Functions and Iteration

SSEP 2022 Afternoon Day 3

Dr. Ab Mosca (they/them)

Slides based on slides courtesy of Jordan Crouser: https://jcrouser.github.io/MassMutual-IntroR/, https://jcrouser.github.io/MassMutual-DataVis/, https://beanumber.github.io/sds192/

Functions

Repetition

- Suppose you're a data scientist given weekly datasets to analyze
- The datasets are similar; each is a `tbl` with three variables
- They come un-tidy, so first you must make them tidy
- For three weeks use the following code:

```
my_df1 %>%
  pivot_wider(names_from = varA, values_from = varB) %>%
  group_by(varC) %>%
  summarise(numObservations = n())

my_df2 %>%
  pivot_wider(names_from = var1, values_from = var2) %>%
  group_by(var3) %>%
  summarise(numObservations = n())

my_df3 %>%
  pivot_wider(names_from = beep, values_from = boop) %>%
  group_by(blerp) %>%
  summarise(numObservations = n())
```

Repetition

```
my_df1 %>%
  pivot_wider(names_from = varA, values_from = varB) %>%
  group_by(varC) %>%
  summarise(numObservations = n())

my_df2 %>%
  pivot_wider(names_from = var1, values_from = var2) %>%
  group_by(var3) %>%
  summarise(numObservations = n())

my_df3 %>%
  pivot_wider(names_from = beep, values_from = boop) %>%
  group_by(blerp) %>%
  summarise(numObservations = n())
```

Work with the person next to you to find similarities in each code chunk above. Is there a common set of steps you take for each dataset?

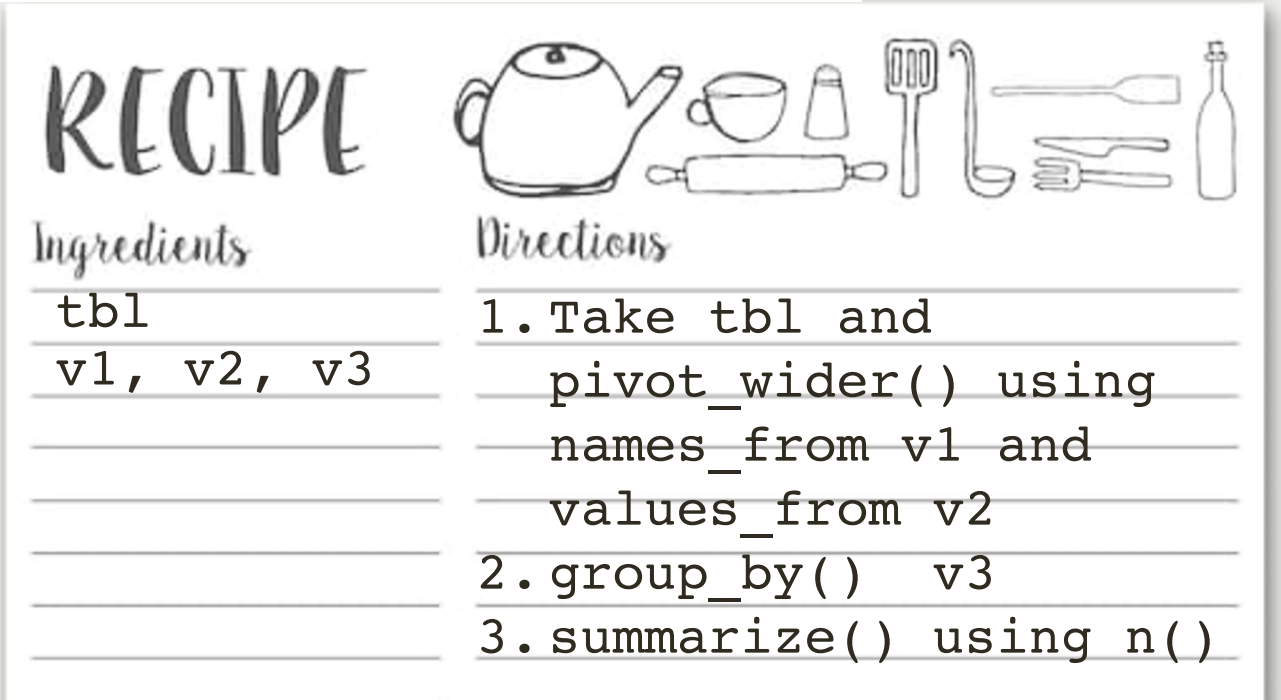
Record your answer on the Jamboard here: Common Steps

