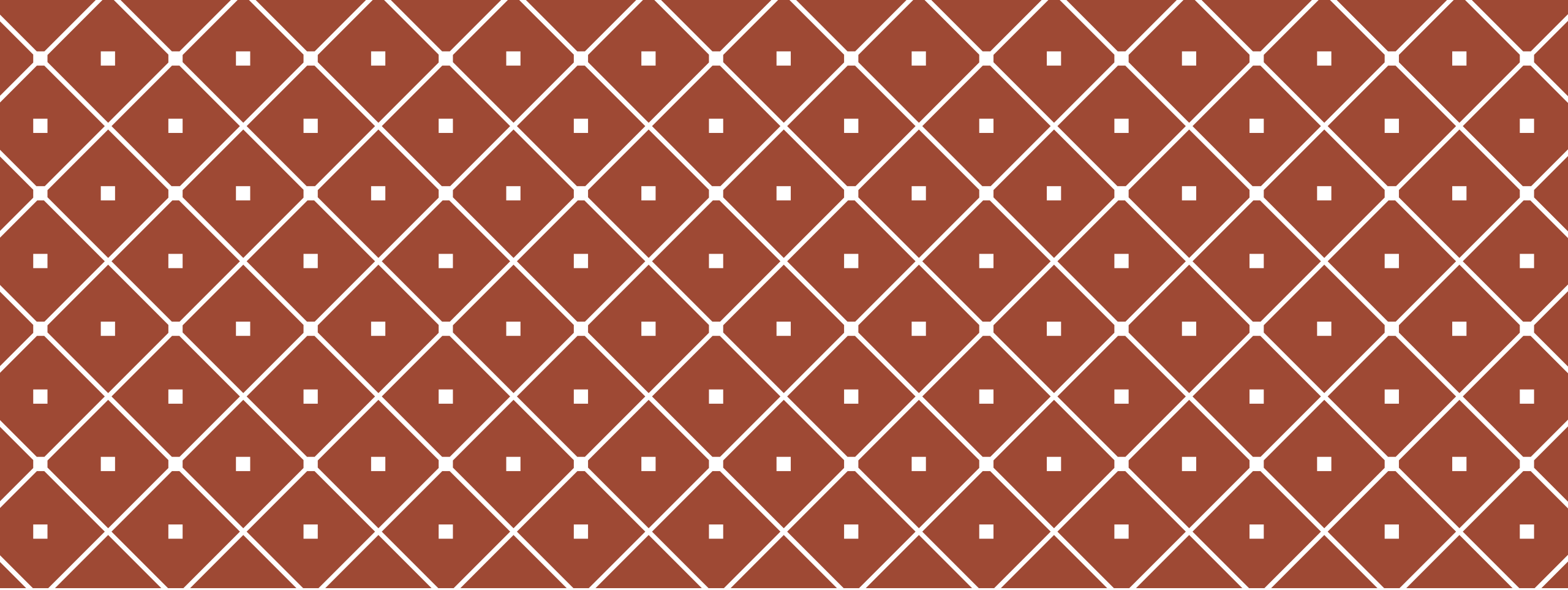
STAT 133

Corrine F. Elliott

Lab #03

OVERVIEW

- Data types
- Vectors & atomicity
- Coercion (explicit and implicit)
- Subsetting (by index, condition, or name)
- Vectorization & recycling
- Matrices & arrays



DATA TYPES

DATA TYPES (PRIMITIVES)

- Integer
- Double
- Logical
- Character
- Complex
- Raw
- Missing / special values

- Useful operations: `typeof()`, `mode()`

INTEGER

- Indicated by a whole number followed by "*L*"

```
x <- 1L
```

- Or, we can call the `integer()` function directly

```
x <- integer(length = 0)   # empty integer
```

- In either case,

```
typeof(x)
## [1] "integer"
```

DOUBLE

- Any numerical value, including whole numbers

```
x <- 1
x <- 2.5
x <- 2/3
```

```
typeof(x)
```

```
## [1] "double"
```

LOGICAL / BOOLEAN

- Assumes value(s) `TRUE` or `FALSE`

```
x <- TRUE
```

- We can also shorten to `T` or `F` , respectively

```
x <- T
```

```
typeof(x)
```

```
## [1] "logical"
```

Recall: case-sensitivity implies `true` and `TRUE` are *not* equivalent!

