

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

1	3	5
2	4	6

(2, 3)



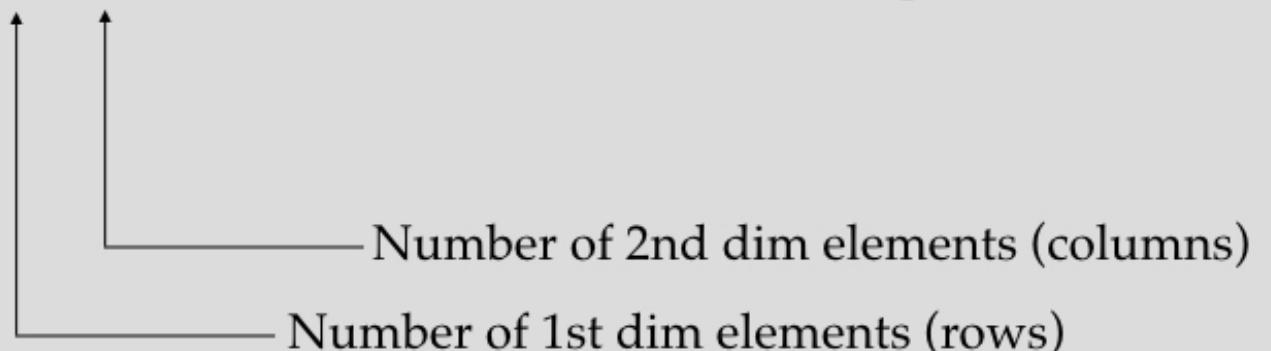
Number of 2nd dim elements (columns)

Number of 1st dim elements (rows)

dimensionality VS dimensions

1	3	5
2	4	6

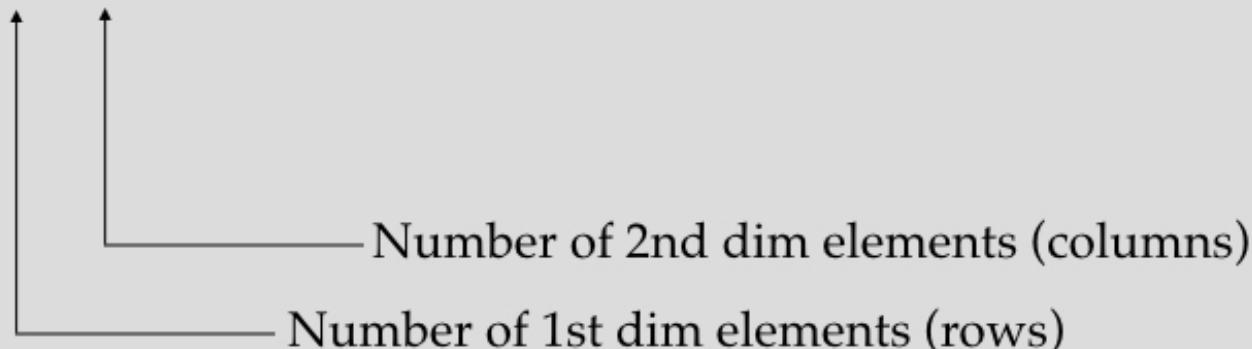
(2, 3)



dimensionality VS dimensions

1	3	5
2	4	6

(2, 3) ← The entire set makes up the **dimensions**



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

Subsetting

Enter the matrix.

1	3	5
2	4	6

x

Column selection

1	3	5
2	4	6

x

1	3
2	4

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...

