Rethinking Arrays in R

April 2019

Davis Vaughan
@dvaughan32
Software Engineer, RStudio

Array manipulation in R is **inconsistent** and doesn't follow natural **intuition**.

Arrays are:

1. *frustrating* to work with.

2. *difficult* to program around.

3. *underpowered*.

Subsetting

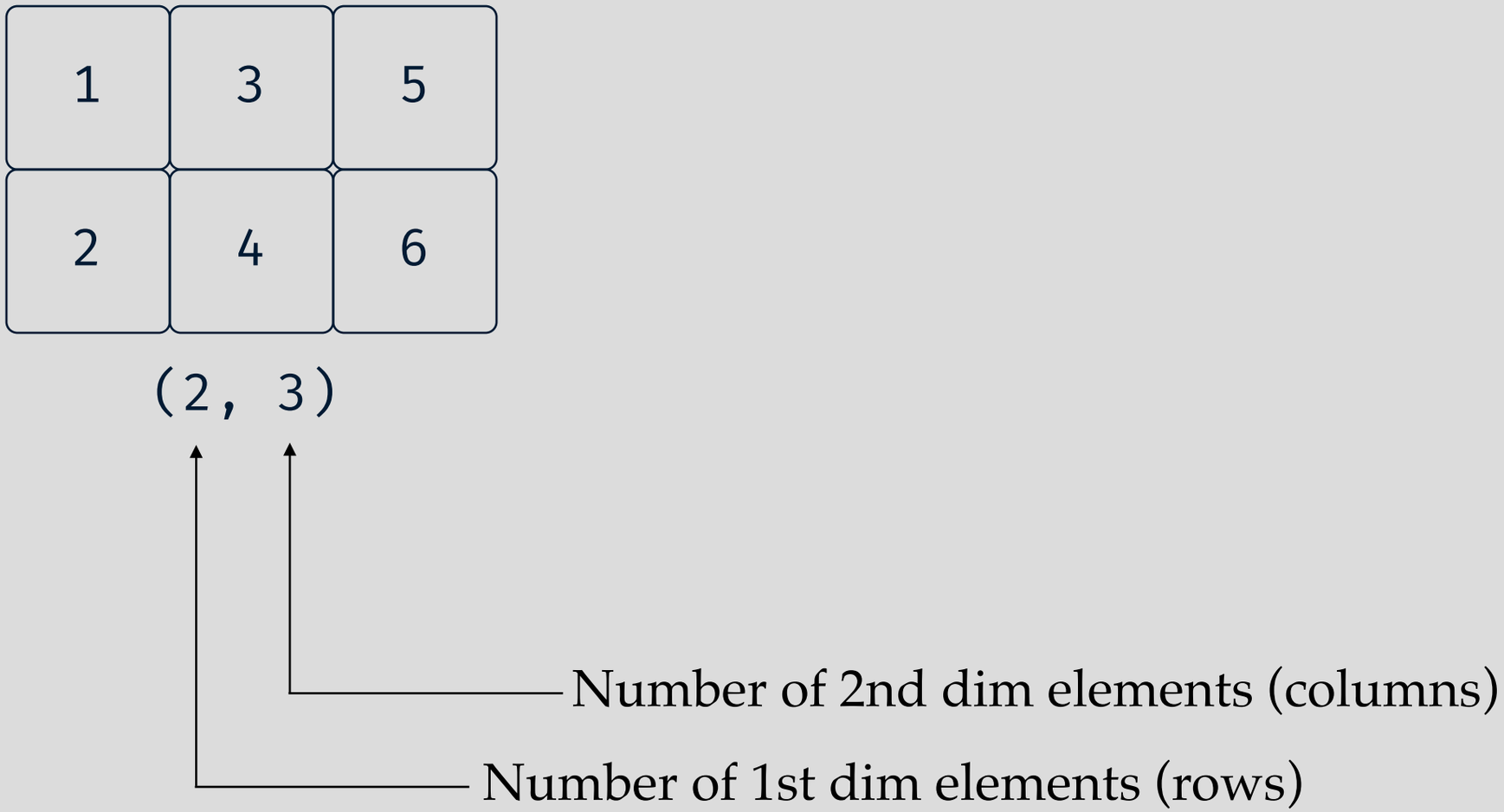
Broadcasting

Manipulation

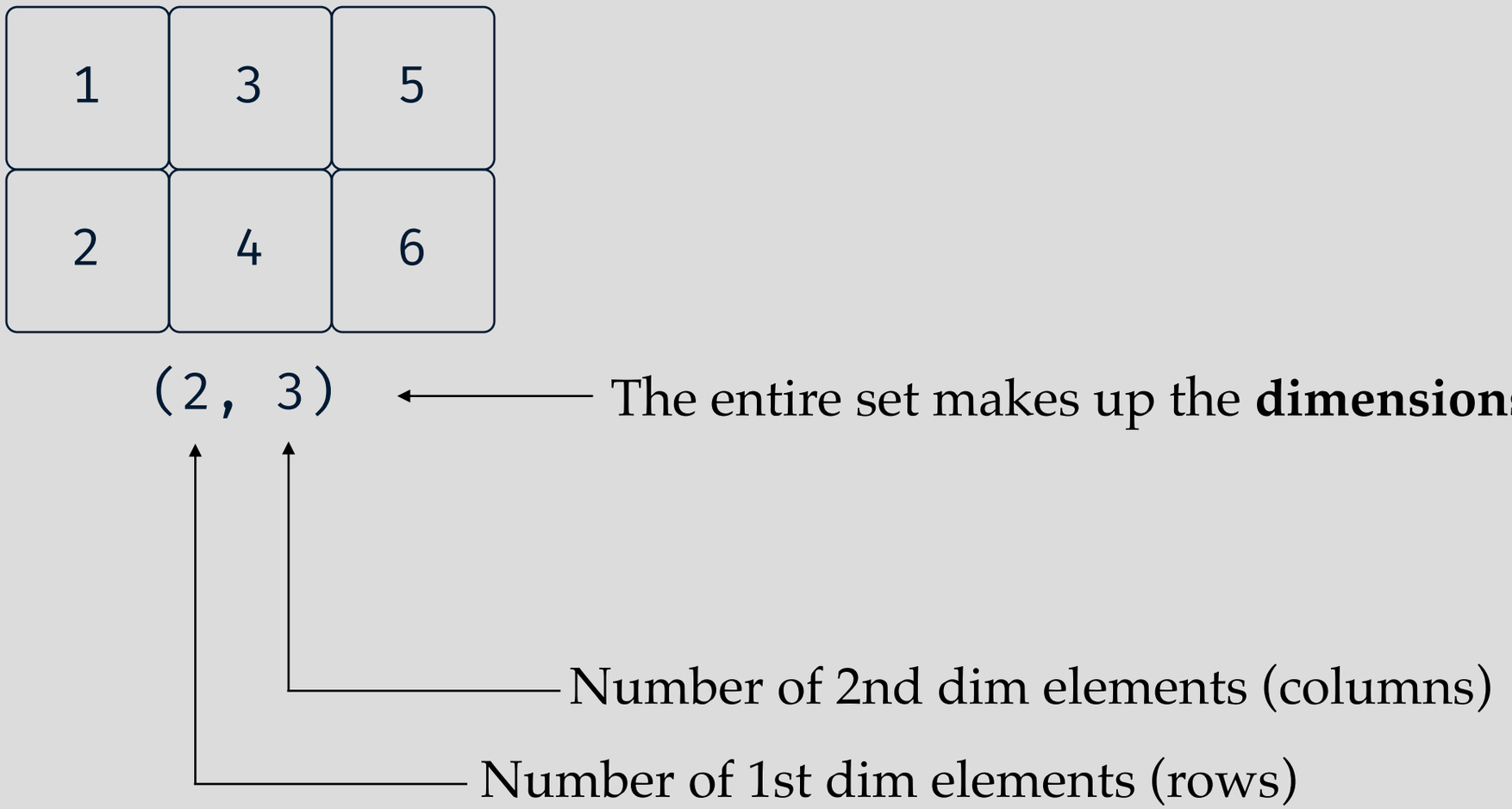
dimensionality:
The number of dimensions in an array.

dimensions:
The set of lengths describing the shape of the array.

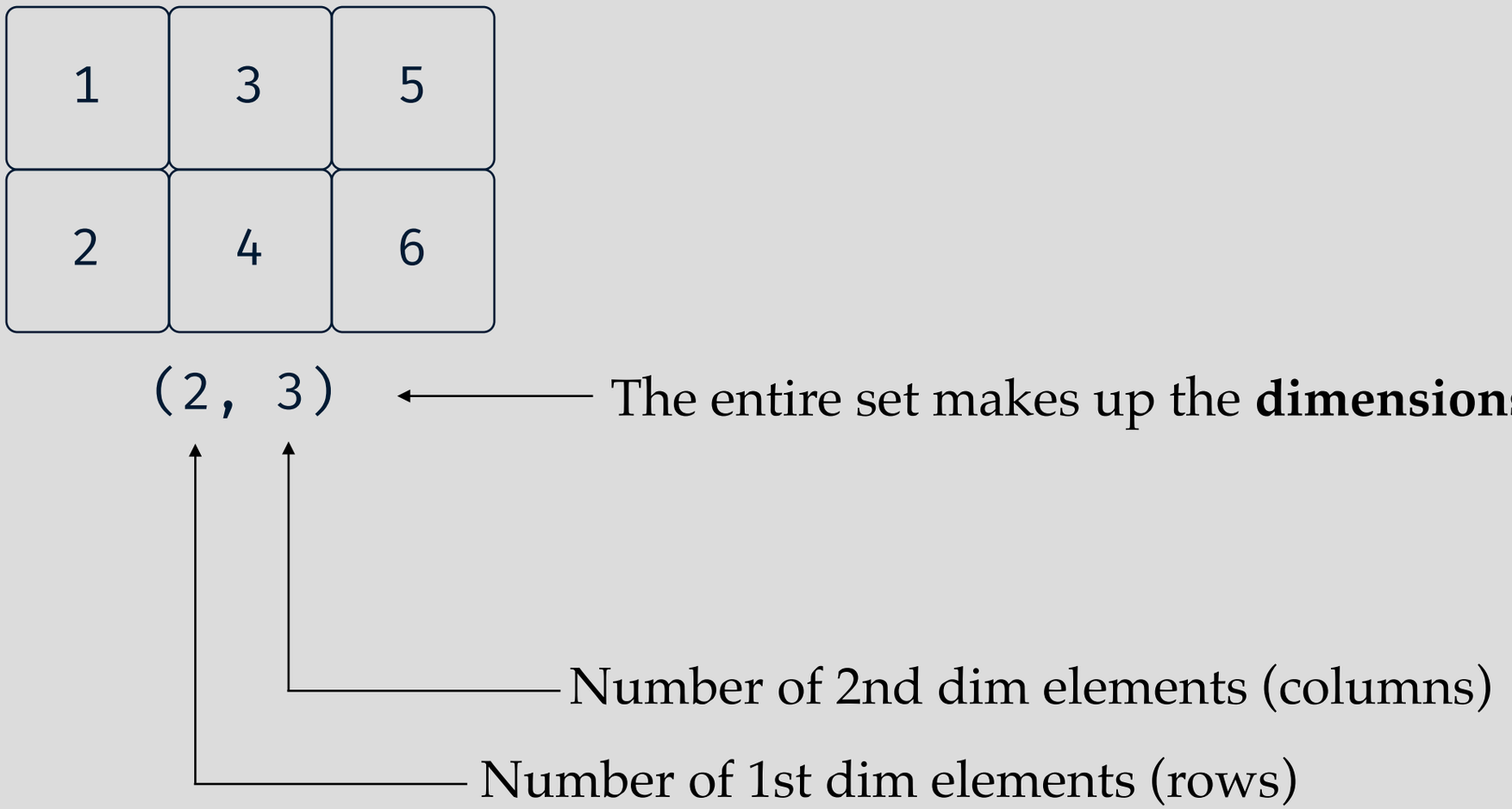
dimensionality VS dimensions



dimensionality VS dimensions



dimensionality VS dimensions



The **dimensionality** is 2
(2D object)

Subsetting

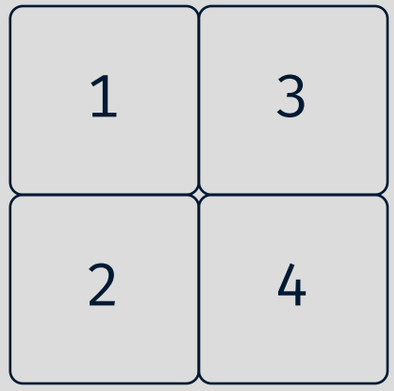
Enter the matrix.

