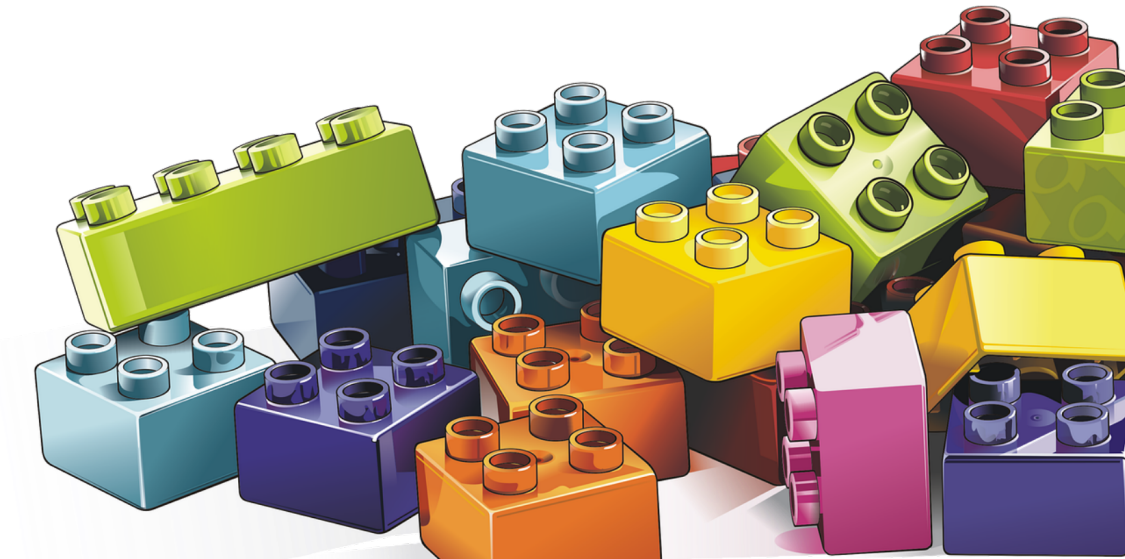
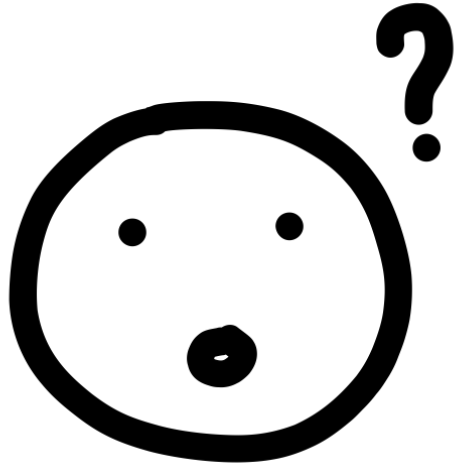
Objects and data types

Session overview

1. Objects and object assignment
2. Data types (and how to define them)
3. Missing values and factors
4. More complex objects



