

Introduction to R

Dr. Kapil Kumar

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

Two-Day National Workshop on Data Analysis with R (20–21 February 2026)
Department of Economics, Chaudhary Charan Singh University, Meerut

20 February 2026 · Day 1 lecture

Plan of the session

1. Getting started: R, RStudio and the workshop data
2. R as a calculator: thinking in vectors
3. Data structures
4. Exploring a data set
5. Data input and output
6. Programming essentials

By the end you can install and run R, compute with whole vectors at once, choose the right data structure, inspect and subset a data frame, read and save CSV, RDS and Excel files with relative paths, and write simple conditions, loops and functions.

Tomorrow (Day 2, *Data Analysis using R*) builds on this: cleaning, summaries, graphs, tests and regression.

PART 1

Getting started

Why R for data analysis?

- **Free and open source**, on Windows, macOS and Linux
- **Vectorised**: one command acts on a whole column
- **Packages**: thousands of tools on CRAN, from econometrics to graphics
- **Reproducible**: a script records every step, so results can be re-run
- Reports with LaTeX and R Markdown

R is more than a calculator It is a programming language, a statistical computing environment, a graphics system, a data-management tool and a platform for reproducible research.

Installing R and RStudio

1. R (the engine)

1. Go to <https://cran.r-project.org>
2. Download R for your operating system
3. Run the installer with default settings

2. RStudio (the workbench)

1. Go to <https://posit.co/download/rstudio-desktop/>
2. Install the free RStudio Desktop *after* R

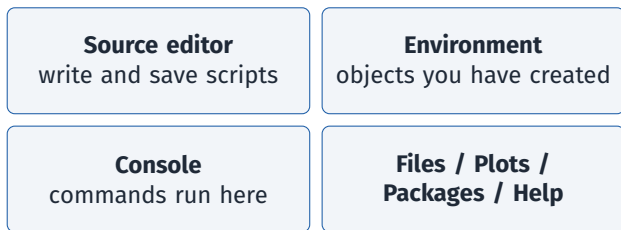
3. Check the installation

Open RStudio and type in the console:

```
print("Hello, R!")  
  
## [1] "Hello, R!"  
  
R.version.string  
  
## [1] "R version 4.3.3 (2024-02-29)"
```

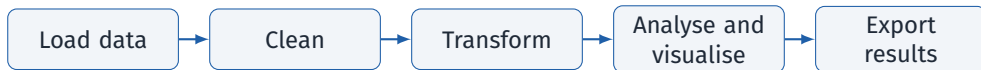
If the message is printed, the setup works.

The RStudio window and a typical workflow



Habit Type code in the *script*, not only in the console. `Ctrl+Enter` runs the current line, and the saved script is your record of the analysis.

A typical analysis



Packages and help

- A **package** adds functions and data sets to base R
- `install.packages()` downloads it once; `library()` loads it in every session
- This lecture uses base R, plus `readxl` and `writexl` for Excel files
- Help: `?read.csv`, `example(mean)`, `help(package = "readxl")`

```
# install once (needs internet)
# install.packages(c("readxl", "writexl"))
library(readxl)    # read .xlsx files
library(writexl)   # write .xlsx files
```

Tip Put all `library()` calls at the top of a script, so a reader sees at once what it needs.

The four workshop data sets

```
socio      <- read.csv("data/socio_economic.csv")
sales      <- read.csv("data/sales.csv")
titanic    <- read.csv("data/titanic.csv")
iris_data  <- read.csv("data/iris.csv")
dim(socio); dim(sales)

## [1] 40  8
## [1] 25  5

dim(titanic); dim(iris_data)

## [1] 891  10
## [1] 150  6
```

- socio: 40 people: region, age, schooling, gender, sector, literacy, income
- sales: monthly sales, marketing spend and revenue by region
- titanic: the classic passenger list (names removed)
- iris_data: Fisher's iris flowers

Data note socio and sales are illustrative teaching data.

