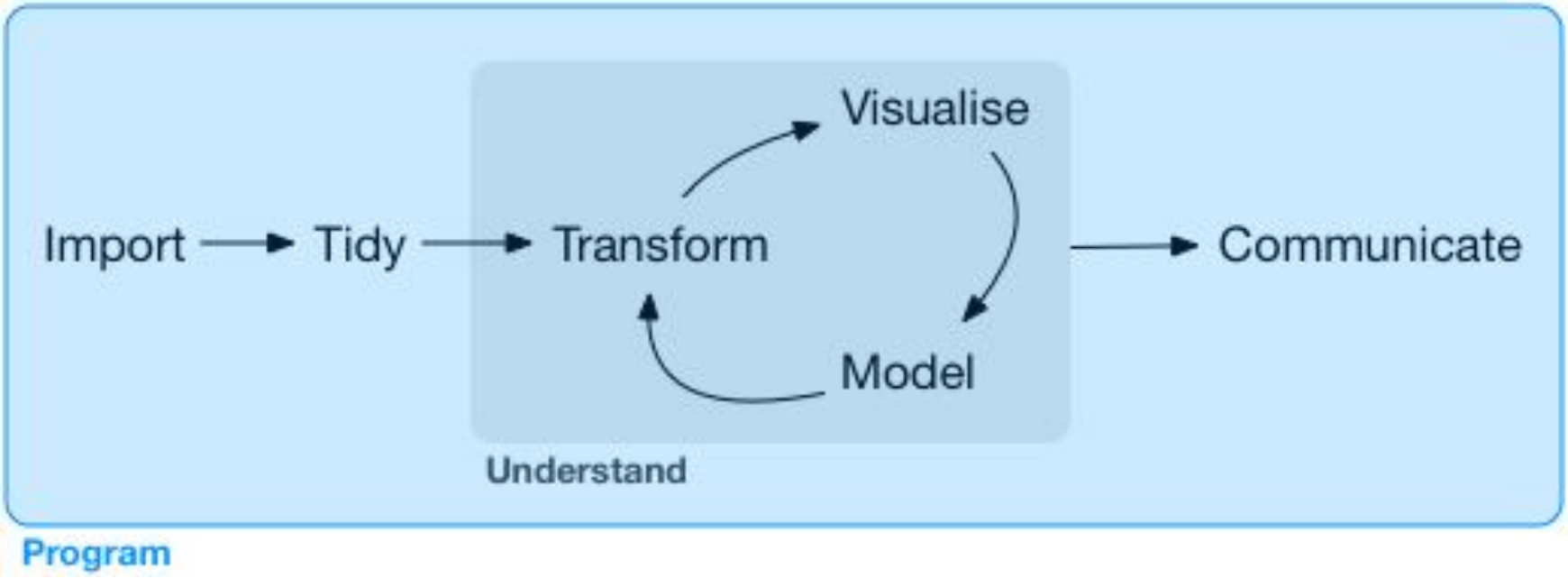
R for Data Science

Chapters 17 - 21

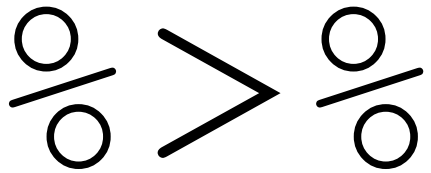
III PROGRAM

Chapter 17 Introduction


Import
Tidy
Transform
Visualise
Model
Understand
Communicate
Program

Chapter 18 Pipes

magrittr



migrittr

Little bunny Foo Foo

Went hopping through the forest

Scooping up the field mice

And bopping them on the head

```
foo_foo <- little_bunny()
```

migrittr

Little bunny Foo Foo

Went hopping through the forest

Scooping up the field mice

And bopping them on the head

```
foo_foo <- little_bunny()

foo_foo_1 <-  hop(foo_foo,
     through = forest)
foo_foo_2 <- scoop(foo_foo_1,
     up = field_mice)
foo_foo_3 <- bop(foo_foo_2,
     on = head)
```

migrittr

Little bunny Foo Foo

```
foo_foo <- little_bunny()
```

Went hopping through the forest

```
foo_foo <-  hop(foo_foo,
     through = forest)
```