Repetition

```
my_df1 %>%
  pivot_wider(names_from = varA, values_from = varB) %>%
  group_by(varC) %>%
  summarise(numObservations = n())

my_df2 %>%
  pivot_wider(names_from = var1, values_from = var2) %>%
  group_by(var3) %>%
  summarise(numObservations = n())

my_df3 %>%
  pivot_wider(names_from = beep, values_from = boop) %>%
  group_by(blerp) %>%
  summarise(numObservations = n())
```

Repetition

```
my_df1 %>%
  pivot_wider(names_from = varA, values_from = varB) %>%
  group_by(varC) %>%
  summarise(numObservations = n())

my_df2 %>%
  pivot_wider(names_from = var1, values_from = var2) %>%
  group_by(var3) %>%
  summarise(numObservations = n())

my_df3 %>%
  pivot_wider(names_from = beep, values_from = boop) %>%
  group_by(blerp) %>%
  summarise(numObservations = n())
```



Functions

- Format:
  - `function_name(argument1, argument2, …)`
- Inputs
  - Arguments
  - Things like: `tbl, string, int`, etc.
- Output
  - Things like: `tbl, string, int`, etc.
- We've been using built in functions! Ex.
  - `ncol(babynames)`
    - Input = `tbl (babynames)`
    - Output = `int` (number of columns in `babynames`)

Functions

- Format:
  - `function_name(argument1, argument2, …)`
- Inputs
  - Arguments
  - Things like: `tbl, string, int`, etc.
- Output
  - Things like: `tbl, string, int`, etc.
- We've been using built in functions! Ex.
  - `ncol(babynames)`
    - Input = `tbl (babynames)`
    - Output = `int` (number of columns in `babynames`)

Work with the person next to you to find another example.

Add your answer to the Jamboard here: Function Examples

User-defined Functions

- We can (temporarily) add functions to R by defining them ourselves
  - We call these user-defined functions
- This is ***very*** useful for repetitive tasks

User-defined Functions

Defining your own functions

```
name_of_function <- function(data, var = "value") {
    ...
    ...
    <valid R code>
    ...
    ...
    return(x)
}
```

User-defined Functions

Defining your own functions

```
name_of_function <- function(data, var = "value") {
    ...
    ...
    <valid R code>
    ...
    ...
    return(x)
}
```

What you want the function to be called

User-defined Functions

Defining your own functions

```
name_of_function <- function data, var = "value") {
    ...
    ...
    <valid R code>
    ...
    ...
    return(x)
}
```

`function` is the key word that tells R "I'm creating a function"

`<-` assigns the function definition to the variable `name_of_function`

User-defined Functions

Defining your own functions

```
name_of_function <- function(data, var = "value") {
    ...
    ...
    <valid R code>
    ...
    ...
    return(x)
}
```

Input / arguments

Inputs

- *arguments*: data, var
- data is required
- var is optional -- has a default value of "value"

User-defined Functions

Defining your own functions

```
name_of_function <- function(data, var = "value") {
  ...
  ...
  <valid R code>
  ...
  ...
  return(x)
}
```

Body

Body

- Defines what the function should do
- Everything between { and }

User-defined Functions

Defining your own functions

```
name_of_function <- function(data, var = "value") {
    ...
    ...
    <valid R code>
    ...
    ...
    return(x)
}
```

