

R Programming

Dr. Kapil Kumar

Associate Professor, Department of Statistics
Chaudhary Charan Singh University, Meerut, India
kapilstats@gmail.com

Extension lecture

Department of Mathematics, Chaudhary Bansi Lal University, Bhiwani

18 October 2025 · Eminent Speaker

Plan of the session

1. R as a programming language
2. Data structures and indexing
3. Control flow
4. Writing functions
5. Vectorisation and the apply family
6. A data-cleaning task, and plotting

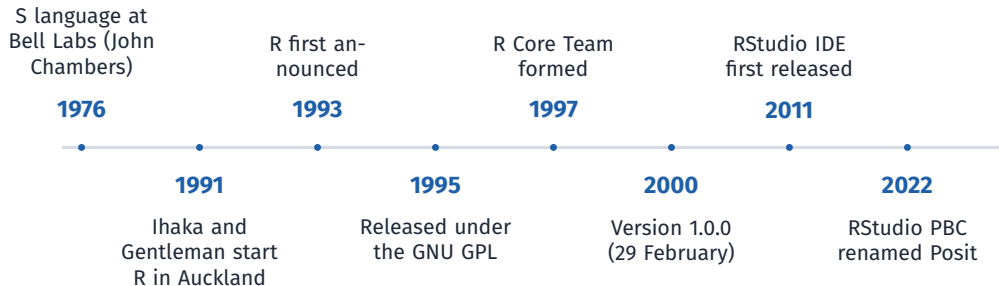
By the end you can choose the right data structure and index it, steer a computation with `if` and loops, write functions (including recursive ones and closures), replace loops by vectorised code, clean a real data file and build a plot layer by layer.

For students and faculty of mathematics: we treat R as a language, not as a menu of statistical procedures.

PART 1

R as a language

From S to R: a short timeline



What R is A free language and environment for computing and graphics, with about 22,000 CRAN packages.

Four ideas behind the language

1. **Everything is an object**, including functions
2. **Every action is a function call**, even `+` and `[`
3. **There are no scalars**: a single number is a vector of length 1, so most operations are vectorised
4. **Lexical scoping**: a function looks up free variables where it was defined

```
`+`(2, 3)           # operator as a function
x <- 7
length(x)           # a vector of length 1
class(mean)         # functions are objects
sapply(list(1L, 2.5, "a", TRUE, 1 + 2i),
        typeof)      # the basic types

## [1] 5
## [1] 1
## [1] "function"
## [1] "integer"  "double"  "character"
## [4] "logical"  "complex"
```

Numbers on a computer

```
0.1 + 0.2 == 0.3           # binary rounding!
isTRUE(all.equal(0.1 + 0.2, 0.3))
sqrt(2)^2 - 2
.Machine$double.eps        # machine epsilon
c(-7 %/% 2, -7 %% 2)        # floor division

## [1] FALSE
## [1] TRUE
## [1] 4.441e-16
## [1] 2.22e-16
## [1] -4  1
```

- Doubles carry about 16 significant digits; compare them with a tolerance, using `all.equal()`, never with `==`
- `%/%` and `%%` follow the floor convention, so $x = y (x \%/\% y) + x \% \% y$ also for negative x
- Integers are written 5L; the largest is 2,147,483,647

Complex numbers and polynomial roots

```
z <- complex(real = 2, imaginary = 3)
c(modulus = Mod(z), argument = Arg(z))
Conj(z)
z * Conj(z)           # |z|^2
sqrt(as.complex(-1))  # i, not NaN
polyroot(c(6, -5, 1)) # 6 - 5x + x^2 = 0

## modulus argument
## 3.6056 0.9828
## [1] 2-3i
## [1] 13+0i
## [1] 0+1i
## [1] 2+0i 3+0i
```

$$z = a + bi: |z| = \sqrt{a^2 + b^2}$$

$$\arg z = \text{atan2}(b, a)$$

$$z\bar{z} = |z|^2 = 13$$

`sqrt(-1)` returns NaN with a warning: R will not leave the real line unless asked.
`polyroot()` takes coefficients in *increasing* powers and returns complex roots.