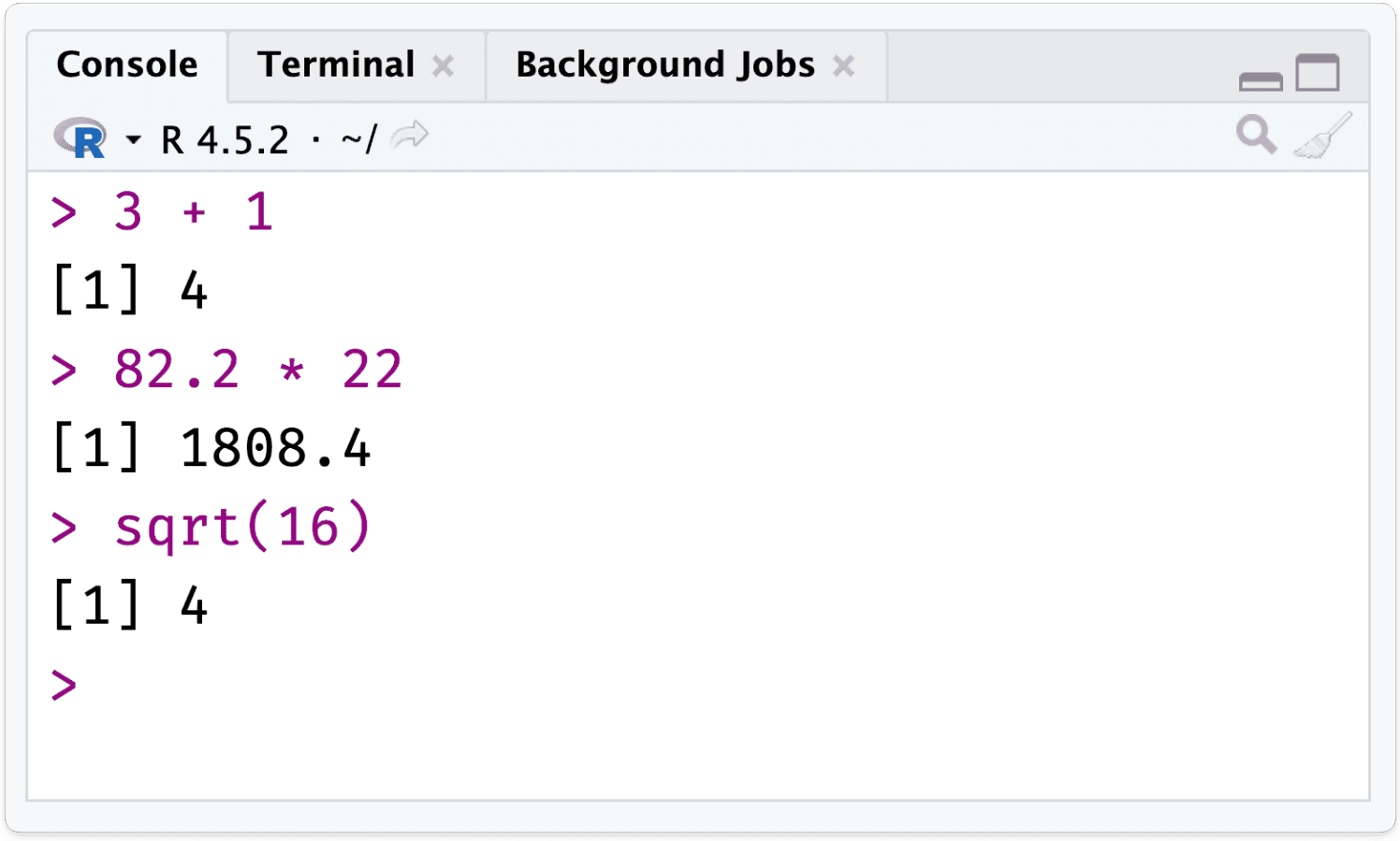


These topics are fairly abstract.

It will make sense later.

Objects and object assignment

The R console can be used interactively, like a calculator.



For more complex operations, we need to store objects in memory.

- Use the result of one calculation as the input for another.
- Load data and analyse it.

We need a way to name and refer to objects.

The assignment arrow

We can store something as an object by giving it a name:

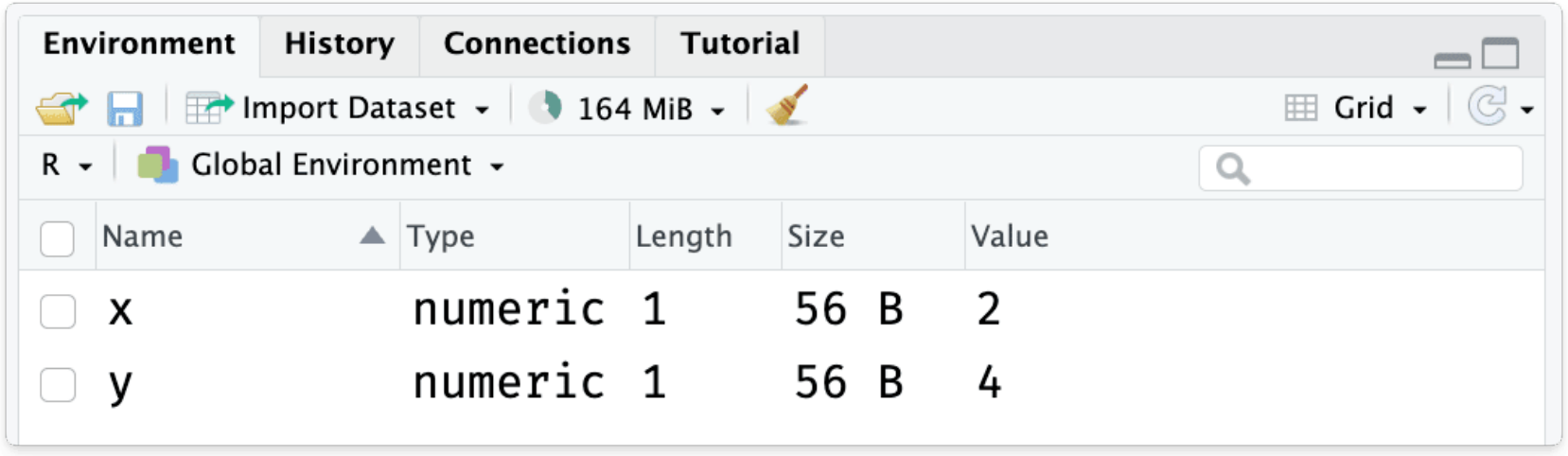
```
x <- 2
y <- 4
```

We can then use stored objects in subsequent calculations:

```
z <- x * y
```

⌨ Have a go now.