Output

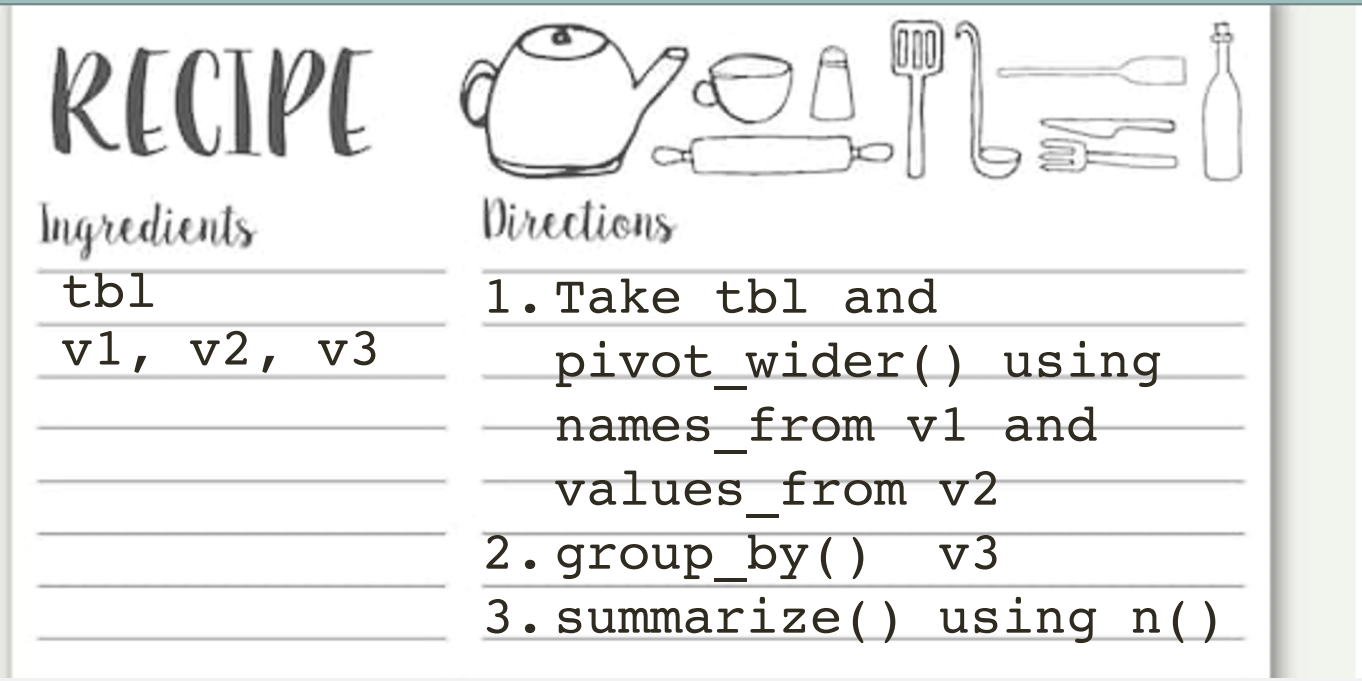
Output

- the *return* value
- by default output of last line in function body
- here, explicitly the object x

Work with the person next to you to turn our recipe into a function.

