

Lecture 05: Sep 7, 2018

# Transforming Data

- *Functions*
- *Classes and Objects*
- *Vectorization*
- *Subsets*

James Balamuta  
STAT 385 @ UIUC

# Announcements

- **hw01** is due tonight **Friday, Sep 7th, 2018 @ 6:00 PM**
- **hw02** is will be released *Tonight*
  - Due on **Friday, Sep 14th, 2018 @ 6:00 PM**
- **Quiz 03** covers Week 2 contents @ [CBTF](#).
  - Window: Sep 11th - 13th
  - Sign up: <https://cbtf.engr.illinois.edu/sched>
  - Demo of CBTF Environment: <https://www.youtube.com/watch?v=6oaPvo4TIFk&t=8s>

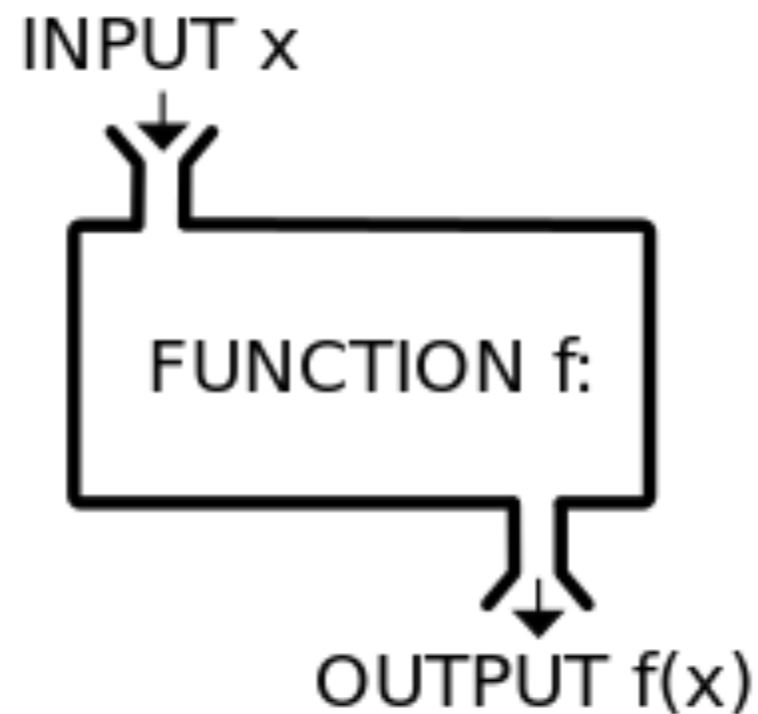
# Lecture Objectives

- **Create** and **call** functions
- **Explain** why functions are useful
- Describe the **difference** between a **class** and **object**.
- Understand the idea behind **vectorization**.
- Apply **recycling** to different vector formulations.
- Generate and use **positional indexes** of a vector.

# Functions

## Definition:

*Functions* are a piece of code that performs a specified task that may or may not depend on parameters and it may or may not return one or more values.



[Source](#)

# Addition Function

... adding values together ...

## Function Name

Actual name of the function that can be called e.g. `add(1, 2)`

## Parameters

Variables that receive expressions that can be used in the function's body

```
add = function(a, b) {
```

## Body

Statements in between `{}` that are run when the function is called

```
    summed = a + b
```

```
    return ( summed )
```

```
}
```

## Return Value

Result made available from running the body statements

```
# Calling the function
```

```
add(1, 2)
```

```
# [1] 3
```

# Motivation

... repeated use of a definition ...

Consider:

```
message("Hello World!")
```

Translates to: **Say "Hello World!" in the console**

What if we wanted to  
**repeat the phrase**  
elsewhere?

# Options

... related to recreating a piece of code ...

1. Retype
2. Copy and paste

## 3. Writing a function

```
say_hello_world = function() {  
  message("Hello World!")  
}
```

```
say_hello_world()  
# Hello World!
```

# Functions are **Powerful**

... expressive pieces of code ...

1

- The *logic flow* is chunked instead of a series of long statements

2

- Decrease the probability of an error by applying a *common definition*

3

- Easier to share code with people that you collaborate with.

# Clarity of Routine

## What's happening here?

```
set.seed(1115)
```

```
sample(6, size = 1)  
# [1] 3
```

```
sample(6, size = 1)  
# [1] 5
```

```
sample(6, size = 1)  
# [1] 5
```

```
# Perform a single dice roll
roll_die = function(num_sides) {

  sample(num_sides, size = 1)

}
```

```
# Set seed for reproducibility
set.seed(1115)
```

```
# Call function with argument
roll_die(6)
# [1] 3
```

# Die Roll

... Adding Documentation via Code ...



[Source](#)

# Argument vs. Parameter

... when to say which ...

Parameter



```
roll_die = function(num_sides) {  
  sample(num_sides, size = 1)  
}
```

```
roll_die(6)
```



Argument

# Default Parameters

... being proactive ...

## Function Name

Actual name of the function that can be called  
e.g. `roll_die()`

## Parameters

Variables that receive a specific data type that can be used in the function's body

## Default Values

The initial values used if the parameters are not supplied on function call

```
roll_die = function(num_sides = 6) {
```

## Body

Statements in between `{}` that are run when the function is called

```
    roll = sample(num_sides, size = 1)  
    return(roll)
```

## Return Value

Result made available from running the body statements

```
}
```

```
set.seed(1115)
```

```
roll_die()
```

```
# [1] 3
```

# Anticipate need of user

```
set.seed(1115)
```

```
roll_die(25)
```

```
# [1] 11
```

# Allow function to still be dynamic

# Generalizing

... enabling multiple rolls ...

## Function Name

Actual name of the function that can be called e.g.  
`roll_n_die(2, 10)`

## Positional Parameter

Variables that must be specified on function call in the order that they appear.