Created objects live in the global environment. We can view them with `ls()` or in the 'Environment' pane:



They stay there until we restart R.

We can remove them with `rm()` or via the 'Environment' pane in RStudio.

In R, almost everything you work with is an object.

- Numbers, strings, vectors, lists, and data frames are objects.
- Functions are objects.
- Even model fits and plots are objects you can store and reuse.

Vectors

Vectors are the simplest object

A vector is a one-dimensional object containing values of a single type.

We can create them with `c()`.

```
x1 <- 5
x2 <- c(5, 7)
x3 <- c("A", "B", "C", "D")
```

Objects of length 1, 2, or 3 are still vectors.

Creating vectors

```r
# By hand
one_to_five <- c(1, 2, 3, 4, 5)

# Using the 'seq' function
lazy <- seq(from = 1, to = 5, by = 1)

# Same, but without naming the arguments
lazier <- seq(1, 5, 1)

# Using ':'
laziest <- 1:5
```

Data types

Every object in R has a type

```
x <- 1
y <- "A"
z <- TRUE
```

We can use `typeof` and `str` to inspect an object's type and structure:

```
> typeof(x)
[1] "double"
> typeof(y)
[1] "character"
> typeof(z)
[1] "logical"
> str(x)
 num 1
```

You'll come across several data types in R:

- Numeric (e.g., `1.0`, `2e12`)
- Integer (e.g., `1L`)
- Character (e.g., `"like this"`)
- Logical (e.g., `TRUE`, `FALSE`)
- Factor
- Missing values (e.g., `NA`)
- Dates, times, and intervals
- ...

R has standard ways to create each type and convert between them.

Numeric vs. integer

Most numbers you type are numeric (stored as double).
You can create an integer explicitly with the `L` suffix.

```
x <- 1          # Numeric (double)
z <- 1.32       # Numeric (double)
y <- 1L         # Integer
```

R provides standard functions to check and convert types:

```
typeof(z)       # "double"

is.numeric(x)   # TRUE
is.integer(y)   # TRUE

as.integer(x)   # 1L
as.numeric(y)   # 1
```

Sidebar: What does double mean in R?

- Most numbers (e.g., `1`, `3.14`) are stored as double.
- A double is a 64-bit floating-point number (IEEE 754).
- Decimals are stored *approximately*, so small rounding effects can appear.

```
> typeof(1)
[1] "double"
> typeof(1L)
[1] "integer"

> sprintf("%.20f", 1)
[1] "1.00000000000000000000"
> sprintf("%.20f", 0.1 + 0.2)
[1] "0.30000000000000004441"
```

Characters (or 'strings')

```
> first <- "Joe"
> last <- "Bloggs"
> age <- "40"

> is.character(first)
[1] TRUE

> paste(first, last)
[1] "Joe Bloggs"

> age + 10
Error in age + 10 : non-numeric argument...
> age <- as.numeric(age)
> age + 10
[1] 50
```

Logical (or 'boolean') values

```
> 5 > 4
[1] TRUE

> "Joe" == "Bloggs"
[1] FALSE

> "Joe" == "Joe"
[1] TRUE

> typeof(TRUE)
[1] "logical"

> str(TRUE)
 logi TRUE
```

```
> !(TRUE)
[1] FALSE

> TRUE & FALSE
[1] FALSE

> TRUE | FALSE
[1] TRUE

> any(TRUE, FALSE, FALSE)
[1] TRUE

> all(TRUE, FALSE, FALSE)
[1] FALSE
```

Type coercion (when types mix)

When you mix types in a vector, R coerces them to a single common type.

```
> c(1, "2")
[1] "1" "2"
> c(TRUE, 2)
[1] 1 2
```

Watch out: one character (or `NA`) can silently change the whole vector.

```
> typeof(c(TRUE, 2))
[1] "double"
> typeof(c(1L, 2.5))
[1] "double"
> typeof(c(1, "2"))
[1] "character"
```