1	3	5
2	4	6

x

1
2

x[, 1, drop = FALSE]

Let's go 3D

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

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]

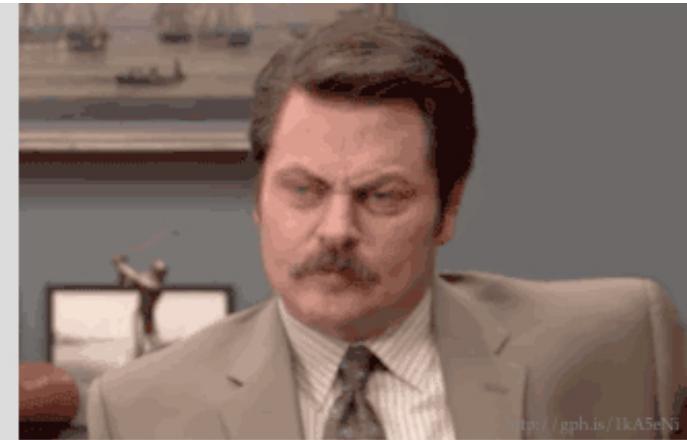
Let's go 3D

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...

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

y

1
2
7
8

y[, 1, , drop = FALSE]

The confusion?

Subsetting is not
dimensionality-stable.

Summary: column selection = 😠

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

* Drops to 2D

Summary: column selection = 😠

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

* Drops to 2D

Summary: column selection = 🎉

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

* Drops to 2D

Summary: column selection = 🎉

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

* Drops to 2D

Summary: column selection = 🎉

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

* Drops to 2D

rray

array 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]

[[always drops to 1D

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

y_rray

1 2 7 8

y_rray[[, 1]]

Broadcasting

Broadcasting has to do with
increasing dimensionality and
recycling dimensions.

Let's do some math

1	3	5
2	4	6

+

1 =

x: (2, 3)

y: (1)

Let's do some math

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + 1 = \begin{array}{|c|c|c|} \hline 2 & 4 & 6 \\ \hline 3 & 5 & 7 \\ \hline \end{array}$$

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

How is y reshaped
so that this works?

Step 1 - increase dimensionality

$$\begin{array}{l} \text{x: } (2, 3) \quad \longleftarrow \text{Dimensionality of 2} \\ \text{y: } (1) \quad \longleftarrow \text{Dimensionality of 1} \\ \hline (?, ?) \end{array}$$

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

$$\begin{array}{l} \text{x: } (2, 3) \\ \text{y: } (1, \textcolor{blue}{1}) \\ \hline (?, ?) \end{array}$$

Step 1 - increase dimensionality

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline \end{array}$$

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

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline \end{array}$$

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

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline \end{array} =$$

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

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array} =$$

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

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array} =$$

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

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} =$$

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

Step 2 - recycle dimensions

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array} =$$

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

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 2 & 4 & 6 \\ \hline 3 & 5 & 7 \\ \hline \end{array}$$

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

What if we started here?

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline \end{array} =$$

$x: (2, 3)$

$y: (1, 1)$

What if we started here?

1	3	5
2	4	6

+  = Error:
non-conformable
arrays

x: (2, 3)

y: (1, 1)

What if we started here?