## Default Parameter

Variables that will use prespecified values if not supplied on function call

## Default Values

The initial values used if the parameters are not supplied on function call

```
roll_n_die = function(num_rolls, num_sides = 6) {
```

## Body

Statements in between {} that are run when the function is called

```
  rolls = sample(num_sides, size = num_rolls,  
                 replace = TRUE)  
  return(rolls)
```

## Return Value

Result made available from running the body statements

```
}
```

```
set.seed(1115)
```

```
roll_n_die(3, 10)
```

```
# [1] 3 5 5
```

```
# Roll the die three times
```

# Your Turn

Clean up the following code by implementing a function that:

1. Generates data from a normal distribution
2. Applies the mean normalization  $x' = \frac{x - \bar{x}}{\max(x) - \min(x)}$

```
set.seed(325)
```

```
x = rnorm(10)
```

```
y = rnorm(10)
```

```
x_nmu = (x - mean(x)) / (max(x) - min(x))
```

```
y_nmu = (y - mean(y)) / (max(y) - min(y))
```

# Classes and Objects

On today's agenda

“Everything that *exists* in *R* is an **Object**.  
Everything that *happens* in *R* is a **Function Call**.  
**Interfaces** to other software are part of *R*”

—John M. Chambers, Extending R (2016) pg. 4

When we talk about  
high performance computing

When we talk  
about functions

Previously

# Vectors

... **1 Dimensional collections** of the **same** kind of **element** ...

```
# Vector of numeric elements
```

```
w = c(9.5, -3.14, 88.9999, 12.0)
```

```
# ^ ^ ^ ^ decimals
```

```
# Vector of integer elements
```

```
x = c(1L, 2L, 3L, 4L)
```

```
# Vector of logical elements
```

```
y = c(TRUE, FALSE, FALSE, TRUE)
```

```
# Vector of character elements
```

```
z = c("a", "b", "c", "d")
```

Previously

# Data Frame

\*

... constructing by hand ...

```
subject_heights = data.frame(id      = c(1, 2, 3, 55),  
                               sex     = c("M", "F", "F", "M"),  
                               height  = c(6.1, 5.5, 5.2, 5.9))
```

id	sex	height
1	M	6.1
2	F	5.5
3	F	5.2
55	M	5.9



**Vectors**

---

\* The "... " row was removed when constructing the data.frame by hand as these observations were *hidden* from the diagram representation.

# Representations

... transcribing mental models ...

**subject\_heights**

id	sex	height
1	M	6.1
2	F	5.5
3	F	5.2
...	...	...
55	M	5.9

Our idea of tabular data

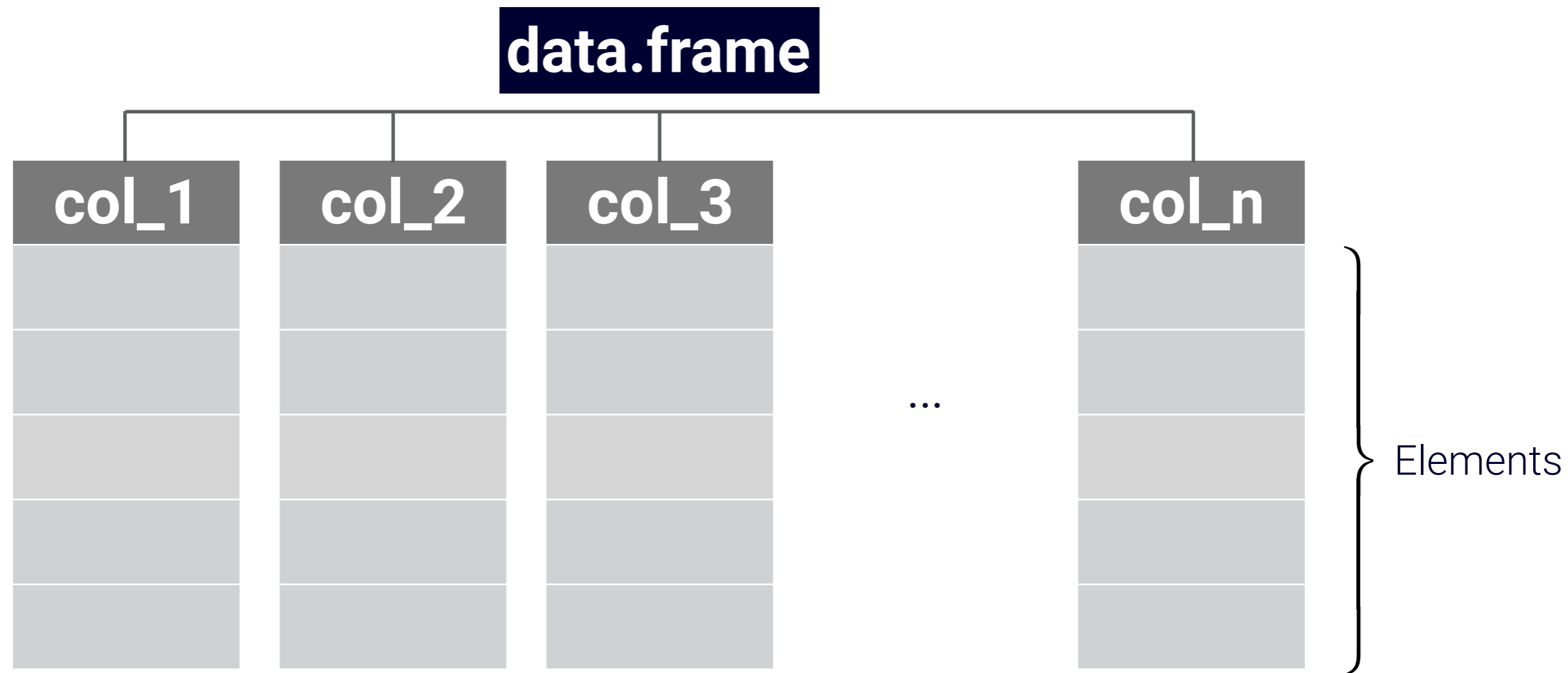
**subject\_heights**

id	sex	height
1	M	6.1
2	F	5.5
3	F	5.2
...	...	...
55	M	5.9

*R*'s idea of tabular data  
Collection of Vectors

# Classes

... **blueprints** for creating *objects* ...



```
my_data = data.frame(col_1 = c( ... ),  
                      col_2 = c( ... ),  
                      col_3 = c( ... ),  
                      ...  
                      col_n = c( ... ))
```