Packages and data for this session

File	Contents
titanic.csv	891 Titanic passengers
Iris.csv	150 iris flowers (UCI copy)
data.csv	two small columns
data.txt	tab-separated, 6 students
Data_Excel.xlsx	Excel sheet, 5 students

The three small files are illustrative; student names were replaced by codes.

```
# install.packages(c("readxl", "moments"))  
library(readxl)      # Excel files  
library(moments)     # skewness check  
col_main <- "#1F5FA8" # plot colours  
col_alt  <- "#B45309"
```

Everything else is base R.

PART 2

Data structures and indexing

Five data structures

Dimensions	One type	Mixed types
1	atomic vector	list
2	matrix	data frame
n	array	

A **factor** is an integer vector with a set of labels (levels), used for categories.

Choosing one Numbers for linear algebra: matrix. A table of variables: data frame. Results of different shapes: list. One variable: vector or factor.

Tools that work on all `str()`, `class()`, `typeof()`, `length()`, `dim()`, `names()`

Atomic vectors and coercion

```
vec <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
c(sum(vec), prod(vec))
typeof(c(TRUE, 2L))      # logical -> integer
typeof(c(1L, 2.5))       # integer -> double
c(1, "a", TRUE)          # all -> character
log_vec <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
sum(log_vec); mean(log_vec) # count, proportion

## [1]      55 3628800
## [1] "integer"
## [1] "double"
## [1] "1"      "a"      "TRUE"
## [1] 3
## [1] 0.6
```

Coercion order logical < integer
< double < complex < character.
Mixing types silently promotes
everything to the highest one.

TRUE counts as 1 and FALSE as 0, so
`sum()` counts and `mean()` gives a
proportion: a trick used constantly.

Six ways to index a vector

```
nums <- c(5, 10, 15, 20, 25)
nums[c(1, 3)]           # by position
nums[nums > 10 & nums < 25] # by a condition
which(nums > 12)         # positions
nums[nums > 20] <- 0      # assign by index
names(nums) <- c("a", "b", "c", "d", "e")
nums[c("b", "e")]        # by name

## [1]  5 15
## [1] 15 20
## [1]  3  4  5
##   b  e
## 10  0
```

- `[ ]` with positive, negative (`nums[-1]` drops the first), logical or character indices
- `which()` converts a logical vector into positions
- Indexing on the left of `<-` *changes* the selected elements: the basis of most data cleaning

Careful Indices start at 1. `nums[0]` is an empty vector and `nums[6]` is NA: neither gives an error.

Matrices

```
mat <- matrix(1:16, nrow = 4, ncol = 4)
mat                # filled column by column
mat[2, 3]          # row 2, column 3
mat[, 2]           # column 2, as a vector

##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
## [1] 10
## [1] 5 6 7 8
```

```
(mat * mat)[1:2, 1:2]      # element-wise
(mat %*% t(mat))[1:2, 1:2] # matrix product

##      [,1] [,2]
## [1,]    1   25
## [2,]    4   36
##      [,1] [,2]
## [1,]  276  304
## [2,]  304  336
```

`matrix()` fills by column unless `byrow = TRUE`; `drop = FALSE` keeps a single column a matrix.

Arrays: more than two dimensions

```
arr <- array(1:24, dim = c(3, 4, 2))
arr[2, 3, 1]           # row 2, column 3, layer 1
arr[, , 2]             # the whole second layer
apply(arr, c(1, 2), sum) # add the two layers

## [1] 8
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
##      [,1] [,2] [,3] [,4]
## [1,]   14   20   26   32
## [2,]   16   22   28   34
## [3,]   18   24   30   36
```

- An array is a vector with a `dim` attribute; a matrix is the 2-d case
- Index with one position per dimension
- `apply()` with `MARGIN = c(1, 2)` keeps rows and columns and collapses the third dimension

Uses: images (height $\times$ width $\times$ colour),
panel data, simulation results
(parameter $\times$ replicate $\times$ statistic).

