

Automation

What to automate?

- Replacing manual copy/paste/renaming with data-cleaning scripts
- Replacing drop-down menu analyses with scripts
- *Replacing redundant code with more efficient code*

Types of code to avoid writing

- Hard coding
 - `ds[1, 1]`
 - `proportion <- ds$counts / 5365`

Types of code to avoid writing

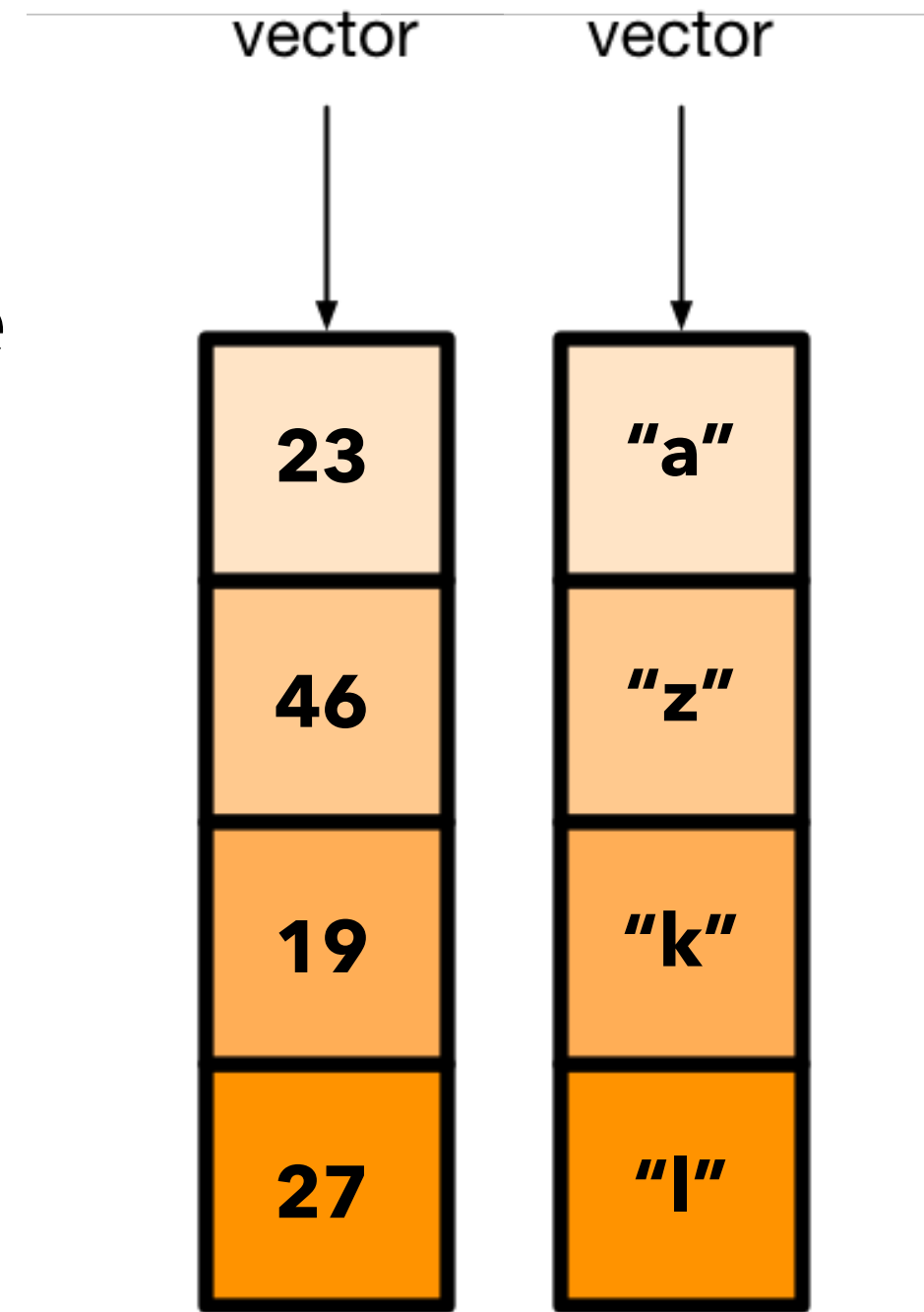
- Hard coding
- Repetitive coding
 - `summarize(m_a = mean(a), m_b = mean(b), m_c = mean(c), m_d = mean(d))`
 - `ds_a <- read_csv("data_a.csv")`
 - `ds_b <- read_csv("data_b.csv")`
 - `ds_c <- read_csv("data_b.csv")`
 - Copy-paste-tweak leads to coding mistakes!