1	3	5
2	4	6

x

Column selection

x



`x[, 1:2]`

One column?

1	3	5
2	4	6

x

?

`x[, 1]`

One column?

1	3	5
2	4	6

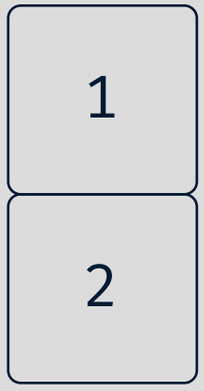
x

```
1      2
```

```
x[, 1]
```

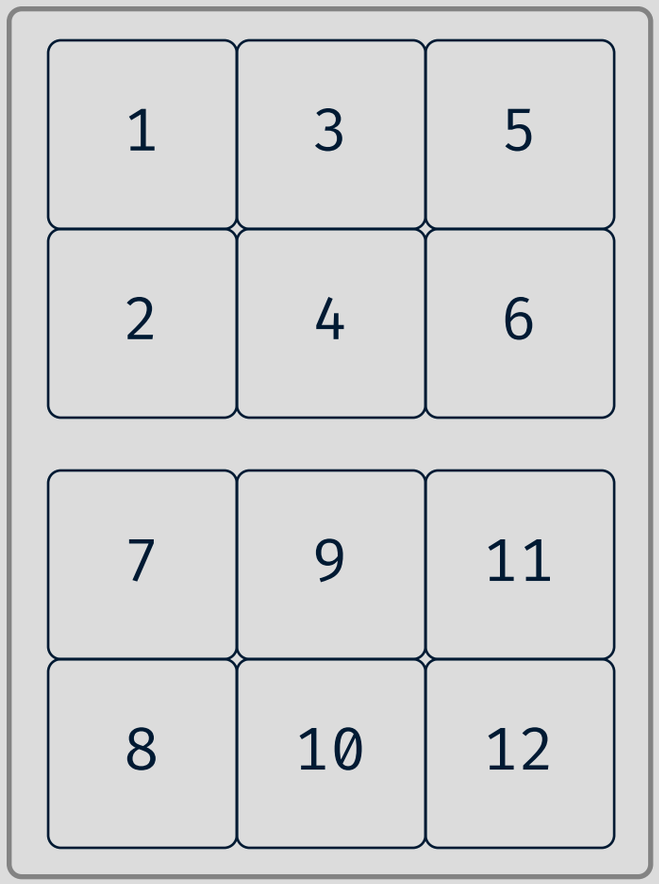
Oh! Let me fix that for you…

`x`




```
x[, 1, drop = FALSE]
```

Let’s go 3D



y

?

y[, 1:2]

Let’s go 3D

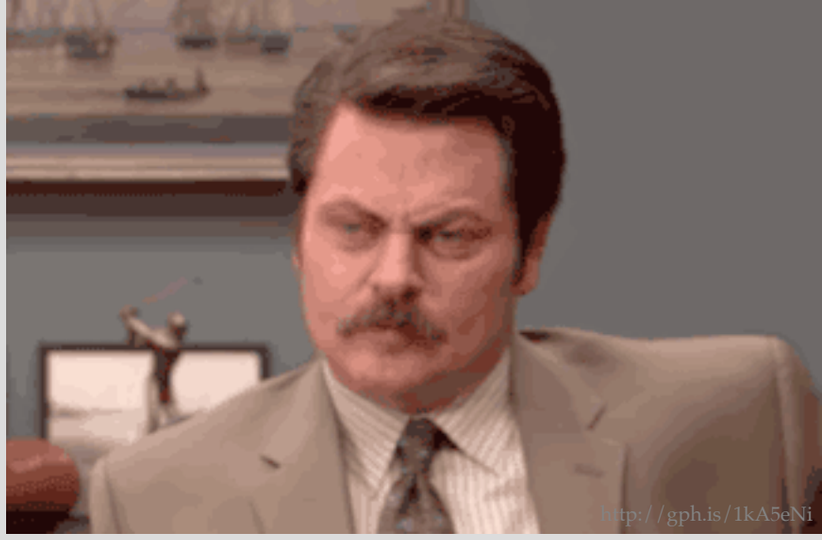
1	3	5
2	4	6
7	9	11
8	10	12

y

```
Error:
incorrect number
  of dimensions
```

y[, 1:2]

Let’s go 3D



1	3	5
2	4	6
7	9	11
8	10	12

y

```
Error:
incorrect number
  of dimensions
```

```
y[, 1:2]
```

Let's go 3D

1	3	5
2	4	6
7	9	11
8	10	12

y

1	3
2	4
7	9
8	10

y[, 1:2,]

One column?

1	3	5
2	4	6
7	9	11
8	10	12

y

?

```
y[, 1,]
```

One column?

1	3	5
2	4	6
7	9	11
8	10	12

y

1	7
2	8

y[, 1,]

One column?

1	3	5
2	4	6
7	9	11
8	10	12

y

1	7
2	8

y[, 1,]

One column?

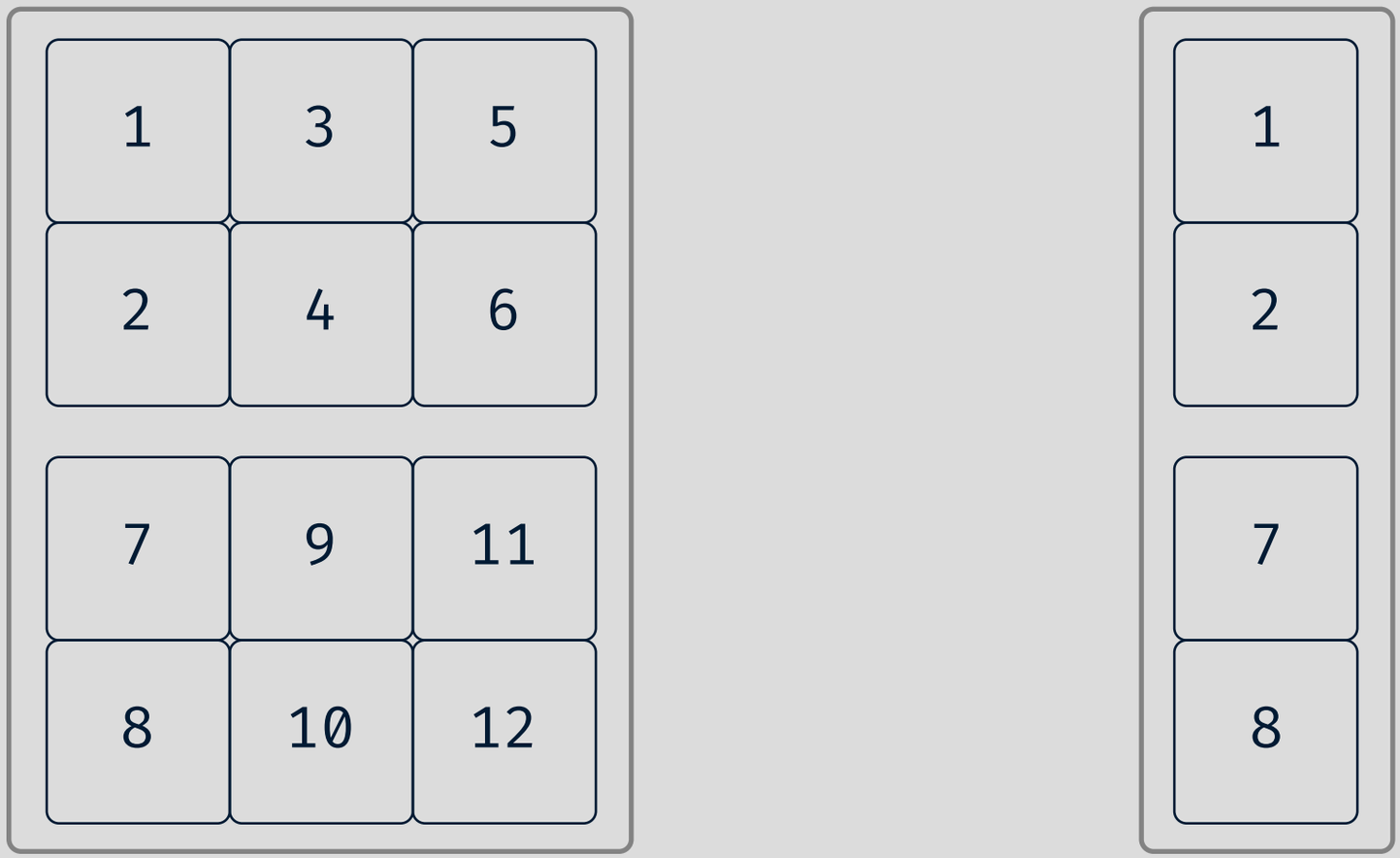
1	3	5
2	4	6
7	9	11
8	10	12

y

1	7
2	8

y[, 1,]

Oh! Let me fix that for you…



y

y[, 1, , drop = FALSE]

The confusion?

Subsetting is not **dimensionality-stable.**

Summary: column selection = 😈

How Many?	Drops?	2D	3D
1	No	`x[, 1, drop = F]`	`x[, 1, , drop = F]`
>1	No	`x[, 1:2]`	`x[, 1:2, ]`
1	Yes	`x[, 1]`	`x[, 1, ]`*
>1	Yes	`x[, 1:2, drop = T]`	`x[, 1:2, , drop = T]`

* Drops to 2D

Summary: column selection = 😈

How Many?	Drops?	2D	3D
1	No	`x[, 1, drop = F]`	`x[, 1, , drop = F]`
>1	No	`x[, 1:2]`	`x[, 1:2, ]`
1	Yes	`x[, 1]`	`x[, 1, ]`*
>1	Yes	`x[, 1:2, drop = T]`	`x[, 1:2, , drop = T]`

* Drops to 2D

Summary: column selection = 🎉

How Many?	Drops?	2D	3D	Proposed
1	No	`x[, 1, drop = F]`	`x[, 1, , drop = F]`	`x[, 1]`
>1	No	`x[, 1:2]`	`x[, 1:2, ]`	`x[, 1:2]`
1	Yes	`x[, 1]`	`x[, 1, ]`*	`extract(x, , 1)`
>1	Yes	`x[, 1:2, drop = T]`	`x[, 1:2, , drop = T]`	`extract(x, , 1:2)`

* Drops to 2D

Summary: column selection = 🎉

How Many?	Drops?	2D	3D	Proposed
1	No	`x[, 1, drop = F]`	`x[, 1, , drop = F]`	`x[, 1]`
>1	No	`x[, 1:2]`	`x[, 1:2, ]`	`x[, 1:2]`
1	Yes	`x[, 1]`	`x[, 1, ]`*	`extract(x, , 1)`
>1	Yes	`x[, 1:2, drop = T]`	`x[, 1:2, , drop = T]`	`extract(x, , 1:2)`

* Drops to 2D

Summary: column selection = 🎉

How Many?	Drops?	2D	3D	Proposed
1	No	`x[, 1, drop = F]`	`x[, 1, , drop = F]`	`x[, 1]`
>1	No	`x[, 1:2]`	`x[, 1:2, ]`	`x[, 1:2]`
1	Yes	`x[, 1]`	`x[, 1, ]`*	`extract(x, , 1)`
>1	Yes	`x[, 1:2, drop = T]`	`x[, 1:2, , drop = T]`	`extract(x, , 1:2)`

* Drops to 2D

rray

`rray` is designed to provide a **stricter** array class.