Lists: one container, many shapes

```
student_info <- list(  
  Name = "Student A", Age = 22,  
  Scores = c(88, 92, 85), Passed = TRUE,  
  Details = data.frame(Subject = c("Math", "English"),  
                        Score = c(95, 85)))  
  
student_info$Scores      # an element  
student_info[["Details"]] # the same with [[ ]]  
class(student_info["Scores"]) # [ ] keeps a list  
  
## [1] 88 92 85  
##   Subject Score  
## 1    Math    95  
## 2 English    85  
## [1] "list"
```

[[]] or []? `[[ ]]` and `$` take an element *out* of the list; `[ ]` returns a smaller list.

Lists hold results of different shapes: model fits, simulation output, the parts of a parsed file. A data frame is a list of equal-length columns.

Data frames: indexing rows and columns

```
states <- data.frame(  
  State = c("Uttar Pradesh", "Maharashtra", "Bihar",  
            "West Bengal", "Madhya Pradesh"),  
  Pop_mn = c(231, 124, 130, 100, 87),  
  Literacy = c(69.7, 82.3, 70.9, 77.1, 73.2))  
states[states$Literacy > 75, c("State", "Pop_mn")]  
states$State[order(-states$Pop_mn)][1:2] # largest two  
states$Big <- states$Pop_mn > 120        # new column  
  
##           State Pop_mn  
## 2 Maharashtra    124  
## 4 West Bengal    100  
## [1] "Uttar Pradesh" "Bihar"
```

Illustrative figures for five states (population in millions, literacy in %), not official statistics.

`df[rows, columns]` Rows by a logical condition, columns by name; an empty position means “all”. `order()` sorts; a minus sign sorts in decreasing order.

Factors and binning with `cut()`

```
department <- factor(c("HR", "Finance", "IT",
                       "HR", "IT"))
as.integer(department)      # stored as codes
table(department)
score <- c(50, 75, 82, 90, 60)
cut(score, breaks = c(0, 60, 80, 100),
     labels = c("Low", "Medium", "High"))

## [1] 2 1 3 2 3
## department
## Finance      HR      IT
##           1      2      2
## [1] Low      Medium High   High   Low
## Levels: Low Medium High
```

- Levels are sorted alphabetically unless you give levels =
- `cut()` turns a number into a category; intervals are $(a, b]$ by default, so 60 is “Low” and 80 “Medium”
- Ordered factors (`ordered = TRUE`) allow `<` and `>`

PART 3

Control flow

Logical and comparison operators

```
x <- c(TRUE, FALSE, TRUE)
y <- c(TRUE, TRUE, FALSE)
x & y                # element-wise AND
x | y                # element-wise OR
c(any(x), all(x), xor(TRUE, FALSE))
3 %in% c(1, 3, 5)
TRUE && FALSE        # single values only

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

== != < > <= >=	comparison
& !	vectorised logic
&&	one value, for if
any(), all(), xor()	summaries
%in%	membership

&& and || stop as soon as the answer is known; since R 4.3 they give an error for vectors longer than one.

if, else if and else

```
marks <- 72
if (marks >= 90) {
  grade <- "A"
} else if (marks >= 75) {
  grade <- "B"
} else if (marks >= 60) {
  grade <- "C"
} else {
  grade <- "D"
}
grade

## [1] "C"
```

- Conditions are checked from the top; the first TRUE wins
- The condition must be a *single* TRUE or FALSE
- else goes on the same line as the closing brace

Common slip if (x = 5) is an error; comparison is ==. And if (NA) fails: test `is.na()` first.

Vectorised decisions: `ifelse()` versus `cut()`

```
grades <- data.frame(Score = c(50, 75, 82, 90, 60))
grades$Grade <- ifelse(grades$Score >= 80, "A",
                      ifelse(grades$Score >= 60, "B", "C"))
grades$Grade2 <- cut(grades$Score, right = FALSE,
                    breaks = c(-Inf, 60, 80, Inf),
                    labels = c("C", "B", "A"))
```

grades

##	Score	Grade	Grade2
## 1	50	C	C
## 2	75	B	B
## 3	82	A	A
## 4	90	A	A
## 5	60	B	B

`ifelse(test, yes, no)` works on a whole vector at once, so it is the vectorised partner of `if`.

Same answer, cleaner code

Nested `ifelse()` grows hard to read. `cut()` with `right = FALSE` uses intervals $[a, b)$ and gives the same grades.

for loops