Add your answer to the Jamboard here: Function Examples

User-defined Functions

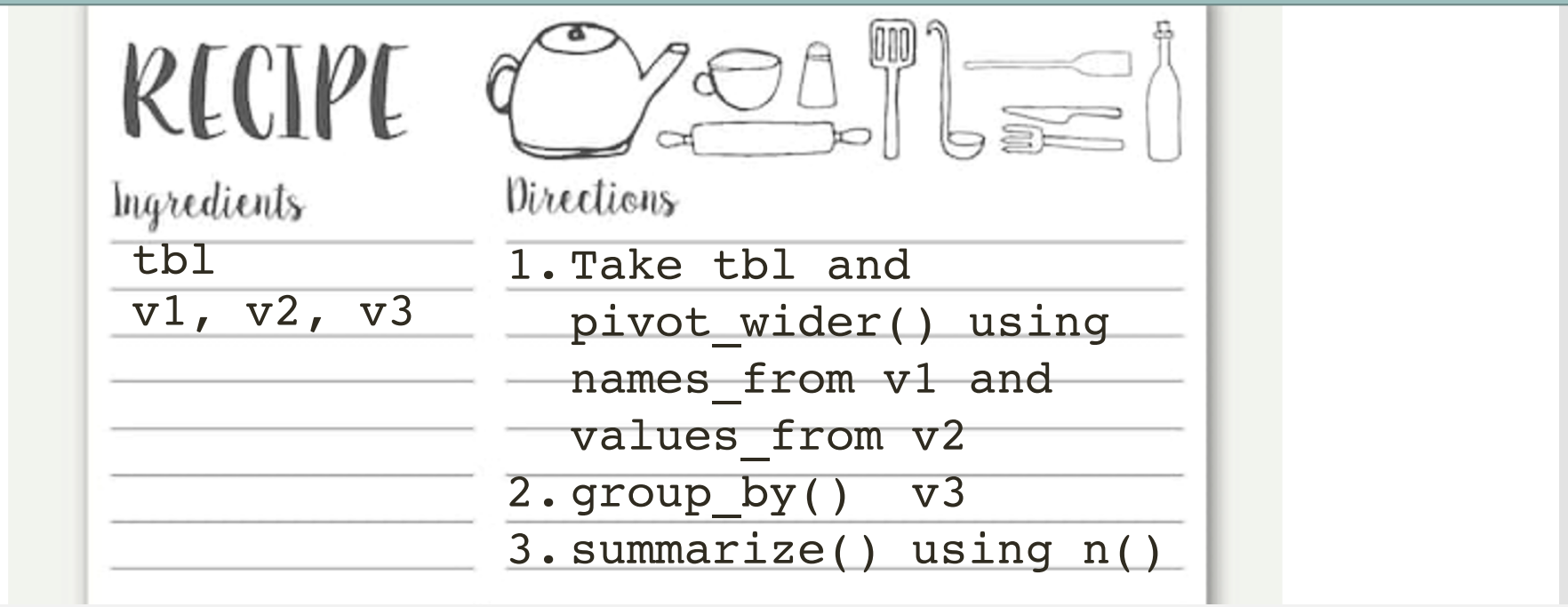



```
name_of_function <- function(data, var = "value") {
    ...
    <valid R code>
    ...
    return(x)
}
```

Work with the person next to you to turn our recipe into a function.

Add your answer to the Jamboard here: Function Examples

User-defined Functions




```
clean_data <- function(data, v1, v2, v3) {
   data %>%
      pivot_wider(names_from = v1, values_from = v2) %>%
      group_by(v3) %>%
      summarize(numObservations = n())
}
```

User-defined Functions

Defining your own functions

- To use a function you defined, you "call" it with the appropriate arguments
- Ex. Let's call `clean_data()` to make `my_df1` from earlier tidy

```
my_df1 %>%
  pivot_wider(names_from = varA, values_from = varB) %>%
  group_by(varC) %>%
  summarise(numObservations = n())
```

```
# Define clean_data
clean_data <- function(data, v1, v2, v3) {
    data %>%
        pivot_wider(names_from = v1, values_from = v2) %>%
        group_by(v3) %>%
        summarize(numObservations = n())
}
# Call clean_data
my_tidy_df1 <- clean_data(my_df1, varA, varB, varC)
```

User-defined Functions

```
# Define clean_data
clean_data <- function(data, v1, v2, v3) {
    data %>%
        pivot_wider(names_from = v1, values_from = v2) %>%
        group_by(v3) %>%
        summarize(numObservations = n())
}
```

Work with the person next to you call `clean_data` for `my_df2` and `my_df3`.

Add your answer to the Jamboard here: Function Examples

```
my_df2 %>%
  pivot_wider(names_from = var1, values_from = var2) %>%
  group_by(var3) %>%
  summarise(numObservations = n())

my_df3 %>%
  pivot_wider(names_from = beep, values_from = boop) %>%
  group_by(blerp) %>%
  summarise(numObservations = n())
```

User-defined Functions

```
# Define clean_data
clean_data <- function(data, v1, v2, v3) {
    data %>%
        pivot_wider(names_from = v1, values_from = v2) %>%
        group_by(v3) %>%
        summarize(numObservations = n())
}
```

Work with the person next to you call `clean_data` for `my_df2` and `my_df3`.