Create an rray

```
library(rray)

x ← matrix(1:6, nrow = 2)
x
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

```
x_rray ← as_rray(x)
x_rray
#> <vctrs_rray<integer>[,3][6]>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

Column subsetting…round two

1	3	5
2	4	6

`x_rray`

1
2

`x_rray[, 1]`

1	3
2	4

`x_rray[, 1:2]`

1	3	5
2	4	6
7	9	11
8	10	12

`y_rray`

1
2
7
8

`y_rray[, 1]`

1	3
2	4
7	9
8	10

`y_rray[, 1:2]`

`rray_extract()` always drops to 1D

1	3	5
2	4	6
7	9	11
8	10	12

`y_rray`

1 2 7 8

`rray_extract(y_rray, , 1)`

Broadcasting

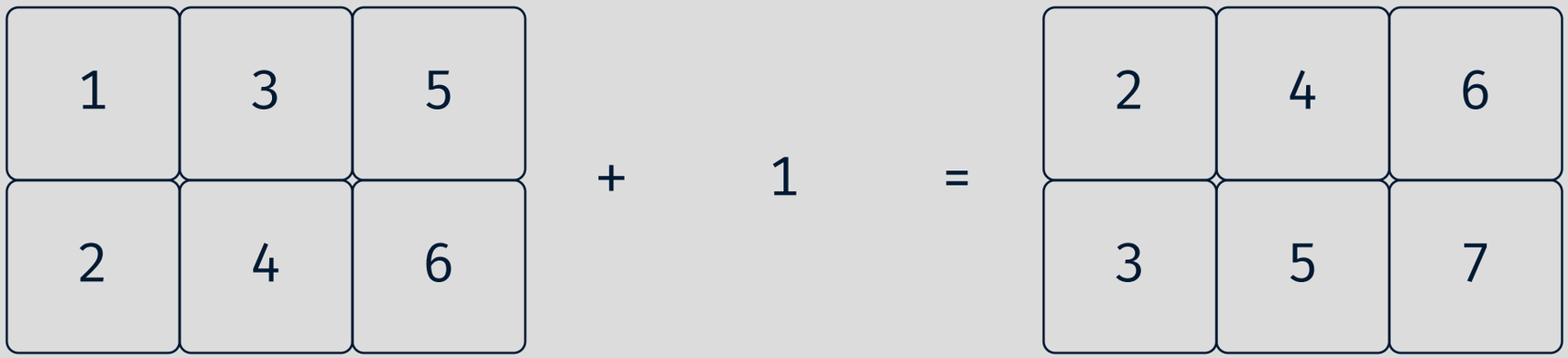
Broadcasting has to do with **increasing dimensionality** and **recycling dimensions.**

Let’s do some math

+ 1 =

x: (2, 3) y: (1)

Let’s do some math



`x: (2, 3)` `y: (1)` `(2, 3)`

How is y reshaped
so that this works?

Step 1 - increase dimensionality

```
x: (2, 3)
y: (1)
```

Dimensionality of **2**

Dimensionality of **1**

```
   (?, ?)
```

Append 1's **to the dimensionality of y**
until it matches the dimensionality of x

```
x: (2, 3)
y: (1, 1)
```

```
   (?, ?)
```

Step 1 - increase dimensionality

\+ 1 =

x: (2, 3) y: (1)

\+ 

=

x: (2, 3) y: (1, 1)

Step 2 - recycle dimensions

```
x: (2, 3)
y: (1, 1)
```

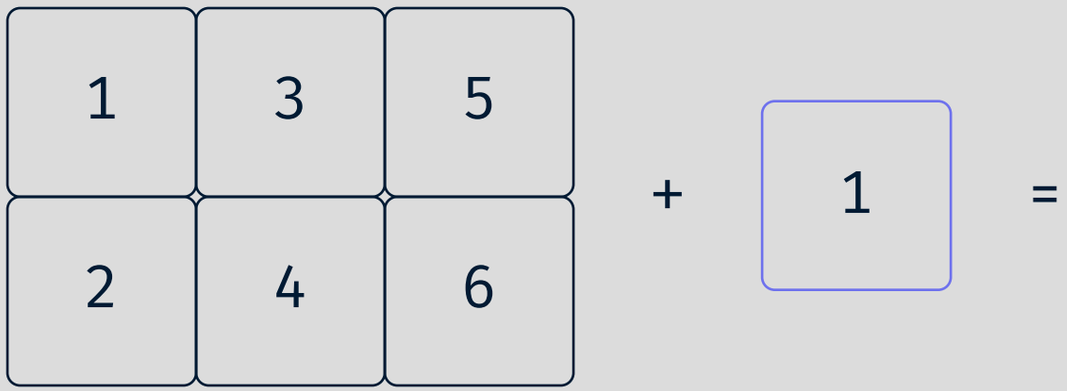
```
   (?, ?)
```

If the rows of `y` were *recycled* to length 2, it would match the length of the rows of `x`

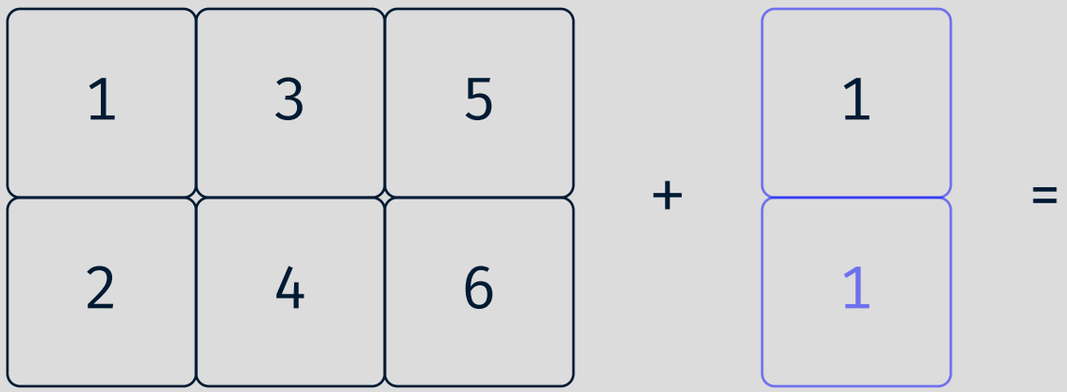
```
x: (2, 3)
y: (2, 1)
```

```
   (2, ?)
```

Step 2 - recycle dimensions



`x: (2, 3)` `y: (1, 1)`



`x: (2, 3)` `y: (2, 1)`

Step 2 - recycle dimensions

```
x: (2, 3)
y: (2, 1)
```

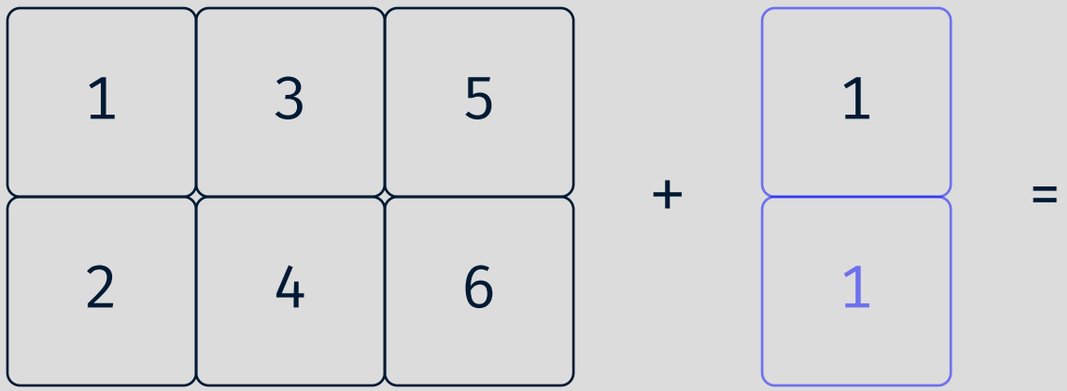
```
   (2, ?)
```

If the columns of `y` were *recycled* to length 3, it would match the length of the columns of `x`

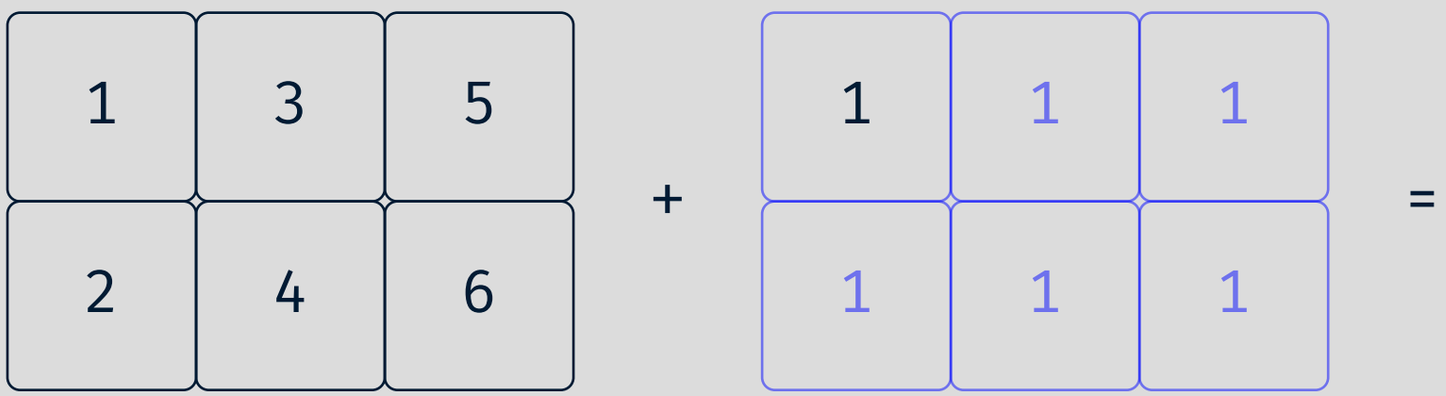
```
x: (2, 3)
y: (2, 3)
```

```
   (2, 3)
```

Step 2 - recycle dimensions

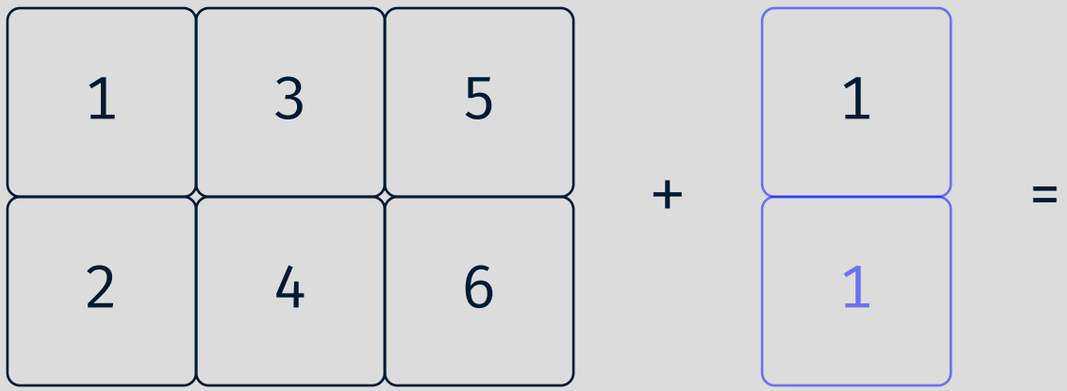


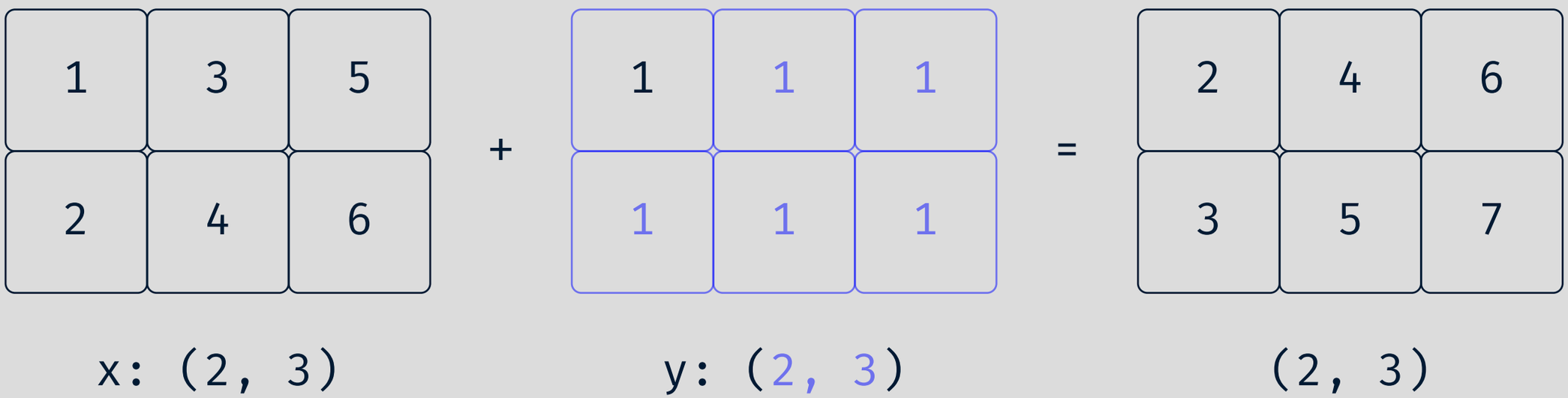
x: (2, 3) y: (2, 1)