1	3	5
2	4	6

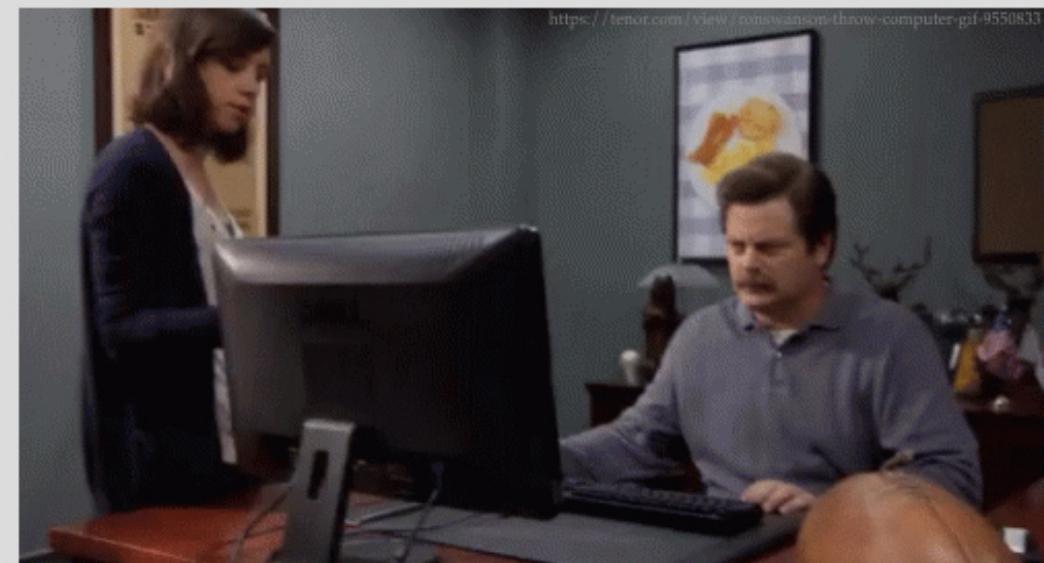
+

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

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 2 & 4 & 6 \\ \hline 3 & 5 & 7 \\ \hline \end{array}$$

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

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

$$y: (\underline{2}, \underline{3})$$

Broadcasting rules:

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

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

rarray broadcasts

```
library(rarray)

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

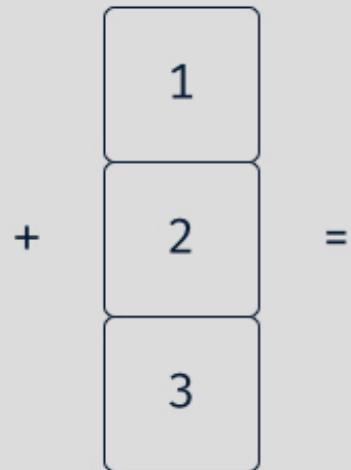
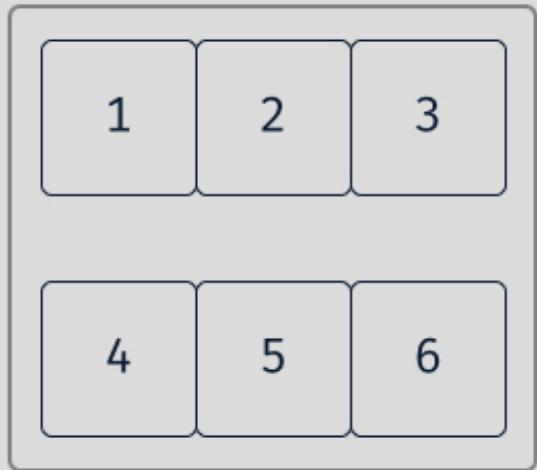
as_rarray(x) + z
#> <vctrs_rray<double>[,3][6]>
#>      [,1] [,2] [,3]
#> [1,]     2     4     6
#> [2,]     3     5     7
```

Let's go 3D

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 4 & 5 & 6 \\ \hline \end{array}$$

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

Let's go 3D



+

=
Can you even
do that??

x: (1, 3, 2)

y: (3, 1)