Dates with `lubridate`

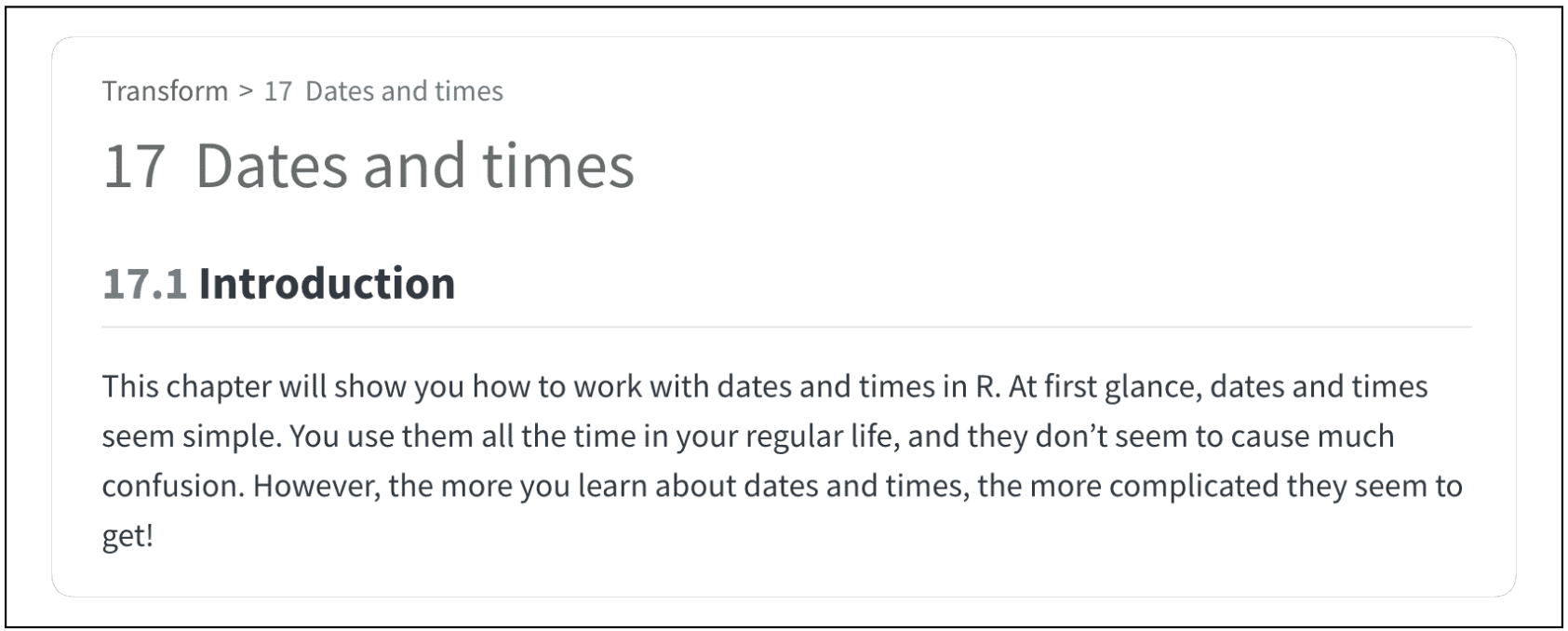
Use `ymd()` to parse ISO-style strings into `Date` objects:

```
> library(lubridate)
> d <- ymd("2026-02-09")
> d
[1] "2026-02-09"
> class(d)
[1] "Date"
```

Use `days()`, `months()`, and `years()` for readable calendar arithmetic:

```
> d + days(7)
[1] "2026-02-16"
> d + months(7)
[1] "2026-09-09"
```

```
> d1 <- ymd("2023-01-22")
> d2 <- ymd("2024-07-12")
> gap <- interval(d1, d2)
> gap
[1] 2023-01-22 UTC--2024-07-12 UTC
> interval(d1, d2) / weeks(1)
[1] 76.71429
```



Transform > 17 Dates and times

17 Dates and times

17.1 Introduction

This chapter will show you how to work with dates and times in R. At first glance, dates and times seem simple. You use them all the time in your regular life, and they don't seem to cause much confusion. However, the more you learn about dates and times, the more complicated they seem to get!

https://r4ds.hadley.nz/datetimes.html

Missing values and factors

Missing values are represented with `NA`

We can check for missing values:

```
> is.na(NA)
[1] TRUE
```

Do not confuse `NA` with `NaN`.

> *NaN, not a number, a numeric value representing an undefined or unrepresentable value.*

```
> 0/0
[1] NaN
> is.nan(0/0)
[1] TRUE
```

```
> is.nan(1)
[1] FALSE
> is.na(NaN)
[1] TRUE
```

Binary values

We can represent binary values as logical (`TRUE`/`FALSE`) or numeric (`0`/`1`).

