

Basics of R Programming

for statistical analysis

Dr. Kapil Kumar

Assistant Professor, Department of Statistics
Central University of Haryana, Mahendergarh, India
kapilstats@gmail.com

One Week Workshop on Statistical Analysis with R (4–8 April 2022)

Department of Mathematics, Faculty of Science & Humanities, SRM University
Delhi-NCR, Sonapat

5 April 2022 · Invited lecture

Plan of the session

1. Getting started with R and RStudio
2. Vectors, data import and export
3. Writing your own functions
4. Control flow, the apply family and simulation
5. Descriptive statistics and graphics

By the end you can create and manipulate vectors; read, check and write data files; write and debug functions; use `if`, `for`, `while` and `apply()`; run a small simulation; compute descriptive statistics; and draw and customise base-R graphs.

PART 1

Getting started

R in a nutshell

- R is a free, open-source **language and environment** for statistical computing and graphics
- It descends from **S**, created by John Chambers and colleagues at Bell Labs in the 1970s
- Ross Ihaka and Robert Gentleman (University of Auckland) began R in the early 1990s; version 1.0.0 appeared on 29 February 2000
- The name echoes S and the initials of its two authors

Why R? Free; about 19,000 CRAN packages (March 2022), grouped by topic in Task Views; excellent graphics; reproducible scripts.

Limitations Data are held in memory (RAM); a steeper learning curve than menu-driven software.

Installing R and RStudio

1. Install R from CRAN:
<https://cran.r-project.org>
2. Install the free RStudio Desktop:
<https://posit.co/download/rstudio-desktop/>
3. Open RStudio and start a script:
File → New File → R Script

Ctrl+Enter runs the current line.
Code typed only in the console is not saved, so work in a script.



The four panes of RStudio. The Environment pane confirms that data have been read.

Packages used in this session

- Base R (base, stats, graphics) does almost everything today; it is loaded automatically
- **moments**: `skewness()` and `kurtosis()`
- Graphics here use base R; **lattice** and **ggplot2** are alternatives
- A package is installed once, but loaded with `library()` in every session

```
# install once
# install.packages("moments")
library(moments)

# colours used in the graphs
col_main <- "#1F5FA8"
col_alt  <- "#B45309"
```

R as a calculator

```
2 + 3 * 5          # * before +
(2 + 3) * 5
5 / 2; 5^2
7 %/% 2; 7 %% 2    # quotient, remainder
x <- 5; sqrt(x)    # assign, then use

## [1] 17
## [1] 25
## [1] 2.5
## [1] 25
## [1] 3
## [1] 1
## [1] 2.236
```

Order of operations ^ first, then * and /, then + and -. Use brackets whenever in doubt.

Naming objects R is case sensitive: x and X differ. Names may contain letters, digits, . and _, and start with a letter (or a dot not followed by a digit). # starts a comment.

Getting help

```
?mean                # help page of a function
help("cor")           # the same, as a call
??correlation         # search all help pages
help.search("regression")
example(mean)          # run the examples
help(package = "MASS") # index of a package
str(swiss)             # structure of a data set
```

```
args(sd)              # the arguments of sd()

## function (x, na.rm = FALSE)
## NULL
```

- Every help page has the same parts: *Description, Usage, Arguments, Value, Examples*
- Read *Arguments* for defaults, e.g. `na.rm = FALSE`
- The examples at the bottom can be copied and run
- In RStudio, place the cursor on a function name and press **F1**

PART 2

Vectors and data

Generating sequences

```
3:6                                # same as c(3, 4, 5, 6)
seq(from = 3, to = 10, by = 0.5)
seq(3, 5, length.out = 6)          # 6 equally spaced values
rep(c(2, 3), times = c(5, 4))      # 2 five times, 3 four times
rep(1:3, times = 3)
rep(1:3, each = 2)

## [1] 3 4 5 6
## [1] 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
## [12] 8.5 9.0 9.5 10.0
## [1] 3.0 3.4 3.8 4.2 4.6 5.0
## [1] 2 2 2 2 2 3 3 3 3
## [1] 1 2 3 1 2 3 1 2 3
## [1] 1 1 2 2 3 3
```