Tools to avoid repetitive coding

- `select(col1:col50)`
- `rename_with(make_clean_names)`
- `mutate(across(selected_vars), list(fxs))`
- `summarize(across(selected_vars), list(fxs))`
- `vroom(list_of_files)`
- What do they all have in common?

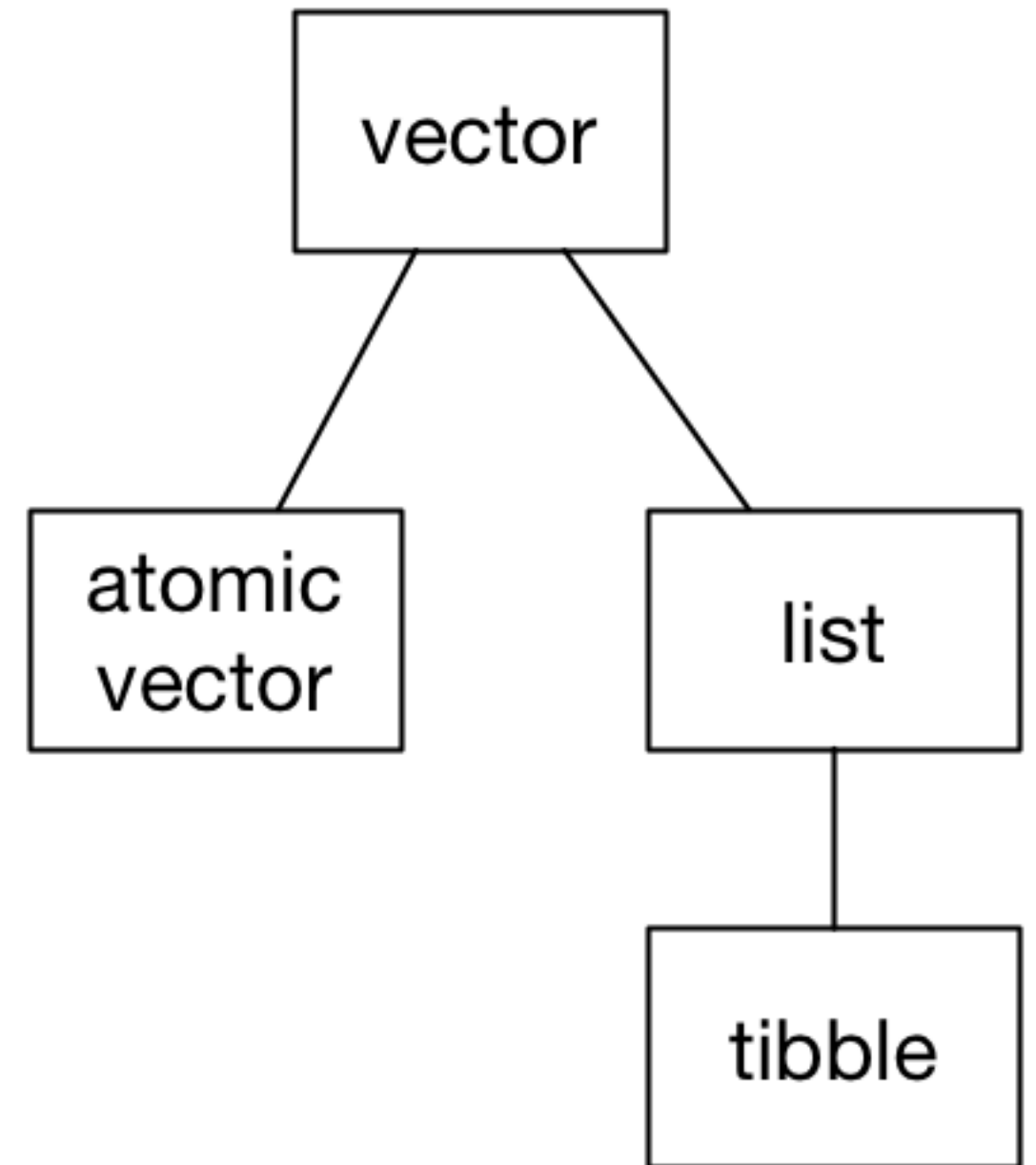
They all operate across **vectors**

- Vectors are collections of values
 - *atomic vectors* contain only a single type
 - numeric vector
 - logical vector
 - character vector
- `v[i]` accesses an element by position
 - `v[1] = 23`
- create vectors with `v <- c(23, 46, 19, 27)`