Scooping up the field mice

```
foo_foo <- scoop(foo_foo_1,
     up = field_mice)
```

And bopping them on the head

```
foo_foo <- bop(foo_foo_2,
     on = head)
```

migrittr

Little bunny Foo Foo

Went hopping through the forest

Scooping up the field mice

And bopping them on the head

```
foo_foo <- little_bunny()

bop(
  scoop(
    hop(foo_foo,
        through = forest),
    up = field_mouse
  ),
)
```

migrittr

Little bunny Foo Foo

Went hopping through the forest

Scooping up the field mice

And bopping them on the head

```
foo_foo <- little_bunny()

foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

`migrittr`

The pipe does not work with:

1. Functions that use the current environment.

2. Functions that use lazy evaluation.

Chapter 19 Functions

“You should consider writing a function whenever you’ve copied and pasted a block of code more than twice (i.e. you now have three copies of the same code).”

There are three key steps to creating a new function:

1. You need to pick a **name** for the function. Here I've used `rescale01` because this function rescales a vector to lie between 0 and 1.
2. You list the inputs, or **arguments**, to the function inside `function`. Here we have just one argument. If we had more the call would look like `function(x, y, z)`.
3. You place the code you have developed in **body** of the function, a `{` block that immediately follows `function(...)`.

`if..else` - use to make conditional statements

`if..else if...else` - use to make conditional statements with more than 2 options

`switch()` - similar to `if..else if` but a cleaner way to encode and reference the options

`cut()` - discretize continuous variables

`stop()` - stops execution of the current expression and executes an error action

`stopifnot()` - if any of the expressions in ... are not all TRUE, stop is called, producing an error message indicating the *first* of the elements of ... which were not true.

… - allows passing of an arbitrary number of inputs

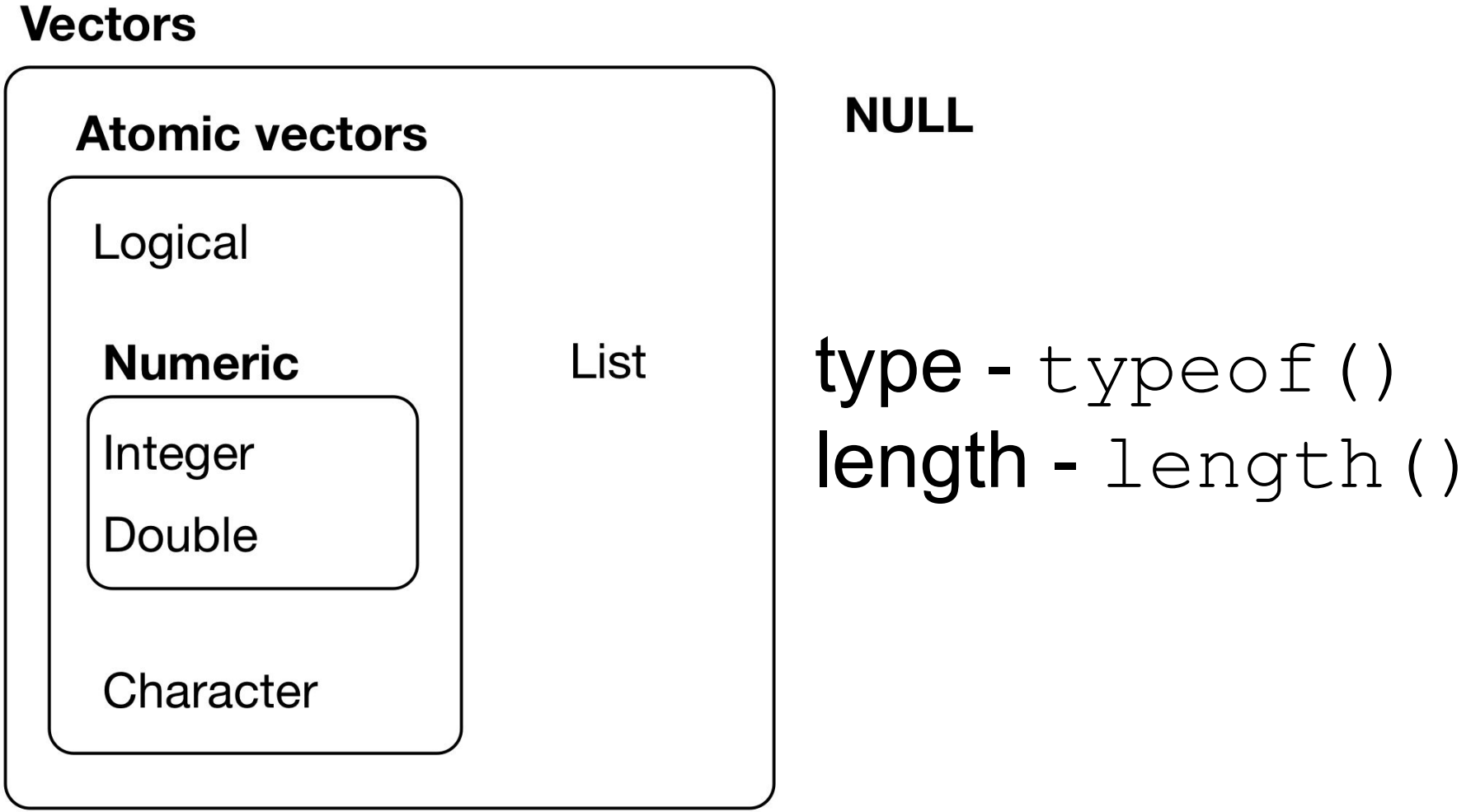
`return()` - explicitly return a value

`invisible()` - hide a value from initial output, but keep as part of object