**Function Call  
to create class**


# Objects

... **instances** of a *class* ...

subject_heights		
id	sex	height
1	M	6.1
2	F	5.5
3	F	5.2
...	...	...
55	M	5.9

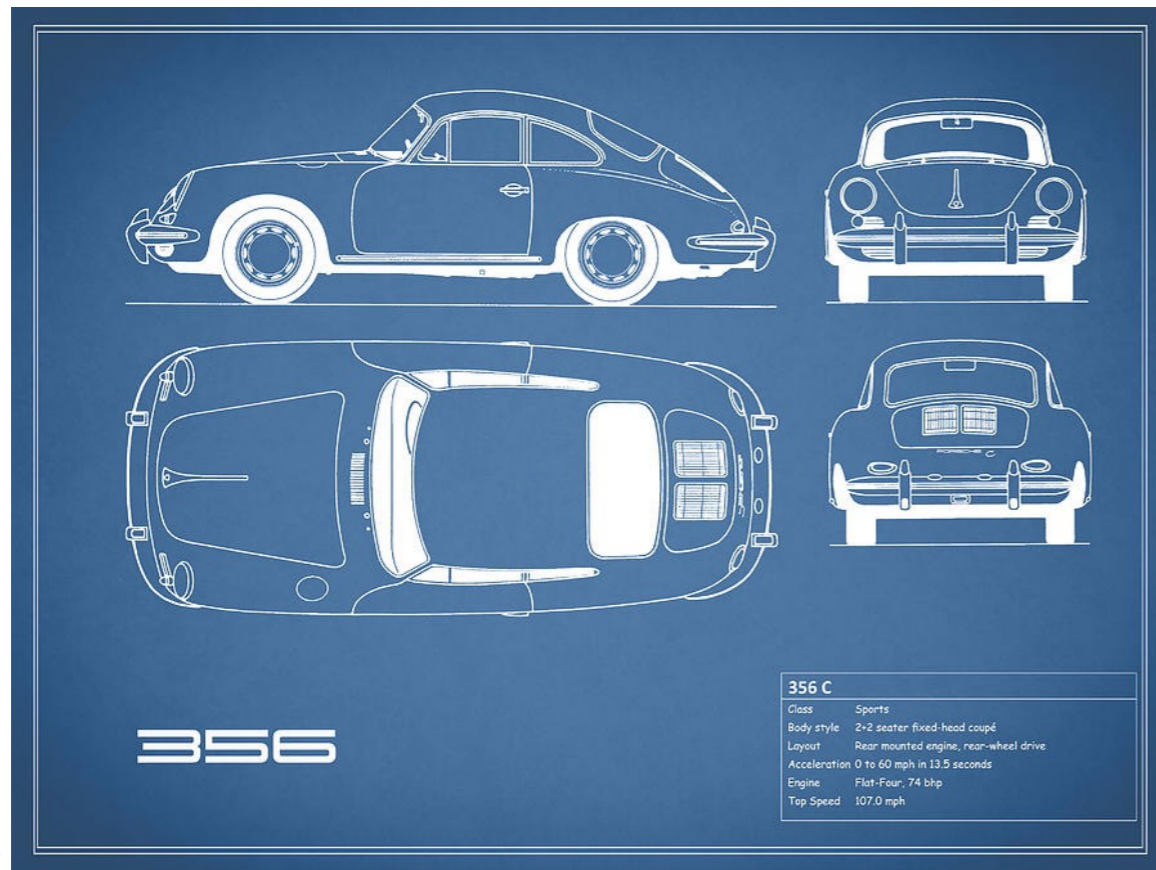
} Items

```
subject_heights = data.frame(id      = c(1, 2, 3, 55),  
                                sex      = c("M", "F", "F", "M"),  
                                height = c(6.1, 5.5, 5.2, 5.9))
```

 **Object**

# Class to Object

... **blueprints** to **instances** ...



[Source](#)



[Source](#)



[Source](#)



[Source](#)

## Objects Have....

### 1. **Class**

"classification"

### 2. **Structure**

"blueprint layout"

```
# Example data set
subject_heights = data.frame(
  id      = c(1, 2, 3, 55),
  sex     = c("M", "F", "F", "M"),
  height  = c(6.1, 5.5, 5.2, 5.9)
)
```

```
class(subject_heights)
```

```
# [1] "data.frame"
```

```
str(subject_heights)
```

```
# 'data.frame': 4 obs. of 3 variables:
```

```
# $ id : num 1 2 3 55
```

```
# $ sex: Factor w/ 2 levels "F","M": 2 1  
1 2
```

```
# $ height: num 6.1 5.5 5.2 5.9
```

# What is the class and structure of ... ?

## twtr\_stock\_prices

time	price
09:30 AM	22.40
09:40 AM	22.38
09:50 AM	22.46
10:00 AM	22.74

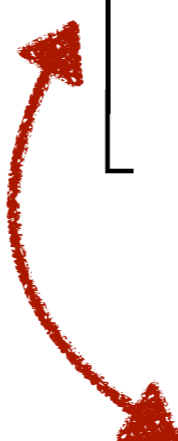
```
id = c(1, 2, 3, 55)
```

# Vectorization

## Definition:

*Vectorization* refers to performing work on multiple items at the same time.

$$f\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_{n \times 1}\right) = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}_{n \times 1}$$


$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}_{4 \times 1} + \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}_{4 \times 1} = \begin{bmatrix} 6 \\ 8 \\ 10 \\ 12 \end{bmatrix}_{4 \times 1}$$

1	+	5	=	6
2	+	6	=	8
3	+	7	=	10
4	+	8	=	12

# Element-wise Mathematics

... adding together individual elements...

