DATA TYPES & STRUCTURES

6 Atomic Types (coercion order)

`logical` < `integer` < `double` < `character` < `complex` < `raw`

typeof() vs class() vs mode()

Object	typeof	class	mode
TRUE	"logical"	"logical"	"logical"
1L	"integer"	"integer"	"numeric"
3.14	"double"	"numeric"	"numeric"
"hi"	"character"	"character"	"character"
1:3	"integer"	"integer"	"numeric"
c(1,2)	"double"	"numeric"	"numeric"
list()	"list"	"list"	"list"
data.frame()	"list"	"data.frame"	"list"
factor("a")	"integer"	"factor"	"numeric"
function()	"closure"	"function"	"function"
NULL	"NULL"	"NULL"	"NULL"
NA	"logical"	"logical"	"logical"

▶ `is.numeric(x)` → TRUE for **both** integer and double!
`is.double(1L)` → FALSE. `is.numeric(1L)` → TRUE.

Key Structures

Matrix: atomic vector + `dim` attr. Filled **column-wise**.
`attr(M,"dim") <- c(2,5)` turns length-10 vec into 2×5 matrix.
`attr(M,"dim") <- NULL` → back to plain vector.
`class(M)` → `c("matrix","array")` when dim set.
Array: atomic vector with `dim` length > 2.
Data frame: `typeof`="list", `class`="data.frame".
`attributes(df)` → list with `names`, `class`, `row.names`.
Attributes: `attr(x,"info")<-"..."` sets metadata.
`attributes(x)` → full list. `attr(x,"dim")` → get one.

Factors

Integer vector with `levels` + `class` attributes.
Levels sorted **alphabetically**: `factor(c("M","F","F"))` → levels `c("F","M")`.
`as.integer(factor(c("M","F","F")))` → `c(2,1,1)`.
▶ Assigning value not in levels → becomes `NA` with warning.
▶ **Numeric factor trap:**
`as.numeric(factor(c(0,1,10,5)))` → `c(1,2,4,3)` (indices!)
Fix: `as.numeric(as.character(x_fac))`

identical() vs ==

`==` is vectorized, `NA` propagates. `identical()` returns single T/F.
`identical(NA, NA)` → TRUE. `NA == NA` → `NA`.
`identical(1, 1L)` → FALSE (type matters!). `identical(1L, 1L)` → TRUE.
`all.equal(1, 1L)` → TRUE (near-equality, ignores type).
`all.equal(0.1+0.2, 0.3)` → TRUE. `0.1+0.2 == 0.3` → FALSE (float!).

Type checking & conversion

`is.numeric()` → TRUE for int AND double. `is.atomic()` → TRUE for vec.
`as.integer(3.9)` → 3 (truncates). `as.double(TRUE)` → 1.
`as.character(123)` → "123". `as.numeric("3.14")` → 3.14.
`is.vector(list())` → TRUE. `is.atomic(list())` → FALSE.
`unlist(list(1,c(2,3)))` → `c(1,2,3)`.

COERCION RULES

Combining different types with c()

`c(TRUE,1L,2.5)` → double `c(1.0,1.0,2.5)`
`c(TRUE,1,"a")` → character `c("TRUE","1","a")`
`c(FALSE,1L)` → integer `c(0L,1L)`

Implicit coercion

TRUE → 1, FALSE → 0 in math.
`sum(c(T,F,T))` → 2. `mean(c(T,F,T))` → 0.667.

as.logical() rules

Input	Result
`as.logical(0)`	FALSE
`as.logical(1)`	TRUE
`as.logical(-1)`	TRUE (any nonzero≠NaN)
`as.logical(NaN)`	NA
`as.logical("TRUE"/"true"/"T")`	TRUE
`as.logical("False"/"false"/"F")`	FALSE
`as.logical("t"/"1"/"0")`	NA

Accepted: "TRUE","True","true","T","FALSE","False","false","F" only.
▶ `as.logical(c("0",1))` → all coerced to **char** first → all `NA`.
`as.numeric("dog")` → `NA` with warning.
`as.integer(TRUE)` → 1L. `as.integer(FALSE)` → 0L.
`as.character(TRUE)` → "TRUE". `as.character(1.5)` → "1.5".