```
f <- function(x) {
  x + y
}
```

In many programming languages, this would be an error, because `y` is not defined inside the function. In R, this is valid code because R uses rules called **lexical scoping** to find the value associated with a name. Since `y` is not defined inside the function, R will look in the **environment** where the function was defined.

Chapter 20 Vectors

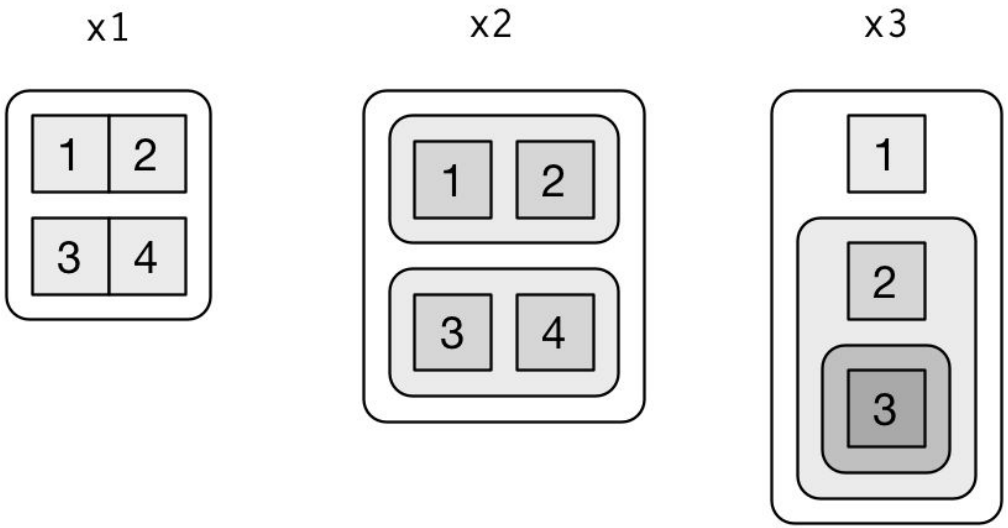


type - `typeof()`

length - `length()`

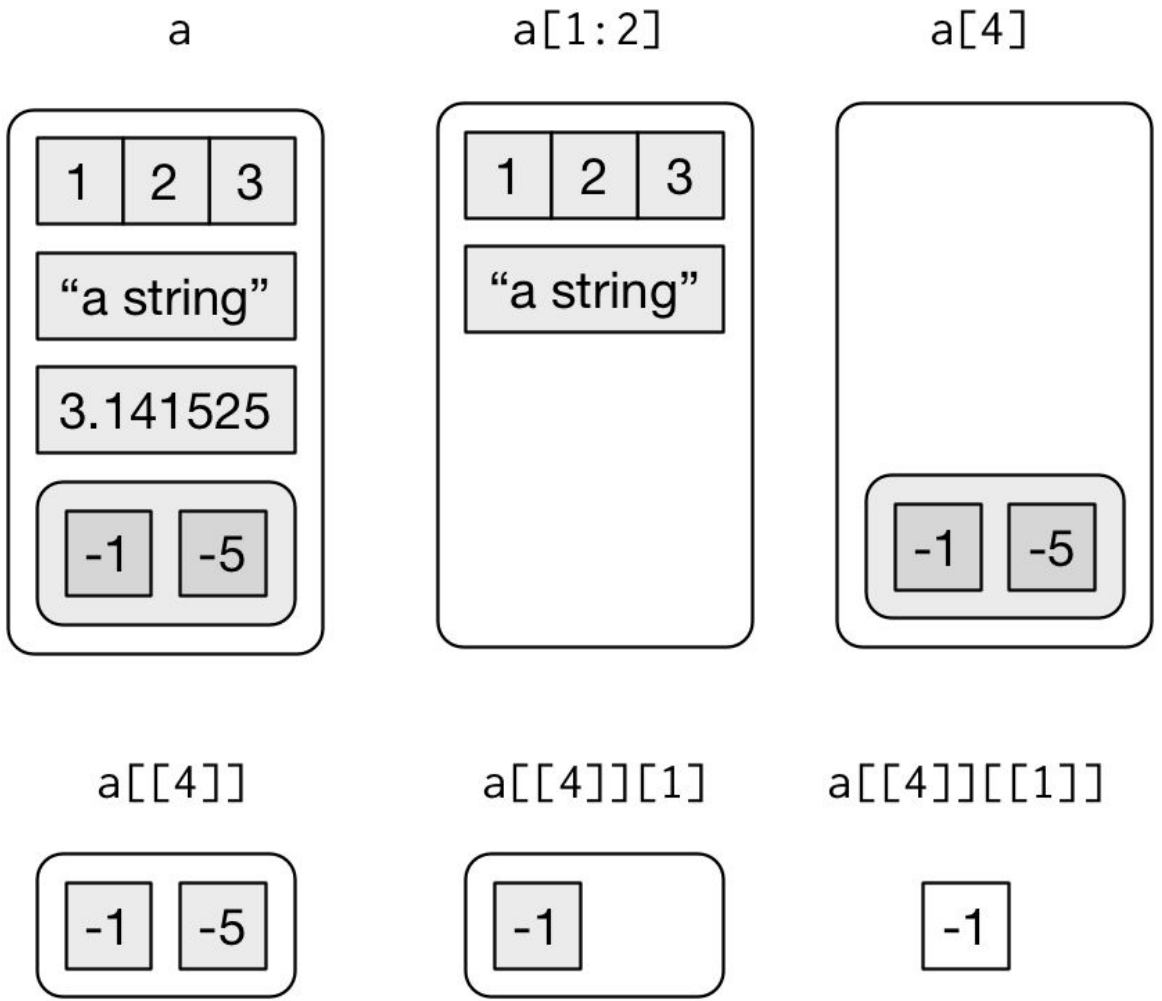
purrr

	lgl	int	dbl	chr	list
`is_logical()`	x				
`is_integer()`		x			
`is_double()`			x		
`is_numeric()`		x	x		
`is_character()`				x	
`is_atomic()`	x	x	x	x	
`is_list()`					x
`is_vector()`	x	x	x	x	x



There are three principles:

1. Lists have rounded corners. Atomic vectors have square corners.
2. Children are drawn inside their parent, and have a slightly darker background to make it easier to see the hierarchy.
3. The orientation of the children (i.e. rows or columns) isn't important, so I'll pick a row or column orientation to either save space or illustrate an important property in the example.


a
1
2
3
“a string”
3.141525
-1
-5
a[1:2]
1
2
3
“a string”
a[4]
-1
-5
a[[4]]
-1
-5
a[[4]][1]
-1
a[[4]][[1]]
-1

Chapter 21 Iteration

for loops

Every for loop has three components:

1. The **output**:
   ```
   output <- vector("double", length(x))
   ```

2. The **sequence**:
   ```
   i in seq_along(df)
   ```

3. The **body**:
   ```
   output[[i]] <- median(df[[i]])
   ```

purrr

`map()` makes a list

`map_lgl()` makes a logical vector

`map_int()` makes an integer vector

`map_dbl()` makes a double vector

`map_chr()` makes a character vector

purrr

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
#>     4     6     8
#> 0.509 0.465 0.423
```