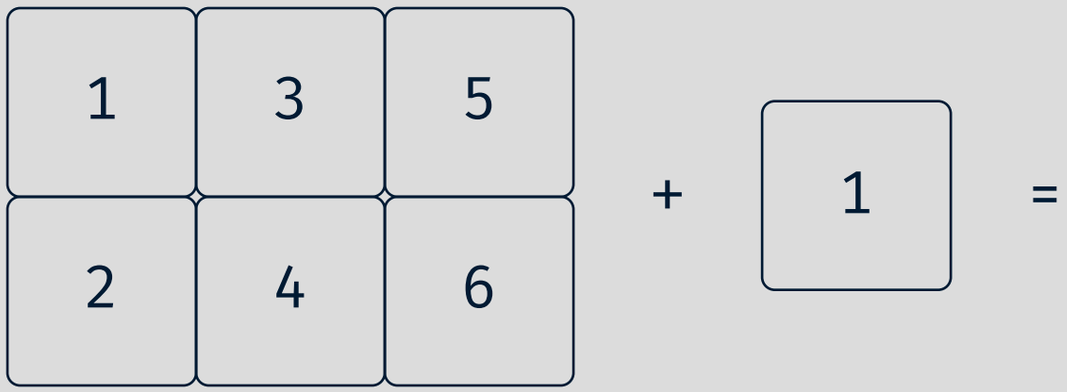
x: (2, 3) y: (2, 3)

Step 2 - recycle dimensions



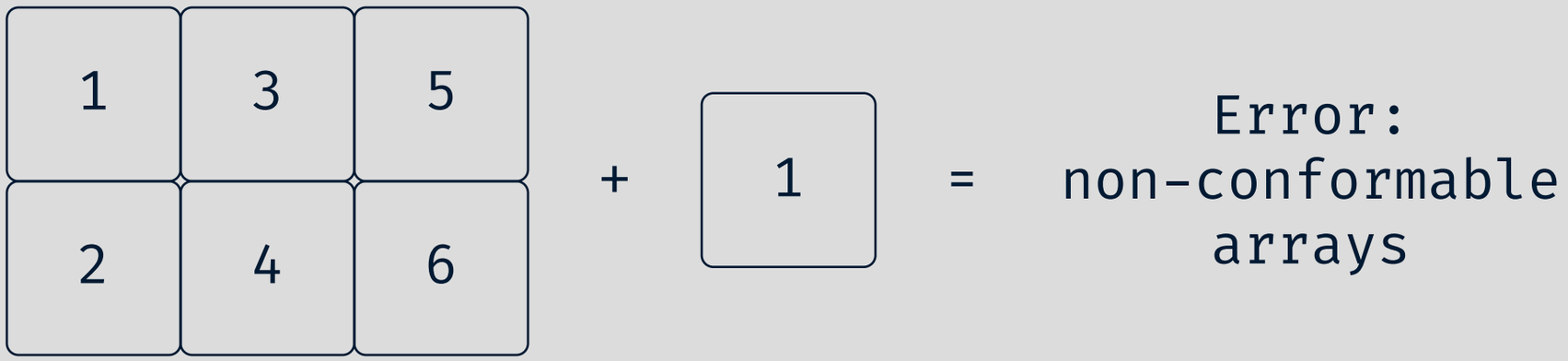


What if we started here?



`x: (2, 3)` `y: (1, 1)`

What if we started here?



What if we started here?

1	3	5
2	4	6

+

1

=

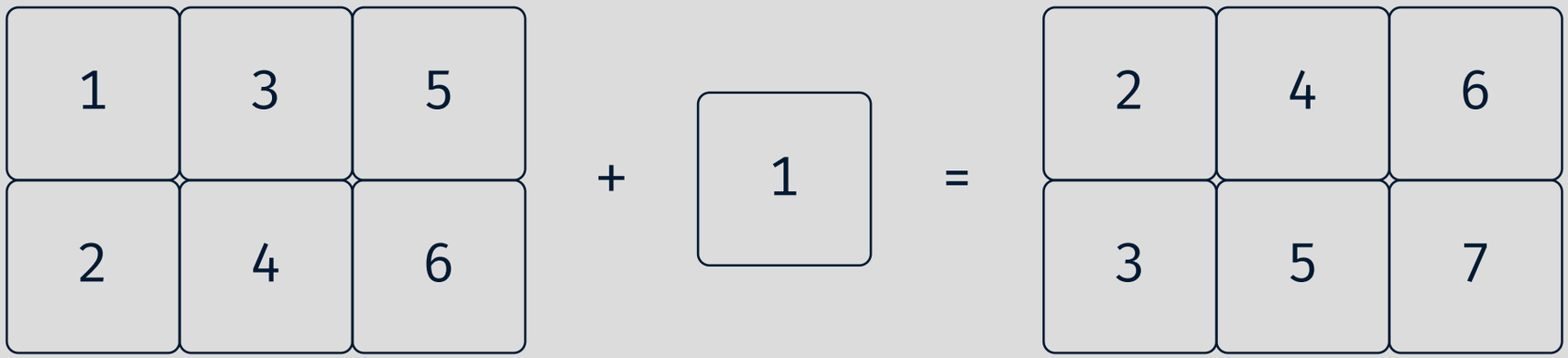
```
Error:
non-conformable
arrays
```

`x: (2, 3)` `y: (1, 1)`

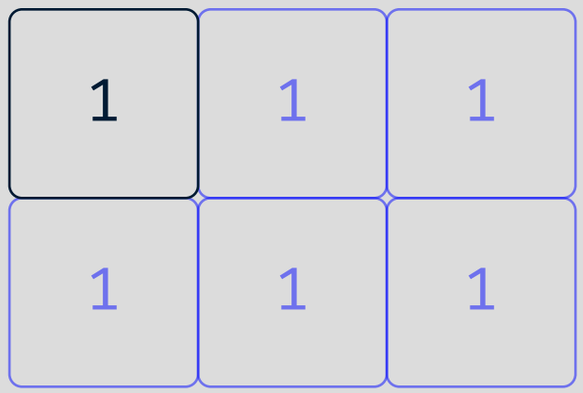
R doesn't broadcast.

We just got lucky that it worked with scalars.

We know the result



`x: (2, 3)` `y: (1, 1)` `(2, 3)`



`y: (2, 3)`

Broadcasting rules:

Match **dimensionality** by *appending 1's*

Match **dimensions** by *recycling* dimensions of length 1

rray broadcasts

```
library(rray)

x ← matrix(1:6, nrow = 2)
x
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6

z ← matrix(1)
z
#>      [,1]
#> [1,]    1

x + z
#> Error in x + z : non-conformable arrays

as_rray(x) + z
#> <vctrs_rray<double>[,3][6]>
#>      [,1] [,2] [,3]
#> [1,]    2    4    6
#> [2,]    3    5    7
```

How?

How?

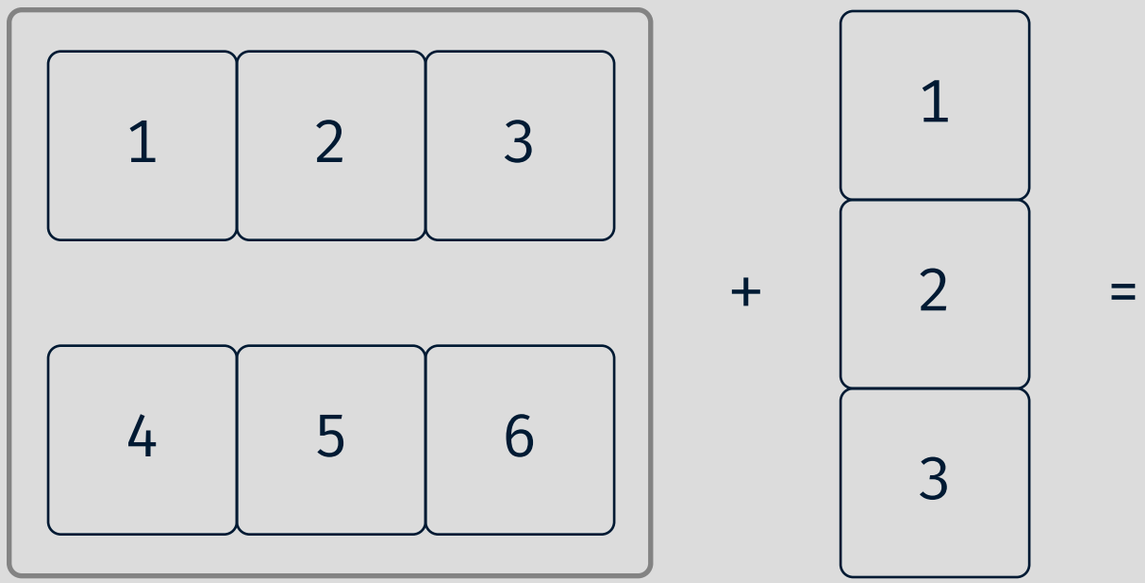
All hail our C++ overlords at QuantStack.

How?

All hail our C++ overlords at QuantStack.

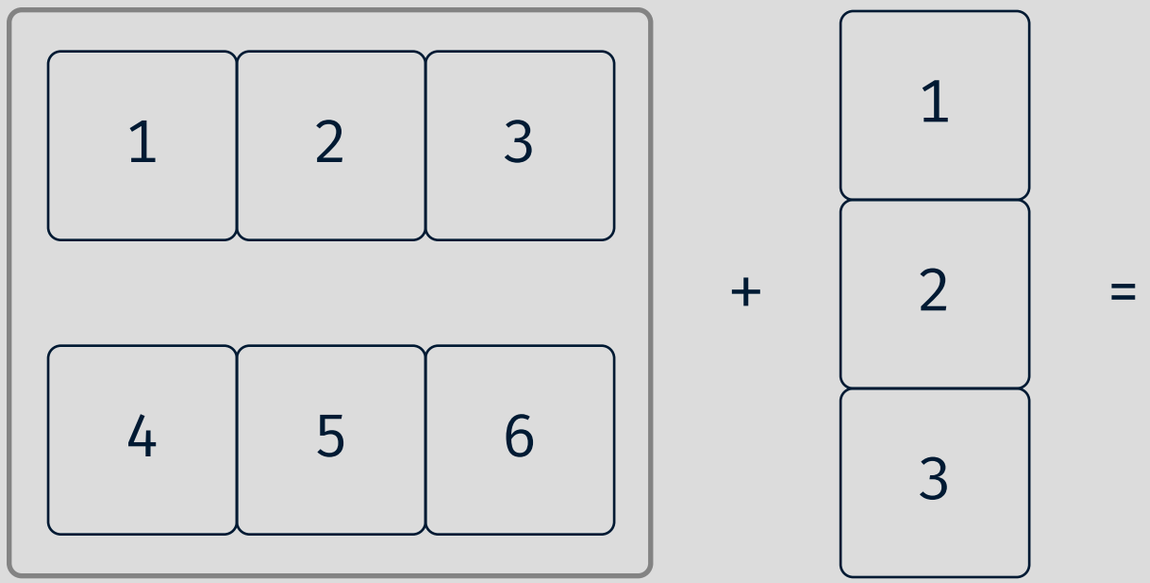
Buy them a beer for creating xtensor.