Coercion in matrix/data.frame

Matrix: all elements same type (coerced). `matrix(c(1,"a"))` → all char.
Data frame: each column can be different type (list of columns).
`cbind()` on mixed types → coerces to most general (matrix).
`data.frame(x=1:3, y=c("a","b","c"))` → preserves types.

SPECIAL VALUES: NA, NULL, NaN, Inf

	NA	NULL	NaN	Inf
Meaning	Missing	Empty	0/0	∞
typeof	"logical"	"NULL"	"double"	"double"
In c()	Kept	Dropped	Kept	Kept
length	1	0	1	1
is.na()	TRUE	lgcl(0)	TRUE	FALSE
is.null()	FALSE	TRUE	FALSE	FALSE
is.finite()	FALSE	—	FALSE	FALSE

NA — Missing / Unknown

`NA == NA` → `NA` (must use `is.na()`).
NA in math: `NA + 1` → `NA`. `NA * 0` → `NA` (not 0!).
`NA & FALSE` → FALSE. `NA | TRUE` → TRUE. (determined!)
Typed: `NA_integer_`, `NA_real_`, `NA_character_`, `NA_complex_`.
NA in vector does NOT change vector's type.

NULL — Empty / Nonexistent

`c(4,5,NULL,3)` → `c(4,5,3)` (NULL silently dropped!).
`length(NULL)` → 0. `NULL + 1` → `numeric(0)`.
`NULL == NULL` → `logical(0)` (not TRUE!). `is.logical(NULL)` → FALSE.
Remove list elements: `l$x <- NULL`. Default arg: `f(x=NULL)`.

NaN and Inf

`0/0` → NaN. `1/0` → Inf. `-1/0` → -Inf. `log(0)` → -Inf.
`Inf - Inf` → NaN. `Inf + Inf` → Inf. `Inf * 0` → NaN.
`is.nan(NaN)` → TRUE. `is.na(NaN)` → TRUE (NaN is also NA!).
`is.infinite(Inf)` → TRUE. `max(c(1,Inf))` → Inf.
`NaN > 1` → NA. `Inf > ie308` → TRUE.

VECTOR ARITHMETIC & OPERATIONS

Element-wise operators (all recycle)

`+ - * / ^ %%` (modulo) `%/%` (integer div)
`5 %% 3` → 2. `5 %/% 3` → 1. `(-7) %% 3` → 2 (always ≥ 0).
`c(10,20,30) * c(1,2,3)` → `c(10,40,90)`.

Math functions (vectorized)

`abs()`, `sqrt()`, `ceiling()`, `floor()`, `round(x,n)`, `trunc()`,
`log()`, `log2()`, `log10()`, `exp()`, `sign()`.
▶ `round(2.5)` → 2 (banker's rounding! rounds to **even**).
`round(3.5)` → 4. `round(0.5)` → 0. `round(1.5)` → 2.

Aggregation functions

`sum()`, `prod()`, `min()`, `max()`, `range()`, `mean()`, `median()`,
`var()`, `sd()`, `length()`, `nchar()`.
▶ All return `NA` if input has NA! Fix: `na.rm=TRUE`.
Cumulative: `cumsum()`, `cumprod()`, `cummax()`, `cummin()`.

Set operations on vectors

`union(x,y)`, `intersect(x,y)`, `setdiff(x,y)`, `setequal(x,y)`.
`%in%`: logical, vectorized on LEFT only.
`c(1,2,3) %in% c(2,4)` → `c(F,T,F)`.

VECTOR RECYCLING

Shorter vector **repeated** to match longer.
▶ Warning only if longer length NOT multiple of shorter.
`c(1,2,3)+c(100,200,300,400,500,600)` → `c(101,202,303,401,502,603)` (no warn)
`c(1,2,3)+c(100,200,300,400,500)` → `c(101,202,303,401,502)` (**warning**)
Scalar recycling: `x * 2` recycles 2. No warning.
Matrix recycling: **column-wise** (down col 1, then col 2...).
Row-wise trick: `t(t(M) + x)`.
Logical recycling: `c(T,F)` recycles for every-other selection.
`cbind`/`rbind` recycle shorter to match longer.