- `from:to` counts in steps of 1
- `seq()` takes a step (`by`) or a length (`length.out`)
- `rep()` repeats a whole vector (`times`) or each element (`each`)
- Sequences serve as indices, grids of x values and group labels

Combining vectors

```
x1 <- c(1, 2, 3, 4, 5, 6, 7)
y1 <- c(2, 1, 4, 3, 2, 5, 6)
c(x1, y1)           # one long vector
rbind(x1, y1)       # stacked as rows

## [1] 1 2 3 4 5 6 7 2 1 4 3 2 5 6
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## x1     1     2     3     4     5     6     7
## y1     2     1     4     3     2     5     6
```

```
cbind(x1, y1)       # side by side

##      x1 y1
## [1,]  1  2
## [2,]  2  1
## [3,]  3  4
## [4,]  4  3
## [5,]  5  2
## [6,]  6  5
## [7,]  7  6
```

One type per vector Mixing numbers and text turns everything into text: `c(1, "a")` gives "1" "a".

A data vector and its summaries

```
x <- c(2, 4, 3, 5, 7, 9, 15)
```

```
length(x); sum(x)
```

```
mean(x); median(x)
```

```
quantile(x)
```

```
summary(x)
```

```
## [1] 7
```

```
## [1] 45
```

```
## [1] 6.429
```

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

```
##    0%   25%   50%   75%  100%
```

```
##    2.0   3.5   5.0   8.0  15.0
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
##      2.00   3.50   5.00   6.43   8.00  15.00
```

- Think of `x` as daily counts over one week
- `quantile()` gives the minimum, quartiles and maximum; `summary()` adds the mean

Reading it The mean (6.43) exceeds the median (5) because one large value, 15, pulls the mean up: the data are right-skewed.

Vectorised arithmetic

```
1 + x
x^2
round(sqrt(x), 3)
round(log(x), 3)      # natural log
gamma(5)              # 4! = 24
c(1, 2, 3, 4) + c(10, 20) # recycling

## [1]  3  5  4  6  8 10 16
## [1]  4 16  9 25 49 81 225
## [1] 1.414 2.000 1.732 2.236 2.646 3.000 3.873
## [1] 0.693 1.386 1.099 1.609 1.946 2.197 2.708
## [1] 24
## [1] 11 22 13 24
```

Element by element Arithmetic and functions such as `log()` and `sqrt()` act on every element at once: no loop is needed.

Recycling The shorter vector is reused. R warns only if the longer length is not a multiple of the shorter.

Sorting, ordering and naming

```
sort(x)
sort(x, decreasing = TRUE)  # same as rev(sort(x))
order(x)                    # positions, smallest first
x[order(x)]                 # same as sort(x)
names(x) <- c("Mon", "Tue", "Wed", "Thu",
              "Fri", "Sat", "Sun")

x

## [1]  2  3  4  5  7  9 15
## [1] 15  9  7  5  4  3  2
## [1] 1  3  2  4  5  6  7
## [1]  2  3  4  5  7  9 15
## Mon Tue Wed Thu Fri Sat Sun
##   2   4   3   5   7   9 15
```

- `sort()` returns the values;
`order()` returns their positions,
which lets us sort a whole data
frame by one column
- `names()` attaches a label to
every element; give exactly one
name per element
- Names travel with the values
through most operations

Extracting elements with []

```
x[2]           # second element
x[c(2, 4)]     # second and fourth
x[-2]          # all but the second
```

```
## Tue
##    4
## Tue Thu
##    4  5
## Mon Wed Thu Fri Sat Sun
##    2  3  5  7  9 15
```

```
x["Sat"]       # by name
x[x > 5]       # by a condition
which(x > 5)   # positions where TRUE
```

```
## Sat
##    9
## Fri Sat Sun
##    7  9 15
## Fri Sat Sun
##    5  6  7
```

Four ways to index position, negative position (drop), name, logical condition; the same rules select rows and columns of a data frame: `data[rows, columns]`.

