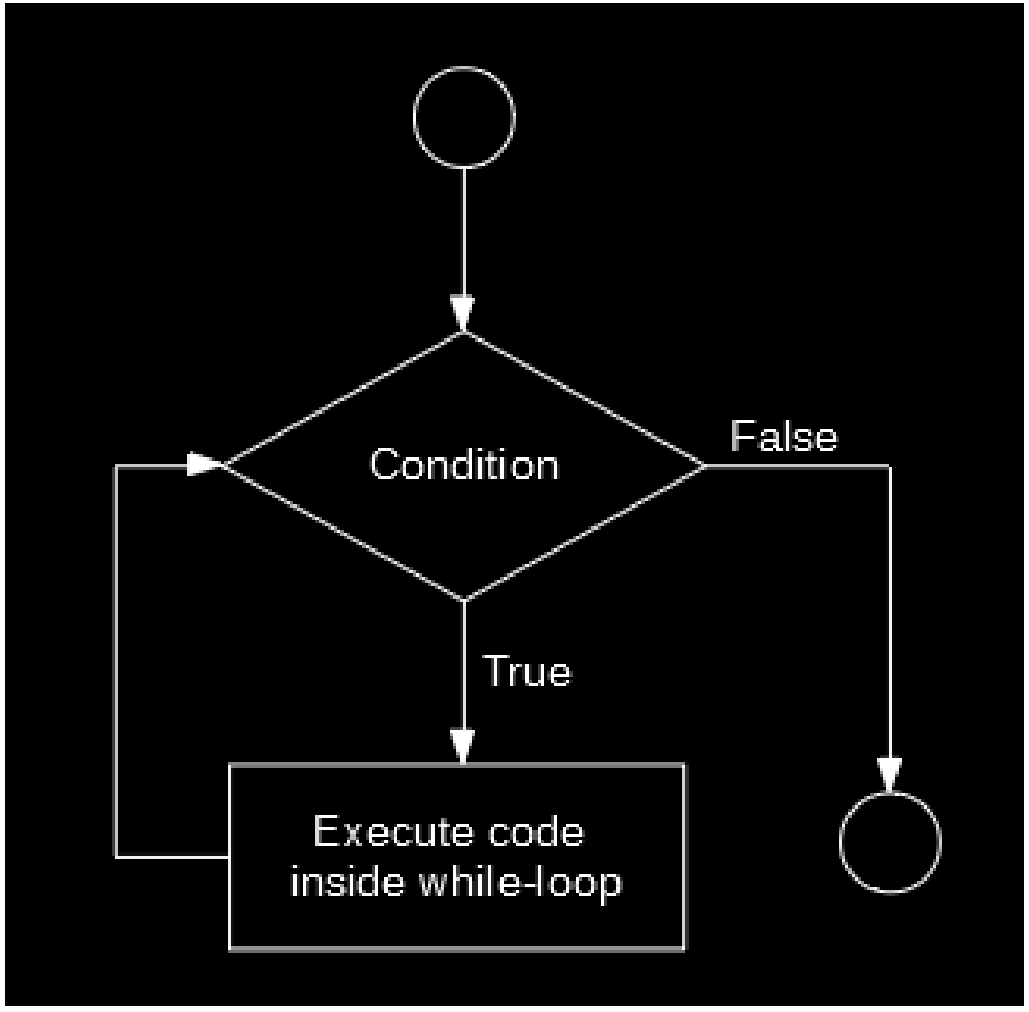
for-loop



for-loop

```r
# number of iterations

n <- 100

# start loop
for (i in 1:n) {

  # BODY

}
```

while-loop



while-loop

```r
# initiate variable for logical statement
x <- 1

# start loop
while (x == 1) {

  # BODY

}
```

Logical statements

Logical statements

```
2+2 == 4 # is equal to
3+3 == 7
4!=7 # is not equal to
6>3
6<7
6<=6
```

Control statements

Control statements

```
condition <- TRUE

if (condition) {
  print("This is true!")
} else {
  print("This is false!")
}
```

```
## [1] "This is true!"
```

Functions

Functions

$$f : X \to Y$$

Functions

$$2 \times X = Y$$

Functions in R

Functions in R are either **built-in** or **user-defined**.