PART 2

R as a calculator

Arithmetic operators

Operator	Meaning
+ -	addition, subtraction
* /	multiplication, division
^	power
%%	remainder (modulus)
%%/	integer division
()	brackets come first

The usual order of operations applies: powers, then $\times$ and $\div$, then $+$ and $-$.

```
# naming elements inside c() labels each result
c(add = 5 + 3, subtract = 10 - 4,
  multiply = 6 * 7, divide = 20 / 5)
c(power = 2^3, remainder = 10 %% 3,
  integer_div = 10 %/% 3)
2 + 3 * 4; (2 + 3) * 4

##      add subtract multiply   divide
##      8       6       42       4
##      power  remainder integer_div
##      8             1           3
## [1] 14
## [1] 20
```

Mathematical functions, with an economist's twist

```
c(sqrt(25), abs(-12), log(100), log10(100))
round(3.14159, 2)
# Rs 100 growing at 7% a year for 10 years
100 * (1 + 0.07)^10
# a log difference is close to the growth rate
log(110) - log(100)

## [1]  5.000 12.000  4.605  2.000
## [1] 3.14
## [1] 196.7
## [1] 0.09531
```

Compound growth: $V_t = V_0 (1+g)^t$

Log difference: $\ln V_1 - \ln V_0 \approx g$

- `log()` is the *natural* logarithm;
`log10()` is base 10
- A 10% rise gives a log difference of 0.0953, close to 0.10: one reason economists model $\ln(\text{income})$

Objects, assignment and vectors

- `name <- value` stores a result in an **object**
- Names are case-sensitive, start with a letter, and contain no spaces:
`gdp_growth`, not `GDP growth`
- A **vector** is an ordered set of values of one type, made with `c()`, `seq()`, `rep()` or `a:b`

```
rate <- 0.07           # assignment
rate * 100
years <- 2021:2026     # integer sequence
years
rep(c("Kharif", "Rabi"), times = 2)
x <- c(10, 20, 30, 40, 50)
length(x); class(x)

## [1] 7
## [1] 2021 2022 2023 2024 2025 2026
## [1] "Kharif" "Rabi"  "Kharif" "Rabi"
## [1] 5
## [1] "numeric"
```

Vectorised arithmetic: a household budget

```
item <- c("Rice", "Milk", "Fuel", "Rent")
price <- c(42, 56, 95, 8000) # rupees per unit
qty <- c(25, 30, 20, 1) # units per month
spend <- price * qty # element by element
names(spend) <- item
spend
sum(spend)
round(100 * spend / sum(spend), 1) # budget shares

## Rice Milk Fuel Rent
## 1050 1680 1900 8000
## [1] 12630
## Rice Milk Fuel Rent
## 8.3 13.3 15.0 63.3
```

Expenditure: $e_i = p_i q_i$

Budget share: $w_i = e_i / \sum_j e_j$

- No loop is needed: `price * qty` multiplies the four pairs at once
- In this illustrative basket rent takes 63.3% of the budget, food (rice and milk) 21.6%

Vectorised operations on a data column

```
head(sales$sales)           # units sold
head(sales$sales * 1.05)    # with 5% growth
head(socio$Annual_Income / 1000) # in thousands
inc_z <- (socio$Annual_Income -
          mean(socio$Annual_Income)) /
          sd(socio$Annual_Income)
round(head(inc_z), 2)       # z-scores

## [1] 615 368 631 695 724 724
## [1] 645.8 386.4 662.6 729.8 760.2 760.2
## [1] 35.00 40.00 26.03 24.86 30.57 16.94
## [1] 1.15 1.78 0.02 -0.13 0.59 -1.13
```

- `data$column` extracts one column as a vector
- Every arithmetic operator and most functions work on the whole column at once
- The *z*-score line combines a vector with two single numbers (the mean and the sd)

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

Comparisons give logical vectors

```
inc <- socio$Annual_Income
head(inc)
head(inc > 30000)
sum(inc > 30000)    # TRUE counts as 1
mean(inc > 30000)   # a proportion

## [1] 35000 40000 26029 24860 30567 16936
## [1] TRUE  TRUE FALSE FALSE  TRUE FALSE
## [1] 12
## [1] 0.3
```

== !=	equal, not equal
> >= < <=	order
& !	and, or, not
%in%	is it in a set?