Add your answer to the Jamboard here: Function Examples

```
my_df2 %>%
  pivot_wider(names_from = var1, values_from = var2) %>%
  group_by(var3) %>%
  summarise(numObservations = n())

my_df3 %>%
  pivot_wider(names_from = beep, values_from = boop) %>%
  group_by(blerp) %>%
  summarise(numObservations = n())
```

```
my_tidy_df2 <- clean_data(my_df2,
                          var1,
                          var2,
                          var3)

my_tidy_df3 <- clean_data(my_df3,
                          beep,
                          boop,
                          blerp)
```

User-defined Functions

- Scope
- **Global environment**
  - The general space in which you're working
- **Global variable**
  - Variable declared in your script outside of the body of a function
  - **Global variables exist everywhere**

User-defined Functions

- Scope
- **Global environment**
  - The general space in which you're working
- **Global variable**
  - Variable declared in your script outside of the body of a function
  - **Global variables exist everywhere**
- Ex.

```
# global variable
global_var <- mtcars

top_cars <- function() {
    # function body has access to global_var
    local_var <- head(global_var)
    return(local_var)
}
# Can print global_var outside of the function too
global_var
```

User-defined Functions

- Scope
- **Local environment**
  - Body of a function
- **Local variable**
  - Variable declared within the body of a function
  - **Local variables exist only within the body** of the function in which they are declared

User-defined Functions

- Default Values
- **Local environment**
  - Body of a function
- **Local variable**
  - Variable declared within the body of a function
  - **Local variables exist only within the body** of the function in which they are declared
- Ex.

```
# global variable
global_var <- mtcars

top_cars <- function() {
    # declare local_var
    local_var <- head(global_var)
    return(local_var)
}
# Cannot print local_var outside of the function
local_var # causes error
```

```
## Error in eval(expr, envir, enclos): object 'local_var' not found
```

User-defined Functions

Default Values

- When defining a function, you can set default values for the arguments
- This can make functions easier to use
- Any default value can be overwritten

User-defined Functions

Default Values

```
my_car_info <- function(mod = "civic", n = 3) {
   mpg %>%
      filter(model == mod) %>%
      select(-manufacturer, -class) %>%
      head(n)
}
my_car_info()
```

```
## # A tibble: 3 x 9
##   model displ  year   cyl trans      drv     cty   hwy fl
##   <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr>
## 1 civic   1.6  1999     4 manual(m5) f        28    33 r
## 2 civic   1.6  1999     4 auto(l4)   f        24    32 r
## 3 civic   1.6  1999     4 manual(m5) f        25    32 r
```

User-defined Functions

Overriding Default Values

```
my_car_info <- function(mod = "civic", n = 3) {
   mpg %>%
      filter(model == mod) %>%
      select(-manufacturer, -class) %>%
      head(n)
}
my_car_info()
```

```
## # A tibble: 3 x 9
##   model displ  year   cyl trans      drv     cty   hwy fl
##   <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr>
## 1 civic   1.6  1999     4 manual(m5) f        28    33 r
## 2 civic   1.6  1999     4 auto(l4)   f        24    32 r
## 3 civic   1.6  1999     4 manual(m5) f        25    32 r
```

```
my_car_info(mod = “jetta”, n = 2)
```

```
## # A tibble: 2 x 9
##   model displ  year   cyl trans      drv     cty   hwy fl
##   <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr>
## 1 jetta   1.9  1999     4 manual(m5) f        33    44 d
## 2 jetta   2    1999     4 manual(m5) f        21    29 r
```

User-defined Functions

Naming Arguments

- Optional

User-defined Functions

Naming Arguments

- Optional

```
my_car_info(mod = "jetta", n = 2)
```

```
## # A tibble: 2 x 9
##   model displ  year   cyl trans      drv     cty   hwy fl
##   <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr>
## 1 jetta   1.9  1999     4 manual(m5) f        33    44 d
## 2 jetta   2    1999     4 manual(m5) f        21    29 r
```

```
my_car_info( "jetta", 2)
```

```
## # A tibble: 2 x 9
##   model displ  year   cyl trans      drv     cty   hwy fl
##   <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr>
## 1 jetta   1.9  1999     4 manual(m5) f        33    44 d
## 2 jetta   2    1999     4 manual(m5) f        21    29 r
```

User-defined Functions

Naming Arguments

- Optional
- But order matters if arguments are unnamed

User-defined Functions

Naming Arguments

- Optional
- But order matters if arguments are unnamed

```
my_car_info(2, "jetta")
```

```
## # A tibble: 0 x 9
## # … with 9 variables: model <chr>, displ <dbl>, year <int>, cyl <int>,
## #   trans <chr>, drv <chr>, cty <int>, hwy <int>, fl <chr>
```

```
my_car_info(n = 2, mod = "jetta")
```

```
## # A tibble: 2 x 9
##   model displ  year   cyl trans      drv     cty   hwy fl
##   <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr>
## 1 jetta   1.9  1999     4 manual(m5) f        33    44 d
## 2 jetta   2    1999     4 manual(m5) f        21    29 r
```