purrr

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

```
models %>%
  map(summary) %>%
  map_dbl(.$r.squared)
#>     4     6     8
#> 0.509 0.465 0.423
```

purrr

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

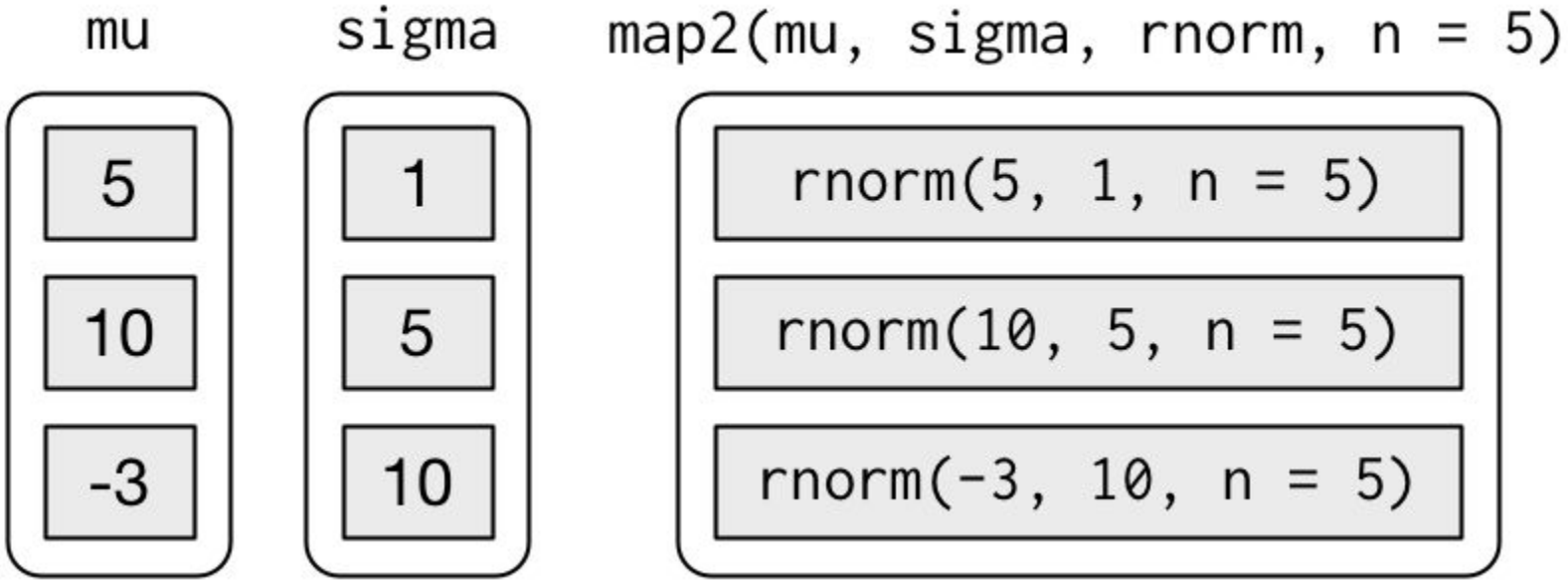
```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
#>     4     6     8
#> 0.509 0.465 0.423
```