Let’s go 3D



`x: (1, 3, 2)` `y: (3, 1)`

Let’s go 3D



Can you even
do that??

x: (1, 3, 2) y: (3, 1)

Step 1 - increase dimensionality

```
x: (1, 3, 2)
y: (3, 1)
```

Dimensionality of **3**

Dimensionality of **2**

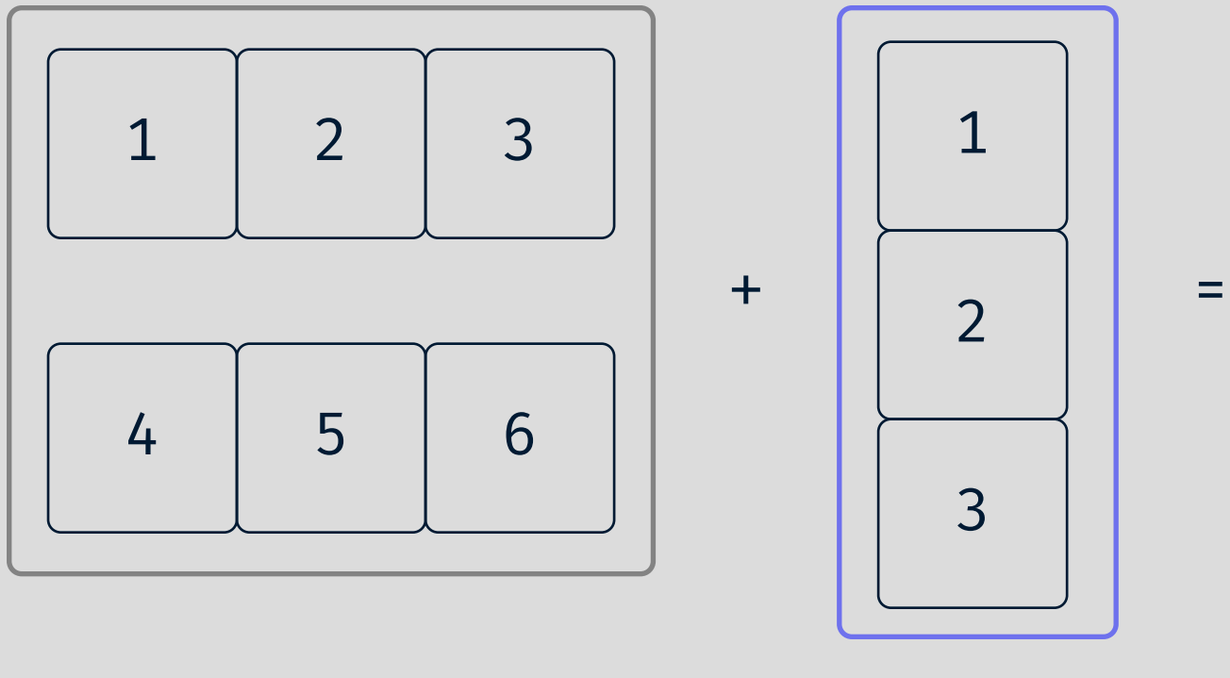
```
   (?, ?, ?)
```

Append 1's **to the dimensionality of y**
until it matches the dimensionality of x

```
x: (1, 3, 2)
y: (3, 1, 1)
```

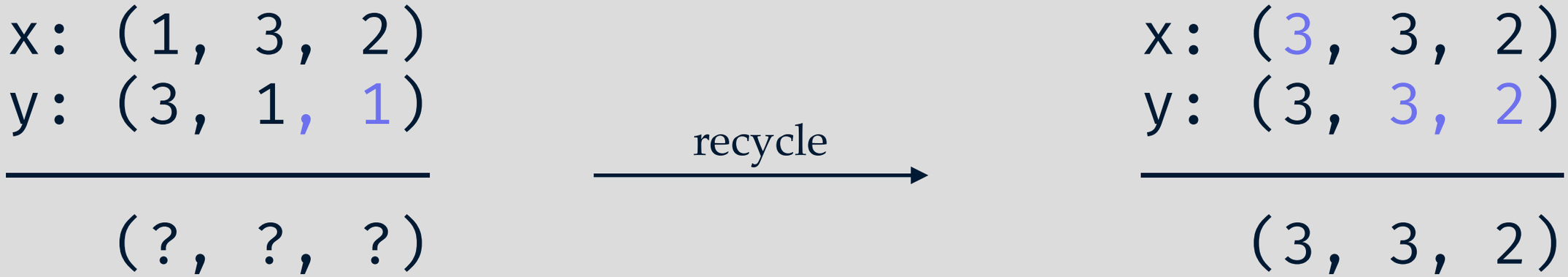
```
   (?, ?, ?)
```

Step 1 - increase dimensionality

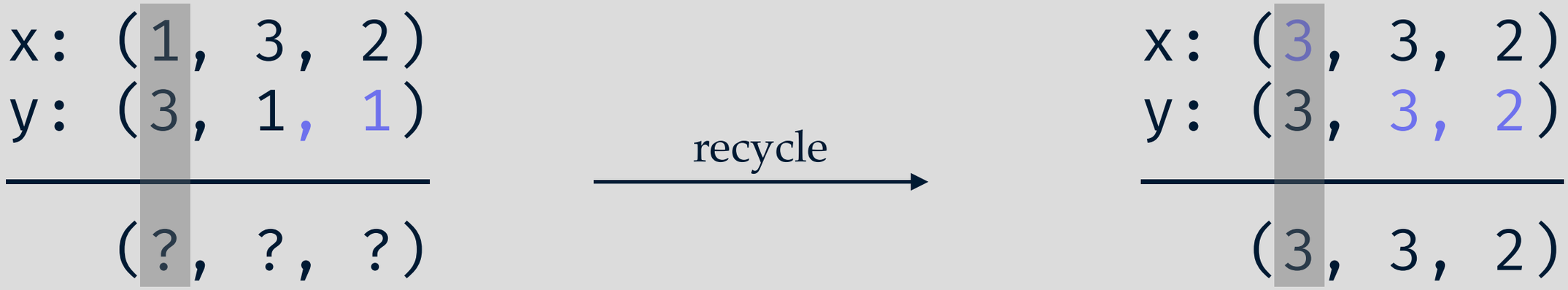


x: (1, 3, 2) y: (3, 1, 1)