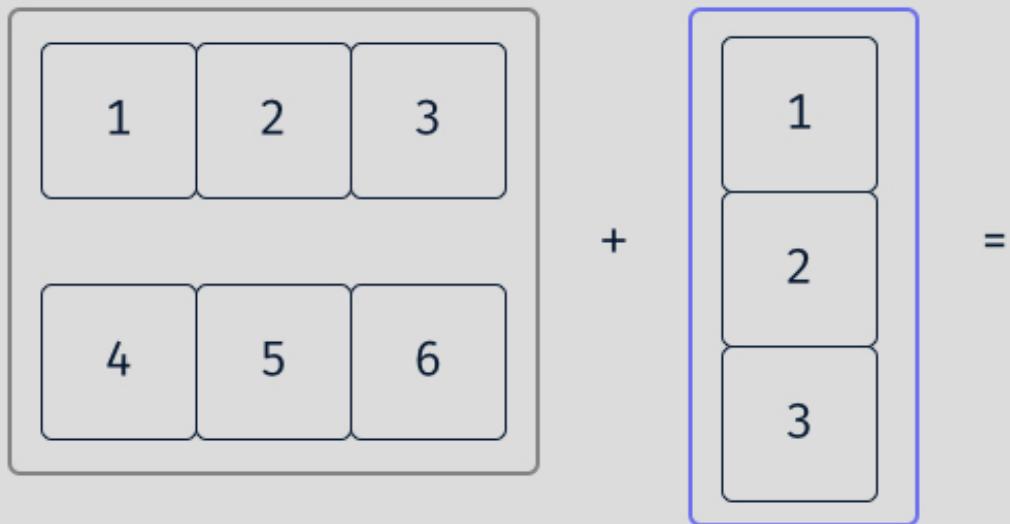
Step 1 - increase dimensionality

$$\begin{array}{rcl} \mathbf{x}: (1, 3, 2) & \longleftarrow & \text{Dimensionality of 3} \\ \mathbf{y}: (3, 1) & \longleftarrow & \text{Dimensionality of 2} \\ \hline & & (?, ?, ?) \end{array}$$

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

$$\begin{array}{rcl} \mathbf{x}: (1, 3, 2) \\ \mathbf{y}: (3, 1, \color{blue}{1}) \\ \hline & & (?, ?, ?) \end{array}$$

Step 1 - increase dimensionality



$$x: (1, 3, 2) \quad y: (3, 1, 1)$$

Step 2 - recycle dimensions

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

(?, ?, ?)

recycle →

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

(3, 3, 2)

Step 2 - recycle dimensions

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

(?, ?, ?)

recycle →

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

(3, 3, 2)

Step 2 - recycle dimensions

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

recycle →

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

(3, 3, 2)

Step 2 - recycle dimensions

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

(?, ?, ?)

recycle →

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

(3, 3, 2)

Step 2 - recycle dimensions

1	2	3
1	2	3
1	2	3

4	5	6
4	5	6
4	5	6

$x: (3, 3, 2)$

+

1	1	1
2	2	2
3	3	3

$y: (3, 3, 2)$

=

Step 2 - recycle dimensions

1	2	3
1	2	3
1	2	3

4	5	6
4	5	6
4	5	6

$x: (3, 3, 2)$

+

1	1	1
2	2	2
3	3	3

1	1	1
2	2	2
3	3	3

$y: (3, 3, 2)$

=

2	3	4
3	4	5
4	5	6

5	6	7
6	7	8
7	8	9

$(3, 3, 2)$

Manipulation

rarray as a toolkit

`rarray_bind()`

`rarray_mean()`

`rarray_duplicate_any()`

`rarray_reshape()`

`rarray_expand_dims()`

`rarray_rotate()`

`rarray_broadcast()`

`rarray_split()`

`rarray_flip()`

`rarray_tile()`

`rarray_max()`

`rarray_unique()`

`rarray_sum()`

...

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()`

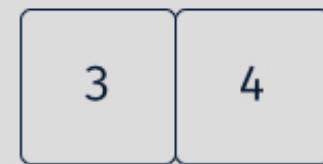
`rray_split()`

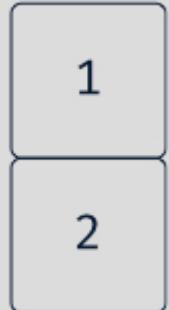
`rray_tile()`

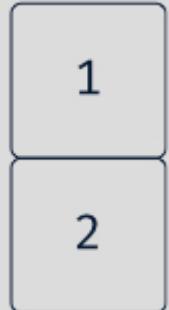
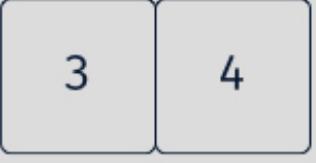
`rray_unique()`

...

How can we bind these together?