CHARACTER (STRING)

- Any combination of characters enclosed by single (') or double quotations (")

```
x <- 'hello'
```

- Single quotes *within* double quotes are interpreted as characters in the string

```
x <- "'How clever!' he thought in amazement."
```

```
typeof(x)
```

```
## [1] "character"
```

COMPLEX NUMBERS

- Indicated by a sum containing a real and an imaginary part, the latter denoted by "*i*"

```
x <- 1 + 3i
```

- Or, we can call the `complex()` function directly

```
x <- complex(real = 1, imaginary = 0)  # same as 1+0i
```

- In either case,

```
typeof(x)
## [1] "complex"
```

RAW

- Convert a value to raw format using the function `as.raw()`

```
x <- as.raw(40)
```

- Raw object stores each byte in hexadecimal

```
print(x)
## [1] 28

typeof(x)
## [1] "raw"
```

TYPE VS. MODE

A bit confusing at the beginning

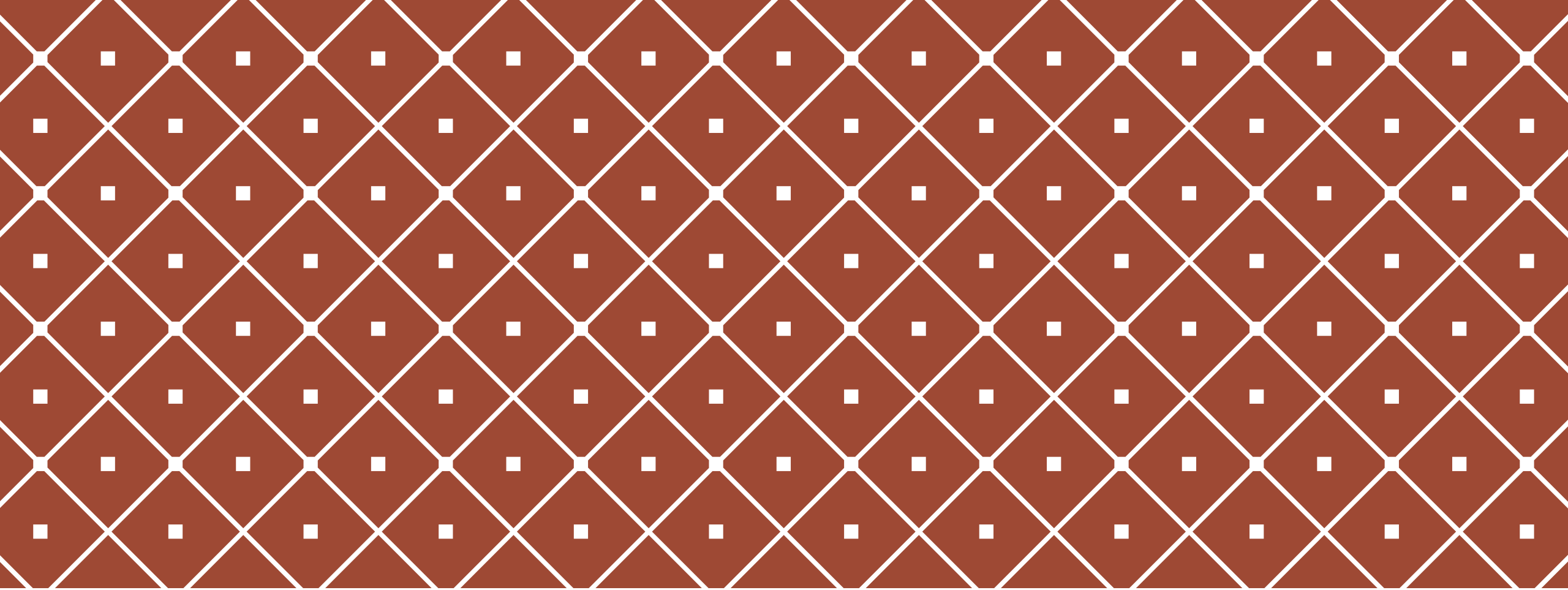
value	example	mode	type
integer	`1L, 2L`	`numeric`	`integer`
real	`1, -0.5`	`numeric`	`double`
complex	`3 + 5i`	`complex`	`complex`
logical	`TRUE, FALSE`	`logical`	`logical`
character	`"hello"`	`character`	`character`