ASSIGNMENT OPERATORS

`<-` → standard (preferred). `=` → only inside current scope.
`->` → right assignment (rare). `<<-` → super assignment (parent env).
▶ `f(x = 3)` passes arg. `f(x <- 3)` assigns AND passes!
`mean(x = 1:5)` → x NOT in global. `mean(x <- 1:5)` → x IS in global.

SUBSETTING ATOMIC VECTORS

4 methods

1. Positive int: `x[c(3,1)]` → 3rd, 1st. Dupes OK: `x[c(1,1)]`.
▶ Reals **truncated**: `x[2.9]` → same as `x[2]`.
2. Negative int: `x[-c(1,3)]` → all except 1st, 3rd.
▶ **Cannot mix +/-:** `x[c(-1,2)]` → **ERROR**.
3. Logical: `x[c(T,F,T,F)]` → 1st, 3rd. Recycled if shorter.
▶ NA in index → NA in output: `x[c(T,NA)]` → `val, NA, val, NA`.
4. Character: (named vec) `y["a"]`. No partial matching with `[`.
Names must match **exactly**: `z["a"]` on `c(abc=1)` → NA.
Lookup tables: `lookup[x]` to translate abbreviations.

Special cases

`x[]` → original. `x[0]` → zero-length vec.
OOB: `x[5]` on length-3 → `NA`. `x[[5]]` → **ERROR**.

SUBSETTING LISTS: [vs [[vs $

	`x[1]`	`x[[1]]` / `x$a`
Returns	**List** (train car)	**Contents** (inside)
typeof	Always "list"	Type of element
Multiple	Yes `x[1:3]`	**No** (single only)
Names	Preserved	Dropped

`x[[c(1,2)]]` → recursive: `x[[1]][[2]]`.
OOB: `l[4]` → list w/ NULL. `l[[4]]` → **ERROR**.
`l[NULL]` → empty list. `l[[NULL]]` → **ERROR**.
`$` does **partial matching**: `l$na` may match `l$name` silently!
`[[` does NOT partial match: `l[["na"]]` → NULL.

On atomic vectors

`x[1]` → keeps names. `x[[1]]` → drops names.
`x[5]` (OOB) → `NA`. `x[[5]]` (OOB) → **ERROR**.

SUBSETTING MATRICES & DATA FRAMES

Matrices

`a[1:2,]` → rows 1–2. `a[,c("B","A")]` → cols B, A.
Single row/col → **simplifies to vector**. Prevent: `drop=FALSE`.
Stored in **column-major**: `a[8]` = 8th element column-wise.
Matrix index: each row = coordinates. `vals[select]` → vector.

Data frames

Syntax	Returns	Style
`df["x"]`	data.frame	list (preserves)
`df[,"x"]`	vector	matrix (simplifies)
`df$x / df[["x"]]`	vector	list (simplifies)
`df[1,]`	data.frame row	matrix-style
`df[c(1,3)]`	cols 1,3 (df)	list (no comma!)

▶ `mtcars[mtcars$cyl<=5]` → **ERROR** (forgot comma — selects cols!)
Fix: `mtcars[mtcars$cyl<=5, ]`
▶ `mtcars$cyl==4|6` → always TRUE (`6` → TRUE).
Fix: `cyl==4 | cyl==6` or `cyl %in% c(4,6)`.
▶ `mtcars[1:13]` → ERROR (13 cols, only 11!). Fix: `mtcars[1:13,]`.

Simplifying vs Preserving

Type	Preserving	Simplifying
Atomic	`x[1]` (names)	`x[[1]]` (no names)
List	`x[1]` (list)	`x[[1]]` (contents)
Factor	`x[1]` (all lvls)	`x[1,drop=T]`
Matrix	`a[1,drop=F]`	`a[1,]` (vector)
Data frame	`df["x"]` (df)	`df[,"x"]` (vector)

LOGICAL OPERATORS

Vectorized vs Non-vectorized

Vectorized	Non-vectorized	Use in
`|` (OR), `&` (AND)	`||`, `&&`	`if()` only
Returns **vector**	Returns **single** value	Length-1 req'd

`||`/`&&`: length>1 → **ERROR** (R≥4.3).