```
factorials <- numeric(5)    # allocate first
for (i in 1:5) {
  fact <- 1
  for (j in 1:i) {
    fact <- fact * j
  }
  factorials[i] <- fact
}
factorials
cumprod(1:5)                # vectorised check

## [1] 1 2 6 24 120
## [1] 1 2 6 24 120
```

- `for (i in seq)` runs once for each element of `seq`
- Create the result vector *before* the loop (`numeric()`, `character()`, `vector("list", n)`); growing it inside is slow
- Prefer `seq_along(x)` to `1:length(x)`, which fails when `x` is empty

while: repeat until a condition fails

```
# add 1 + 2 + ... until the total exceeds 20
sum_val <- 0
num <- 0
while (sum_val <= 20) {
  num <- num + 1
  sum_val <- sum_val + num
}
c(last_term = num, total = sum_val)

## last_term      total
##           6      21
```

```
# e = sum of 1/k! until terms are tiny
term <- 1; total <- 1; k <- 0
while (term > 1e-10) {
  k <- k + 1
  term <- term / k
  total <- total + term
}
k                                # terms used
total - exp(1)                   # error

## [1] 14
## [1] -8.149e-13
```

A while loop suits iterations whose length is unknown in advance: series, fixed-point and Newton iterations. Make sure the condition can become FALSE.

repeat, break and next

```
i <- 1
repeat {                                # first i with i^2 > 50
  if (i^2 > 50) break
  i <- i + 1
}
i

## [1] 8
```

```
evens <- c()
for (i in 1:10) {
  if (i %% 2 != 0) next # skip odd numbers
  evens <- c(evens, i)
}
evens

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

In short repeat runs until a break; break leaves any loop at once; next skips to the next iteration. (For the second example, `(1:10)[(1:10) %% 2 == 0]` does the same without a loop.)

PART 4

Writing functions

Anatomy of a function

```
add_numbers <- function(a, b) {  
  a + b                # last value is returned  
}  
add_numbers(b = 25, a = 10)  # match by name  
greet <- function(name = "Guest") {  
  paste("Hello,", name)  
}  
greet()                  # the default is used  
greet("Alice")  
  
## [1] 35  
## [1] "Hello, Guest"  
## [1] "Hello, Alice"
```

Parts arguments (with optional defaults), a **body**, and a **value**: the last expression evaluated, or what `return()` gives.

Arguments are matched by name first, then by position. Defaults make a function convenient without hiding its options. `args(greet)` shows them.

Scope: where does a function find a variable?

```
x <- 10
check_scope <- function() {
  x <- 5          # a local x
  x
}
check_scope()
x                # global x unchanged

## [1] 5
## [1] 10
```

```
x <- 1
f <- function() {
  y <- 2
  g <- function() x + y + 3 # y from f, x global
  g()
}
f()

## [1] 6
```

Lexical scoping A function looks first among its own variables, then where it was *defined*, then outwards to the global environment.

Functions that make functions: closures

```
make_power <- function(n) {  
  function(x) x^n          # remembers its n  
}  
square <- make_power(2)  
cube <- make_power(3)  
square(4)  
cube(2)  
sapply(1:5, cube)  
  
## [1] 16  
## [1] 8  
## [1] 1 8 27 64 125
```

`make_power` maps $n \mapsto (x \mapsto x^n)$: a function whose value is a function.

Each call creates a new environment holding its own n ; the returned function keeps that environment. Uses: likelihoods with fixed data, families of kernels, counters.

Recursion

```
factorial_rec <- function(n) {  
  if (n == 0) return(1)           # base case  
  n * factorial_rec(n - 1)  
}  
fibonacci <- function(n) {  
  if (n <= 1) return(n)          # F0 = 0, F1 = 1  
  fibonacci(n - 1) + fibonacci(n - 2)  
}  
c(factorial_rec(5), factorial(5))  
sapply(0:10, fibonacci)  
  
## [1] 120 120  
## [1] 0 1 1 2 3 5 8 13 21 34 55
```

$$n! = n \cdot (n - 1)!, \quad 0! = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0, F_1 = 1$$

A recursive function needs a **base case** that stops the calls and a step that moves towards it. The code mirrors the mathematics line by line.