Iteration

Iteration

- Suppose you have `f(val)`, which returns a value, and you want to use this function on values `1 – 100`

Iteration

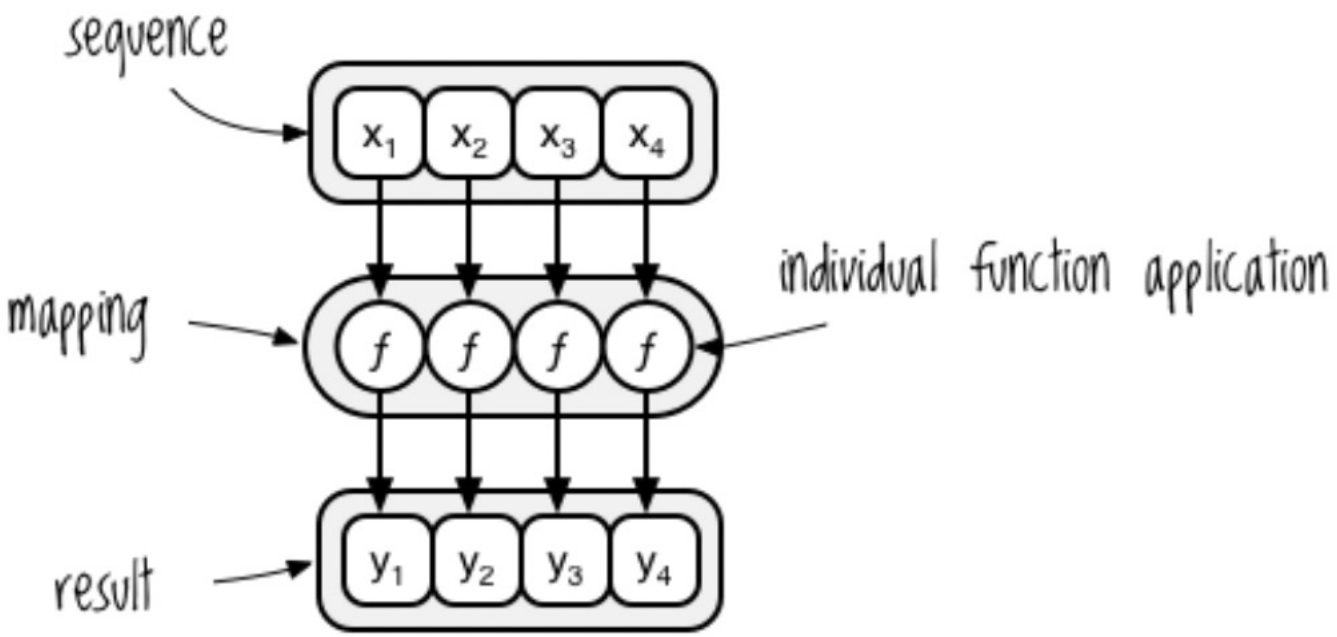
- Suppose you have `f(val)`, which returns a value, and you want to use this function on values `1 – 100`
- One option for doing this would be to call `f(val)` for `1 – 100`
  - `f(1)`
  - `f(2)`
  - `f(3)`
  - `f(4)`
  - `f(4)`
  - `f(6)`
  - `...okay, I'm already bored`

Iteration

- Suppose you have `f(val)`, which returns a value, and you want to use this function on a vector of values `1 – 100`
- One option for doing this would be to call `f(val)` for `1 – 100`
  - `f(1)`
  - `f(2)`
  - `f(3)`
  - `f(4)`
  - `f(4)`
  - `f(6)`
  - `...okay, I'm already bored`
- Uh-oh…not only am I bored, but
  - I accidentally called `f(4)` twice and skipped `f(5)`
  - and I'm going to get all my results separately, but I'd really like them in a vector