Vectors vs. lists vs. tibbles

- List: a mixed-type vector
 - `x <- list("c", 1, TRUE)`
 - Lists can contain lists, whereas vectors are unnested
- Tibbles are lists of vectors
 - All vectors are the same length
 - Each vector has a different type



Named elements help us get away from hard coding

No names

Access by position only

```
> x <- c("X", "Y", "Z")
> x
[1] "X" "Y" "Z"
> x[1]
[1] "X"
```

With names

Access by position or name

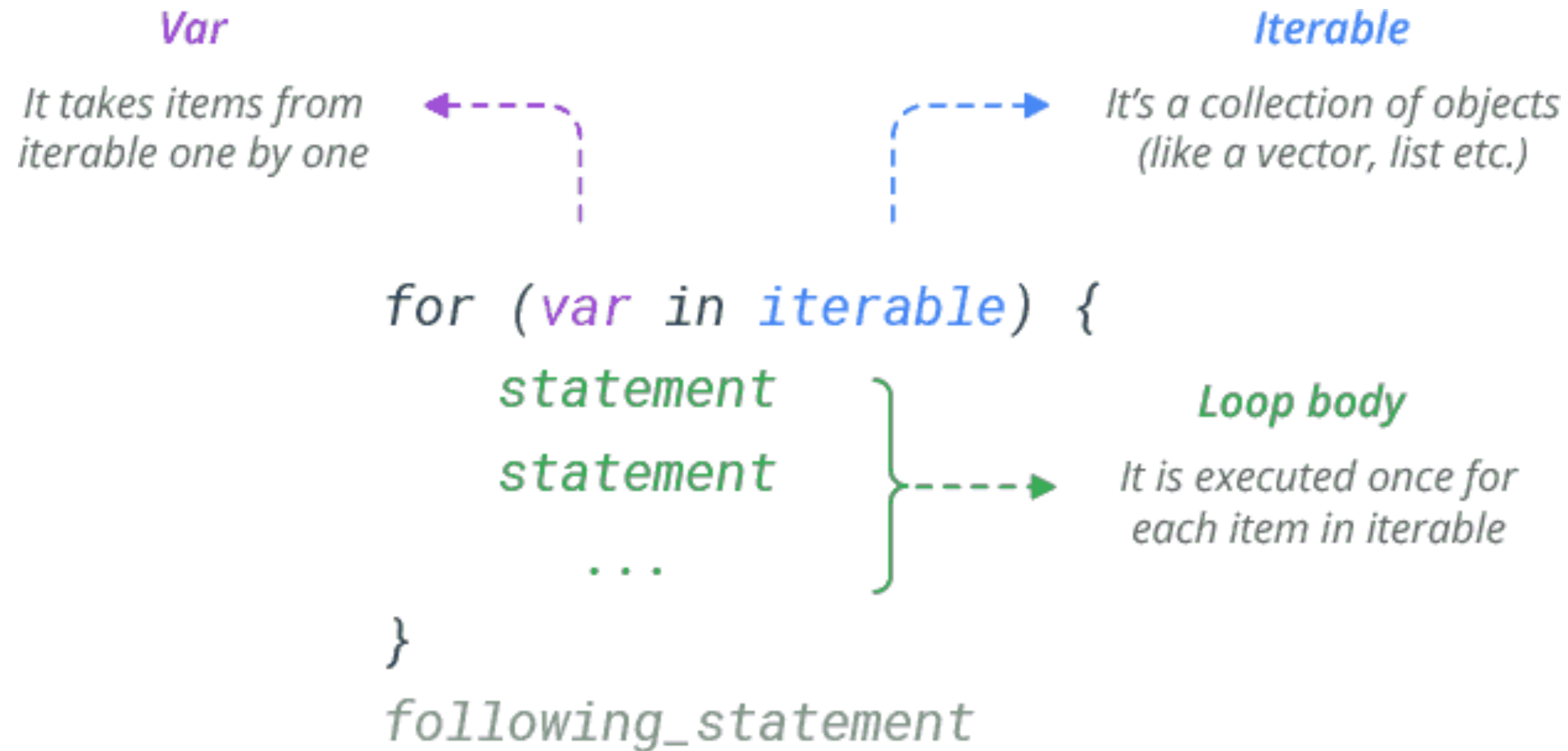
```
> y <- c(first = "X", second = "Y", third = "Z")
> y
  first second  third
    "X"    "Y"    "Z"
> y["second"]
second
    "Y"
> names(y)
[1] "first" "second" "third"
> set_names(y, c("a", "b", "c"))
  a  b  c
"X" "Y" "Z"
```