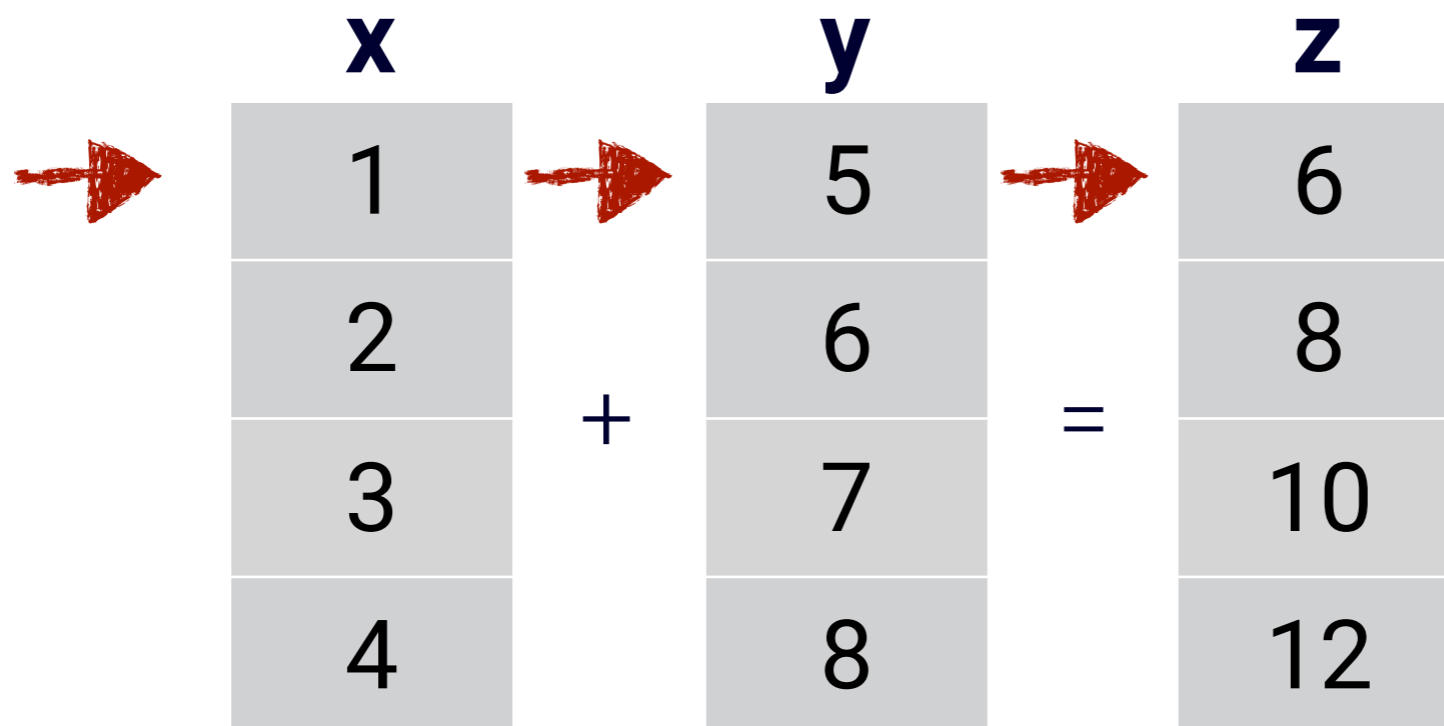
```
x = c(1, 2, 3, 4)
```

```
y = c(5, 6, 7, 8)
```

```
z = x + y
```

```
z
```

```
# [1] 6 8 10 12
```



# Binary Vectorized Ops

... **element**-wise changes...

```
x = c(1, 2, 3, 4); y = c(5, 6, 7, 8)
```

```
x + y
```

# Addition

```
# [1] 6 8 10 12
```

```
x - y
```

# Subtraction

```
# [1] -4 -4 -4 -4
```

```
x * y
```

# Multiplication

```
# [1] 5 12 21 32
```

```
x / y
```

# Division

```
# [1] 0.20 0.333 0.429 0.500
```

```
x ^ y
```

# Exponentiation

```
# [1] 1 32 729 65536
```

```
x %/% y
```

# Integer Division

```
# [1] 0 0 0 0
```

```
x %% y
```

# Modulus

```
# [1] 1 2 3 4
```

# Modulus Operator

... a mod q ...