Recursion versus iteration

```
fib_iter <- function(n) {  
  if (n <= 1) return(n)  
  a <- 0; b <- 1  
  for (i in 2:n) {  
    s <- a + b; a <- b; b <- s  
  }  
  b  
}  
  
c(fib_iter(25), fibonacci(25)) # same value  
2 * fib_iter(26) - 1 # calls made by fibonacci(25)  
  
## [1] 75025 75025  
## [1] 242785
```

Reading it The recursive version recomputes the same values again and again: 242,785 function calls for F_{25} , against 24 additions in the loop.

R does not optimise tail calls, so very deep recursion runs into its nesting limit. Recursion is elegant for trees and divide-and-conquer; loops are safer for long sequences.

Writing statistical functions: skewness and kurtosis

```
skewness_manual <- function(x) {  
  n <- length(x); z <- (x - mean(x)) / sd(x)  
  sum(z^3) * n / ((n - 1) * (n - 2))  
}  
kurtosis_manual <- function(x) {  
  n <- length(x); z <- (x - mean(x)) / sd(x)  
  a <- n * (n + 1) / ((n - 1) * (n - 2) * (n - 3))  
  a * sum(z^4) - 3 * (n - 1)^2 / ((n - 2) * (n - 3))  
}  
x <- mtcars$mpg      # G1 and G2 (ours), g1 (moments):  
c(skewness_manual(x), kurtosis_manual(x), skewness(x))  
  
## [1]  0.67238 -0.02201  0.64044
```

With $z_i = (x_i - \bar{x})/s$:

$$G_1 = \frac{n}{(n-1)(n-2)} \sum z_i^3$$

$$G_2 = a \sum z_i^4 - b, \text{ with}$$

$$a = \frac{n(n+1)}{(n-1)(n-2)(n-3)}$$

$$b = \frac{3(n-1)^2}{(n-2)(n-3)}$$

Bias-adjusted, as in Excel's SKEW and KURT; **moments** reports the plain ratio g_1 .

A mode function, and its edge cases

```
get_mode <- function(x) {  
  ux <- unique(x)  
  counts <- tabulate(match(x, ux))  
  ux[which.max(counts)]  
}  
get_mode(c(5, 5, 6, 7, 8, 5, 6))  
get_mode(c("it", "the", "it", "a")) # text works  
get_mode(c(3, 3, 6, 6)) # tie: first one wins  
  
## [1] 5  
## [1] "it"  
## [1] 3
```

- `match()` gives each value's position in `ux`; `tabulate()` counts those positions
- `which.max()` returns the *first* maximum, so ties are hidden

Improve it Return *all* modes: `ux[counts == max(counts)]`. Test every function on edge cases: ties, NA, empty input.

PART 5

Vectorisation and apply

Think in vectors

```
n <- 1000
total <- 0 # loop version
for (i in 1:n) total <- total + i^2
total
sum((1:n)^2) # vectorised
n * (n + 1) * (2 * n + 1) / 6 # closed form
set.seed(42)
u <- runif(10000)
mean(u < 0.25) # a proportion

## [1] 333833500
## [1] 333833500
## [1] 333833500
## [1] 0.254
```

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

Vectorised code is shorter, closer to the mathematics and much faster, because the loop runs in compiled C inside `sum()`. A logical vector averaged gives a proportion: here an estimate of $P(U < 0.25) = 0.25$ for $U \sim U(0, 1)$.

lapply(), sapply() and vapply()

```
numbers <- list(1, 2, 3, 4)
squares <- lapply(numbers, function(x) x^2)
class(squares)
unlist(squares)
sapply(numbers, function(x) x * 2)
vapply(iris[1:4], mean, FUN.VALUE = numeric(1))

## [1] "list"
## [1] 1 4 9 16
## [1] 2 4 6 8
## Sepal.Length Sepal.Width Petal.Length Petal.Width
##          5.843          3.057          3.758          1.199
```

Which one? `lapply()` always returns a list. `sapply()` simplifies to a vector or matrix when it can. `vapply()` states the type of each result, so it fails loudly instead of returning a surprise.