Reading a CSV file and spotting missing values

```
data1 <- read.csv("data/data1.csv")
data1
sapply(data1, class)
```

```
##      A  B C  D
## 1 10 12 1 14
## 2 25 24 2 36
## 3 36 36 5 69
## 4 14 85 9 58
## 5 10 12 9 NA
##           A           B           C           D
## "integer" "integer" "integer" "integer"
```

```
is.na(data1$D)
complete.cases(data1)
data1[complete.cases(data1), ]
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE
## [1]  TRUE  TRUE  TRUE  TRUE FALSE
##      A  B C  D
## 1 10 12 1 14
## 2 25 24 2 36
## 3 36 36 5 69
## 4 14 85 9 58
```

Empty cells become NA (not available);
`complete.cases()` flags rows without NA.

District yield data: a first look

```
yld <- read.table("data/data2.txt", header = TRUE) # tab-delimited text file
dim(yld); sum(is.na(yld))                          # size; missing values
head(yld[, 1:6], 3)

## [1] 58 10
## [1] 0
##   Dist Yield MARG_HH_F HH_SIZE NetArea Croppedarea
## 1    1 19575     4015    6.35  274837    405032
## 2    2 23483     9155    6.70  326387    472441
## 3    3 19442    12700    6.49  335831    427985
```

Data note Illustrative district-level farm data (58 districts) from the original lecture; source and units are not documented. Irrigated area never exceeds sown area.

Subsetting rows and columns

```
med <- median(yld$Yield)
low <- subset(yld, Yield < med,
              select = c(Dist, Yield, Rainfall, Fert))
nrow(low)
head(low, 3)

## [1] 29
##   Dist Yield Rainfall  Fert
## 6    6 10850    1179 164793
## 9    9 12419    620 176072
## 10   10 10483    383 124266
```

`subset()` keeps the rows that meet a condition and the columns named in `select`.

```
wet <- yld$Rainfall > 1500 # TRUE/FALSE
yld[wet, c("Dist", "Yield", "Rainfall")]

##   Dist Yield Rainfall
## 17    17 14975    1633
## 18    18 15986    1678
## 21    21 17331    1770
```

Reading it 29 of 58 districts lie below the median yield of 15,412; only 3 have rainfall above 1500.

Grouping with `cut()` and `aggregate()`

```
yld$RainClass <- cut(yld$Rainfall,
                     breaks = c(0, 700, 1100, Inf),
                     labels = c("Low", "Medium", "High"))
table(yld$RainClass)
aggregate(Yield ~ RainClass, data = yld, FUN = mean)

##
##      Low Medium   High
##      17     25     16
##      RainClass Yield
## 1      Low 17058
## 2   Medium 14178
## 3     High 15905
```

- `cut()` turns a number into a labelled **factor**: rainfall up to 700, 700–1100, above 1100
- `aggregate()` (or `tapply()`) applies a function within each group

Reading it Low-rainfall districts have the highest mean yield. Is that typical of the class, or driven by one district? The box plot later answers this.

Writing results and saving graphs

```
write.csv(low, "data/low_yield.csv",  
          row.names = FALSE)  
check <- read.csv("data/low_yield.csv")  
dim(check)           # read it back  
  
## [1] 29  4
```

```
pdf("yield_hist.pdf", width = 7, height = 5)  
hist(yld$Yield)           # draw on the device  
dev.off()                 # close the file
```

- `write.csv()` and `write.table()` save data frames as text files
- Read a written file back to check it
- `png()` works like `pdf()`; `saveRDS()` keeps one R object

row.names = FALSE Without it, row numbers are written as an extra unnamed first column, which reappears every time the file is read.

PART 3

Writing functions

Anatomy of a function

- A function packages a calculation for reuse, without copying code
- It takes **arguments**, runs its **body** and **returns** a value

```
fname <- function(arg1, arg2 = 10) {  
  result <- arg1 + arg2  # the body  
  return(result)         # value handed back  
}
```

arg2 = 10 sets a default value.

```
square <- function(number) {  
  answer <- number * number  
  return(answer)  
}  
  
square(3)  
square(0.8)  
square(1:5)      # a whole vector  
  
## [1] 9  
## [1] 0.64  
## [1] 1 4 9 16 25
```

Keep functions in a script and load them
with `source("myfunctions.R")`.

Two kinds of errors

Syntax error: R cannot read the code.

```
square <- function(number) {  
  answer <- number * number  
  return(number      # ")" is missing  
}
```

What R says unexpected '}' : count brackets; RStudio highlights matching pairs.