Reading it 12 of the 40 people
(30%) earn more than 30,000.

Logical indexing: the most useful idea in R

```
inc[inc > 40000]                                # values that pass the test

## [1] 46375 43422

socio[socio$Annual_Income > 38000, c("id", "gender", "sector", "Annual_Income")]

##      id gender      sector Annual_Income
## 2    2 Female    Services          40000
## 28 28 Female    Services          46375
## 36 36 Female Agriculture          43422
```

A logical vector inside `[ ]` keeps the elements (rows) where it is TRUE; in `[rows, columns]` the condition picks rows and the names pick columns. Conditions combine with `&` (and) and `|` (or).

The recycling rule

```
a <- c(10, 20, 30, 40)
b <- c(1, 2)
a + b      # b is reused: 1, 2, 1, 2
a * 2      # 2 is a vector of length 1

## [1] 11 22 31 42
## [1] 20 40 60 80
```

Careful With lengths 4 and 3, R still recycles but warns: *longer object length is not a multiple of shorter object length*. Treat that warning as an error in your code.

Workshop insight: think in vectors

Instead of looping over elements, apply an operation to the whole vector. The code is shorter, easier to read, faster, and has fewer places for mistakes.

PART 3

Data structures

Six data structures you will meet

Structure	Dimensions	Elements	Typical use
vector	1	one type	a variable: incomes, prices
factor	1	categories (levels)	gender, sector, region
matrix	2	one type	linear algebra, input-output tables
array	3 or more	one type	region $\times$ quarter $\times$ year
data frame	2	one type per column	a data set (rows = cases)
list	any	anything	model output, mixed results

Check before you analyse `class()` tells you what an object is; `str()` shows its structure compactly. Most surprises in R come from an object not being the type you expected.

Vectors and their types

```
v_num <- c(2.5, 4, 6)
v_chr <- c("North", "South")
v_lgl <- c(TRUE, FALSE, TRUE)
class(v_num); class(v_chr); class(v_lgl)
c(1, "two", TRUE)      # mixed: all become text
as.numeric(c("12", "7.5"))
sum(v_lgl)             # logicals count as 0/1

## [1] "numeric"
## [1] "character"
## [1] "logical"
## [1] "1"      "two"    "TRUE"
## [1] 12.0    7.5
## [1] 2
```

- All elements of a vector share one type: numeric, character or logical
- Mixing types *coerces* everything to the most general one (character)
- `as.numeric()`, `as.character()`, `as.integer()` convert types

Watch out Text that is not a number, such as "1,250" or "abc", becomes NA on conversion (with a warning).

Matrices: household spending as Qp

```
Q <- matrix(c(25, 30, 20,  
             10, 45, 35), nrow = 2, byrow = TRUE,  
            dimnames = list(c("HH1", "HH2"),  
                             c("Rice", "Milk", "Fuel")))  
p <- c(42, 56, 95)      # prices  
Q  
Q %*% p                  # matrix product  
  
##      Rice Milk Fuel  
## HH1    25   30   20  
## HH2    10   45   35  
##      [,1]  
## HH1 4630  
## HH2 6265
```

$$e = Qp, \quad e_h = \sum_j q_{hj} p_j$$

- Rows are households, columns are goods; every entry is a number
- `%*%` is matrix multiplication; `*` would multiply element by element; `t()` transposes
- HH2 spends 6,265 rupees on these three goods

Matrices from data, and arrays

```
iris_mat <- as.matrix(iris_data[, 2:5]) # numeric
dim(iris_mat)
round(colMeans(iris_mat), 2)

## [1] 150    4
## Sepal.Length Sepal.Width Petal.Length
##           5.84           3.05           3.76
## Petal.Width
##           1.20
```