Short-circuit evaluation

`FALSE && y` → FALSE (y not evaluated). `TRUE || y` → TRUE (y not evaluated).
`FALSE || y` → evaluates y → error if length>1. `TRUE && y` → same.

NA logic

	TRUE	FALSE
`_ | NA`	TRUE	NA
`_ & NA`	NA	FALSE

Rule: if result determined without NA, R returns it.
`xor(T,T)` → F. `xor(T,F)` → T. `!NA` → NA.
`any(logical(0))` → FALSE. `all(logical(0))` → TRUE.
"10" < "2" → TRUE (alphabetical!). "a" < "A" → TRUE.

Comparison operators

`==`, `!=`, `<`, `>`, `<=`, `>=` (vectorized, recycling).
`%in%`: left-vectorized. `c(1,2) %in% c(2,3)` → `c(F,T)`.
`is.___()`: `is.na`, `is.null`, `is.numeric`, `is.character`, `is.logical`, `is.integer`, `is.double`, `is.list`, `is.data.frame`, `is.factor`, `is.function`, `is.matrix`, `is.array`, `is.atomic`.

CONTROL FLOW

if() — requires single non-NA logical

Condition	Result
Length > 1	ERROR: "condition has length > 1"
NA	ERROR: "missing value where T/F needed"
Length 0	ERROR: "argument is of length zero"

`else` **must** be on same line as closing `}`.
Once TRUE, remaining `else if` blocks **skipped**.
`ifelse(cond,yes,no)` → **vectorized**. Evaluates ALL branches.
`case_when(c1~v1, c2~v2, TRUE~default)`: vectorized multi-cond.
Use `any()`/`all()` to reduce vector to single logical for `if()`.

Loops

`for(x in vec){...}`: iterates any vector/list.
`while(cond){...}`: after exit, counter one past end.
`repeat{...; if(...) break}`: infinite until `break`.
`break` → exit loop. `next` → skip to next iteration.
`for(i in 1:10){if(i%%2==0) break; cat(i)}` → prints 1 only.
`for(i in 1:10){if(i%%2==0) next; cat(i)}` → 1 3 5 7 9.

Pre-allocate!

`res <- rep(NA, n)` then fill. NOT `c(res,val)` (100x slower).
▶ `1:length(1)` when empty → `c(1,0)` (loops twice!).
Fix: `seq_along(1)` or `seq_len(n)` → `integer(0)` when empty.

Vectorization

`rowSums(x)`/`ncol(x)` much faster than loop + `mean()`.
`apply(x,1,mean)` slower than `rowSums` approach.
`colSums()`, `colMeans()`, `rowMeans()` → fast built-ins.
Vectorized ops avoid explicit loops and are 10–100x faster.

Operator precedence (high to low)

`^` > unary `-/+` > `%%/%/%*%/%in%` > `*/` > `+/-` > `< > <= >= == !=` > `!` > `&/&&` > `|/||` > `~` > `->` `->>` > `<-` `<<-`
▶ `!x == y` means `(!x) == y`, not `!(x==y)`.
`-2^2` → -4 (power before negation!). `(-2)^2` → 4.
`1:5 + 1` → `c(2,3,4,5,6)` (`:` binds tighter than `+`).
`1:(5+1)` → `c(1,2,3,4,5,6)`.

FUNCTIONS

`name <- function(arg1, arg2=default) { body }`
`typeof(f)` → `"closure"`. `mode(f)` → `"function"`.
Implicit return: last expression. **Explicit:** `return()`.
Return multiple: `list()`. One line: `f <- function(x) x*x`.
Arg matching: 1) exact name, 2) positional.
`f(y=2, 1, 3)` → y=2, then 1 → x, 3 → z (positional fill).
Missing used arg → "arg missing, no default". Extra → "unused arg".
Unused arg in body → **no error** (lazy evaluation).
Scoping: vars inside function are **local**. Must assign: `x <- f(x)`.
`rm(list=ls())` inside function only removes **local** objects.
`...` (dots): passes extra args to inner functions.
`f <- function(x, ...) plot(x, ...)` → forwards to `plot()`.