Logic error: the code runs but is wrong.

```
square_bad <- function(number) {  
  answer <- number * number  
  return(number)      # wrong object  
}  
square_bad(3)  
  
## [1] 3
```

Logic errors give no message, so test every function on a known case: `square(3)` must give 9.

A function for the standard error of the mean

```
set.seed(2022)           # reproducible
y1 <- rnorm(100, mean = 5, sd = 1)
mean(y1); sd(y1)
sqrt(var(y1) / length(y1)) # by hand
se_mean <- function(x) sqrt(var(x) / length(x))
se_mean(y1)

## [1] 5.139
## [1] 1.02
## [1] 0.102
## [1] 0.102
```

$$SE(\bar{x}) = \frac{s}{\sqrt{n}}$$

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

Reading it Base R has no SE function, so we wrote one. Its estimate, 0.102, is close to the true $1/\sqrt{100} = 0.1$.

Making the function robust to missing values

```
y1[50] <- NA; se_mean(y1)  # NA spreads
se_mean <- function(x) {
  n <- sum(!is.na(x))      # non-missing
  sqrt(var(x, na.rm = TRUE) / n)
}
se_mean(y1)
samp_size <- function(x)
  c(n = sum(!is.na(x)), NAs = sum(is.na(x)))
samp_size(y1)

## [1] NA
## [1] 0.103
##    n NAs
##  99   1
```

- Arithmetic with NA gives NA
- `na.rm = TRUE` drops missing values inside `var()`, `mean()`
- `samp_size()` reports the usable n and the NAs

Why not `length()`? It counts the NA too: $n = 100$, not 99. The original `sum(x)/mean(x)` gives the right count, less transparently.

Returning several results: normal MLEs

```
normal_mle <- function(x) {  
  n <- length(x)  
  var_est <- var(x) * (n - 1) / n  
  list(mean_mle = mean(x), sd_mle = sqrt(var_est))  
}  
set.seed(2022); x <- rnorm(100, 5, 2)  
est <- normal_mle(x)  
est$mean_mle; est$sd_mle; sd(x) # $ picks one  
  
## [1] 5.277  
## [1] 2.03  
## [1] 2.04
```

MLEs for $N(\mu, \sigma^2)$: $\hat{\mu} = \bar{x}$,

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 = \frac{n-1}{n} s^2$$

A `list()` returns several named results.

Reading it $\hat{\sigma} = 2.03$ (divisor n) is slightly below `sd()` = 2.04 (divisor $n - 1$); the gap vanishes as n grows.

A loop inside a function: a double sum

```
series_sum <- function(x) {  
  s <- 0  
  n <- length(x)  
  for (i in 1:n) {  
    s <- s + sum((1 / x[i] - 1 / x[-i])^2)  
  }  
  s  
}  
  
series_sum(1:10)  
a <- 1 / (1:10)           # check  
2 * 10 * sum((a - mean(a))^2)  
  
## [1] 13.84  
## [1] 13.84
```

$$S = \sum_{i=1}^n \sum_{j \neq i} \left(\frac{1}{x_i} - \frac{1}{x_j} \right)^2$$
$$= 2n \sum_{i=1}^n (a_i - \bar{a})^2, \quad a_i = 1/x_i$$

`x[-i]` drops element i , so only the outer sum needs a loop. **Check:** the closed form gives the same 13.8376.

PART 4

Control flow and apply

if and else: the median by hand

Control structures

if/else: act on a condition

for: repeat a fixed number of times

while: repeat while a condition holds

break/next: leave / skip a pass

Correction The original `xs[n/2]` for odd n truncates 12.5 to 12: the wrong element.

```
set.seed(7)
x <- round(rchisq(25, df = 2), 2)
xs <- sort(x)
n <- length(x)
if (n %% 2 == 0) {                                # n even
  med <- (xs[n / 2] + xs[n / 2 + 1]) / 2
} else {                                           # n odd
  med <- xs[(n + 1) / 2]
}
med
median(x)                                         # agrees

## [1] 1.77
## [1] 1.77
```

for loops

```
for (k in 1:3) {  
  cat(k, "squared is", k^2, "\n")  
}  
states <- c("Haryana", "Punjab", "Delhi")  
for (i in seq_along(states)) {  
  cat(i, states[i], "\n")  
}
```

```
## 1 squared is 1  
## 2 squared is 4  
## 3 squared is 9  
## 1 Haryana  
## 2 Punjab  
## 3 Delhi
```

- The loop variable takes each value of the vector in turn
- `seq_along(x)` gives $1, 2, \dots, \text{length}(x)$; safer than $1:\text{length}(x)$, which gives $1\ 0$ for an empty vector
- `cat()` prints compactly

Loops or vectors? Loops are clear for beginners, but many can be replaced by one vectorised call: $(1:3)^2$ gives all three squares.