`colMeans()` averages each column. An **array** (right) adds a third index: region $\times$ quarter $\times$ year, filled with the values 1 to 24.

```
units <- array(1:24, dim = c(3, 4, 2),
  dimnames = list(c("North", "South", "East"),
    paste0("Q", 1:4),
    c("2024", "2025")))
units[, , "2025"]
units["South", "Q3", "2024"]

##           Q1 Q2 Q3 Q4
## North  13 16 19 22
## South  14 17 20 23
## East   15 18 21 24
## [1] 8
```

Lists: containers for anything

```
res <- list(file = "sales.csv",
            months = nrow(sales),
            revenue = summary(sales$Monthly_Revenue))
names(res)

## [1] "file"      "months"    "revenue"

res$months

## [1] 25

res[["revenue"]]

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      105     161     250     239     305     370
```

- A list holds objects of different types and lengths
- Extract with `$name` or `[[ ]]`; single `[ ]` returns a smaller list
- Results of `t.test()` and `lm()` are lists: that is how we pull out a *p*-value or coefficients on Day 2

Factors: categorical variables

```
sector <- factor(socio$sector)
table(sector)           # levels and counts
band <- factor(c("Low", "High", "Medium", "Low"),
               levels = c("Low", "Medium", "High"),
               ordered = TRUE)

band

## sector
## Agriculture      Industry      Services
##           12           9           19
## [1] Low      High      Medium Low
## Levels: Low < Medium < High
```

- A factor stores each category once (its *levels*) and an integer code per case
- Models and tables treat factors as groups, not numbers
- An *ordered* factor knows that Low < Medium < High, so band > "Low" works
- By default levels are sorted alphabetically; set levels = to choose the order (and the reference group)

Data frames: building one by hand

```
hh <- data.frame(  
  hh_id   = 101:105,  
  state   = c("UP", "Haryana", "UP", "Delhi", "Bihar"),  
  members = c(4, 6, 3, 5, 7),  
  income  = c(18500, 22000, 31000, 45000, 12500),  
  landed  = c(TRUE, TRUE, FALSE, FALSE, TRUE))
```

hh

##	hh_id	state	members	income	landed
## 1	101	UP	4	18500	TRUE
## 2	102	Haryana	6	22000	TRUE
## 3	103	UP	3	31000	FALSE
## 4	104	Delhi	5	45000	FALSE
## 5	105	Bihar	7	12500	TRUE

- Rows are **observations** (households), columns are **variables**
- Each column is a vector, so columns can have different types
- All columns must have the same length

Data note A made-up example of five households, monthly income in rupees.

Structure, summary and a new variable

```
str(hh)
hh$per_capita <- hh$income / hh$members # new column
round(hh$per_capita)

## 'data.frame': 5 obs. of 5 variables:
## $ hh_id : int  101 102 103 104 105
## $ state  : chr  "UP" "Haryana" "UP" "Delhi" ...
## $ members: num  4 6 3 5 7
## $ income : num  18500 22000 31000 45000 12500
## $ landed: logi  TRUE TRUE FALSE FALSE TRUE
## [1] 4625 3667 10333 9000 1786
```

```
summary(hh[, c("members", "income")])

##      members      income
## Min.      :3    Min.    :12500
## 1st Qu.:4      1st Qu.:18500
## Median :5      Median :22000
## Mean     :5      Mean    :25800
## 3rd Qu.:6      3rd Qu.:31000
## Max.     :7      Max.     :45000
```

Reading it Household 104 has the highest income per member: 10,333 rupees.