useRs typically talk about the **mode**

Figure credit: Prof. Gaston Sanchez

MISSING / SPECIAL VALUES

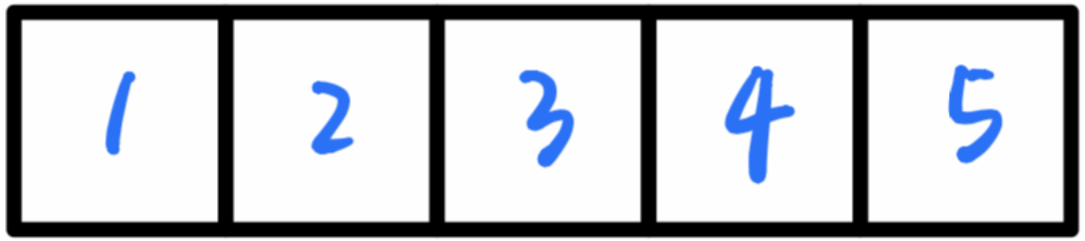
```
NA      # Not Available (missing value / placeholder)
        # assumes a different type depending on context

NULL    # has zero length

Inf     # positive infinity (e.g., divide by zero)
-Inf    # negative infinity

NaN     # Not a Number (e.g., divide 0/0)
```



VECTORS & ATOMICITY

VECTORS

- Think of as contiguous cells, each containing a single value:



- Simplest data structure in R: A *variable* is a vector with unit length

- Useful operations: `length()`, `table()`, `rev()`, `names()`

CREATING VECTORS

- We have made use already of the *combine* function and colon notation:

```
vec <- c(1, 9, 12)    # 1 9 12
vec <- 1:3    # 1 2 3
```

- The *sequence* function accepts an initial value, final value, and step size or length:

```
vec <- seq(from = 1, to = 7, by = 2)    # 1 3 5 7
vec <- seq(from = 1, to = 7, length.out = 3)    # 1 4 7
```

- The *replicate* function accepts an input vector and replicate specification:

```
vec <- rep(x = 1:3, each = 2)    # 1 1 2 2 3 3
vec <- rep(x = 1:3, times = 2)    # 1 2 3 1 2 3
```

ATOMICITY / ATOMIC STRUCTURE

- A vector can contain *exactly one* type of data

- We can apply the `typeof()` function to a vector determine the type of its contents

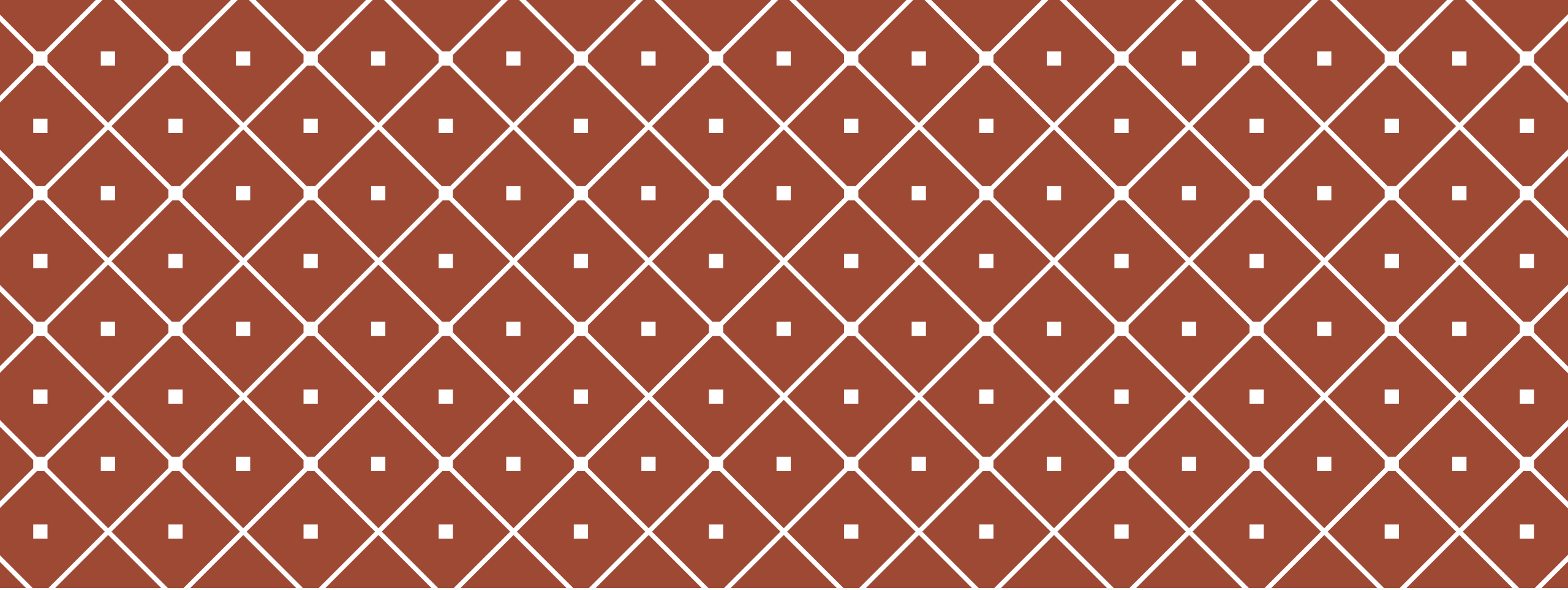
- We can also create an 'empty' vector with a destined type; placeholder value varies:

```
vec <- vector(mode = "double", length = 3)  # 0 0 0
vec <- vector(mode = "logical", length = 3)  # F F F
vec <- vector(mode = "character", length = 3)  # "" "" ""
```