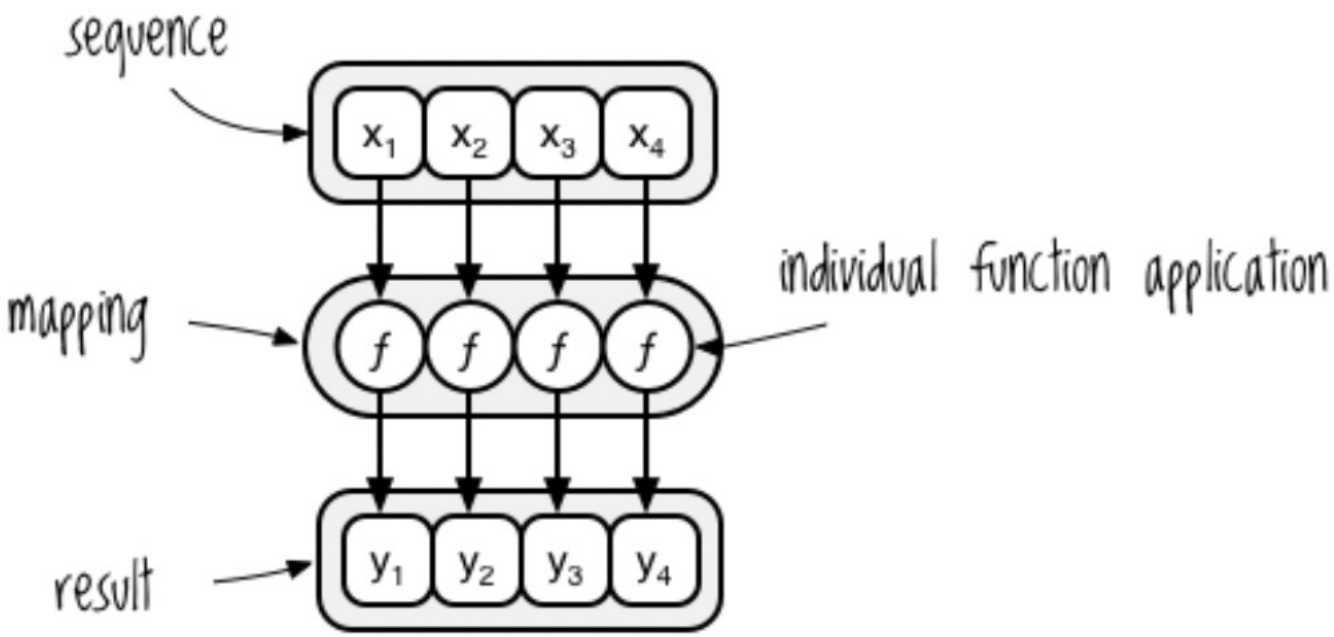
Iteration

- What if instead, we could apply `f()` to our vector of values, and get a vector of results?



Iteration

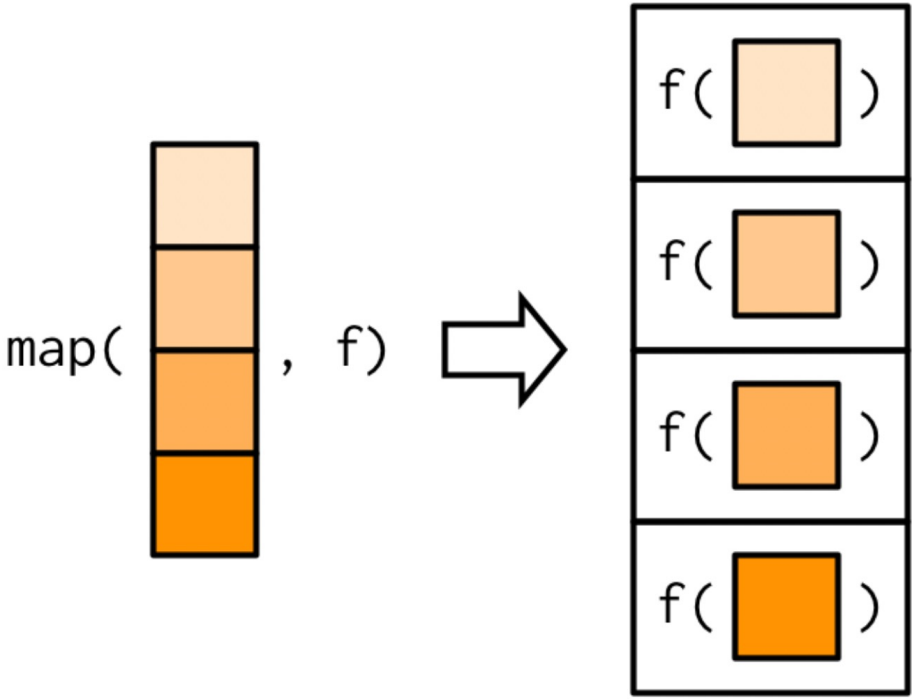
- What if instead, we could apply `f()` to our vector of values, and get a vector of results?