`function(x) x^2` is an **anonymous function**: written where it is used, never named.

apply() for margins, tapply() for groups

```
apply(mat, 1, max)           # row maxima
means <- tapply(mtcars$mpg, mtcars$cyl, mean)
sds <- tapply(mtcars$mpg, mtcars$cyl, sd)
ns <- tapply(mtcars$mpg, mtcars$cyl, length)
round(rbind(means, sds, ns), 2)

## [1] 13 14 15 16
##           4      6      8
## means 26.66 19.74 15.10
## sds   4.51  1.45  2.56
## ns    11.00  7.00 14.00
```

tapply(X, INDEX, FUN) Split x by the groups in INDEX, apply FUN to each piece, and collect the answers.

Here: mean, SD and size of fuel economy (mpg) for cars with 4, 6 and 8 cylinders. We plot these on a later slide.

PART 6

Cleaning and plotting

Reading three kinds of file

```
names(read.csv("data/data.csv"))           # odd name!
d_csv <- read.csv("data/data.csv",
                  fileEncoding = "UTF-8-BOM")
names(d_csv)
d_txt <- read.table("data/data.txt", header = TRUE,
                   sep = "\t")
d_xl <- read_excel("data/Data_Excel.xlsx", sheet = 1)
c(txt = nrow(d_txt), excel = nrow(d_xl))

## [1] "X...A" "B"
## [1] "A" "B"
##   txt excel
##    6     5
```

A real-world trap Excel often saves CSV files with an invisible byte-order mark. R then reads the first column name as X...A (or i...A on Windows); `fileEncoding = "UTF-8-BOM"` fixes it.

`read.table()` needs `sep = "\t"` for tabs; `read_excel()` returns a tibble, a compact data frame.

A cleaning task: the Titanic passenger list

```
titanic <- read.csv("data/titanic.csv")
titanic$Name <- NULL; titanic$Ticket <- NULL
sum(duplicated(titanic))           # repeated rows?
colSums(is.na(titanic))[c("Age", "Fare")]
c(Cabin = sum(titanic$Cabin == ""),
  Embarked = sum(titanic$Embarked == ""))
sum(titanic$Fare == 0)             # zero fares?

## [1] 0
##   Age Fare
##  177    0
##   Cabin Embarked
##    687        2
## [1] 15
```

Reading it 891 passengers, no duplicates. Age has 177 NAs; blanks in Cabin and Embarked are invisible to `is.na()`. 15 zero fares need a question to whoever made the file.

Names and tickets identify people and are not needed: drop them early.

Cleaning step by step

```
titanic$Embarked[titanic$Embarked == ""] <- NA
titanic$Embarked[is.na(titanic$Embarked)] <-
  names(which.max(table(titanic$Embarked))) # mode
med_age <- ave(titanic$Age, titanic$Pclass,
              FUN = function(a) median(a, na.rm = TRUE))
titanic$Age[is.na(titanic$Age)] <-
  med_age[is.na(titanic$Age)] # by class
titanic$Survived <- factor(titanic$Survived,
                          labels = c("No", "Yes"))
titanic$Cabin <- NULL # mostly blank
colSums(is.na(titanic))[c("Age", "Embarked")]

##      Age Embarked
##      0         0
```

- Blank to NA, then the most common port
- Missing ages get the *median of their class* (37, 29, 24 years), not one overall value
- 0/1 becomes a labelled factor
- Cabin (77% blank) is dropped

Write down every such decision.