PART 4

Exploring a data set

First look: `str()`

```
str(socio)           # chr = text, int = whole numbers

## 'data.frame': 40 obs. of  8 variables:
## $ Region          : chr  " West " " East " " North " " South " ...
## $ id              : int   1 2 3 4 5 6 7 8 9 10 ...
## $ age             : int  46 50 55 44 53 22 48 51 41 36 ...
## $ education_years: int  16 7 8 2 14 19 NA 5 3 14 ...
## $ gender          : chr   "Female" "Female" "Female" "Female" ...
## $ sector          : chr   "Services" "Services" "Agriculture" "Services" ...
## $ Literacy        : int   81 86 90 88 93 93 93 84 92 83 ...
## $ Annual_Income   : int  35000 40000 26029 24860 30567 16936 10695 27098 23208 22046 ...
```

Already visible: Region has extra spaces (" West ") and education_years has NAs. Day 2 cleans both.

head(), tail() and summary()

```
head(socio, 3) # tail() shows the last rows
summary(socio[, c("age", "Literacy", "Annual_Income")])
```

```
##      Region id age education_years gender      sector Literacy Annual_Income
## 1    West   1  46                16 Female    Services        81        35000
## 2    East   2  50                7 Female    Services        86        40000
## 3   North   3  55                8 Female Agriculture       90        26029
##
##      age      Literacy      Annual_Income
## Min.   :19.0   Min.   :40.0   Min.   :10695
## 1st Qu.:28.5   1st Qu.:52.8   1st Qu.:21114
## Median :46.0   Median :80.5   Median :25754
## Mean   :41.1   Mean   :71.8   Mean   :25891
## 3rd Qu.:52.2   3rd Qu.:89.2   3rd Qu.:30307
## Max.   :60.0   Max.   :95.0   Max.   :46375
```

Selecting columns

```
socio$age[1:6]           # $ gives a vector
socio[1:6, "age"]         # same, by name
socio[1:3, c("gender", "sector")]
names(socio)[3:5]         # column names

## [1] 46 50 55 44 53 22
## [1] 46 50 55 44 53 22
##   gender      sector
## 1 Female    Services
## 2 Female    Services
## 3 Female Agriculture
## [1] "age"           "education_years"
## [3] "gender"
```

Two ways to ask `data$column` for one column; `data[rows, columns]` for anything else. Leaving `rows` empty means *all rows*.

Selecting by *name* is safer than by position: the code still works if columns are added or reordered.

Selecting rows: positions, conditions and `subset()`

Rows by position: `socio[2:3, ]`. Rows by condition: `&` = all conditions hold, `|` = at least one holds. `subset()` reads more naturally and drops rows where the condition is NA.

```
socio[socio$gender == "Male" & socio$sector == "Industry" & socio$age < 40,  
      c("id", "age", "Annual_Income")]  
subset(socio, Annual_Income > 45000 | age >= 59,  
       select = c(id, age, sector, Annual_Income))
```

```
##      id age Annual_Income  
## 10 10  36          22046  
## 13 13  27          25000  
## 14 14  35          15427  
##      id age  sector Annual_Income  
## 24 24  59 Services          17129  
## 28 28  46 Services          46375  
## 38 38  60 Services          28153
```

Missing values

```
sum(is.na(titanic$Age))      # missing ages
mean(titanic$Age)           # NA spreads
mean(titanic$Age, na.rm = TRUE) # drop NAs first
sum(titanic$Cabin == "")    # blanks are not NA

## [1] 177
## [1] NA
## [1] 29.7
## [1] 687
```

- NA marks a missing value; any calculation with it gives NA
- `na.rm = TRUE` drops the missing values first

Reading it 177 of 891 passengers (20%) have no recorded age.

Hidden gaps 687 cabins are empty strings, which `is.na()` does not count. The next section reads them as NA.

PART 5

Data input and output

Paths and project folders

- **Absolute** path, works on one computer only:
D:/OneDrive/Desktop/talk/data/titanic.csv
- **Relative** path, works for anyone who copies the folder:
data/titanic.csv
- Use forward slashes /, even on Windows
- Open the folder as an *RStudio project*: the working directory is then set for you

Rule Keep raw data read-only; write cleaned data and results to new files.

workshop/

data/

socio_economic.csv
sales.csv
titanic.csv
iris.csv

output/

analysis.R
workshop.Rproj

Reading CSV files carefully

```
file.exists("data/titanic.csv") # path correct?
titanic2 <- read.csv("data/titanic.csv",
                    header = TRUE, sep = ",",
                    na.strings = c("", "NA"))
na_count <- colSums(is.na(titanic2))
na_count[na_count > 0]

## [1] TRUE
##      Age      Cabin Embarked
##      177      687         2
```

With `na.strings`, blank cabins and the two blank ports are now NA.

