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 & SPECIAL VALUES

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.
`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.

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 > 1e308` → TRUE.

OPERATORS, RECYCLING & PRECEDENCE

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)`.

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.

Logical operators

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

`||`/`&&`: length>1 → **ERROR** (R≥4.3).
`==`, `!=`, `<`, `>`, `<=`, `>=` (vectorized, recycling).

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.

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)`.

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**.

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: `x[1]` → keeps names. `x[[1]]` → drops names.

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)

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(l)` when empty → `c(1,0)` (loops twice!).
Fix: `seq_along(l)` or `seq_len(n)` → `integer(0)` when empty.

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()`.

S3 OOP

Setup — one example for everything

`f <- function(x) UseMethod("f")` *generic (must have UseMethod)*
`f.j <- function(x) x+2` *method for class "j"*
`f.k <- function(x) x+10` *method for class "k"*
`f.default <- function(x) x+100` *fallback if no match*

How dispatch works: f(obj) traces

`f(obj)` → R reads `class(obj)`, tries `f.class1`, `f.class2`, ..., `f.default`, else ERROR.

`k <- 1` *no class attr, k is a plain double*
`f(k)` → try `f.double`? **No** → `f.numeric`? **No** → `f.default` → 1 + 100 = 101

`class(k) <- "j"` *now k has class "j"*
`f(k)` → try `f.j`? **Yes** → 1 + 2 = 3

`f(structure(10, class=c("k","l")))` *class vector: try left to right*
→ try `f.k`? **Yes** (first match) → 10 + 10 = 20

`f(structure(10, class=c("m","n")))` *no f.m or f.n*
→ `f.m`? No → `f.n`? No → `f.default` → 10 + 100 = 110

Implicit classes (no class attr set)

7 is double → dispatch tries: `f.double` → `f.numeric` → `f.default`

7L is integer → dispatch tries: `f.integer` → `f.numeric` → `f.default`
`f.integer <- function(x) 100*x`
`f(7)` → 107 (double path). `f(7L)` → 700 (integer path).

Traps (using same example)

▶ `f.j(7)` → 9: direct call **skips dispatch**, just runs 7 + 2!
▶ `f <- function(x) UseMethod("g")` → dispatches `g.class` not `f.class`!
▶ Generic without `UseMethod()` → methods **never** called.
▶ No matching method + no `.default` → **ERROR**.

Creating S3 objects

```
obj <- structure(list(name="apple"), class="fruit")
```
Or: `obj <- list(...); class(obj) <- "fruit"`
Inheritance: `class(obj) <- c("apple","fruit")` (tries left to right).
▶ Use `inherits(obj,"fruit")`, NOT `class(obj)=="fruit"`.

R6 OOP

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

Example: Clock class

```
Clock <- R6Class("Clock", public = list(
hour=0, minute=0, second=0,
add_hour = function(x) {
self$hour <- (self$hour + 1)%%24
invisible(self) },
add_minute = function(x) {
self$minute <- (self$minute + 1)%%60
invisible(self) },
add_second = function(x) {
self$second <- (self$second + 1)%%60
invisible(self) },
print = function(...)  {
cat("Hour:", self$hour, "\n")
cat("Minute:", self$minute, "\n")
cat("Second:", self$second, "\n")
invisible(self) }))
```

Usage: new(), chaining, $set()

```
clock1 <- Clock$new()
clock1$add_hour()$add_hour()$add_second()
clock1    →    Hour:  2 / Minute:  0 / Second:  1
Clock$set("public", "clock_type", "analog")
clock1    →    still Hour:  2 / Minute:  0 / Second:  1
```
▶ `$set()` only affects **new** objects, not existing ones!

Key concepts

`self$` → access own fields/methods. `invisible(self)` → chaining.
`$initialize()` → overrides `$new()`. `$print()` → custom print.
▶ **Without `invisible(self)`, chaining FAILS!**
`clock1$add_hour()$add_hour()` → ERROR: "$ operator is invalid for atomic vectors" (method returns NULL, not self).

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!

DPLYR

```
df <- data.frame(name=c("A","B","C","A"),
age=c(20,25,30,22), score=c(80,90,70,85))
```

	name	age	score
1	A	20	80
2	B	25	90
3	C	30	70
4	A	22	85

select() — choose columns

`df |> select(name, age)` → keeps only name, age cols
`df |> select(-score)` → same (drop score)
`df |> select(n=name)` → renames name to n

filter() — keep rows

`df |> filter(age > 22)` → rows 2,3 (B/25/90, C/30/70)
`df |> filter(name=="A", age>20)` → row 4 only (comma=AND)
▶ `filter(df, name = "A")` → **error**! Use == not =.

mutate() — add/transform columns

`df |> mutate(pass = score >= 80)` → adds col:

name	age	score	pass
A	20	80	TRUE
B	25	90	TRUE
C	30	70	FALSE
A	22	85	TRUE

Keeps all cols. `transmute()` → only new cols.

arrange() — sort rows

`df |> arrange(age)` → 20, 22, 25, 30
`df |> arrange(desc(score))` → 90, 85, 80, 70

group_by + summarize

```
df |> group_by(name) |>
summarize(avg=mean(score), n=n())
```

name	avg	n
A	82.5	2
B	90	1
C	70	1

`n()` → count rows. `n_distinct(col)` → unique count.
`group_by + mutate` → adds col within group.
`ungroup()` to remove grouping after.