purrr

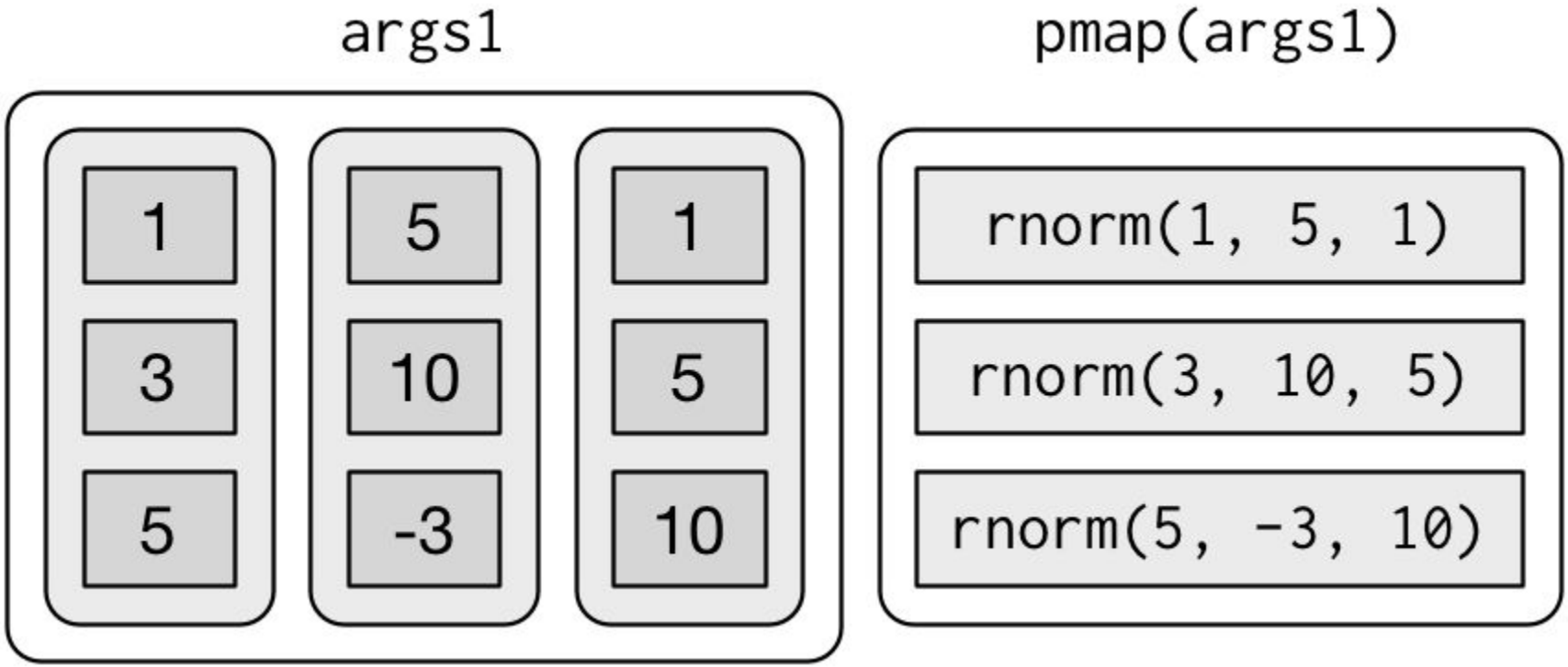
```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
```

```
y <- y %>% transpose()
str(y)
```