USEFUL BASE R FUNCTIONS

Searching & matching

`which(x > 5)` → **indices** where TRUE. Ignores NA.
`which.min(x)`, `which.max(x)` → index of min/max.
`match(x, table)` → 1st index of x in table (`NA` if none).
`match(c(3,5,1), c(1,2,3))` → `c(3,NA,1)`.

Sequence & repetition

`seq(from, to, by)` or `seq(from, to, length.out=n)`.
`seq_along(x)` → safe `1:length(x)`. `seq_len(n)` → safe `1:n`.
`rep(x, times=3)` → repeat whole 3x. `rep(x, each=3)` → each 3x.
`rep(c(1,2), times=c(3,2))` → `c(1,1,1,2,2)`.

Sorting & ordering

`sort(x)` → sorted values. `rev(x)` → reversed.
`order(x)` → indices that would sort. `x[order(x)]` → `sort(x)`.
`df[order(df$age),]` → sort df by age. `order(x, decreasing=T)`.

Counting & uniqueness

`table(x)` → freq table. `table(x,y)` → cross-tab.
`unique(x)` → unique values. `duplicated(x)` → logical.
`nrow()`, `ncol()`, `dim()`, `str()`, `summary()`, `head()`, `tail()`.

Apply family

`lapply(x, f)` → always returns **list**.
`sapply(x, f)` → tries to simplify to vector/matrix.
`vapply(x, f, FUN.VALUE)` → like sapply with type safety.
`apply(M, 1, f)` → rows. `apply(M, 2, f)` → cols.
`tapply(x, group, f)` → by group. `mapply(f, x, y)` → parallel.
`do.call(f, list(a,b))` → same as `f(a,b)`.
▶ `sapply` returns list OR vector (unpredictable). Use `vapply`.

Missing data

`complete.cases(df)` → logical, rows with no NA.
`na.omit(df)` → remove rows with any NA.
`is.na(x) <- 3` → sets `x[3]` to NA.

ENVIRONMENTS & SCOPING

Environment: bag of name → value bindings. **Not ordered**.
`e[[1]]` → ERROR. `[` doesn't work (only `[[` and `$`).
`e$d <- NULL` does NOT remove d. Use `rm(d, envir=e)`.
Copy-on-modify: `y<-x`; modify y copies, x unchanged.
Name binding: `y<-x` binds to **same** object until modified.

4 special environments

`globalenv()`: user workspace. `baseenv()`: base package.
`emptyenv()`: ultimate ancestor (**no parent**).
`search()` → search path. `library()` inserts package.

Lexical scoping (KEY RULE)

R looks where function was **DEFINED** (enclosing env),
NOT where it was **CALLED** (calling env).

4 function environments

Type	What
Enclosing	Where function was **created**
Binding	Where function's **name** exists
Execution	**Temporary**, created per call
Calling	Where function was **called from**

`parent.frame()` → calling env (dynamic scoping).
`get("y", envir=parent.frame())` → looks in calling env.

SUPER ASSIGNMENT & SCOPING EXAMPLES

`<-` **never** creates in current env; modifies in parent.
Climbs scope; if not found, creates in global.
▶ **Stops at FIRST scope where variable exists!**

Ex1: Nested (lexical scoping)

```
x<-1; y<-1; z<-1
f<-function(){y<-2; g<-function(){z<-3; x+y+z}; g()}
```
`f()` → 6 (x=1 global, y=2 f, z=3 g). Global x,y,z still 1.

Ex2: Separate functions (KEY!)

```
y<-10; g<-function(){y}; h<-function(){y<-100; g()}
```
`h()` → 10 (g defined in global, finds global y=10).
▶ g() finds **global** y, NOT h()'s local y=100.

Ex3: parent.frame()

```
g<-function(){get("y",envir=parent.frame())}
h<-function(){y<-100; g()}; h() → 100 (calling env).
```
`g()` → 10 (calling env is global).

Ex4: Super assignment

```
x<-1; y<-1; z<-1
f<-function(){y<-2; y<-4; g<-function(){z<-3; x+y+z}; g()}
```
`f()` → 6. Global: y=4, z=3. `y<-2` local, `y<-4` global.