12 %% 7  
# [1] 5

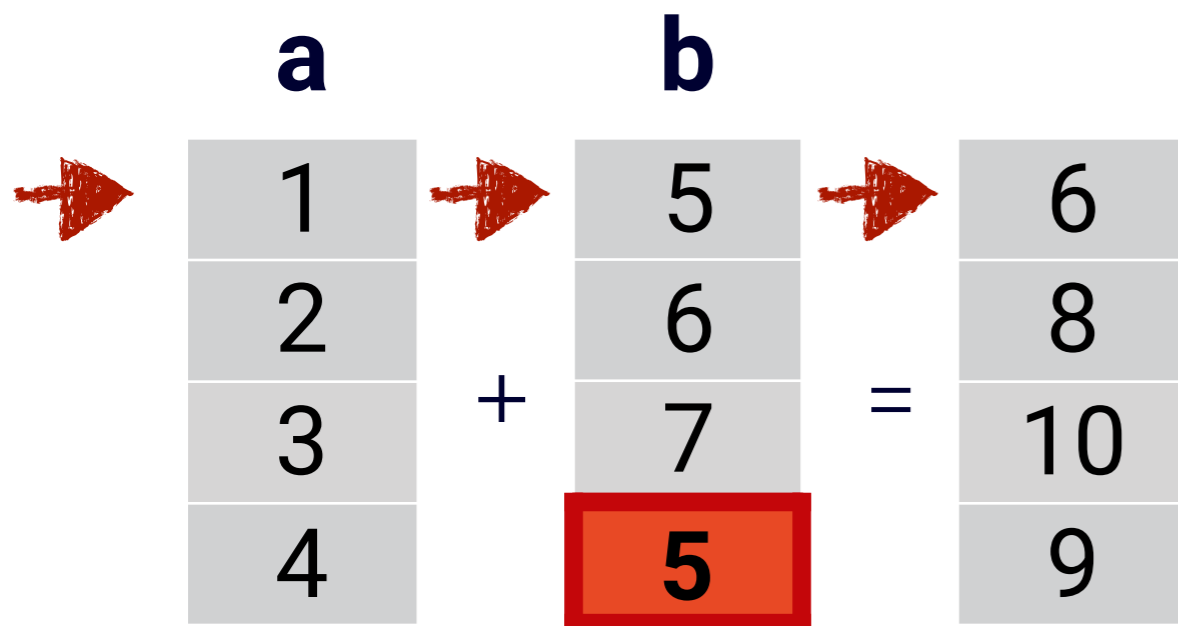
$$\# a = n * q + r \Rightarrow 12 = 1 * 7 + 5$$

outer(9:1, 2:9, `%%`) # Compute the cross between X & Y

	mod 2 <int>	mod 3 <int>	mod 4 <int>	mod 5 <int>	mod 6 <int>	mod 7 <int>	mod 8 <int>	mod 9 <int>
x = 1	1	1	1	1	1	1	1	1
x = 2	0	2	2	2	2	2	2	2
x = 3	1	0	3	3	3	3	3	3
x = 4	0	1	0	4	4	4	4	4
x = 5	1	2	1	0	5	5	5	5
x = 6	0	0	2	1	0	6	6	6
x = 7	1	1	3	2	1	0	7	7
x = 8	0	2	0	3	2	1	0	8
x = 9	1	0	1	4	3	2	1	0

# Recycling

... *R* cares ...



```
# Create two vectors with  
# differing number of elements
```

```
a = c(1, 2, 3, 4)
```

```
b = c(5, 6, 7)
```

```
# Count elements
```

```
length(a)
```

```
# [1] 4
```

```
length(b)
```

```
# [1] 3
```

```
# Compute element-wise
```

```
# addition
```

```
res = a + b
```

```
# Look at result
```

```
res
```

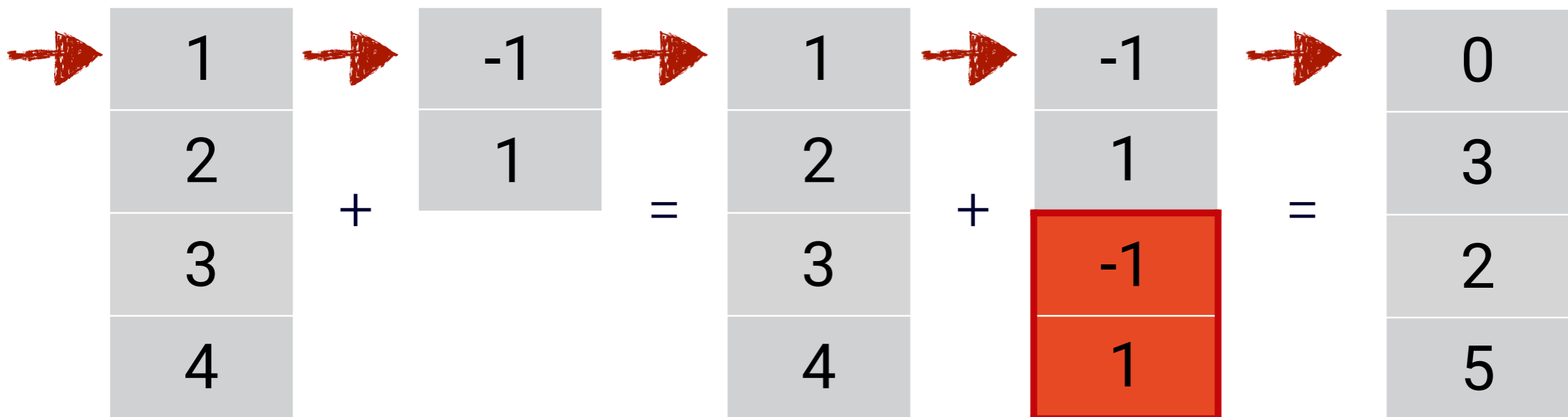
```
# [1] 6 8 10 9
```

# Rules to Recycle

... guidelines of *R* ...

1. **Determine** if a vector is *shorter* than the other.
2. **Recycling** (or an **expansion**) of the *shorter* vector until it matches the longer's length.
3. **Application** of a binary operation (e.g +, -, \*, /, %, and %%)

```
c(1, 2, 3, 4) + c(-1, 1)  
# [1] 0 3 2 5
```



# Your Turn

Explain what happens if we have a vector and add a single value

```
a = 2  
x = c(1, 2, 3, 4)  
x + a
```

How could we harness recycling to compute a confidence interval in one line?

$$\hat{p} \pm z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} = \left[ \hat{p} - z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \hat{p} + z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \right]$$

Plus or Minus

# Scalars are Vectors

... single values are disguised vectors ...