Load built-in functions from a R-package:

```
# install a package
install.packages("<PACKAGE NAME>")

# load a package
library(<PACKAGE NAME>)
```

Functions in R

Functions have three elements:

1. *formals()*, the list of arguments that control how you call the function
2. *body()*, the code inside the function
3. *environment()*, the data structure that determines how the function finds the values associated with the names (not the focus of this course)

Functions in R

```
myfun <- function(x, y){

  # BODY
  z <- x + y

  # What the function returns
  return(z)
}
```

Formals

```
formals(myfun)
```

```
## $x
##
##
## $y
```

Functions in R

```
myfun <- function(x, y){

  # BODY
  z <- x + y

  # What the function returns
  return(z)
}
```

Body

```
body(myfun)
```

```
## {
##     z <- x + y
##     return(z)
## }
```

Functions in R

```
myfun <- function(x, y){

  # BODY
  z <- x + y

  # What the function returns
  return(z)
}
```

Environment

```
environment(myfun)
```

```
## <environment: R_GlobalEnv>
```

Functions in R

Example of a function: a simple power function

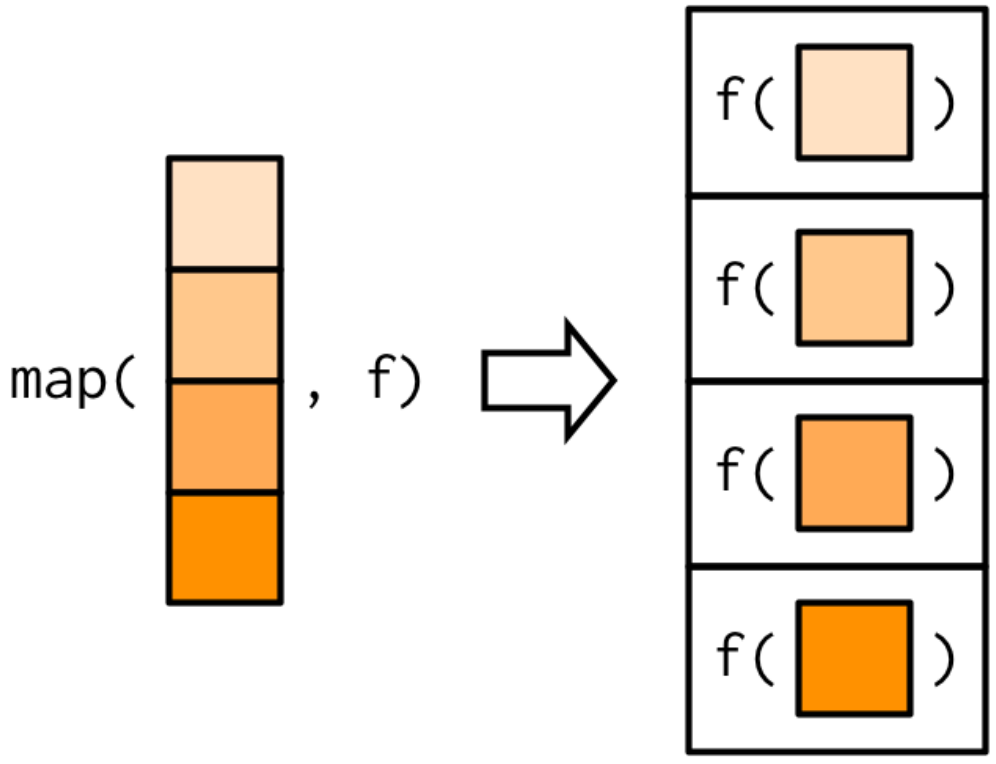
```
powerFunction <- function(base, exponent){

  results <- base ^ exponent

  return(results)
}

powerFunction(exponent = 2, base = 3)
powerFunction(base = 2, exponent = 3)
powerFunction(2, 3)
powerFunction(c(2,4,3), 3)
```

Step-up your game: Functionals

Functionals

- A functional is a function that takes a function as an input and returns a vector or a list as output.
- Functionals are alternative to for-loops.
- Functionals can be programmed using the `apply` family (`apply`, `lapply`, `tapply`), or the `purrr::map()` family.