Ex5: <<- stops at first scope

```
f<-function(){y<-2; z<-10; y<-4; g<-function(){z<-3; x+y+z}; g()}
```
`f()` → 6. Global z=1 unchanged! `z<-3` modifies f's z=10.

Danger of <<-

```
foo_bad<-function(foo,x){foo<<-c(foo,x^2)}; bar<-10; foo_bad(bar,6)
```
`<-` targets name "foo", not passed variable. bar unchanged!

DPLYR CORE VERBS

All: 1st arg = df, return new df. Pipe: `|>` ("then").

select() — choose columns

`select(df, name, age)`, `select(df, -col)`, `select(df, a:d)`.
Rename: `select(df, new=old)`. Also: `rename(df, new=old)`.
Helpers: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `everything()`, `all_of(v)`, `any_of(v)`, `last_col()`, `where(is.numeric)`.

filter() — keep rows

`filter(df, species == "Human")`. Comma = AND.
`filter(df, x %in% c("a","b"))` for multiple values.
▶ `filter(df, species = "X")` → error (use `==` not `=`).
▶ `species=="A"|"B"` → all TRUE (`"B"` → `TRUE`).

mutate() — add/transform columns

Keeps all cols. Can reference just-created cols in same call.
`.before = everything()` puts new cols first.
`transmute()` → like mutate but drops original cols.

Other verbs

`arrange(df, desc(col))` → sort. `slice(df, 5:10)`.
`slice_sample(n=5)`, `slice_max(col, n=3)`, `slice_min()`.
`distinct(df, col1, .keep_all=TRUE)` → unique rows.
`pull(df, col)` → vector. `relocate(df, col, .after=ref)`.
`count(df, col)` → shortcut for `group_by + summarize(n=n())`.
`if_else(cond, T, F)`: strict types. `case_when(c~v, TRUE~d)`.

Useful mutate functions

`pmin()/pmax()` → row-wise min/max. `coalesce()` → first non-NA.
`cumsum()`, `cummean()`. `lead(x,n)/lag(x,n)` → shift.
`between(x,a,b)` → $a \leq x \leq b$. `ntile(x,n)` → n bins.
`n()` → rows in group. `n_distinct(x)` → unique count.
`row_number()`, `min_rank()`, `dense_rank()` → ranking.

GROUP_BY, SUMMARIZE, JOINS

```
df |> summarize(m=mean(x, na.rm=T), cnt=n())
df |> group_by(sp) |> summarize(m=mean(ht))
```
→ one row/group.
`group_by + mutate` → within-group (z-scores). `ungroup()`.
Multiple: `group_by(a,b)`. Each summarize peels one layer.

Joins

Join	Keeps
`left_join(x,y)`	All x rows, match from y
`right_join(x,y)`	All y rows, match from x
`inner_join(x,y)`	Only matching (intersection)
`full_join(x,y)`	All rows from both (union)
`semi_join(x,y)`	x rows w/ match, no cols added
`anti_join(x,y)`	x rows w/ **no** match

`left_join(x, y, by=join_by(a == b))`. Same name: `join_by(x)`.
No match → `NA`. Non-unique → **Cartesian product**.
Set ops: `intersect()`, `union()`, `setdiff()` on rows.

TIDYR: RESHAPING

```
pivot_longer(df, cols=c1:c3, names_to="year", values_to="n")
cols: -country, starts_with("20"), 2:4. Unpivoted cols duplicated.
pivot_wider(df, id_cols=city, names_from=sz, values_from=amt)
values_fill=0. ▶ "NYC" vs "New York" → separate rows!
```
They are **inverse** operations. `separate()`/`unite()` for string cols.

STRINGS & REGEX

```
paste("a",1:3,sep="") → c("a1","a2","a3"). paste0=sep="".
collapse="," → "a1,a2,a3" (single string).
cat() → prints w/o quotes, returns invisible(NULL).
nchar("hi") → 2. substr("hello",2,4) → "ell".
sprintf("%d items at $%.2f", 3, 9.5) → "3 items at $9.50".
```

stringr functions

Function	Returns
`str_detect(s,p)`	logical vector
`str_extract(s,p)`	matched text or NA
`str_match(s,p)`	matrix: full + groups
`str_replace(s,p,r)`	1st match replaced
`str_split(s,p)`	list of pieces
`str_count/locate/sub`	count / position / substring

`_all` suffix: ALL matches. `str_extract_all` → **list**.