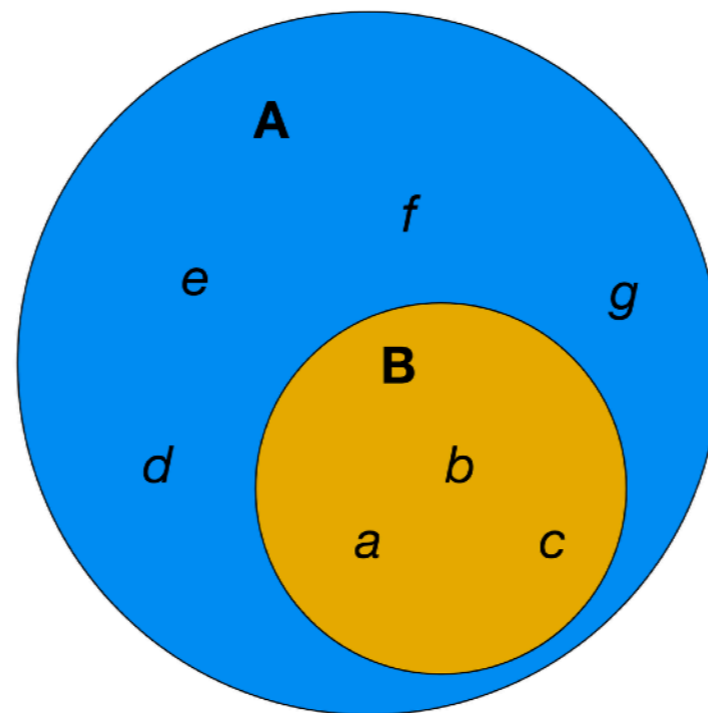
```
a = 2  
length(a)  
# [1] 1
```

```
a_vec = c(2)  
length(a_vec)  
# [1] 1
```

# Subsets

## Definition:

*Subsets* are a way to take a selection of values in a larger collection and create a smaller collection with them.



$$B \subset A$$
$$\{a, b, c\} \subset \{a, b, c, d, e, f, g\}$$

# Brackets [ ] Subset

... for elements in a vector ...