Iterating over vectors/lists

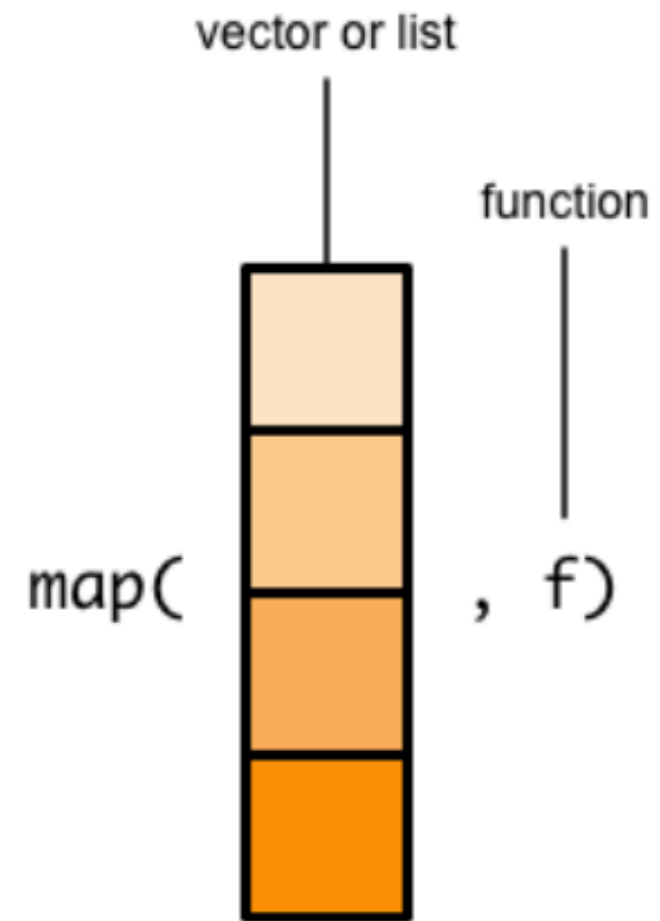
Three main options to iterate/loop

- Loop -> perform a set of actions for each element of a vector/list
 - for loops
 - map (tidyverse *purrr* package)
 - lapply (base R, like map but harder to use)