For example, create a new binary variable with a conditional expression:

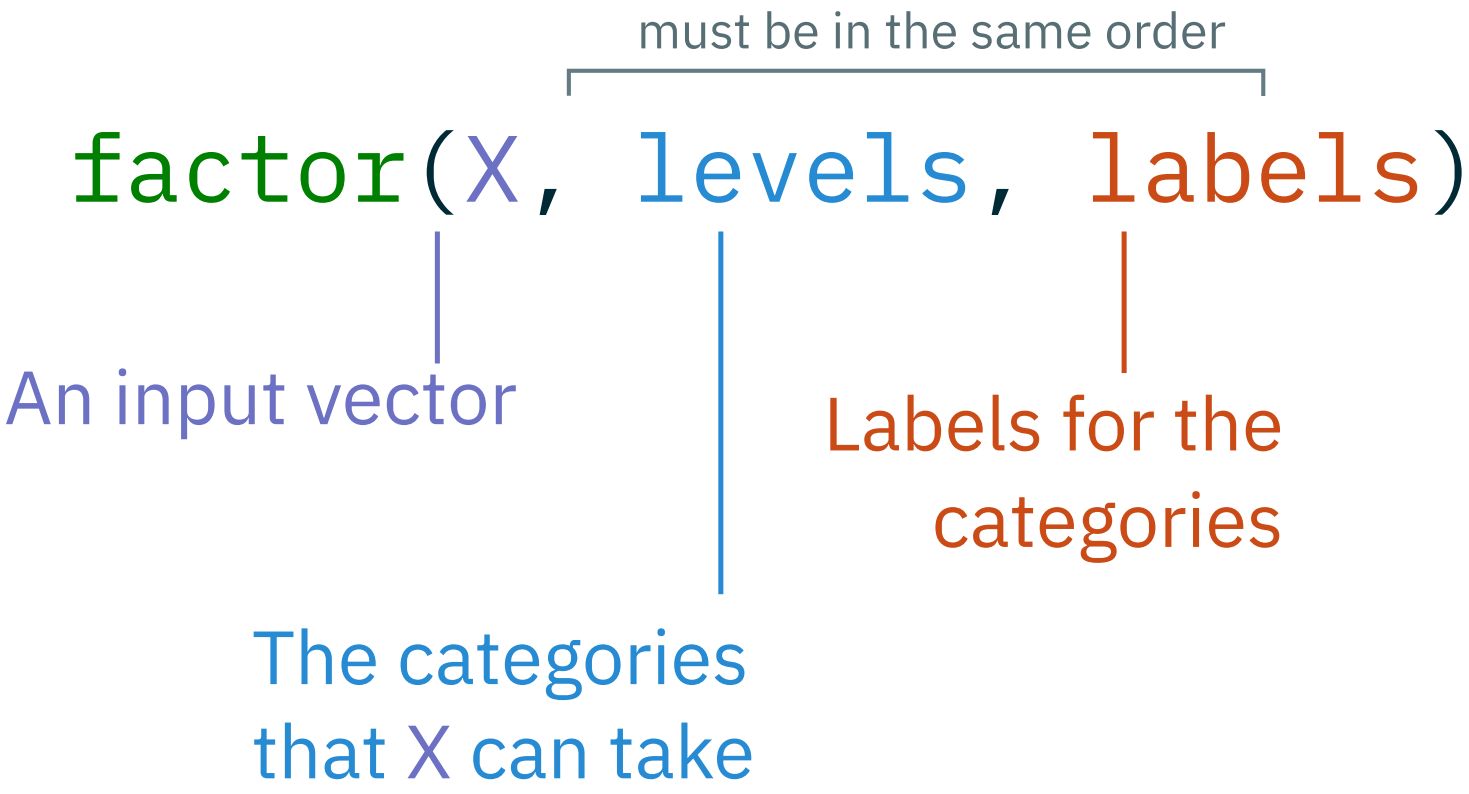
```
> mtcars$ineff <- mtcars$mpg < 15,
> mtcars$ineff
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[7] FALSE FALSE  TRUE  TRUE   TRUE FALSE
```

We can then convert it to numeric (`FALSE` becomes `0`; `TRUE` becomes `1`).

```
> as.numeric(mtcars$ineff)
 [1] 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 [...]
```

Factors

Factors represent categorical data (e.g., nominal or ordinal) in terms of a numeric value and an associated label.