Nested loops and `while`

```
A2 <- matrix(1:6, nrow = 2, ncol = 3)
for (i in seq_len(nrow(A2))) {
  for (j in seq_len(ncol(A2))) {
    cat("A2[", i, ", ", j, "] =",
        A2[i, j], "\n")
  }
}
```

```
## A2[ 1 , 1 ] = 1
## A2[ 1 , 2 ] = 3
## A2[ 1 , 3 ] = 5
## A2[ 2 , 1 ] = 2
## A2[ 2 , 2 ] = 4
## A2[ 2 , 3 ] = 6
```

```
h <- 0; n <- 0 # 1 + 1/2 + 1/3 + ...
while (h <= 3) {
  n <- n + 1
  h <- h + 1 / n
}
n; h

## [1] 11
## [1] 3.02
```

Reading it The sum first exceeds 3 after 11 terms. Use `while` when the number of passes is unknown.

apply(): rows and columns of a matrix

```
A1 <- matrix(1:20, nrow = 5)
A1
apply(A1, 1, mean)      # MARGIN 1: rows
apply(A1, 2, mean)      # MARGIN 2: columns

##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
## [1]  8.5  9.5 10.5 11.5 12.5
## [1]   3   8  13  18
```

`apply(X, MARGIN, FUN, ...)`

MARGIN = 1: FUN on each row

MARGIN = 2: FUN on each column

- One readable line replaces an explicit loop
- Arguments after FUN are passed on:
`apply(X, 2, mean, na.rm = TRUE)`
- `rowMeans()`, `colMeans()` are fast shortcuts

apply() on data: the CV of every variable

```
cv <- function(x) 100 * sd(x) / mean(x)
num <- yld[, 2:10]           # numeric columns
cvs <- round(apply(num, 2, cv), 1)
sort(cvs)
```

##	HH_SIZE	Yield	NetArea
##	7.5	29.5	38.1
##	Croppedarea	Rainfall	Netirrig
##	38.5	39.1	45.3
##	Fert GrossIrrigated	MARG_HH_F	
##	45.8	47.6	76.7

- Our own `cv()` is passed to `apply()` like any built-in function
- The CV has no units, so variables in different units can be compared

Reading it Household size is the most stable variable (CV 7.5%); the number of marginal households varies most (76.7%). Yield: 29.5%.

Coefficient of variation: $CV = 100 \times \frac{s}{\bar{x}} \%$

lapply() and sapply() on a list

```
data_list <- list(A1 = A1, B1 = 1:5)
lapply(data_list, mean)
sapply(data_list, mean)

## $A1
## [1] 10.5
##
## $B1
## [1] 3
##
##      A1      B1
## 10.5  3.0
```

The apply family

`apply()`: margins of a matrix

`lapply()`: each element of a list;
returns a *list*

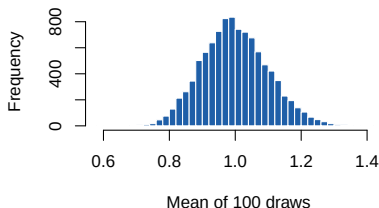
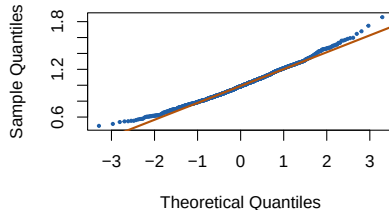
`sapply()`: as `lapply()`, but *simplifies*
to a vector or matrix

`tapply()`: a function within groups

A list holds objects of any kind and size;
`unlist()` flattens the result of `lapply()`. A
data frame is a list of columns.