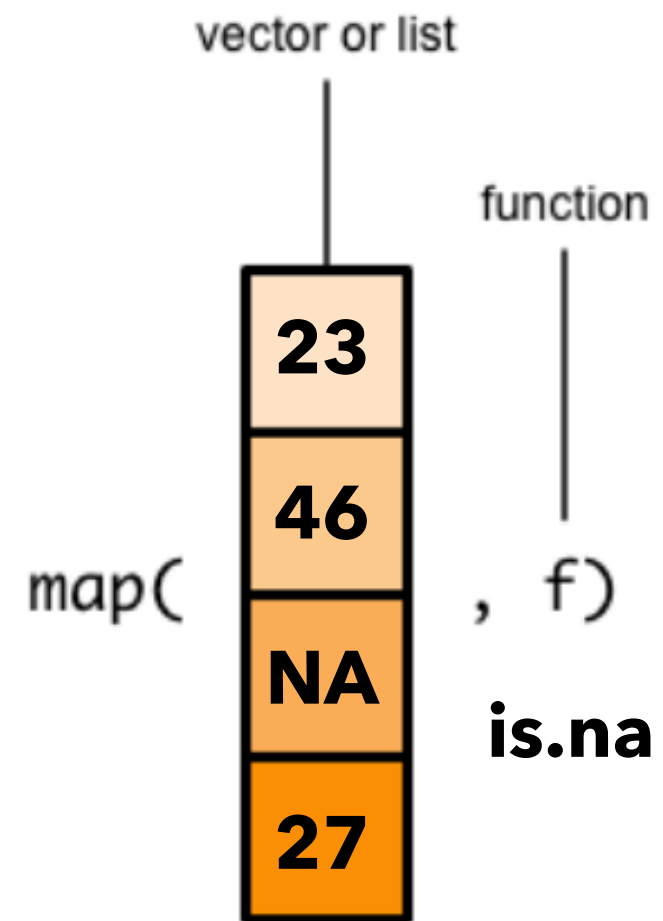
For loops



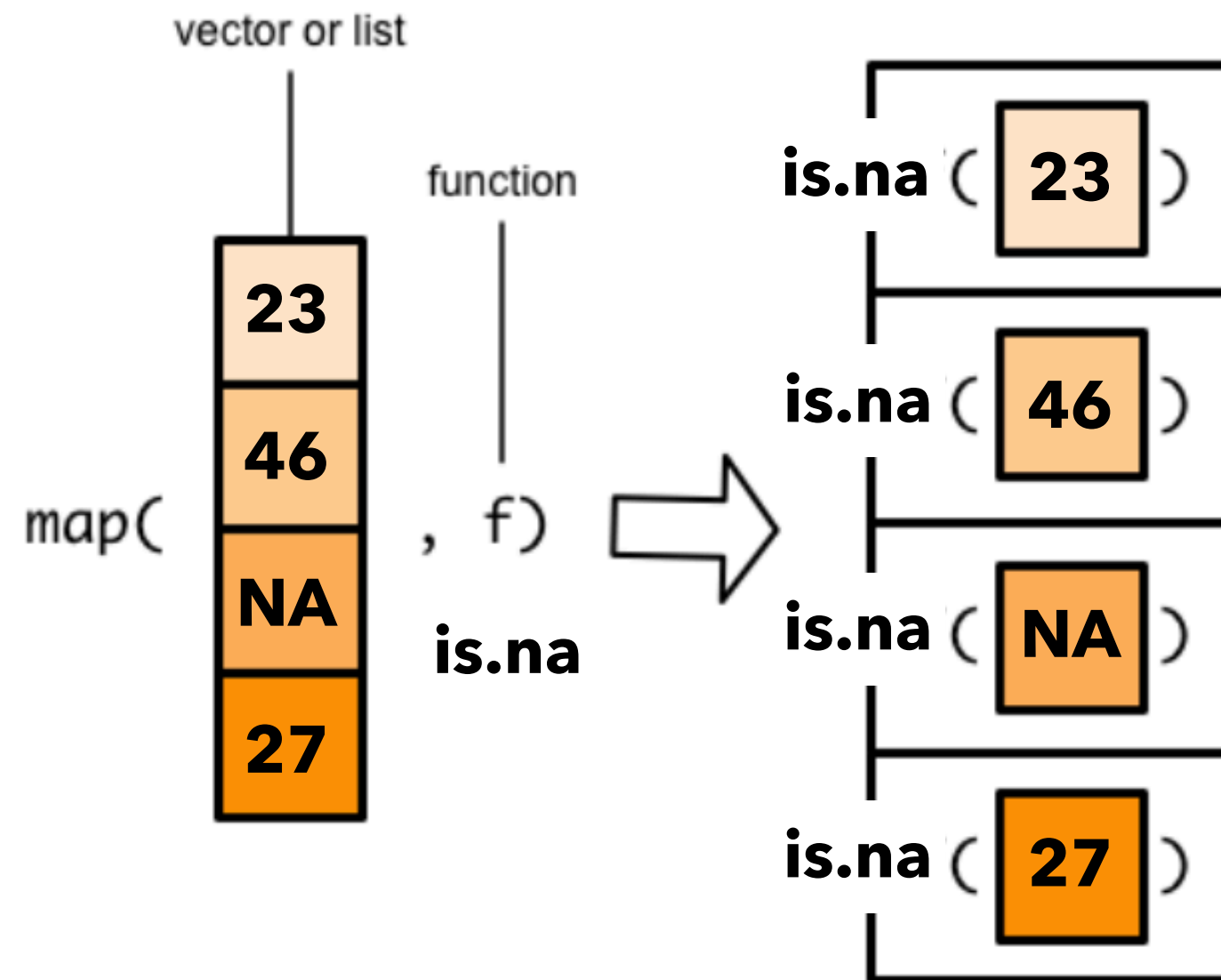
map



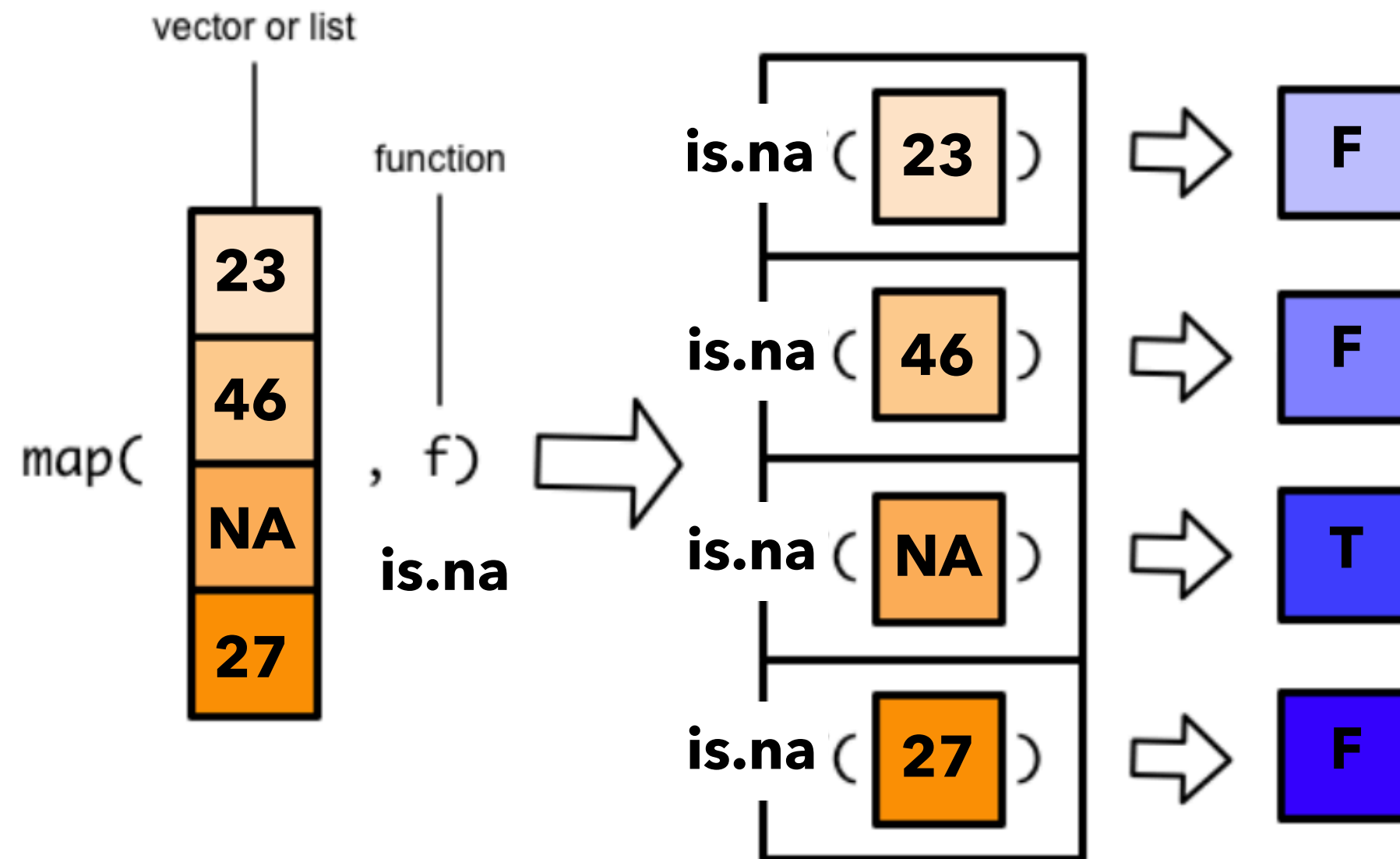
map



map



map



Different map outputs

- map returns a list
- map_lgl returns a vector of logicals
- map_chr returns a vector of characters
- vector/list names get carried through

```
> v <- c(23, 46, NA, 27)
> map(v, is.na)
[[1]]
[1] FALSE

[[2]]
[1] FALSE

[[3]]
[1] TRUE

[[4]]
[1] FALSE
```

```
> v <- c(23, 46, NA, 27)
> map_lgl(v, is.na)
[1] FALSE FALSE TRUE FALSE
> map_chr(v, is.na)
[1] "FALSE" "FALSE" "TRUE"  "FALSE"
```

```
> v <- c(w = 23, x = 46, y = NA, z = 27)
> map_chr(v, is.na)
      w      x      y      z
"FALSE" "FALSE" "TRUE" "FALSE"
```


Other considerations

- Write code from the inside out
 - Test the basic function(s) with a single unit of data to make sure it works
 - Then, revise it to iterate over a list/vector
- Writing custom functions can make map easier (more on this next week)

Automation tutorial

"259-automation"

What are functions?