Functionals: representation with `map()`



Example of a functional

```
# Install purrr and load
# library(purrr)

# Set a user-defined function
triple <- function(x) x * 3

# With lapply
lapply(1:3, triple)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 9
```

```
# Apply the function to a vector. map_dbl Returns a vector
map_dbl(1:3, triple)
```

```
## [1] 3 6 9
```

The "apply" functional

`apply` applies a function to columns or rows of a matrix. It loops over rows (`MARGIN = 1`) or columns (`MARGIN = 2`) of a matrix.

```
# Empty matrix with 2 rows and 4 columns
mymatrix <- matrix(c(1,2,3,11,12,13,1,10),
                   nrow = 2,
                   ncol = 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3   12    1
## [2,]    2   11   13   10
```

Apply a sum function on each column

```
apply(mymatrix, MARGIN = 2, sum)
```

```
## [1]  3 14 25 11
```

Recap

Loops and functionals

for-loops	functionals
Gets messy fast	Elegant and readable, allow for better syntax
Long code	Short code, based on functions
Standard base R	`apply` family: standard R. `map` family: `purrr` package

Tutorials

Tutorial 1: A Function to Compute the Mean

Starting point: we should be aware of how the mean is defined:

$$\bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right) = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

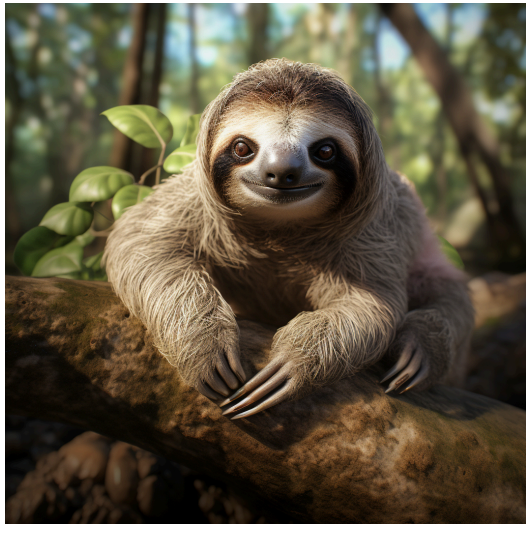
.

Tutorial 1: A Function to Compute the Mean

```
#####################################
# Mean Function:
# Computes the mean, given a
# numeric vector.

meaN <- function(x){

}
```

Tutorial 2: on slow and fast sloths

We can use loops to simulate natural processes over time. Write a program that calculates the populations of two kinds of sloths over time. At the beginning of year 1, there are **1000 slow sloths** and **1 fast sloth**. This one fast sloth is a new mutation that is genetically able to use roller blades. Not surprisingly, being fast gives it an advantage, as it can better escape from predators.

A slow sloth	A fast sloth in its natural element
	

Tutorial 2: on slow and fast sloths

Each year, each sloth has one offspring. There are no further mutations, so slow sloths beget slow sloths, and fast sloths beget fast sloths. Also, each year 40% of all slow sloths die each year, while only 30% of the fast sloths do.

At the beginning of year one there are 1000 slow sloths. Another 1000 slow sloths are born. But, 40% of these 2000 slow sloths die, leaving a total of 1200 at the end of year one. Meanwhile, in the same year, we begin with 1 fast sloth, 1 more is born, and 30% of these die, leaving 1.4.

Beginning of Year	Slow Sloths	Fast Sloths
1	1000	1
2	1200	1.4
3	1440	1.96

Enter the first year in which the fast sloths outnumber the slow sloths.

Tutorial 3: A loop function

Be the function

```
appendsums <- function(lst){

  #Repeatedly append the sum of the current last three elements
  #of lst to lst.
}
```

Create a function that repeatedly appends the sum of the current last three elements of the vector lst to lst. Hint: use the `append` and `tail` functions. Your function should loop 25 times. To check if your function is correct, run:

```
sum_three = c(0, 1, 2)
appendsums(sum_three)

# Solution for testing:
sum_three[10] == 125
```

Q&A