Outliers: flag them, do not delete them blindly

```
iqr_flag <- function(v) {  
  q <- quantile(v, c(0.25, 0.75), na.rm = TRUE)  
  fence <- q[2] + 1.5 * (q[2] - q[1])  
  v > fence  
}  
sum(iqr_flag(titanic$Fare))           # "outliers"  
table(titanic$Pclass[iqr_flag(titanic$Fare)])  
max(titanic$Fare)  
  
## [1] 116  
##  
##    1    2    3  
## 104    5    7  
## [1] 512.3
```

Reading it The rule flags 116 fares, and 104 of them are first-class tickets: real, expensive fares in a right-skewed variable, not errors.

The original notes deleted every passenger outside the Age fences: 66 people, infants and the elderly. Flag, investigate, and if needed transform (e.g. $\log(1 + \text{Fare})$) instead.

Checking a file against its source: Iris.csv

```
iris_csv <- read.csv("data/Iris.csv")
which(duplicated(iris_csv[, -1])) # ignoring the Id
differ <- rowSums(iris_csv[, 2:5] != iris[, 1:4]) > 0
iris_csv[differ, 2:5]             # the csv copy
iris[differ, 1:4]                 # R's built-in iris

## [1] 35 38 143
## Sepal.Length Sepal.Width Petal.Length Petal.Width
## 35          4.9          3.1          1.5          0.1
## 38          4.9          3.1          1.5          0.1
## Sepal.Length Sepal.Width Petal.Length Petal.Width
## 35          4.9          3.1          1.5          0.2
## 38          4.9          3.6          1.4          0.1
```

Reading it With the Id column `duplicated()` finds nothing: duplicates hide behind an ID. And two rows of this widely shared copy differ from Fisher's data in R's iris: a known transcription error. Clean data can still be wrong data.

Standardising with your own function

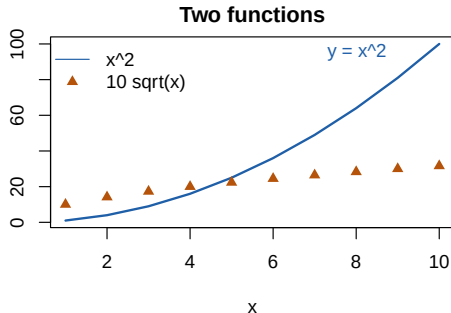
```
normalise <- function(v) {  
  (v - min(v)) / (max(v) - min(v))  
}  
iris_num <- iris_csv[, 2:5]  
names(iris_num) <- c("SL", "SW", "PL", "PW")  
sapply(lapply(iris_num, normalise), range)  
iris_z <- scale(iris_num) # mean 0, SD 1  
round(apply(iris_z, 2, sd), 3)  
  
##      SL SW PL PW  
## [1,]  0  0  0  0  
## [2,]  1  1  1  1  
## SL SW PL PW  
##  1  1  1  1
```

$$\text{Min-max: } x' = \frac{x - \min x}{\max x - \min x}$$
$$\text{z-score: } z = \frac{x - \bar{x}}{s}$$

One small function plus `lapply()` handles every column; no copy-and-paste. Distances, PCA and clustering need variables on a common scale.

Base graphics: building a plot layer by layer

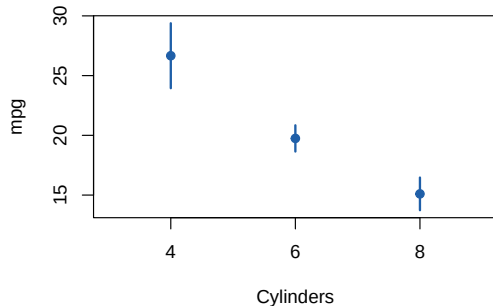
```
x <- 1:10
y <- x^2
par(mar = c(4, 4, 2, 1))
plot(x, y, type = "n", xlab = "x", ylab = "")
lines(x, y, col = col_main, lwd = 2)
points(x, 10 * sqrt(x), col = col_alt, pch = 17)
text(9, 95, "y = x^2", col = col_main, pos = 2)
legend("topleft", bty = "n",
      legend = c("x^2", "10 sqrt(x)"),
      col = c(col_main, col_alt),
      lty = c(1, NA), pch = c(NA, 17))
title(main = "Two functions")
```



`plot(type = "n")` sets up axes only;
`lines()`, `points()`, `text()`, `legend()` and
`title()` then add one layer each.