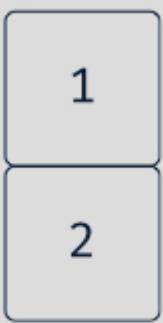


```
cbind(  ,  )
```

```
cbind(  ,  )
```

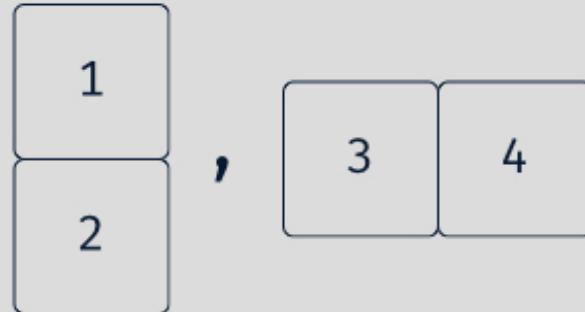
Error:
number of rows
of matrices must match

```
rbind(  ,  )
```

```
rbind(  ,  )
```

Error:
number of columns
of matrices must match

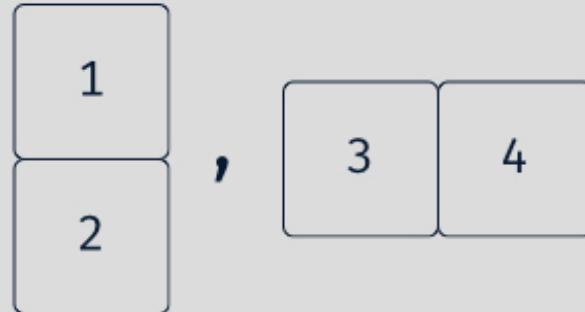
```
rray_bind(
```



```
, axis = 1)
```



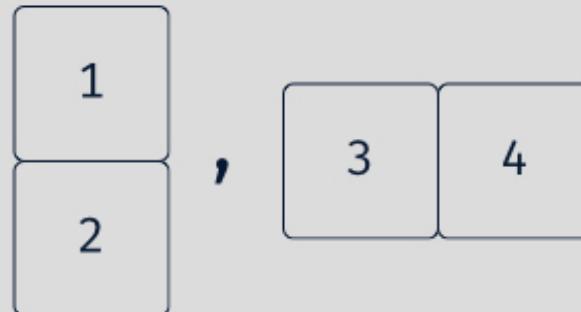
```
rray_bind(
```



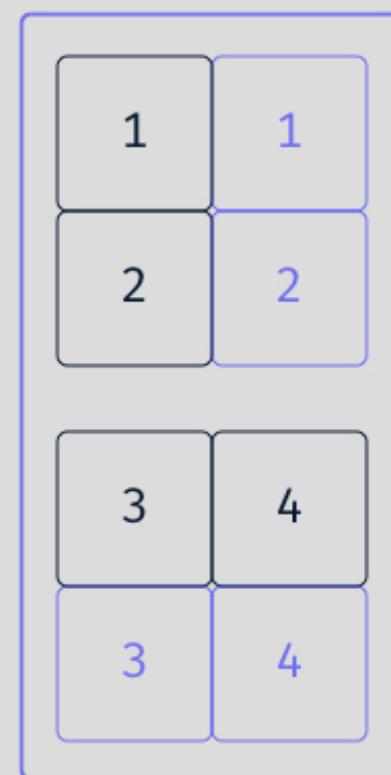
```
, axis = 2)
```



```
rray_bind(
```



```
, axis = 3)
```



rarray as a toolkit

`rarray_bind()`

`rarray_mean()`

`rarray_duplicate_any()`

`rarray_reshape()`

`rarray_expand_dims()`

`rarray_rotate()`

`rarray_broadcast()`

`rarray_split()`

`rarray_flip()`

`rarray_tile()`

`rarray_max()`

`rarray_unique()`

`rarray_sum()`

...

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), "/")`

1	3	5
2	4	6

.200	.600	1
.333	.667	1

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

1	3	5
2	4	6

.500	.750	.833
1	1	1

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

1	3	5
2	4	6

.167	.500	.833
.333	.667	1

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

1	3	5
2	4	6

.200	.600	1
.333	.667	1

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

1	3	5
2	4	6

.500	.750	.833
1	1	1

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

$$\begin{array}{|c|c|c|} \hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} \quad / \quad \begin{array}{|c|c|c|} \hline 2 & 4 & 6 \\ \hline 2 & 4 & 6 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline .500 & .750 & .833 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

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*.