NAMED VECTOR

```
vec <- c('a', 'b', 'c', 'd', 'e')  # "a" "b" "c" "d" "e"
names(vec) <- c('A', 'B', 'C', 'D', 'E')
print(vec)
```

```
  A   B   C   D   E
"a" "b" "c" "d" "e"
```



COERCION

IMPLICIT COERCION

- What happens if we try to create a vector containing a mix of data types?
  - R tries to fix the problem for you, by *coercing* one data type into another
  - *Implicit* because the user does not request the operation explicitly; moreover, R does not warn the user

- Coercion follows an underlying hierarchy:

logical < integer < double < complex < character

EXPLICIT COERCION

- Recall that we used the function `as.raw()` to convert a numeric (double) value to raw

- Similar functions exist for performing other conversions

  `as.integer()`, `as.double()`, `as.character()`,

  `as.logical()`, `as.complex()`

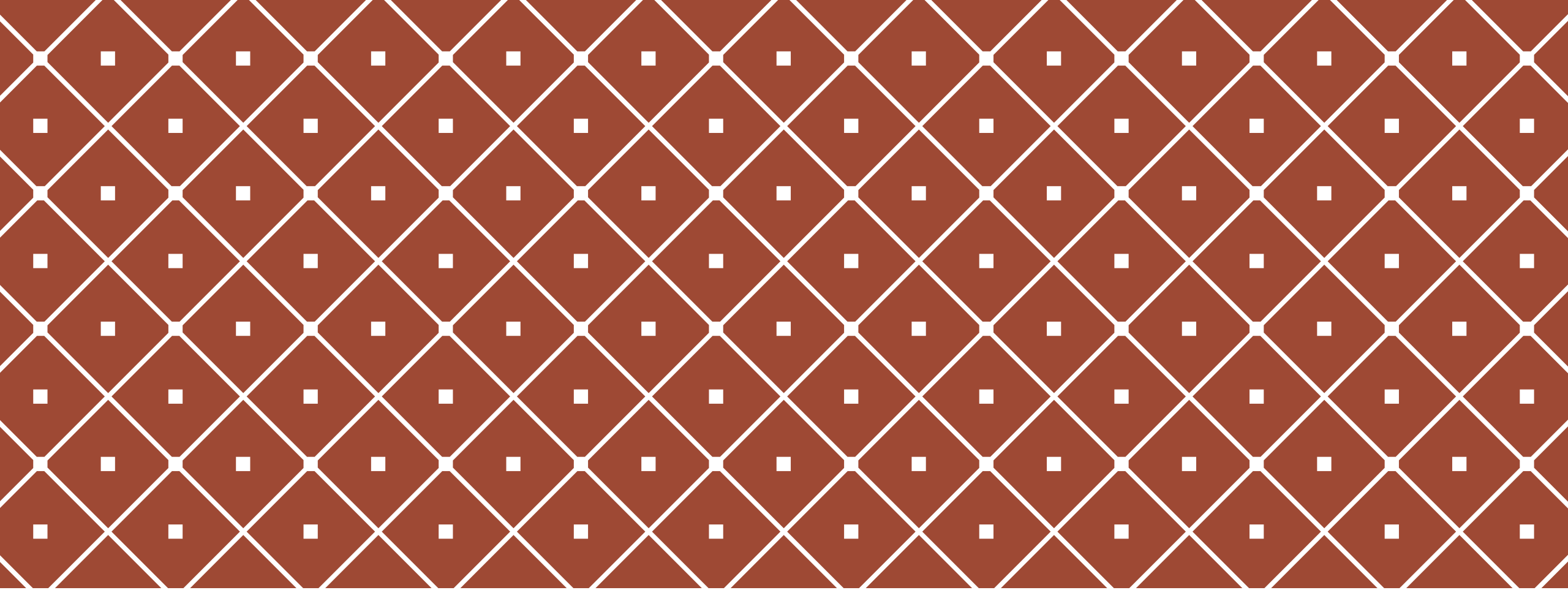
- These functions do not affect the value of the input variable