Other verbs

`df |> count(name)` → A:2, B:1, C:1
`df |> distinct(name)` → A, B, C
`df |> slice(1:2)` → first 2 rows
`df |> pull(age)` → c(20,25,30,22) (vector!)

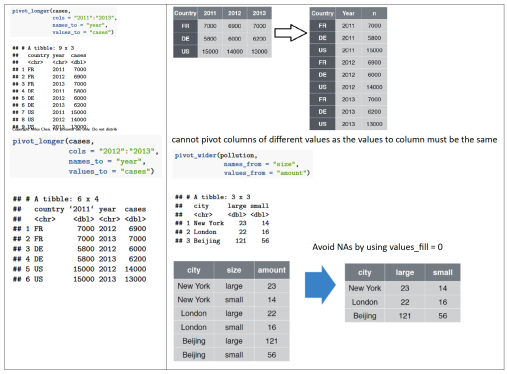
Joins

```
x = data.frame(id=c(1,2,3), n=c("A","B","C"))
y = data.frame(id=c(2,3,4), s=c(90,80,70))
```

Join	Result (id, n, s)
left_join(x,y)	1/A/NA, 2/B/90, 3/C/80
right_join(x,y)	2/B/90, 3/C/80, 4/NA/70
inner_join(x,y)	2/B/90, 3/C/80
full_join(x,y)	1/A/NA, 2/B/90, 3/C/80, 4/NA/70
semi_join(x,y)	2/B, 3/C (no y cols added)
anti_join(x,y)	1/A (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



STRINGS & REGEX

stringr functions

`s <- c("apple 3", "banana 12", "cherry")`

Function	Example & Result
str_detect(s,p)	str_detect(s,"\\d") → T, T, F
str_extract(s,p)	str_extract(s,"\\d+") → "3","12",NA
str_match(s,p)	str_match(s,"(\\w+) (\\d+)") → matrix col1=full, col2="apple","banana",NA
str_replace(s,p,r)	str_replace(s,"a","X") → "Xpple 3",...
str_replace_all	str_replace_all(s,"a","X") → "Xpple 3","bXnXnX 12",...
str_split(s,p)	str_split("a.b.c","\\.") → list(c("a","b","c"))
str_count(s,p)	str_count(s,"a") → 1, 3, 0

_all suffix: ALL matches. `str_extract_all` → **list** (not vector!).

Regex metacharacters (escape: \\)

Symbol	Meaning	Example
.	any single char	"h.t" → "hat","hot"
^	start of string	"^Hi" → "Hi there"
$	end of string	"end$" → "the end"
\\d	digit (0–9)	"\\d+" → "123"
\\w	word (a–z,0–9,_)	"\\w+" → "hi_2"
\\s	whitespace	"\\s+" → " "
\\b	word boundary	"\\bcat\\b" skips "catch"
[^...]	negation in []	[^aeiou] = not vowel

Caret ^ — 3 meanings by position

"^Hi" → matches "Hi there", not "say Hi" (start anchor)
"[^aeiou]" → matches "b","c","d",... (negation inside [])
"[abc^]" → matches a,b,c,^ (literal when not first in [])
Inside []: most metachar literal. Exceptions:] - ^ \.

Quantifiers & Greedy/Lazy

Quant.	Meaning	Example
*	0 or more	"ab*" → "a","ab","abbb"
+	1 or more	"ab+" → "ab","abbb"
?	0 or 1 (optional)	"ab?" → "a" or "ab"
{n}	exactly n	"b{2}" → "bb"
{n,}	n or more	"b{2,}" → "bb","bbb"
{n,m}	between n and m	"b{1,2}" → "b","bb"

Greedy (default): longest match. Add ? after 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 */+).

WHEN DOES R OUTPUT NA / NaN / NULL / Inf?

NA — "I don't know"

Situation	Example
Any math with NA	NA + 1 → NA, NA * 0 → NA
Comparison with NA	NA == NA → NA, NA > 5 → NA
Logic (undetermined)	NA & TRUE → NA, NA | FALSE → NA
OOB with [	x[10] on length-3 → NA
No match	match(5, c(1,2)) → NA
Coercion failure	as.numeric("dog") → NA
Bad factor level	assign value not in levels → NA

NaN — "Undefined math" (is also NA!)

Situation	Example
0 divided by 0	0/0 → NaN
Inf minus Inf	Inf - Inf → NaN
Inf times 0	Inf * 0 → NaN
Negative sqrt/log	sqrt(-1) → NaN, log(-1) → NaN

`is.na(NaN)` → TRUE (NaN is a kind of NA). `is.nan(NA)` → FALSE.

NULL — "Nothing exists"

Situation	Example
Nonexistent list element	list()$x → NULL
Dropped in c()	c(1, NULL, 3) → c(1, 3)
NULL in math	NULL + 1 → numeric(0)
Length is 0	length(NULL) → 0

Inf — "Infinity"

Situation	Example
Divide by zero	1/0 → Inf, -1/0 → -Inf
Log of zero	log(0) → -Inf
Inf in math	Inf + 1 → Inf

Quick contrast

0/0 → NaN. 1/0 → Inf. NA+1 → NA. NULL+1 → numeric(0).