Object to subset

Brackets to subset

```
my_vec[????]
```

Value

Previously

# Four Ways to Subset

**empty**

include *all* values

**integer**

**+num**: include value(s)

**0**: remove *all* values

**-num**: remove value(s)

**logical**

**TRUE**: include value(s)

**FALSE**: remove value(s)

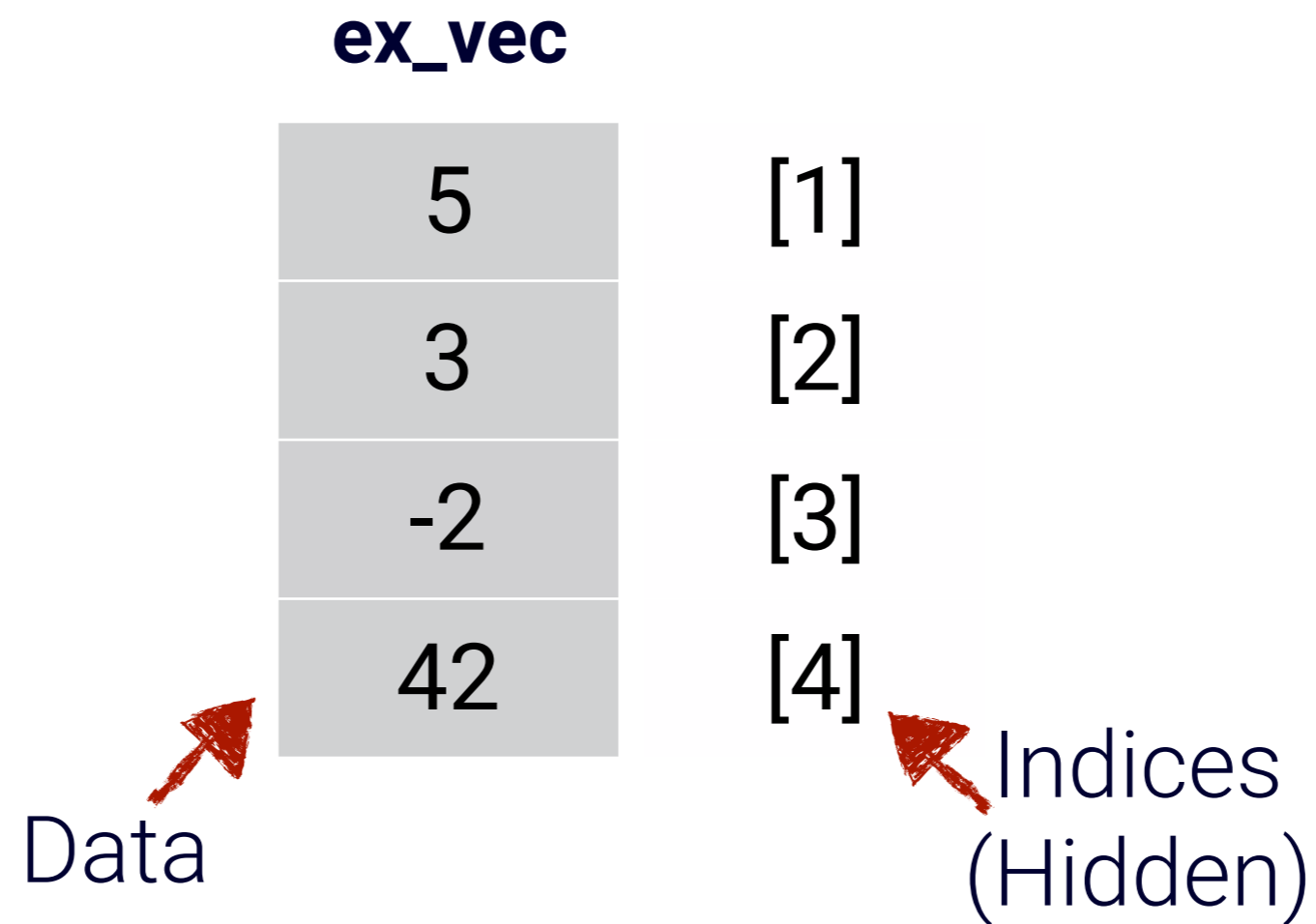
**character**

**"name"**: include value(s)

# Positional Indexes

... the location of the data in the vector ...

```
ex_vec = c(5, 3, -2, 42)
```



---

\* *One-based indexing* means the counting of elements starts at **1** and goes to **n**.  
*Zero-based indexing* means the counting of elements begins at **0** and ends at **n-1**.

# Create example vector

```
ex_vec = c(5, 3, -2, 42)
```

# Get value in **first** position

```
ex_vec[1]
```

```
# [1] 5
```

# Get value in **fourth** position

```
ex_vec[4]
```

```
# [1] 42
```

# Find length of vector

```
last_pos = length(ex_vec)
```

# Get last value vector in vector

```
ex_vec[last_pos]
```

```
# [1] 42
```



# Location of result vector

## Retrieve Single Value

**ex\_vec**

5	<b>[1]</b>
3	[2]
-2	[3]
42	[4]

**ex\_vec**

5	[1]
3	[2]
-2	[3]
42	<b>[4]</b>

```
# Create example vector  
ex_vec = c(5, 3, -2, 42)
```

```
# Get values in  
# second and third position  
ex_vec[c(2, 3)]  
# [1] 3 -2
```

```
# Retrieve values in  
# second and third position  
# with colon operator  
ex_vec[2:3]  
# [1] 3 -2
```

```
# Colon operator creates a  
# sequence from:to  
# e.g. 2:3 is c(2, 3)
```

## Retrieving Multiple Values

ex_vec	
5	[1]
3	[2]
-2	[3]
42	[4]

## Retrieving Multiple Values by Removing Positions

```
# Create example vector
```

```
ex_vec = c(5, 3, -2, 42)
```

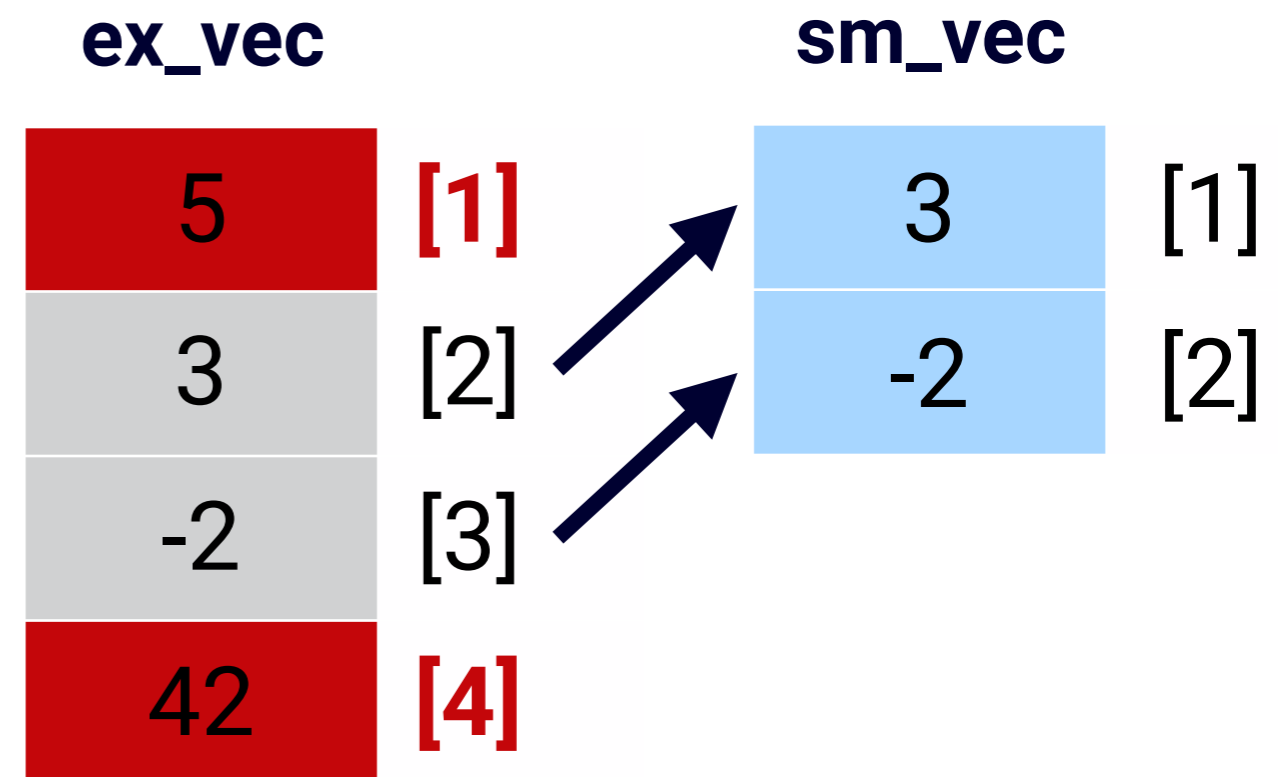
```
# Get values not in the  
# first or fourth position
```

```
sm_vec = ex_vec[c(-1, -4)]
```

```
# Display short vector
```

```
sm_vec
```

```
# [1] 3 -2
```



```
# Create example vector  
ex_vec = c(5, 3, -2, 42)
```

```
# Count elements in vector  
length(ex_vec)  
# [1] 4
```

```
# Select past vector length  
ex_vec[5]  
# [1] NA
```

```
# Out of Bounds occurs  
# when position > length(vec)  
# End up with missingness
```