EXPLICIT COERCION (CONT.)

- Not all data types are interchangeable; invalid attempts return `NA`

- Any value can be treated as a character string
- Most character strings cannot be coerced

- Double to integer: truncate decimal places
- Numeric values (double or integer) to logical:
  - `0` treated as `FALSE`
  - Any non-zero element, including a negative value, treated as `TRUE`



SUBSETTING

BRACKET NOTATION

```
vec <- c('a', 'b', 'c', 'd', 'e')  # "a" "b" "c" "d" "e"
```

- We use square brackets to extract values from a data structure
- When subsetting a vector, brackets can accept ***one*** or ***a vector*** of …
  - *Indices*: numerical positions of the desired elements

```
vec[1]   # "a"
vec[ c(3,1,1) ]   # "c" "a" "a"
```

Note: in R, indices start from `1` rather than `0`

BRACKET NOTATION

```
vec <- c('a', 'b', 'c', 'd', 'e')  # "a" "b" "c" "d" "e"
names(vec) <- c('A', 'B', 'C', 'D', 'E')
```

- We use square brackets to extract values from a data structure
- When subsetting a vector, brackets can accept ***one*** or ***a vector*** of …
  - *Indices*: numerical positions of the desired elements

```
vec[1]   # "a"
vec[ c(3,1,1) ]   # "c" "a" "a"
```

  - *Logical* values: extract elements associated with `TRUE` values

```
vec[ c(T,F,T,F,F) ]   # "a" "c"
```

  - *Names*: character strings associated with the desired elements

```
vec[ c("B","D") ]   #   B   D
                    # "b" "d"
```

CONDITIONAL SUBSETTING

```
vec <- c('a', 'b', 'c', 'd', 'e')  # "a" "b" "c" "d" "e"
x <- 1:5   # 1 2 3 4 5
```

- Suppose we wish to extract elements using logic
- We can specify the Boolean flags explicitly

```
vec[ c(T,F,T,F,F) ]   # "a" "c"
```

- Or we can provide a *conditional statement* equivalent to bools

```
vec[ x > 3 ]   # "d" "e"
vec[ x %in% c(1,3) ]   # "a" "c"
```

Operator	Definition
==	Is equal to
! !=	Not [negation operator] Is not equal to
> >=	Greater than Greater than or equal to
< <=	Less than Less than or equal to
|	Or
&	And
%in%	Belongs to the set

A NOTE ON SPECIAL VALUES

- You may be tempted to use subsetting to remove missing elements with a statement like