Common import problems

- ; as separator: `sep = ";"`
- no header row:
`header = FALSE`
- numbers stored as text:
`"1,250"`
- accents:
`fileEncoding = "UTF-8"`
- file not found: check the path

Writing a CSV file and reading it back

```
dir.create("output", showWarnings = FALSE)
high_income <- subset(socio, Annual_Income > 35000)
write.csv(high_income, "output/high_income.csv",
          row.names = FALSE)
check <- read.csv("output/high_income.csv")
dim(check)

## [1] 4 8

identical(check$id, high_income$id)

## [1] TRUE
```

- `dir.create()` makes the output/folder if it is missing
- `row.names = FALSE` avoids an extra unnamed column of row numbers
- Reading the file back is a quick check that the export worked
- CSV is plain text: any software can open it

RDS files keep R objects intact

```
socio$sector <- factor(socio$sector)
saveRDS(socio, "output/socio.rds")
back_rds <- readRDS("output/socio.rds")
class(back_rds$sector)

## [1] "factor"

write.csv(socio, "output/socio_copy.csv",
          row.names = FALSE)
back_csv <- read.csv("output/socio_copy.csv")
class(back_csv$sector)

## [1] "character"
```

CSV or RDS? CSV stores only text, so the factor comes back as character. RDS stores the object itself (types, factor levels, attributes) but is readable only by R.

Use CSV to share data, RDS to save your own intermediate results.

Excel files with `writexl` and `readxl`

```
write_xlsx(list(sales = sales, socio = socio),
            "output/workshop.xlsx") # two sheets
excel_sheets("output/workshop.xlsx")

## [1] "sales" "socio"

sales_x <- read_excel("output/workshop.xlsx",
                      sheet = "sales")

dim(sales_x)

## [1] 25  5

class(sales_x)

## [1] "tbl_df"      "tbl"        "data.frame"
```

- Much official and survey data arrives as `.xlsx`
- `read_excel()` reads one sheet; choose it by name or number
- The result is a *tibble*, a modern data frame that works with all the code in these lectures

Workshop insight Good input and output habits make an analysis easy to debug, share and reproduce.

PART 6

Programming essentials

Decisions: if, else and ifelse()

```
avg_income <- mean(socio$Annual_Income)
if (avg_income > 30000) {
  print("Average income is above 30,000")
} else {
  print("Average income is 30,000 or less")
}
socio$band <- ifelse(socio$Annual_Income >= 30000,
                    "Upper", "Lower")

table(socio$band)

## [1] "Average income is 30,000 or less"
##
## Lower Upper
##    28    12
```

- if tests *one* condition and runs one block of code
- ifelse() tests every element and returns a vector: ideal for new variables
- Chain else if for more branches

Here Mean income is 25,891, so the else branch runs.

Repetition: for loops

```
for (sp in unique(iris_data$Species)) {  
  x <- iris_data$Sepal.Length[iris_data$Species == sp]  
  cat(sp, "mean sepal length:", round(mean(x), 2), "\n")  
}
```

```
## Iris-setosa mean sepal length: 5.01  
## Iris-versicolor mean sepal length: 5.94  
## Iris-virginica mean sepal length: 6.59
```

```
total <- 0  
for (i in 1:nrow(sales)) total <- total + sales$sales[i]  
c(loop = total, vectorised = sum(sales$sales))
```

```
##      loop vectorised  
##    14124      14124
```

- for (i in sequence) { ... } runs the block once for each value
- Loops suit step-by-step work: simulations, iterative algorithms, values that depend on the previous step
- For simple totals and means, the vectorised function is shorter and faster

while, break and next

```
i <- 0; running <- 0
while (running < 3000) {  # stop at 3000 units
  i <- i + 1
  running <- running + sales$sales[i]
}
cat("months:", i, " units:", running, "\n")
for (k in 1:10) {
  if (k %% 2 != 0) next    # skip odd numbers
  if (k > 6) break        # leave the loop
  cat(k, " ")
}

## months: 5  units: 3033
## 2 4 6
```

- while repeats as long as its condition is TRUE: make sure something inside changes it
- next skips to the next round; break leaves the loop
- repeat { . . . if (cond) break } runs until a break

Reading it Cumulative sales first reach 3,000 units in month 5 (3,033 units).