# Out of Bounds

... selects **missingness** ...

	ex_vec
[1]	5
[2]	3
[3]	-2
[4]	42
[5]	

Indices

Data

How can we  
**generate indices explicitly?**

# Generating Sequences

... types of sequences ...

```
ex_vec = c(5, 3, -2, 42)
```

```
1:length(ex_vec)
```

```
# [1] 1 2 3 4
```

```
seq(1, length(ex_vec))
```

```
# [1] 1 2 3 4
```

```
seq_len(length(ex_vec))
```

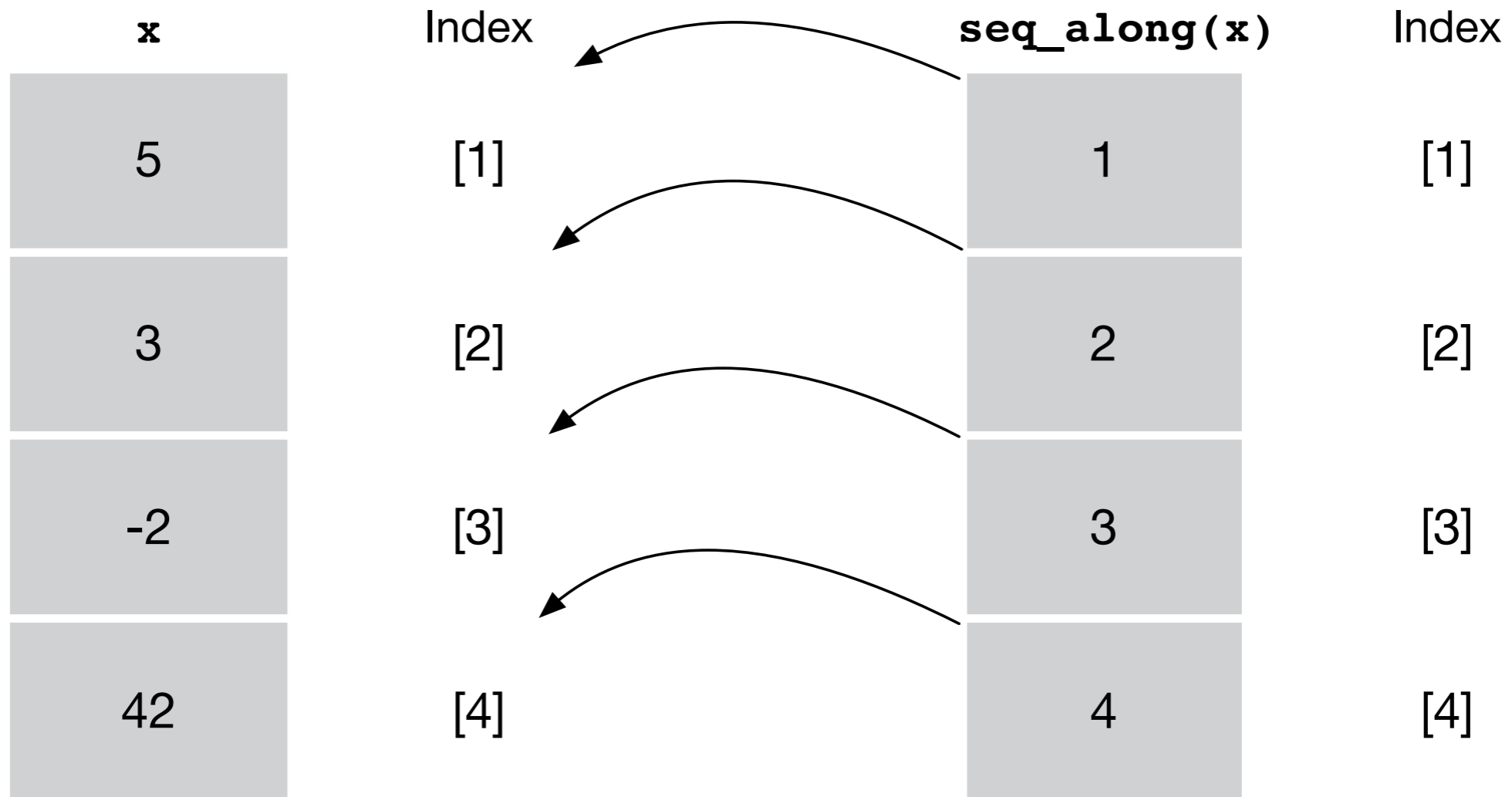
```
# [1] 1 2 3 4
```

```
seq_along(ex_vec)
```

```
# [1] 1 2 3 4
```

# seq\_along

... what happens ...



# Your Turn

Using **all** sequence methods, create sequences for the following vectors. Are all approaches the same?

```
int_vec = c(8L, -2L, 5L, 0L)  
empty_vec = numeric(0)
```

# Acknowledgements

# Recap

- **Functions**

- Functions allow for dynamic input and output
- Provide a way to standardize and share code

- **Classes and Objects**

- Each object in  $R$  has a class

- **Vectorization**

- Elements are always represented in a collection
- Expressive statements that allow operations on the object

- **Subsets**

- Retrieving smaller quantities

# Acknowledgements

- [Bob Rudis](#)' minimalist gestalt C++ function diagram

This work is licensed under the  
Creative Commons  
Attribution-NonCommercial-  
ShareAlike 4.0 International  
License