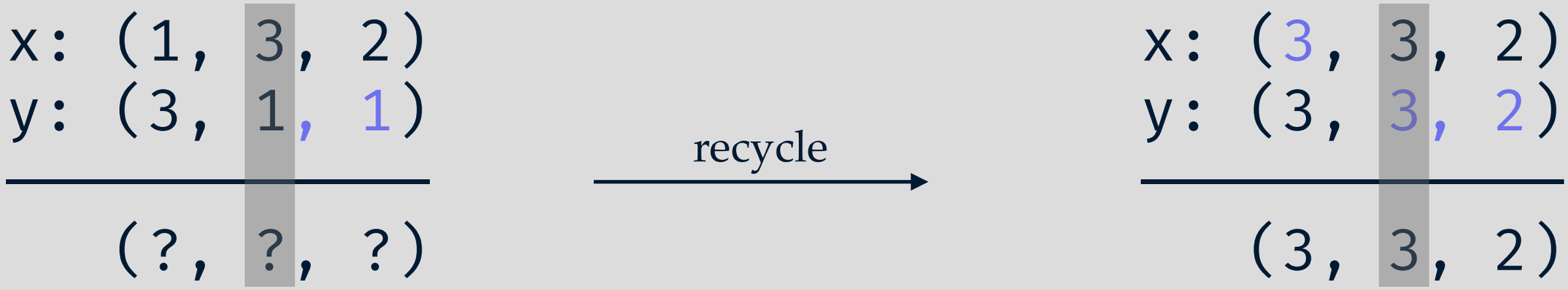
Step 2 - recycle dimensions



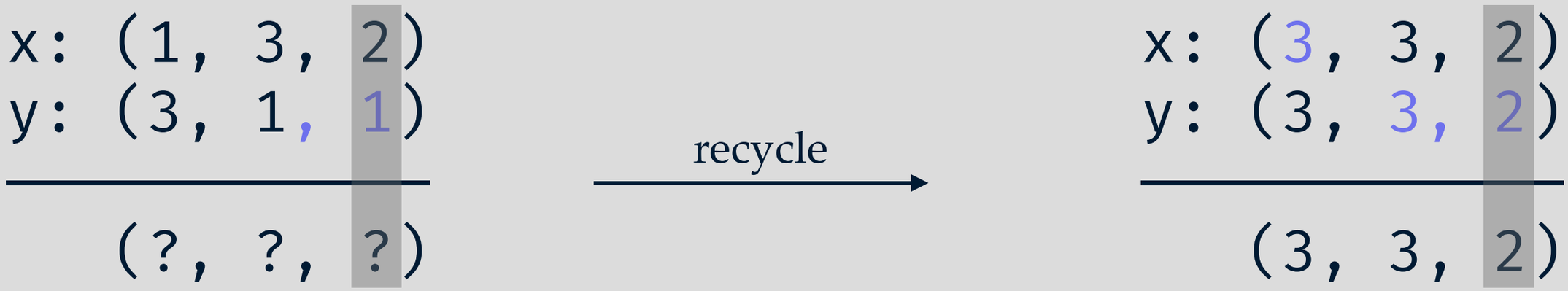
Step 2 - recycle dimensions



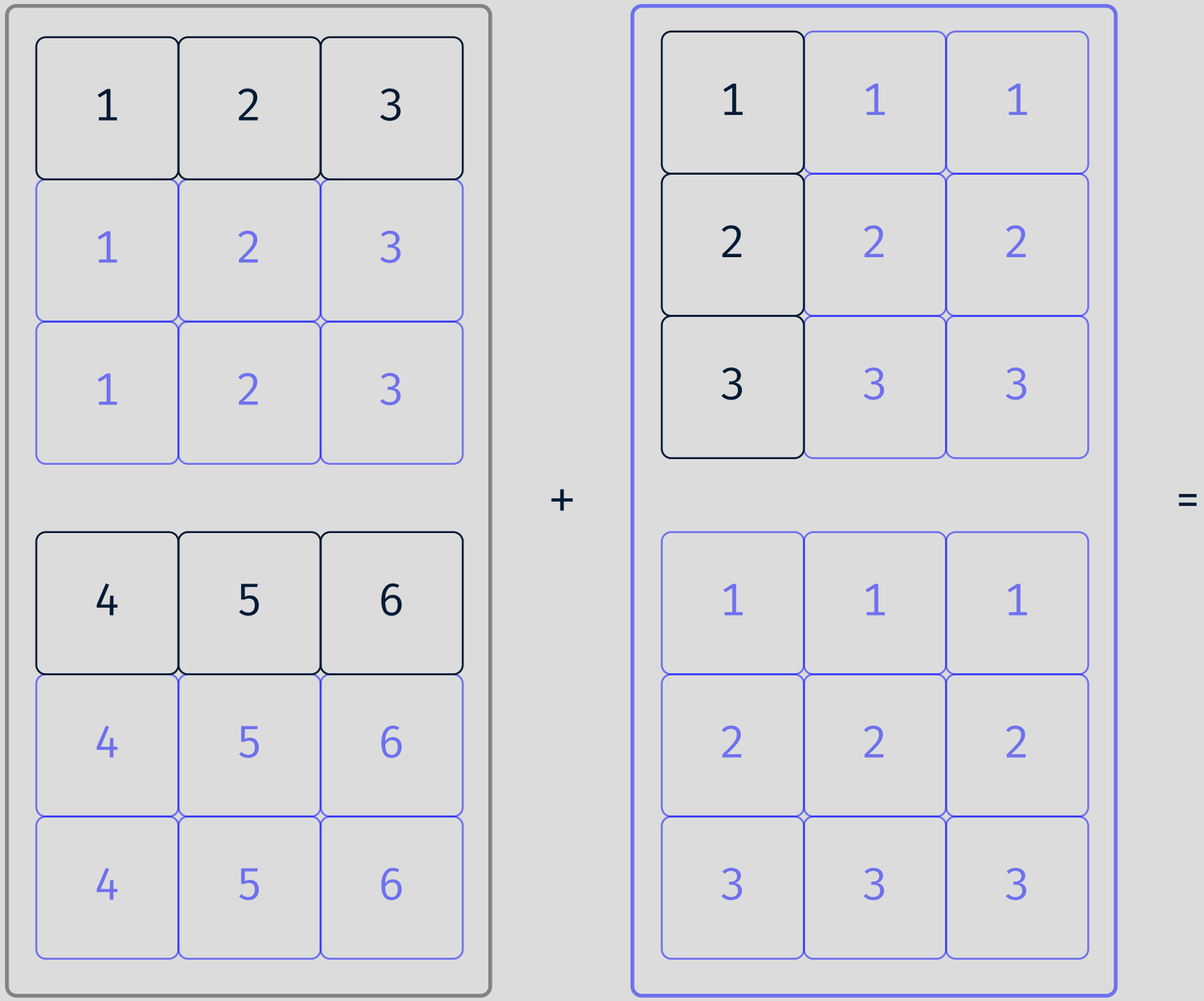
Step 2 - recycle dimensions



Step 2 - recycle dimensions



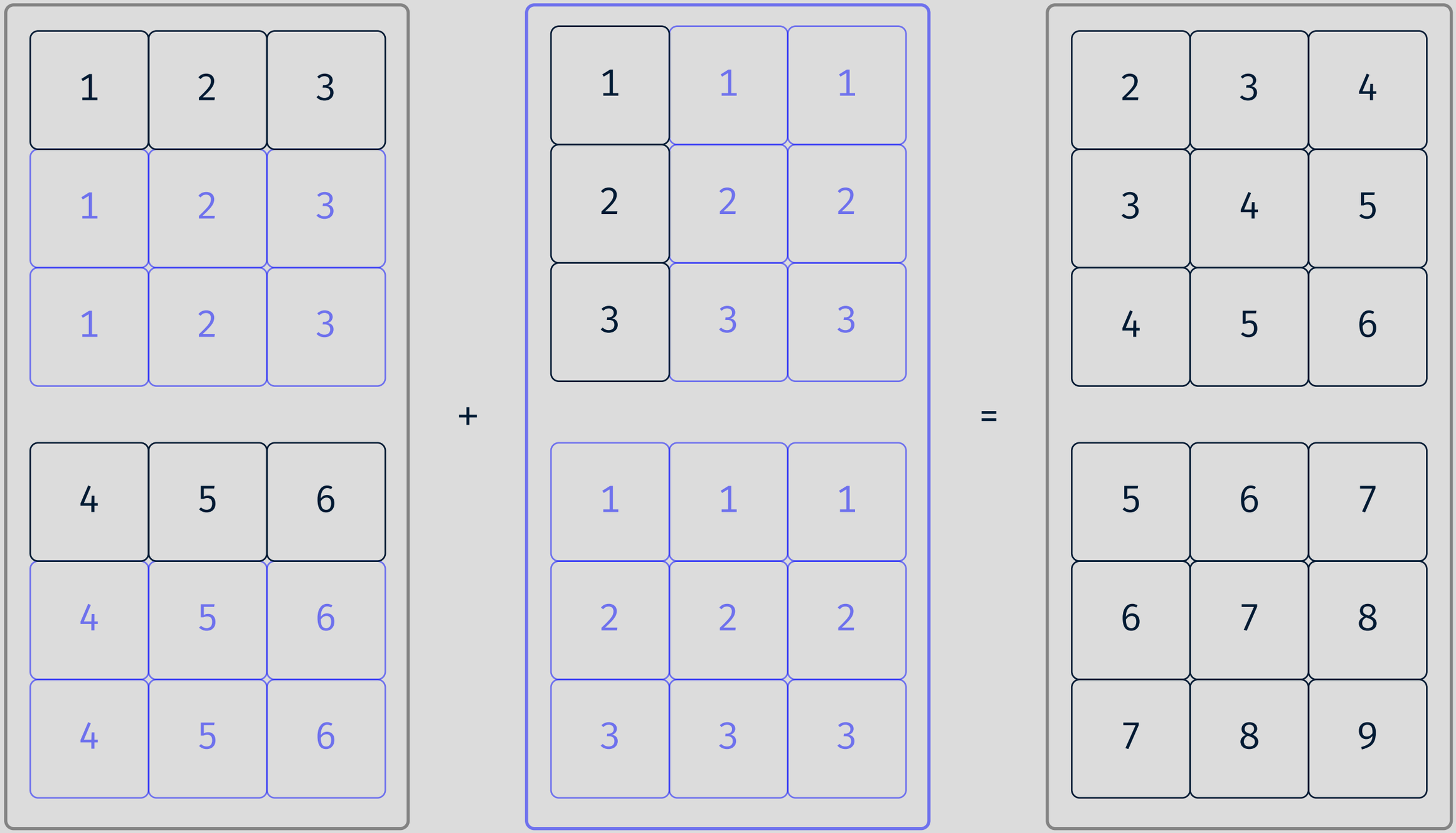
Step 2 - recycle dimensions



x: (3, 3, 2)

y: (3, 3, 2)

Step 2 - recycle dimensions



x: (3, 3, 2)

y: (3, 3, 2)

(3, 3, 2)

Manipulation

`rray` as a toolkit

`rray_bind()`

`rray_duplicate_any()`

`rray_expand_dims()`

`rray_broadcast()`

`rray_flip()`

`rray_max()`

`rray_sum()`

`rray_mean()`

`rray_reshape()`

`rray_rotate()`

`rray_split()`

`rray_tile()`

`rray_unique()`

…

The best part?

The best part?

They all work with base R.

`rray` as a toolkit

`rray_bind()`

`rray_duplicate_any()`

`rray_expand_dims()`

`rray_broadcast()`

`rray_flip()`

`rray_max()`

`rray_sum()`

`rray_mean()`

`rray_reshape()`

`rray_rotate()`

`rray_split()`

`rray_tile()`

`rray_unique()`

…

`rray` as a toolkit

`rray_bind()`

`rray_duplicate_any()`

`rray_expand_dims()`

`rray_broadcast()`

`rray_flip()`

`rray_max()`

`rray_sum()`

`rray_mean()`

`rray_reshape()`

`rray_rotate()`

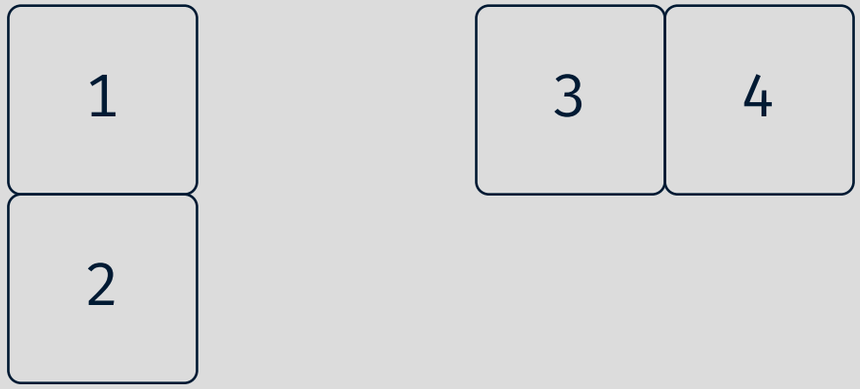
`rray_split()`

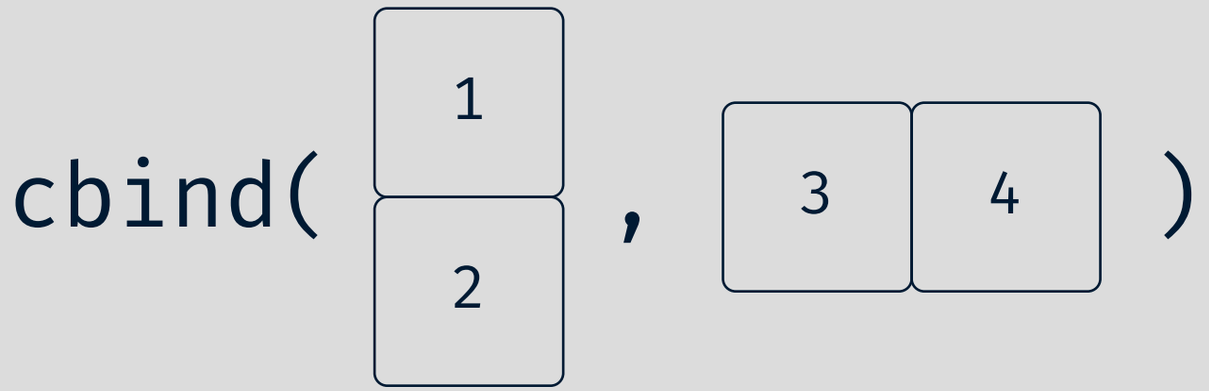
`rray_tile()`

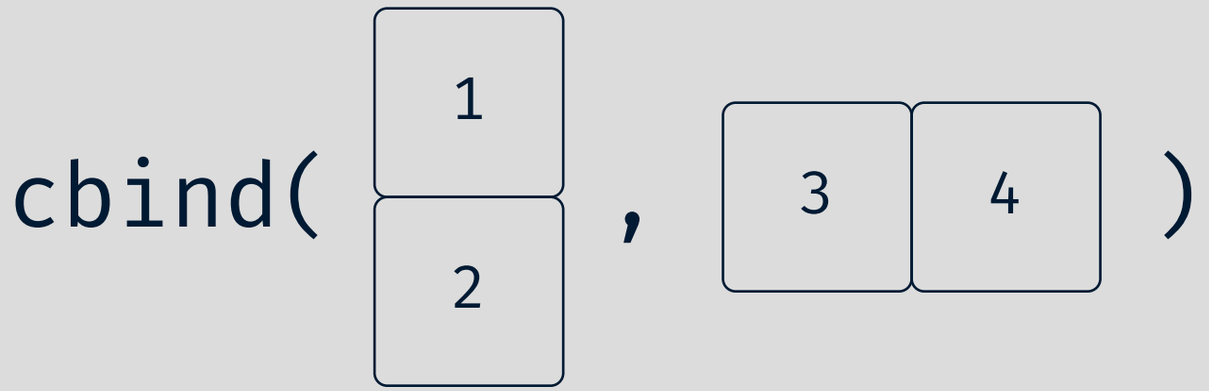
`rray_unique()`

…

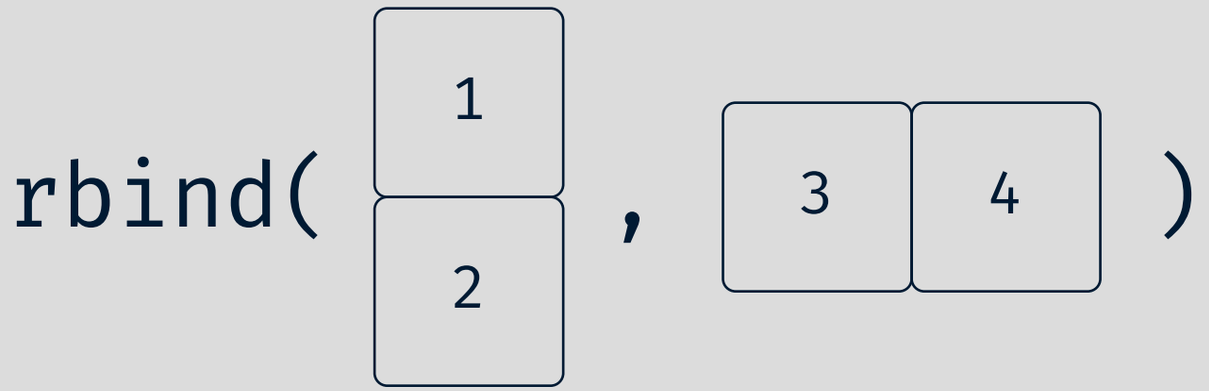
How can we bind these together?