Regex metacharacters (escape: \\)

`.` → any char. `[^aeiou]` → negation. `\\d/\\w/\\s` → digit/word/space.
`^` → start. `$` → end. `\\b` → word boundary.
Caret 3 meanings: `^[0-9]` start; `[^0-9]` negate; `[0-9^]` literal.
Inside `[]`: most metachar literal. Exceptions: `]` `-` `^` `\`.

Quantifiers & Greedy/Lazy

`*` 0+, `+` 1+, `?` 0/1, `{n}`, `{n,}`, `{n,m}`.
Greedy (default): longest. Add `?` for **lazy** (shortest).
`str_extract("Peter Piper","P.*r")` → "Peter Piper" (greedy).
`str_extract("Peter Piper","P.*?r")` → "Peter" (lazy).

Groups & Backreferences

`(abc)` → capture. `(?:abc)` → non-capture. `|` → alternation.
▶ Order matters: `(Mr|Mrs)` on "Mrs." → "Mr" (first wins!).
`\\1`/`\\2` in replacement reference groups.
`str_match` → matrix: col1=full, col2+=groups.

Lookarounds (zero-width, not consumed)

`(?=...)` → pos ahead. `(?!...)` → neg ahead.
`(?<=...)` → pos behind. `(?<!...)` → neg behind.
Lookbehind: **bounded length only** (no `*`/`+`).

S3 OOP

Generic function OOP: methods belong to **functions**.
Generic calls `UseMethod("name")`. Methods: `generic.class()`.
Create: `structure(list(), class="fruit")` or `class(y)<-"fruit"`.
No formal definition; just set class attr. Class can be vector.
▶ Use `inherits(x,"fruit")` not `class(x)=="fruit"`.

Dispatch order

`UseMethod("gen.class1", gen.class2, ...)` → `gen.default`.
No match + no default → ERROR. Direct `f.j(x)` bypasses dispatch.
`class(k)<-NULL` → implicit class (double/numeric then default).

Dispatch examples

```
f<-function(x) UseMethod("f"); f.jc<-function(x) x+2; f.k<-function(x) x+10
k<-1; f(k) → ERROR (no f.double/f.numeric/f.default).
class(k)<-"jc"; f(k) → 3. Add f.default<-function(x) x+100.
structure(10,class=c("k","l")) → f.k first. c("m","n") → f.default.
```
`f(7)` → 107 (double). `f.j(7)` → 9 (direct, bypasses dispatch).
▶ 7 is double; `7L` is integer! `f.integer<-function(x) 100*x`
`f(7)` → 107 (double!). `f(7L)` → 700 (integer!).

UseMethod mismatch

`f<-function(x) UseMethod("g")` → dispatches `g.class` not `f.class`!

R6 OOP

Encapsulated OOP: methods belong to **objects**. `library(R6)`.

```
Cls <- R6Class("Cls", public=list(val=0, set=function(x){
self$val<-x; invisible(self)})). Instance: x <- Cls$new().
```
`self$` → own fields. `invisible(self)` → method chaining.
`$initialize()` → overrides `$new()`. `$print()` → custom print.
Methods bound at creation; redefining class won't update old objects.
`$set("public","field",val)` → add after creation (new objs only).

Inheritance & Reference semantics

```
R6Class("Child", inherit=Parent). super$method() for parent.
class(x) → c("Child","Parent","R6").
y <- x → SAME object! y$set(10) changes x too!
```
`y <- x$clone()` → independent. `$clone(deep=TRUE)` → deep copy.
Functions can modify R6 args without assignment: `f(x)` changes x!