Functions

- A series of code statements designed to accomplish a particular task
- Functions take input(s) and return output(s)
- `mean(x, na.RM = FALSE)`
 - fx name = mean
 - fx arguments = x, na.RM
 - fx body = code used to calculate the mean
 - fx output = the mean value that is returned

Defining your "hyp" function

a name you assign




```
hyp <- function(side_a, side_b) {  
  a <- side_a^2  
  b <- side_b^2  
  h <- sqrt(a + b)  
  return(h)  
}
```

Defining your "hyp" function

a name you assign

argument names you assign



```
hyp <- function(side_a, side_b) {  
  a <- side_a^2  
  b <- side_b^2  
  h <- sqrt(a + b)  
  return(h)  
}
```

Defining your "hyp" function

a name you assign

argument names you assign

code body

```
hyp <- function(side_a, side_b) {  
  a <- side_a^2  
  b <- side_b^2  
  h <- sqrt(a + b)  
  return(h)  
}
```

Defining your "hyp" function

a name you assign

argument names you assign

code body

```
hyp <- function(side_a, side_b) {  
  a <- side_a^2  
  b <- side_b^2  
  h <- sqrt(a + b)  
  return(h)  
}
```

define the output value

Defining your "hyp" function

a name you assign

argument names you assign

code body

```
hyp <- function(side_a, side_b) {  
  a <- side_a^2  
  b <- side_b^2  
  h <- sqrt(a + b)  
  return(h)  
}
```

define the output value

```
> hyp(3,4)  
[1] 5
```

use like any other function

Why do we write functions?

- Save time/typing
 - We could all write code to calculate a mean, but imagine doing that every time we needed to

```
> x <- c(1, 2, 3, 4, 5)
> mean_x <- sum(x)/length(x)
> y <- c(3, 4, 5, 6, 7)
> mean_y <- sum(y)/length(y)
> z <- c(10, 11, 12, 13, 14, 15)
> mean_z <- sum(z)/length(z)
> mean_x
[1] 3
> mean_y
[1] 5
> mean_z
[1] 12.5

> vector_mean <- function(v) sum(v)/length(v)
> vector_mean(x)
[1] 3
> vector_mean(y)
[1] 5
> vector_mean(z)
[1] 12.5
```

Why do we write functions?

- Save time/typing
 - We could all write code to calculate a mean, but imagine doing that every time we needed to
- Encapsulate computation
 - Functions execute in their own *environment*
 - We just want the output, not all of the intermediary steps of the computation
- Introduce abstraction
 - Good functions are general-purpose tools

Packages extend R functionality with functions

- base package = functions for data types, basic math, and other generic functions
- other packages = sets of functions that extend the base R language
 - May be written using base R functions or a mix of base R and functions from other packages
 - dplyr depends on base R but also on other packages (such as tibble and tidyselect)

Packages are just a set of function definitions

- calling “arrange” before `library(dplyr)` won’t work, because `arrange` is defined by `dplyr`
- calling “vector_mean” before assigning a function to `vector_mean` won’t work for the same reason
- Loading a package with `library()` is just a shortcut for defining lots of functions

What's special about packages?

- The short answer: nothing
 - The functions you write vs. use from a package operate the same way when you execute them
 - Just because a function is in a package does not guarantee that it "works"

What's special about packages?

- The longer answer
 - Packages on CRAN can be installed with `install.packages`
 - Packages on CRAN have to pass a stringent set of automated checks every day to make sure they run
 - Large user bases make it more likely that errors are found
 - `devtools::install_github()` lets you install not-vetted packages, which may be experimental/buggy

Other thoughts

- There's little cost to writing functions
 - Even if it's something you'll use 3-4 times, it could clean up your code and make it easier to run
- Write functions that do one thing well
 - Functions that are trying to do too many things can be hard to name and hard to debug
- Revise your functions over time
 - Add arguments to make them more general