Simulation with `replicate()`

```
sim_mean <- function(n) mean(rexp(n))      # mean of n draws from Exp(1)
set.seed(2022)
m100 <- replicate(10000, sim_mean(100))
m20  <- replicate(1000, sim_mean(20))
par(mfrow = c(1, 2), mar = c(4, 4, 1, 1))
hist(m100, breaks = 40, col = col_main, border = "white", main = "", xlab = "Mean of 100 draws")
qqnorm(m20, main = "", pch = 19, cex = 0.4, col = col_main); qqline(m20, col = col_alt, lwd = 2)
```



What the simulation shows

```
mean_sd <- function(x) {  
  c(mean = mean(x, na.rm = TRUE),  
    sd = sd(x, na.rm = TRUE))  
}  
  
round(mean_sd(m100), 4); 1 / sqrt(100)  
round(mean_sd(m20), 4); 1 / sqrt(20)  
  
##    mean    sd  
## 1.0000 0.1005  
## [1] 0.1  
##    mean    sd  
## 0.9990 0.2104  
## [1] 0.2236
```

$X \sim \text{Exp}(1)$: $E(X) = \text{Var}(X) = 1$,
so $\bar{X}_n \approx N(1, 1/n)$ for large n .

- Means of 100 draws: average 1.000, SD 0.1005; theory says 1 and 0.1
- The histogram is bell-shaped although each draw is skewed: the **central limit theorem**
- With $n = 20$ the Q-Q plot bends up at the right: some skewness (0.39) remains

PART 5

Statistics and graphics

Mean, trimmed mean and median

```
y <- yld$Yield
mean(y)
mean(y, trim = 0.1) # 10% cut from each end
median(y)

## [1] 15499
## [1] 15311
## [1] 15412
```

Trimmed mean: drop $[n\alpha]$ values at each end, average the rest.

Reading it District 24 has a yield of 34,050, far above the rest. Trimming 10% moves the mean from 15,499 to 15,311, close to the median.

Correction `trim` is a *fraction*, not a count. The original `trim = 2` is treated as 0.5 and silently returns the median.

The mode: a function of our own

```
mode_value <- function(x) {  
  ux <- unique(x)  
  ux[which.max(tabulate(match(x, ux)))]  
}  
A3 <- c(2, 1, 2, 3, 1, 2, 3, 4, 1, 5, 5, 3, 2, 3)  
mode_value(A3)  
table(A3)           # a tie!  
A4 <- c("o", "it", "the", "it", "it"); mode_value(A4)  
  
## [1] 2  
## A3  
## 1 2 3 4 5  
## 3 4 4 1 2  
## [1] "it"
```

```
modes <- function(x) {  
  tab <- table(x)  
  names(tab)[tab == max(tab)]  
}  
modes(A3)  
  
## [1] "2" "3"
```

Correction `which.max()` returns only the first maximum, but A3 is bimodal (2 and 3 each occur 4 times); `modes()` returns both.

Spread and shape

```
c(var = var(y), sd = sd(y), IQR = IQR(y))
quantile(y, c(0.10, 0.90))
c(cv = cv(y), se = sd(y) / sqrt(length(y)))
c(skew = skewness(y), kurt = kurtosis(y))
```

```
##      var      sd      IQR
## 20844998    4566    5191
##   10%   90%
## 10284 19527
##      cv      se
## 29.46 599.50
##   skew   kurt
## 0.987 6.269
```

With $m_r = \frac{1}{n} \sum (x_i - \bar{x})^r$:

skewness $\sqrt{b_1} = m_3/m_2^{3/2}$

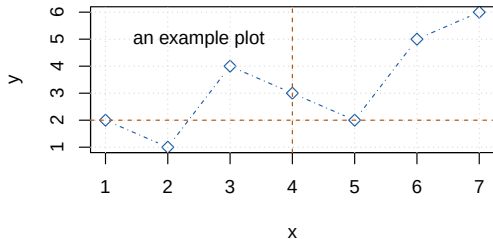
kurtosis $b_2 = m_4/m_2^2$ (= 3 for a normal)

Reading it Skewness 0.99 > 0:
a long right tail. Kurtosis 6.27
> 3: heavier tails than a normal,
largely because of district 24.