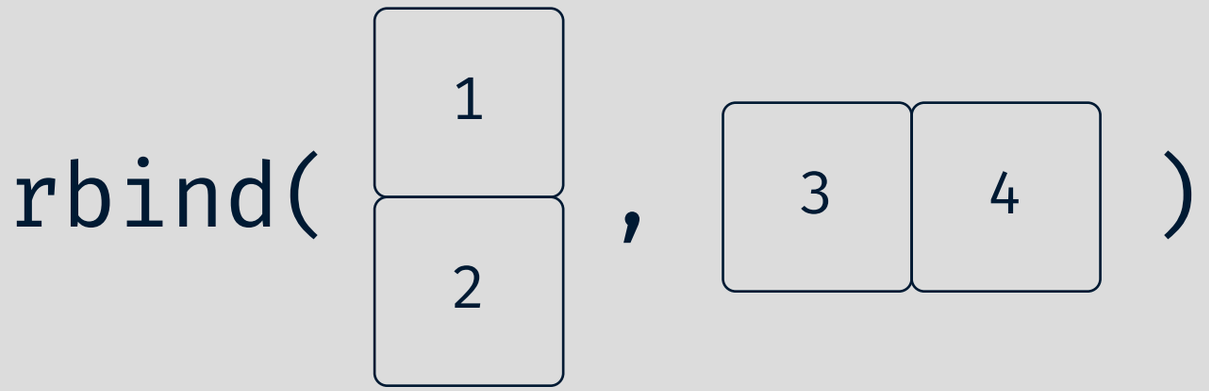



cbind(
1
2
,
3
4
)

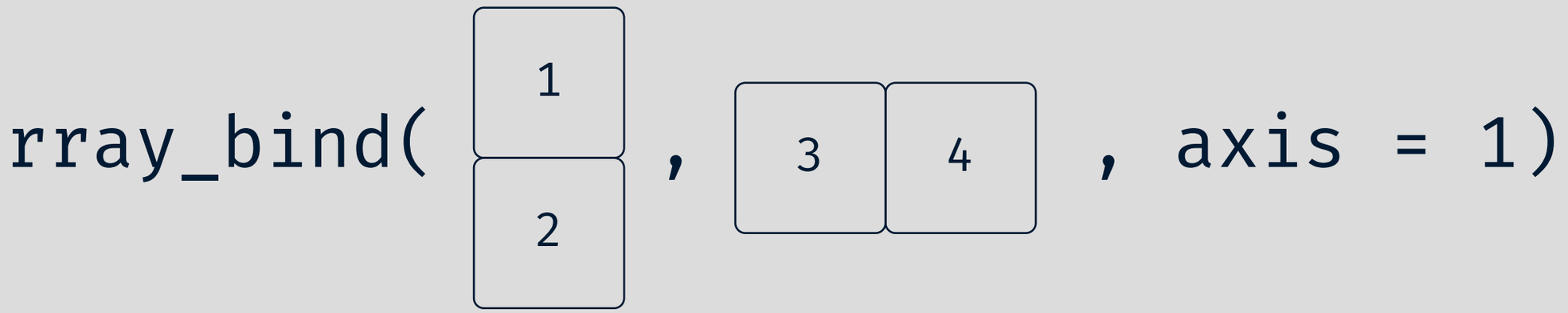


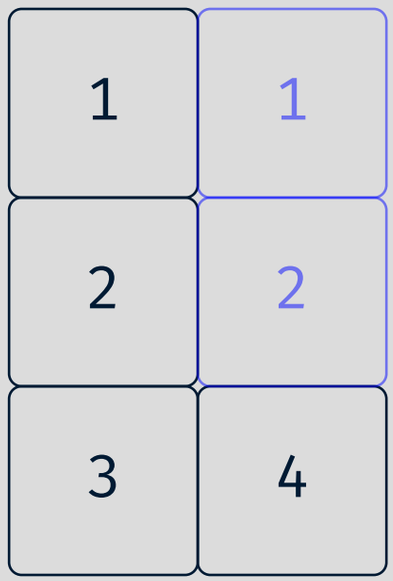
Error:
number of rows
of matrices must match

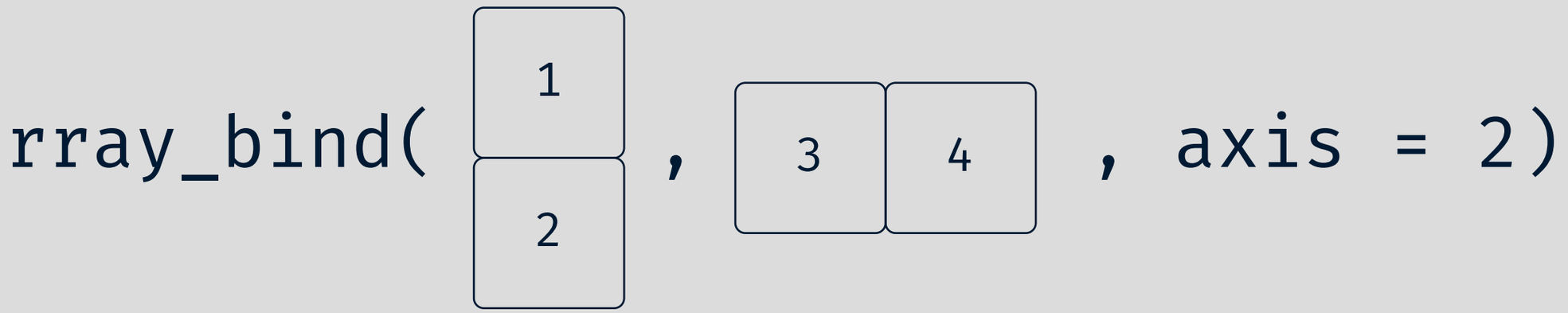

rbind(
1
2
,
3
4
)



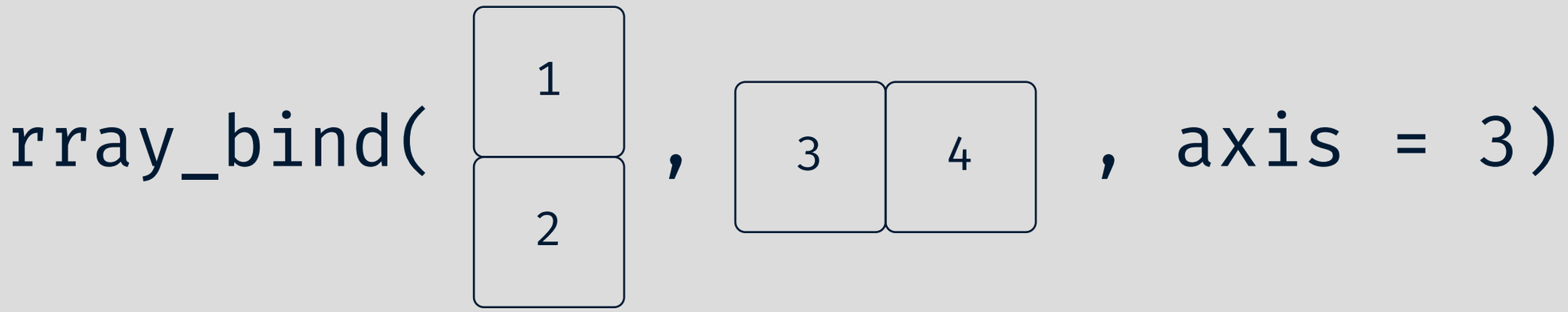

```
Error:
number of columns
of matrices must match
```

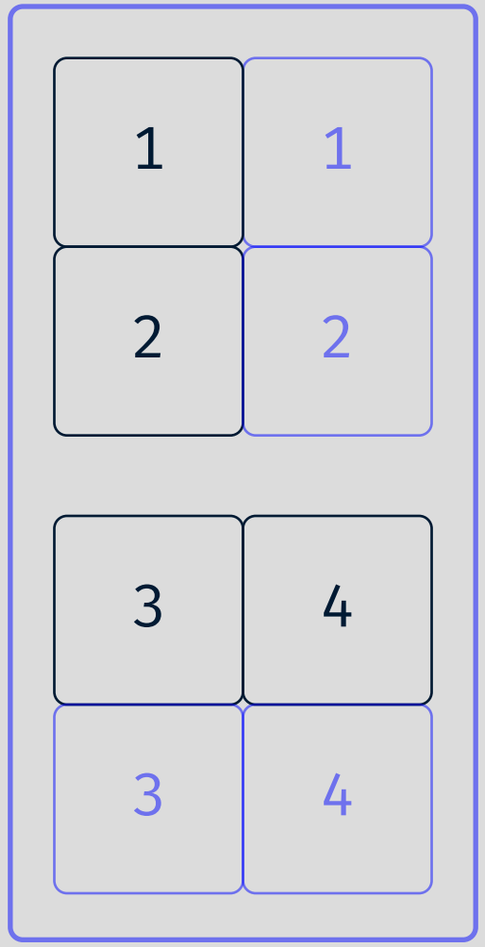

rray_bind(
1
2
,
3
4
, axis = 1)


1
1
2
2
3
4


rray_bind(
1
2
,
3
4
, axis = 2)


1
3
4
2
3
4


rray_bind(
1
2
,
3
4
, axis = 3)


1
1
2
2
3
4
3
4

`rray` as a toolkit

`rray_bind()`

`rray_duplicate_any()`

`rray_expand_dims()`

`rray_broadcast()`

`rray_flip()`

`rray_max()`

`rray_sum()`

`rray_mean()`

`rray_reshape()`

`rray_rotate()`

`rray_split()`

`rray_tile()`

`rray_unique()`

…

What if we want to “normalize”
by dividing by the max value?
Along columns? Along rows?

1	3	5
2	4	6

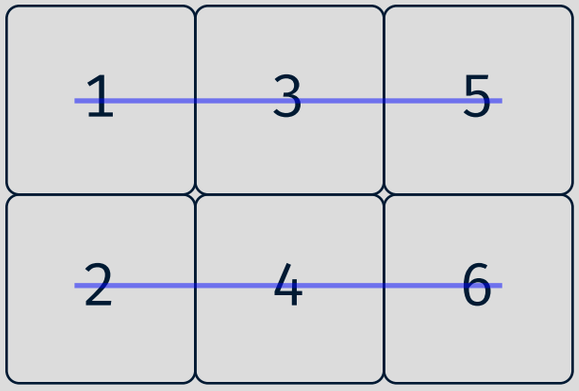
x

```
x / max(x)
```

1	3	5
2	4	6

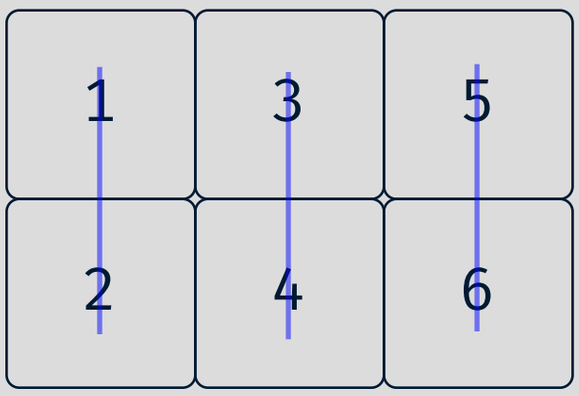
.167	.500	.833
.333	.667	1

```
sweep(x, 1, apply(x, 1, max), “/")
```