COMMON TRAPS — QUICK REFERENCE

Trap	Reality
`typeof(df)`	`"list"` not `"data.frame"`
`typeof(factor)`	`"integer"` not `"factor"`
`7 vs 7L`	double vs integer (S3 dispatch!)
`1:0`	`c(1,0)` not empty! Use `seq_len`
`c(1,NULL,3)`	`c(1,3)` (NULL dropped)
`NA == NA`	`NA` (use `is.na()`)
`NA * 0`	`NA` (not 0!)
Matrix fill	Column-wise (not row-wise)
`as.numeric(factor)`	Gives indices, not values
`x[2.9]`	Same as `x[2]` (truncated)
`<-`	Stops at first scope w/ var
R6 `y<-x`	Shared ref (use `$clone()`)
Greedy regex	Longest; `?` for shortest
`[` on list	Always returns list
`if(vec)`	ERROR if length>1
`f.j(7)`	Bypasses S3 dispatch
`UseMethod("g")`	Dispatches `g.class` not `f.class`
`df[1:13]` no comma	Selects columns, not rows!
`as.logical("t"/"1")`	`NA` (only full words)
`as.logical(c("0",1))`	All `NA` (char coercion first)
`factor` bad assign	`NA` with warning
`"10" < "2"`	`TRUE` (alphabetical!)
`any(logical(0))`	`FALSE`
`all(logical(0))`	`TRUE`
`(Mr|Mrs)`	`"Mr"` first wins on "Mrs"
`round(2.5)`	2 (banker's rounding)
`is.numeric(1L)`	`TRUE` (int IS numeric)
`identical(1, 1L)`	`FALSE` (type matters)
`sapply empty`	Returns `list()` not vector
`l$na`	Partial match → wrong result
`f(x=3) vs f(x<-3)`	`=` arg only; `<-` also assigns
`Inf - Inf`	`NaN` (not 0)
`(-7) %% 3`	2 (R modulo always ≥ 0)
`nchar(NA)`	`NA` (not 2!)
`NULL + 1`	`numeric(0)` (not error)
`log(0)`	`-Inf` (not error)
`Inf * 0`	`NaN` (not 0)
`is.na(NaN)`	`TRUE` (NaN is NA!)
`NULL == NULL`	`logical(0)` (not TRUE)
`e$d <- NULL` in env	Does NOT remove d
`for(i in 1:0)`	Loops **twice** (1,0)

ERROR HANDLING

`stop("msg")` → raises error. `warning("msg")` → warning.
`message("msg")` → message (to stderr).
`tryCatch(expr, error=function(e) ..., warning=function(w) ...)`.
`try(expr)` → returns result or try-error object (no stop).
`stopifnot(x > 0)` → stops if condition FALSE.
`suppressWarnings(expr)`, `suppressMessages(expr)`.

MISC R TRICKS

`switch(x, "a"=1, "b"=2, 3)` → match x; last = default.
`Sys.time()` → current datetime. `proc.time()` → CPU time.
`system.time(expr)` → time an expression.
`sample(1:10, 5)` → 5 random from 1:10 (no replace).
`sample(1:10, 5, replace=TRUE)` → with replacement.
`set.seed(42)` → reproducible randomness.
`print(x)` returns x invisibly. `cat()` returns `invisible(NULL)`.
`invisible(x)` → return x without printing.
`exists("x")` → TRUE if x exists in scope.
`get("x")` → get value by name string. `assign("x", 5)` → set.
`nargs()` → number of args passed to current function.
`missing(x)` → TRUE if arg x was not supplied.
`is.function(mean)` → TRUE. `is.primitive(sum)` → TRUE.
`environment(f)` → enclosing env of f. `body(f)` → function body.
`formals(f)` → list of formal arguments with defaults.
`methods("print")` → all print methods. `methods(class="factor")`.
`paste(1:3, collapse="+")` → `"1+2+3"`.
`strsplit("a.b.c", "`

`.")` → `list(c("a","b","c"))`.
`gsub("old","new",x)` → replace all in base R.
`sub("old","new",x)` → replace first in base R.
`grepl("pattern",x)` → logical (base R `str_detect`).
`grep("pattern",x)` → indices of matches.
`regmatches(x, regexpr(p, x))` → base R extract.
`trimws(" hi ")` → `"hi"` (trim whitespace).
`startsWith("abc","ab")` → TRUE. `endsWith("abc","bc")` → TRUE.
`sprintf("%05d", 42)` → `"00042"` (zero-padded).
`sprintf("%.1e", 1234)` → `"1.2e+03"` (scientific).