High-level and low-level graphics

```
x2 <- c(1, 2, 3, 4, 5, 6, 7)
y2 <- c(2, 1, 4, 3, 2, 5, 6)
par(mar = c(4, 4, 1, 1))
plot(x2, y2, type = "b",      # points and lines
     pch = 5, lty = 4,      # symbol, line type
     col = col_main, xlab = "x", ylab = "y")
grid()                      # low-level additions
abline(h = 2, v = 4, col = col_alt, lty = 2)
text(2.5, 5, "an example plot")
```

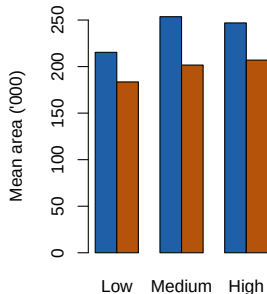
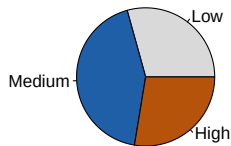
Other options: lwd line width, main title;
`colors()` lists the 657 named colours.



Two kinds *High-level* functions start a plot (`plot()`, `hist()`, `barplot()`, `curve()`); *low-level* ones add to it (`lines()`, `abline()`, `text()`, `legend()`).

Bar charts and a pie chart

```
counts <- table(yld$RainClass)
area <- aggregate(cbind(NetArea,
  Netirrig) ~ RainClass,
  data = yld, FUN = mean)
par(mfrow = c(1, 2), mar = c(3, 4, 1, 1))
barplot(t(area[, 2:3]) / 1000,
  beside = TRUE,
  names.arg = area$RainClass,
  col = c(col_main, col_alt),
  ylab = "Mean area ('000)")
pie(counts, col = c("grey85",
  col_main, col_alt))
```

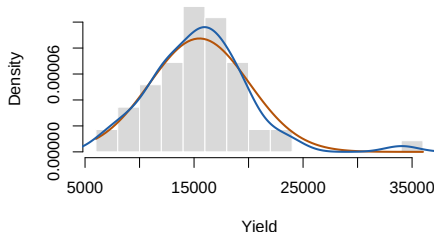


Left: grouped bars (beside = TRUE); blue net sown, amber net irrigated area. Right: districts per rainfall class.

Histogram with a fitted normal curve

```
par(mar = c(4, 4, 1, 1))
hist(y, breaks = 12, freq = FALSE,
     main = "", col = "grey85",
     border = "white", xlab = "Yield")
curve(dnorm(x, mean(y), sd(y)), add = TRUE,
      col = col_alt, lwd = 2)
lines(density(y), col = col_main, lwd = 2)
```

freq = FALSE: density scale, so curves can be added. Amber: normal curve (sample mean, SD); blue: kernel density.

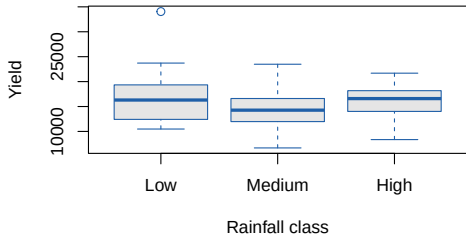


Reading it Most districts look roughly normal, but the right tail is longer than the normal curve allows: the skewness and kurtosis, seen.

Box plots: comparing groups

```
par(mar = c(4, 4, 1, 1))
boxplot(Yield ~ RainClass, data = yld,
        col = "grey90", border = col_main,
        xlab = "Rainfall class",
        ylab = "Yield")
```

Box: quartiles and median; whiskers reach the most extreme values within $1.5 \times \text{IQR}$ of the box; points beyond are possible outliers.

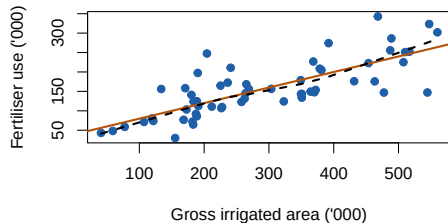


Reading it Medians (16,304, 14,268, 16,590) put Low and High close. The extreme value, a low-rainfall district, lifted that mean.

Scatter plot with fitted lines

```
irr <- yld$GrossIrrigated / 1000
fert <- yld$Fert / 1000
par(mar = c(4, 4, 1, 1))
plot(irr, fert, pch = 19, col = col_main,
     xlab = "Gross irrigated area ('000)",
     ylab = "Fertiliser use ('000)")
fit <- lm(fert ~ irr)
abline(fit, col = col_alt, lwd = 2)
lines(lowess(irr, fert), lty = 2, lwd = 2)
```

Dashed: a `lowess()` smooth. It agrees with the straight line.