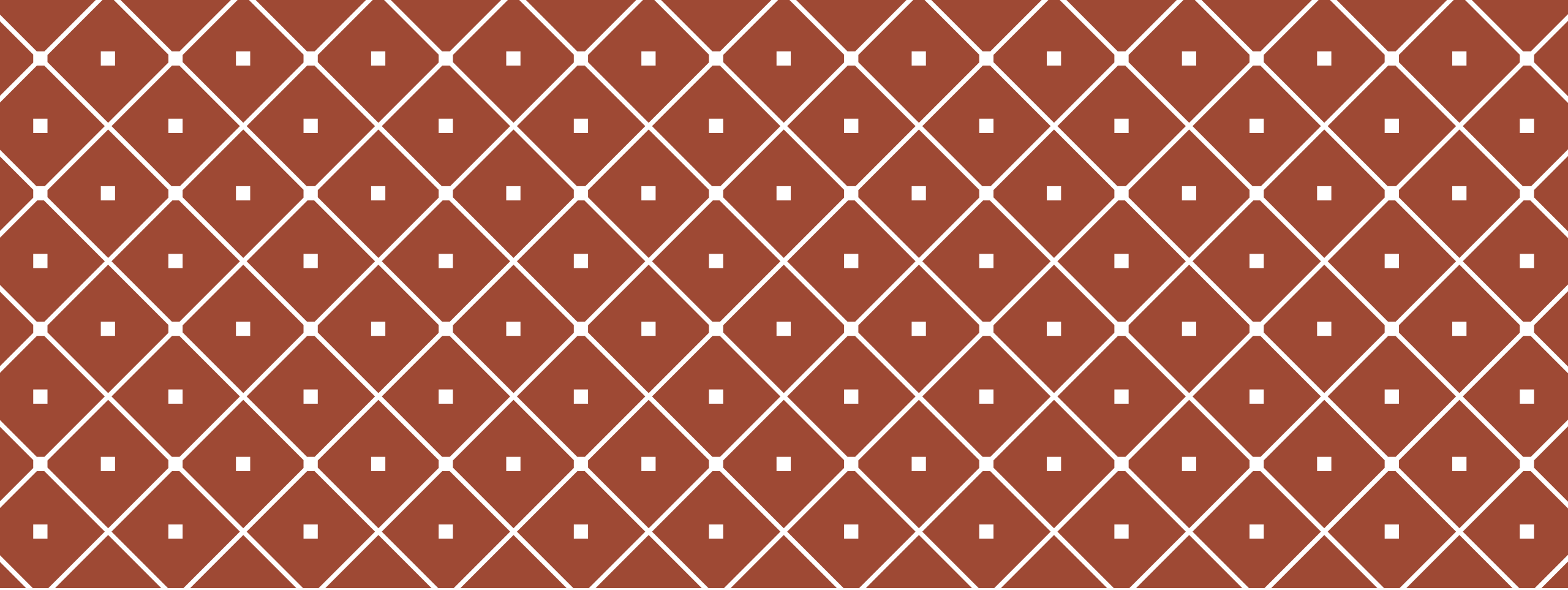
```
vec[ vec != NA ]
```

- ***DON'T.***

- Different systems store special values (`NA`, `NULL`, `Inf`, `-Inf`, `NaN`) differently
- Such statements do not operate consistently as the user might expect.

- Instead of using conditional operators (`==`, `!=`), you can use the base R functions

```
is.na(), is.null(), is.nan(), is.finite(), is.infinite()
```

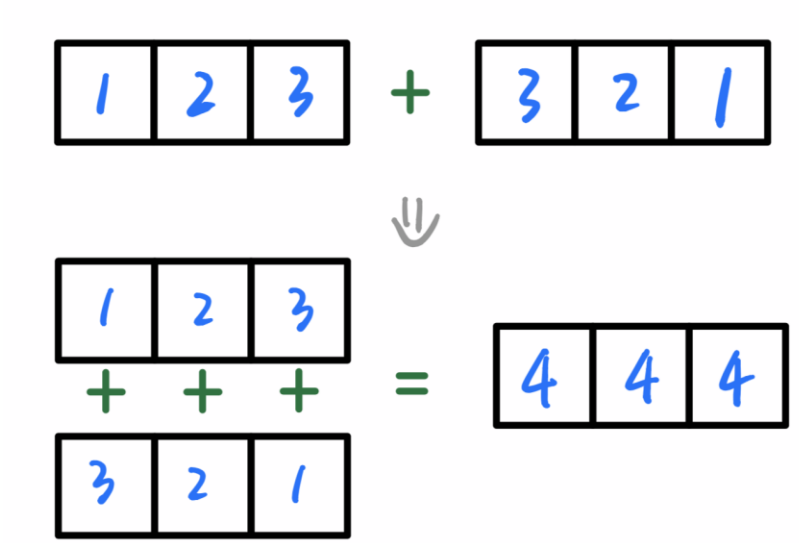


VECTORIZATION & RECYCLING

VECTORIZED OPERATIONS

- A *vectorized* or computation is performed *element-wise*

```
c(1, 2, 3) + c(3, 2, 1)    # 4 4 4
```