Error bars with `segments()`

```
se <- sds / sqrt(ns)           # from tapply()
xpos <- 1:3
par(mar = c(4, 4, 1, 1))
plot(xpos, means, pch = 19, col = col_main,
     xlim = c(0.5, 3.5),
     ylim = range(means - 2 * se, means + 2 * se),
     xaxt = "n", xlab = "Cylinders", ylab = "mpg")
axis(1, at = xpos, labels = names(means))
segments(xpos, means - 2 * se, xpos,
         means + 2 * se, col = col_main, lwd = 2)
```

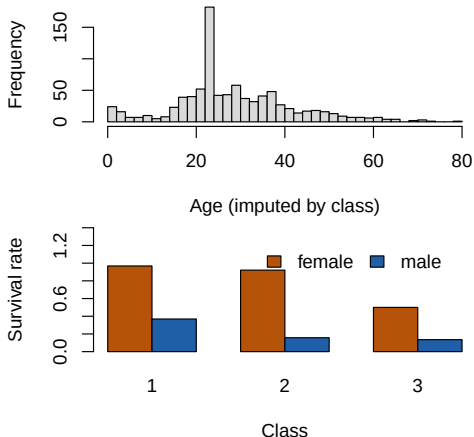


Mean mpg $\pm$ 2 SE: from 26.7 mpg (4 cylinders) down to 15.1 (8 cylinders); the bars do not overlap.

Plotting the cleaned Titanic data

```
rates <- tapply(titanic$Survived == "Yes",  
               list(titanic$Sex, titanic$Pclass), mean)  
par(mfrow = c(2, 1), mar = c(4, 4, 0.5, 1))  
hist(titanic$Age, breaks = 30, col = "grey85",  
     main = "", xlab = "Age (imputed by class)")  
barplot(rates, beside = TRUE,  
        col = c(col_alt, col_main),  
        xlab = "Class", ylab = "Survival rate",  
        ylim = c(0, 1.4), legend.text = TRUE,  
        args.legend = list(bty = "n", horiz = TRUE))
```

Spikes at the class medians; women fared better in every class.



Exercises

1. Write `cv(x)` returning the coefficient of variation in %, with an `na.rm` argument passed on to `mean()` and `sd()`.
2. For $n = 1, \dots, 30$ count the steps of the Collatz map ($n \rightarrow n/2$ if even, $3n + 1$ if odd) until $n = 1$. Use `while` inside a function and `sapply()` outside it.
3. Rewrite `fibonacci()` so that it stores values already computed (memoisation), and compare its speed with `system.time()`.
4. Make `get_mode()` return all modes, and test it on ties, NA and an empty vector.
5. Re-impute the Titanic ages by the median within each class *and* sex using `ave()`, and compare the histograms.

Key take-aways

1. **Know your structure:** vector, matrix, array, list, data frame, factor; most surprises come from silent coercion.
2. **Indexing is the core skill:** position, negation, condition, name, and assignment through an index.
3. **Control flow:** `if` for single decisions, `ifelse()/cut()` for vectors, loops when the length is unknown.
4. **Functions:** arguments, defaults, scope, closures and recursion; test them on edge cases.
5. **Vectorise** where you can, and use the apply family instead of copy-and-paste.
6. **Clean data deliberately:** blanks are not NA, outliers are not errors, and a shared file can still be wrong.

References

- Ihaka, R., and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3), 299–314.
- Jones, O., Harden, S., and Crawley, M. J. (2022). *The R Book*, 3rd Ed., Wiley, Hoboken.
- R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna. <https://www.R-project.org>
- Rizzo, M. L. (2019). *Statistical Computing with R*, 2nd Ed., CRC Press, Boca Raton.
- Wickham, H., Çetinkaya-Rundel, M., and Grolemund, G. (2023). *R for Data Science*, 2nd Ed., O'Reilly, Sebastopol. (Free online: r4ds.hadley.nz)
- Software: CRAN <https://cran.r-project.org>; RStudio Desktop <https://posit.co/download/rstudio-desktop/>

Thank you

Questions and discussion