If you're familiar with Stata, this is similar to 'values' and 'value labels'.

```
> marital
 [1] "Never married" "Divorced"      "Widowed"
 [4] "Never married" "Divorced"      "Married"
 [7] "Never married" "Divorced"      "Married"
[10] "Married"       "Married"       "Married"
[13] "Married"       "Married"       "Divorced"
[...]

> table(marital)
marital
     Divorced       Married Never married
         3383         10117          5416
    No answer     Separated       Widowed
           17           743          1807

> typeof(marital)
[1] "character"
```

```
> marital_f <- factor(marital)
> marital_f
[1] Never married      Divorced       Widowed
[5] Divorced          Married        Never married
[9] Married           Married        Married
Levels: Divorced Married Never married No answer
        Separated Widowed

> typeof(marital_f)
[1] "integer"

> as.numeric(marital_f)
[1]  3 1 6 3 1 2 3 1 2 2 2 2 2 2 1
[16] 2 6 3 2 2 2 2 3 6 6 6 6 6 1 6
[31] 6 2 2 3 2 3 3 3 3 3 2 2 1 3 3
[46] 3 2 2 2 2 3 2 2 2 2 1 1 1 3 3
```

```
> marital_n
[1]  3 1 6 3 1 2 3 1 2 2 2 2 2 2 1
[16] 2 6 3 2 2 2 2 3 6 6 6 6 6 1 6
[31] 6 2 2 3 2 3 3 3 3 3 2 2 1 3 3

> categories <- c("Divorced",
>                 "Married",
>                 "Never married",
>                 "No answer",
>                 "Separated",
>                 "Widowed")

> marital_f <- factor(
>   marital_n,
>   levels = 1:6,
>   labels = categories
> )
```

More complex objects

Matrices

A matrix is a rectangular array of data:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

They can be created with the `matrix` or `array` functions.

```
> x <- 1:20
> matrix(x, ncol = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

By default, `matrix` fills by column. We can instead fill by row with the `byrow` option:

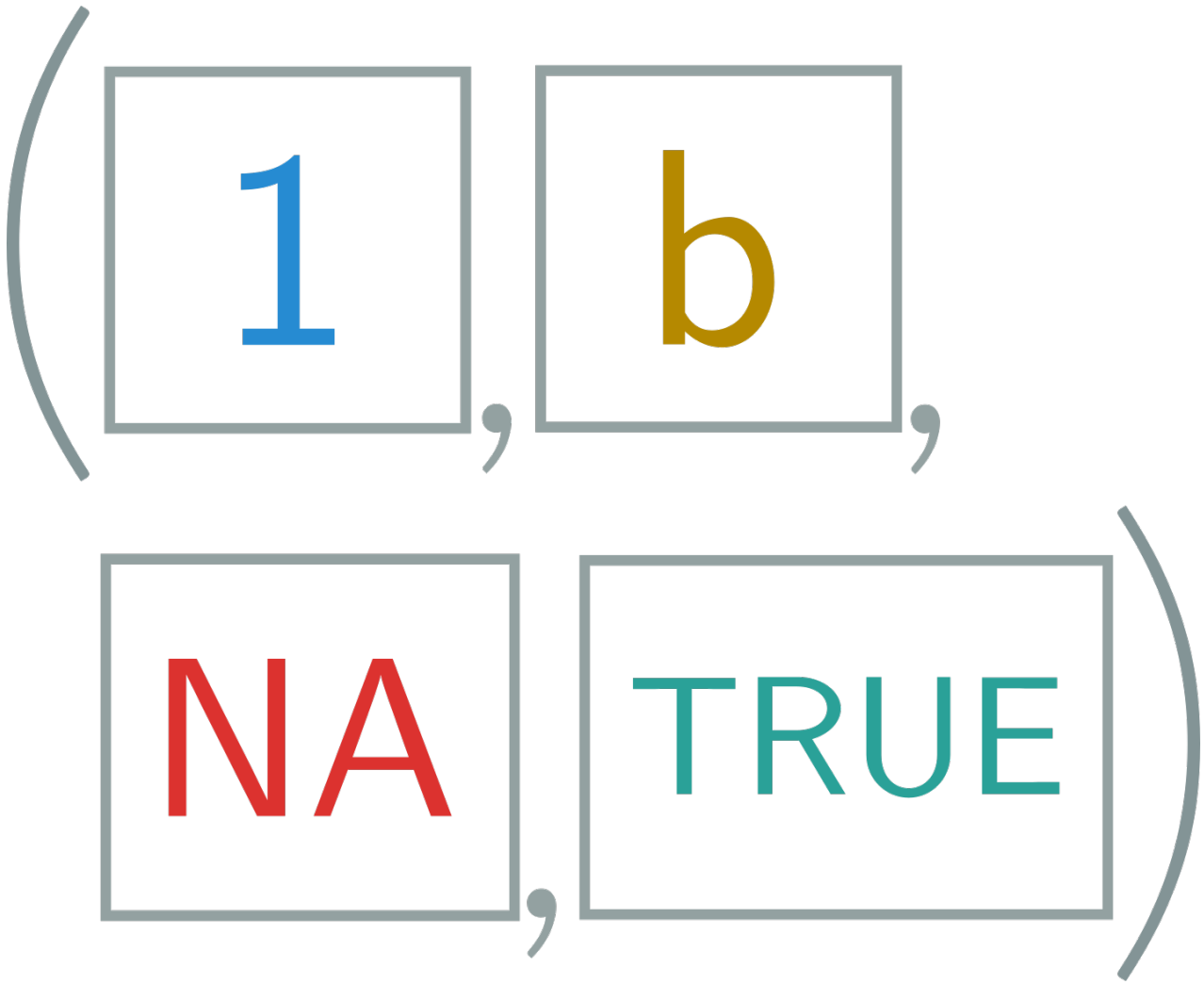
```
matrix(x, ncol = 5, byrow = TRUE)
```