VECTORIZED OPERATIONS

- A *vectorized* or computation is performed *element-wise*

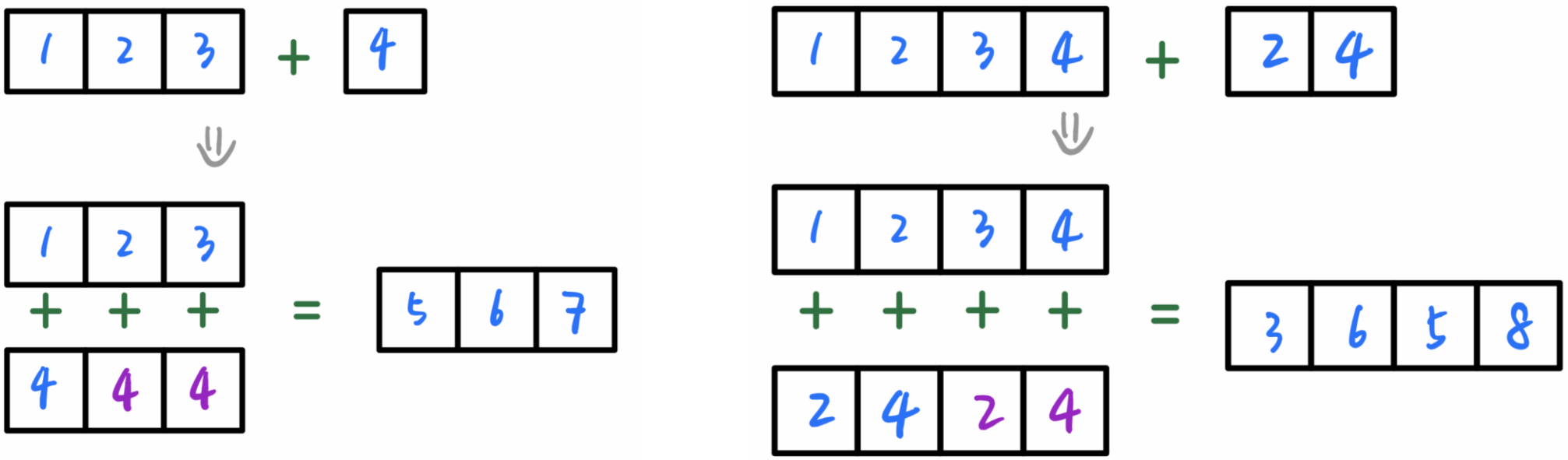
```
c(1, 2, 3) + c(3, 2, 1)    # 4 4 4
```

- Most operations in R are vectorized

```
c(1, 2, 3) * c(3, 2, 1)    # 3 4 3
c(1, 2, 3) ^ c(1, 2, 3)    # 1 4 27
sqrt( c(1, 4, 9) )         # 1 2 3
abs( c(-1, -2, -3) )       # 1 2 3
```

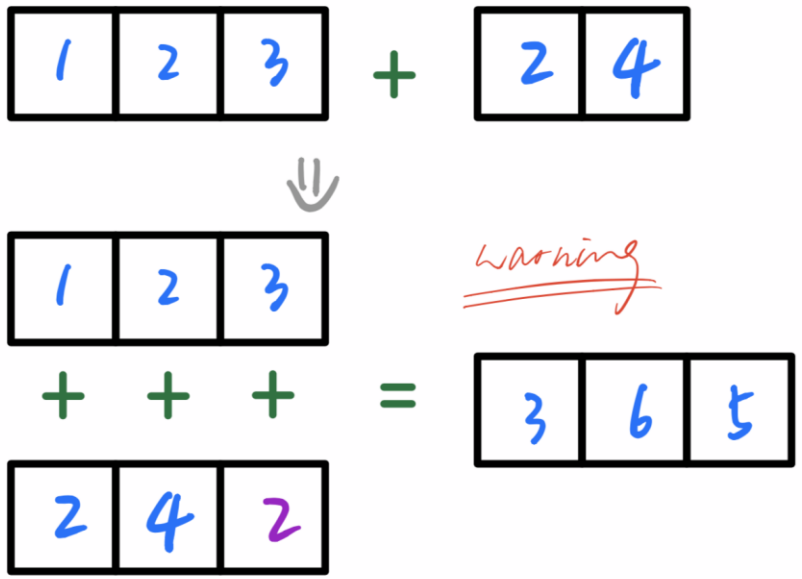
RECYCLING

- If two vectors have disparate length, the shorter is *recycled* to match the longer