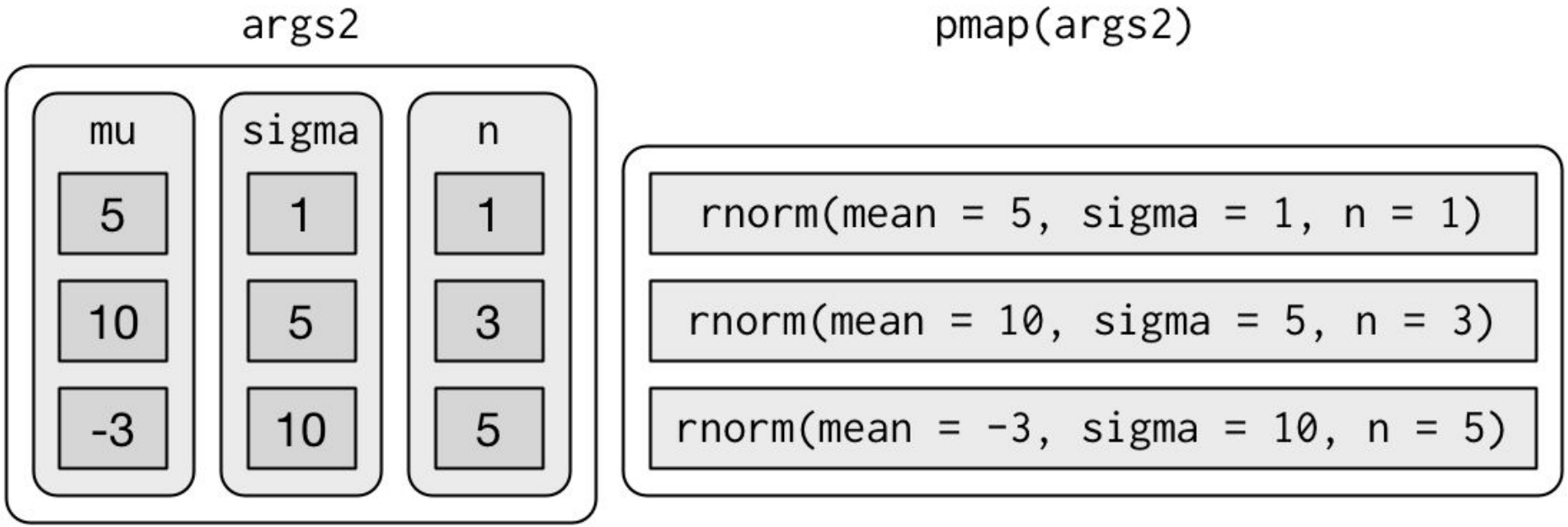
purrr::map2()


mu
sigma
map2(mu, sigma, rnorm, n = 5)
5
10
-3
1
5
10
rnorm(5, 1, n = 5)
rnorm(10, 5, n = 5)
rnorm(-3, 10, n = 5)

purrr::pmap()


args1
1
3
5
5
10
-3
1
5
10
pmap(args1)
rnorm(1, 5, 1)
rnorm(3, 10, 5)
rnorm(5, -3, 10)

purrr::pmap()


args2
mu
5
10
-3
sigma
1
5
10
n
1
3
5
pmap(args2)
rnorm(mean = 5, sigma = 1, n = 1)
rnorm(mean = 10, sigma = 5, n = 3)
rnorm(mean = -3, sigma = 10, n = 5)

purrr::invoke_map()