Writing your own functions

```
grow <- function(value, rate = 0.10) {  
  value * (1 + rate)      # last line is returned  
}  
grow(100000)              # default rate  
grow(100000, rate = 0.20)  
cv <- function(x) {  
  100 * sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)  
}  
cv(socio$Annual_Income)  # coefficient of variation  
  
## [1] 110000  
## [1] 120000  
## [1] 30.6
```

`f <- function(args) { body }`
 $CV = 100 s/\bar{x} (\%)$

- Arguments can have defaults (rate = 0.10)
- The value of the last expression is returned; `return()` is optional
- Write a function when you would otherwise copy the same code three times

The apply family: a function over many columns or groups

```
num <- iris_data[, 2:5]
sapply(num, mean)           # each column

## Sepal.Length Sepal.Width Petal.Length Petal.Width
##          5.843          3.054          3.759          1.199

sapply(num, function(x) max(x) - min(x)) # range

## Sepal.Length Sepal.Width Petal.Length Petal.Width
##          3.6          2.4          5.9          2.4

tapply(socio$Annual_Income, socio$sector, mean)

## Agriculture      Industry      Services
##          27673          23659          25823
```

- `lapply()` applies a function to each element of a list (a data frame is a list of columns) and returns a list
- `sapply()` does the same and simplifies the result to a vector
- `tapply()` applies a function within groups: mean income by sector
- `function(x) ...` written inline is an *anonymous* function

A first look at recursion

```
fact_rec <- function(n) {  
  if (n == 0) return(1)      # base case  
  n * fact_rec(n - 1)        # recursive step  
}  
fact_loop <- function(n) {  
  result <- 1  
  for (i in seq_len(n)) result <- result * i  
  result  
}  
c(rec = fact_rec(5), loop = fact_loop(5),  
  built_in = factorial(5))  
  
##      rec      loop built_in  
##     120     120     120
```

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

- A recursive function calls itself on a smaller problem
- It must have a **base case**, or it never stops
- Elegant for tree-like problems, but loops or built-ins (`factorial()`, `sum()`) are usually faster for data work

Key take-aways

1. **Work in scripts inside a project folder**, and read data with relative paths.
2. **Think in vectors**: most operations act on whole columns at once.
3. **Logical indexing** (`data[condition, columns]`) is the key to subsetting.
4. **Know your structures**; check them with `class()` and `str()`.
5. **Inspect every import**: `str()`, `summary()`, missing values, blanks.
6. **Save wisely**: CSV to share, RDS to keep R objects, `.xlsx` for Excel users.
7. **Functions and loops** automate repetition; prefer vectorised code where it exists.

Next session (Day 2): cleaning, describing, plotting, testing and modelling these data in *Data Analysis using R*.

References

- Jones, O., Harden, S., and Crawley, M. J. (2022). *The R Book*, 3rd Ed., Wiley, Hoboken.
- Wickham, H., Çetinkaya-Rundel, M., and Golemund, G. (2023). *R for Data Science*, 2nd Ed., O'Reilly, Sebastopol. Free online: <https://r4ds.hadley.nz>
- Verzani, J. (2014). *Using R for Introductory Statistics*, 2nd Ed., CRC Press, Boca Raton.
- Dalgaard, P. (2008). *Introductory Statistics with R*, 2nd Ed., Springer, New York.
- R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna. <https://www.R-project.org>
- Software: CRAN <https://cran.r-project.org>; RStudio Desktop <https://posit.co/download/rstudio-desktop/>

Thank you

Questions and discussion