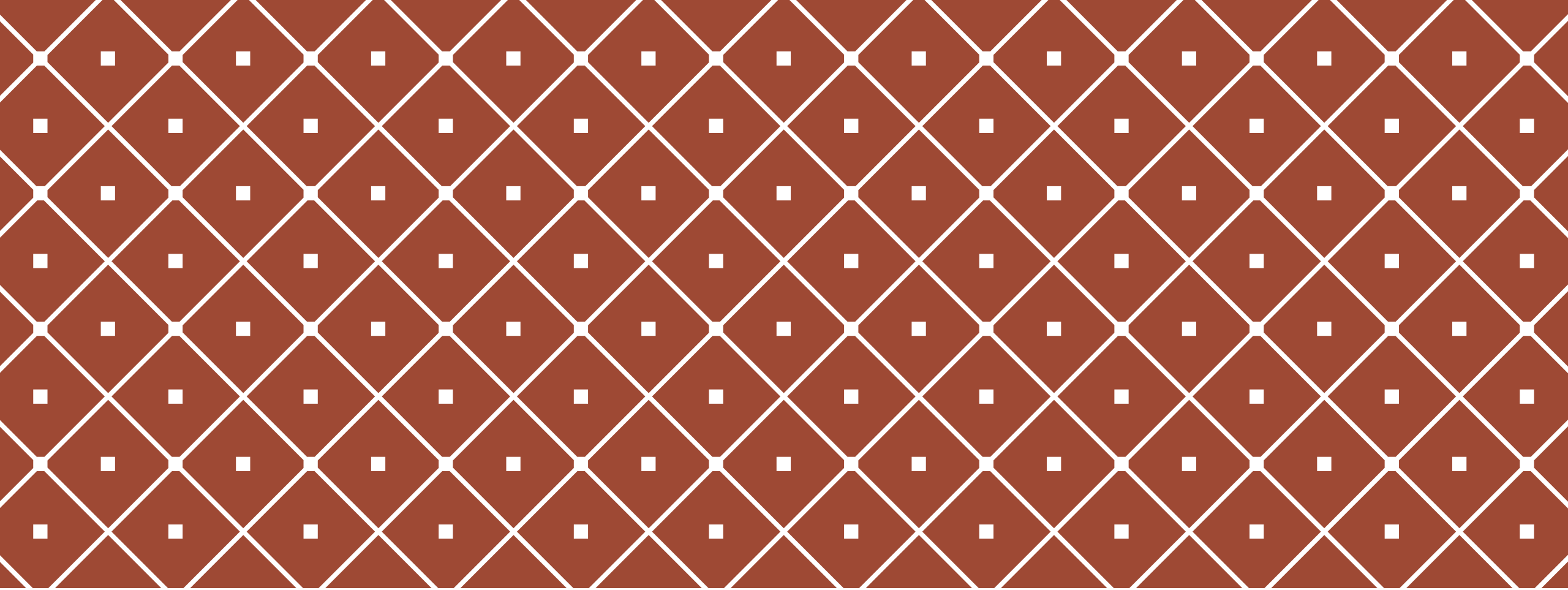


RECYCLING

- If two vectors have disparate length, the shorter is *recycled* to match the longer




```
c(1,2,3) + c(2,4)

Warning message:
In c(1, 2, 3) + c(2, 4) :
  longer object length is not a multiple
of shorter object length
```



MATRICES & ARRAYS

ARRAYS

- A *matrix* is an atomic data structure with two dimensions (rows and columns)
- An *array* is an atomic data structure with more than two dimensions

- Useful functions:

  `matrix()`

  `nrow()`, `ncol()`, `dim()`,

  `rbind()`, `cbind()`,

  `rownames()`, `colnames()`

```
HM
```

```
##                   height mass
## Luke Skywalker       172   77
## Darth Vader          202  136
## Leia Organa          150   49
## Owen Lars            178  120
## Beru Whitesun lars   165   75
```

BRACKET NOTATION & ARRAYS

- We can subset an array using bracket notation
- Requires supplying in index specification for *each* dimension

```
HM[3, ]

HM[, "mass"]

HM[height == 172, "mass"]
```

```
HM
```

```
##                   height mass
## Luke Skywalker       172   77
## Darth Vader          202  136
## Leia Organa          150   49
## Owen Lars            178  120
## Beru Whitesun lars   165   75
```