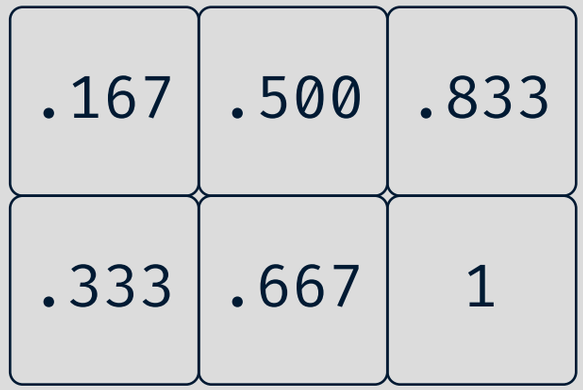
.200	.600	1
.333	.667	1

```
sweep(x, 2, apply(x, 2, max), “/")
```

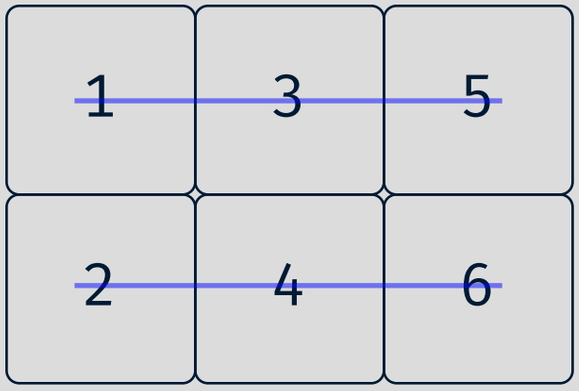


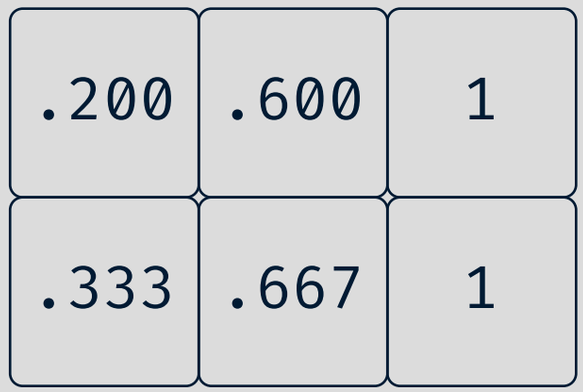
.500	.750	.833
1	1	1

```
x / rray_max(x)
```

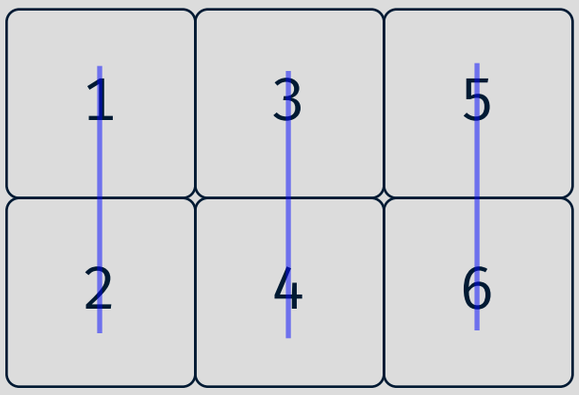


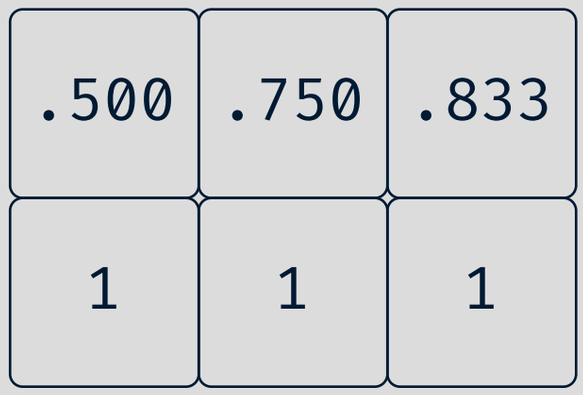

```
x / rray_max(x, axes = 2)
```



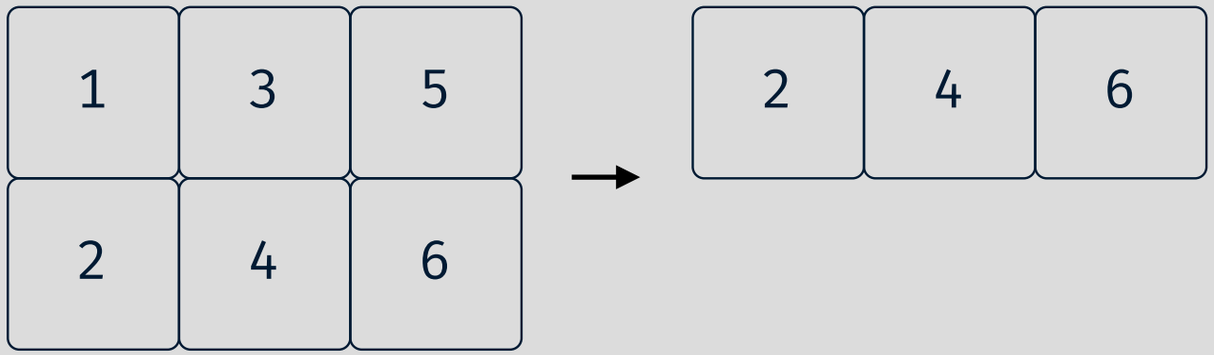



```
x / rray_max(x, axes = 1)
```

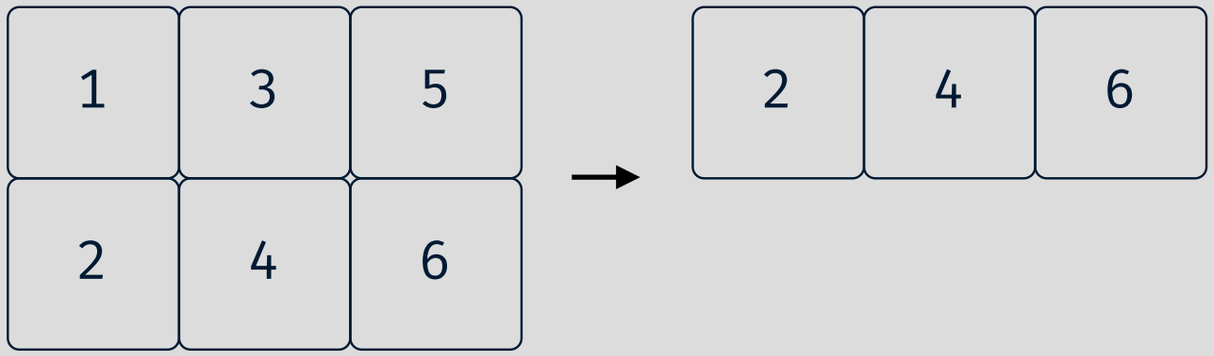




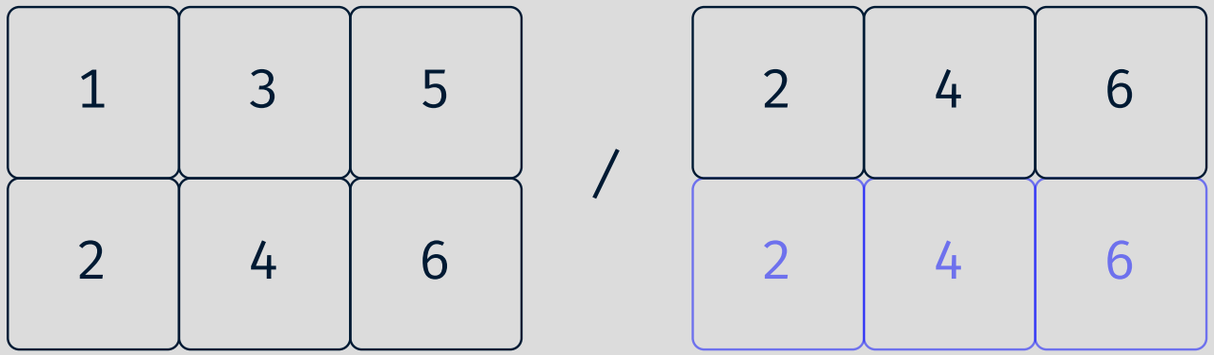
rray_max(x, axes = 1)



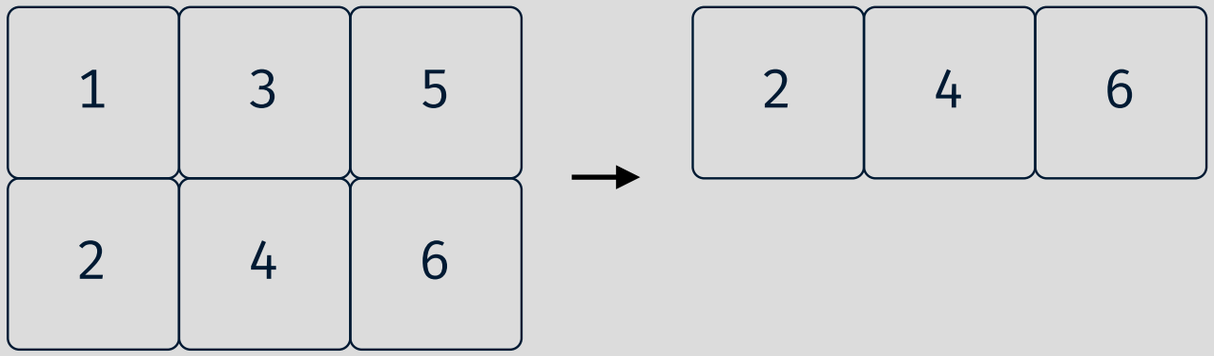
```
rray_max(x, axes = 1)
```



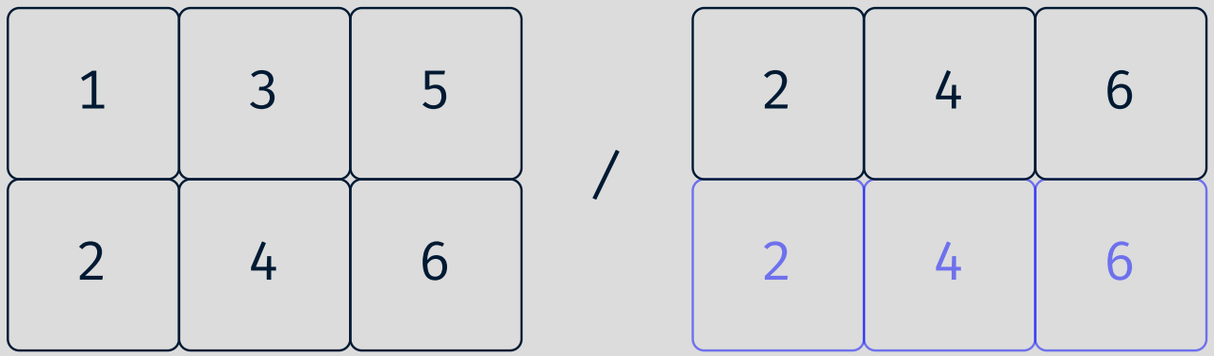

```
x / rray_max(x, axes = 1)
```

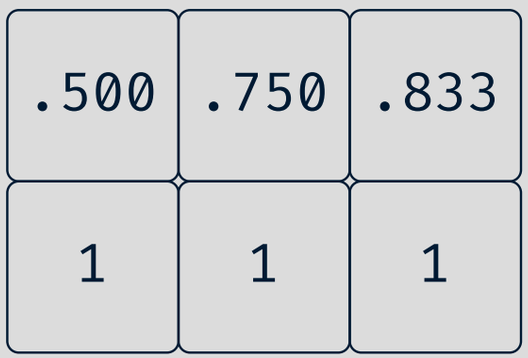


```
rray_max(x, axes = 1)
```




```
x / rray_max(x, axes = 1)
```





Arrays are:

1. *frustrating* to work with.

2. *difficult* to program around.

3. *underpowered*.

Arrays are:

1. *intuitive* to work with.

2. *predictable* to program around.

3. *powerful*.

GitHub

https://github.com/DavisVaughan/rray

Website

https://davisvaughan.github.io/rray/