```
> matrix(x, ncol = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20

> matrix(x, ncol = 5, byrow = TRUE)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
```

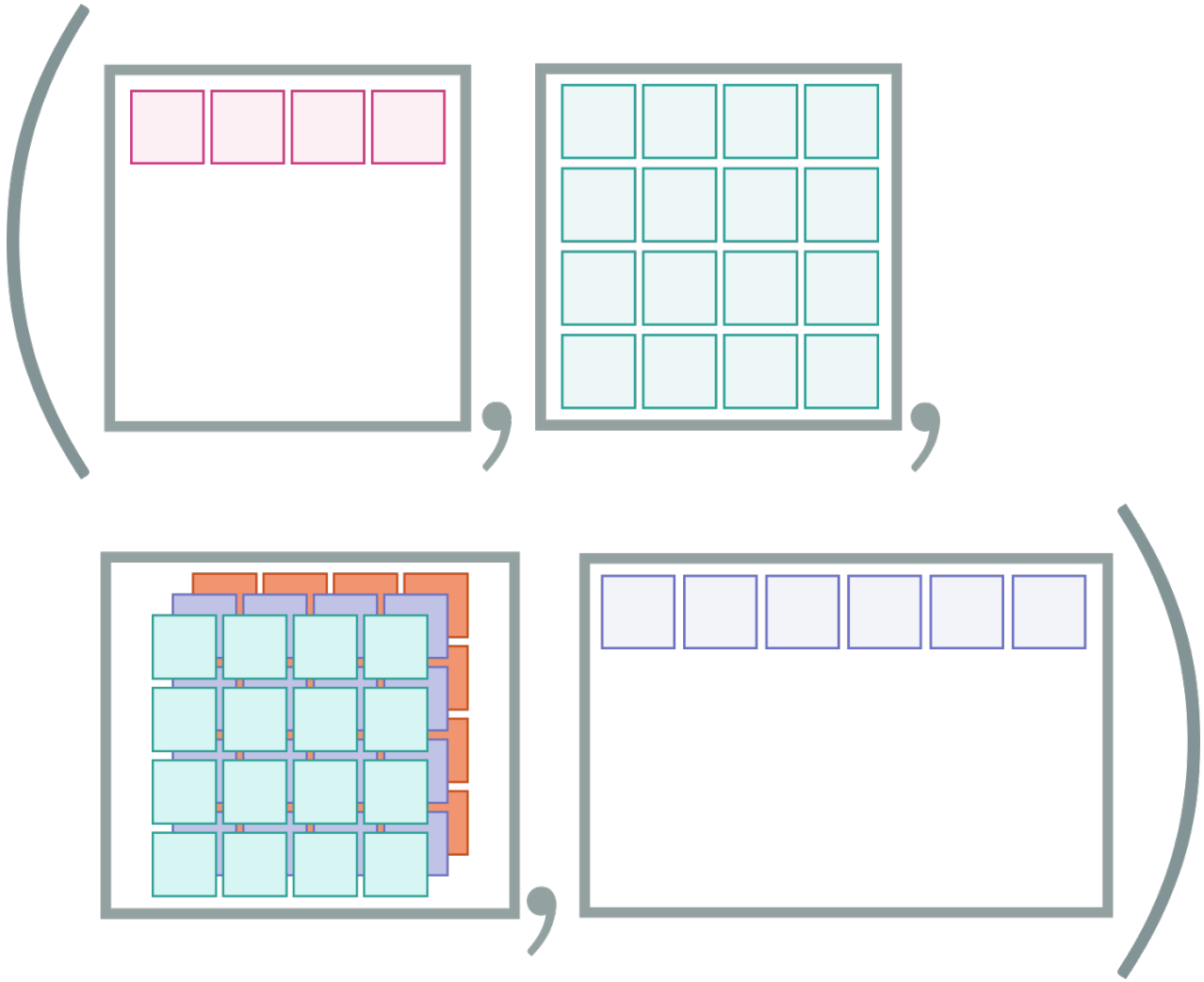
Lists

A list can contain objects of different types (including other lists).



Lists

A list can contain objects of different types (including other lists).



Defing a list

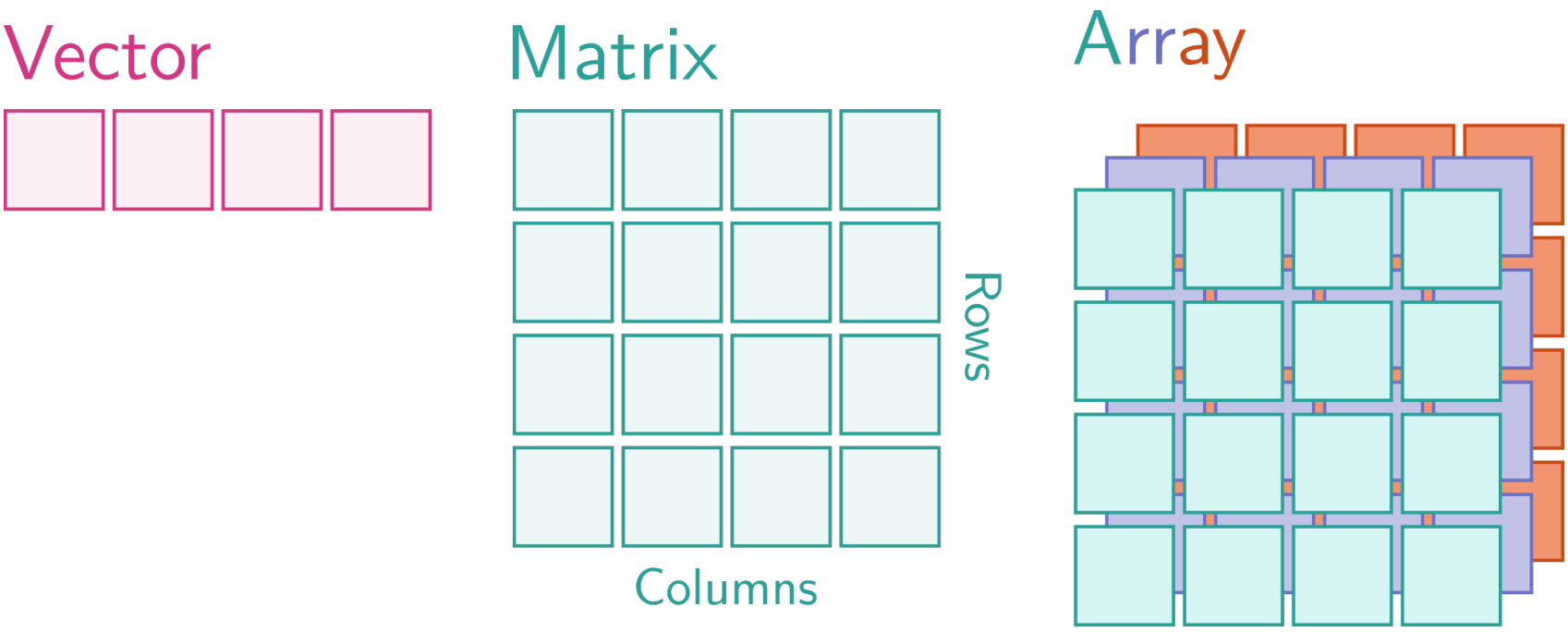
```r
my_list_unnamed <- list(
  12.1,
  "Tuesday",
  c(TRUE, FALSE, TRUE),
  matrix(1:20, ncol = 4)
)

# Same idea, but with names:
my_list_named <- list(
  number = 12.1,
  day = "Tuesday",
  flags = c(TRUE, FALSE, TRUE),
  mat = matrix(1:20, ncol = 4)
)
```

We'll cover how to select elements (e.g., `number`) from an object in the next session.

Arrays

An `array` is a vector with one or more dimensions.



We don’t often use them.