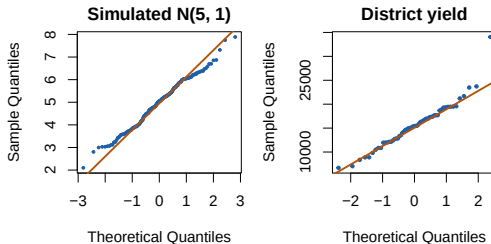


Reading it $r = 0.78$: more irrigated land goes with more fertiliser, about 0.4 units per unit of area (association, not cause).

Normal Q-Q plots

```
set.seed(2022)
rn <- rnorm(200, mean = 5, sd = 1)
par(mfrow = c(1, 2), mar = c(4, 4, 2, 1))
qqnorm(rn, main = "Simulated N(5, 1)",
       pch = 19, cex = 0.4, col = col_main)
qqline(rn, col = col_alt, lwd = 2)
qqnorm(y, main = "District yield",
       pch = 19, cex = 0.5, col = col_main)
qqline(y, col = col_alt, lwd = 2)
```

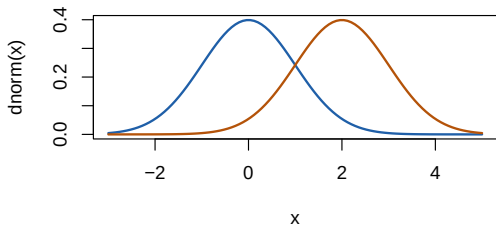
Sample against normal quantiles: a straight line means the data look normal.



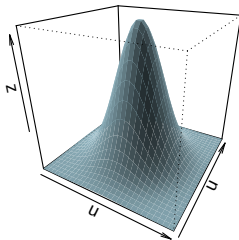
Reading it The simulated sample follows the line; the yield data bend away at the top (Shapiro-Wilk $p = 0.0057$).

Curves and a 3-D surface

```
par(mar = c(4, 4, 1, 1))  
curve(dnorm(x), -3, 5, lwd = 2, col = col_main)  
curve(dnorm(x, mean = 2), add = TRUE,  
      lwd = 2, col = col_alt)
```



```
u <- seq(-3, 3, length.out = 30)  
z <- outer(dnorm(u), dnorm(u)) # f(a) f(b)  
par(mar = c(0, 0, 0, 0))  
persp(u, u, z, theta = 30, phi = 25, shade = 0.4,  
       col = "lightblue", border = NA)
```



Key take-aways

1. **Work in scripts**, with data in a `data/` folder and relative paths, so the analysis reruns from top to bottom.
2. **Vectors are the basic unit**: arithmetic, indexing and most functions act on all elements at once.
3. **Look at data before analysing it**: `str()`, `summary()`, `is.na()`, a plot.
4. **Write a function** whenever you repeat a calculation, and test it on a case with a known answer.
5. **Handle missing values deliberately** with `na.rm` or `complete.cases()`.
6. **Loops and the apply family** automate repetition; `replicate()` turns a function into a simulation study.
7. **Graphs reveal** what summary numbers hide: skewness, outliers, groups.

References

- Dalgaard, P. (2008). *Introductory Statistics with R*, 2nd Ed., Springer, New York.
- Jones, O., Harden, S., and Crawley, M. J. (2022). *The R Book*, 3rd Ed., Wiley, Hoboken.
- Rizzo, M. L. (2019). *Statistical Computing with R*, 2nd Ed., CRC Press, Boca Raton.
- Verzani, J. (2014). *Using R for Introductory Statistics*, 2nd Ed., CRC Press, Boca Raton.
- Wickham, H., Çetinkaya-Rundel, M., and Golemund, G. (2023). *R for Data Science*, 2nd Ed., O'Reilly, Sebastopol. Free online: <https://r4ds.hadley.nz>
- R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna. <https://www.R-project.org>
- CRAN: <https://cran.r-project.org>; RStudio Desktop: <https://posit.co/download/rstudio-desktop/>

Thank you

Questions and discussion