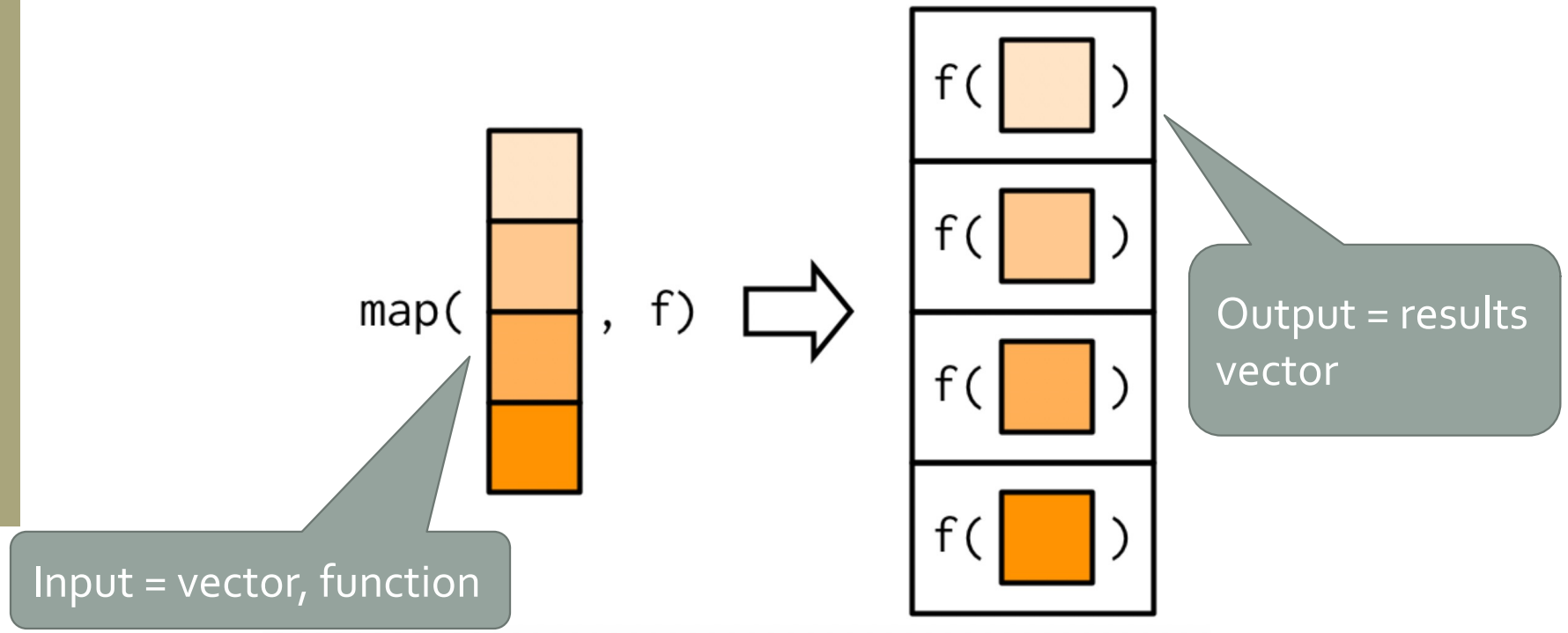
- We call this mapping, and it falls into the category of something called functional programming

- The library `purrr` makes functional programming with R easier
- Find the `purr` cheatsheet here: https://www.rstudio.com/resources/cheatsheets/

- The library `purrr` makes functional programming with R easier
- Find the `purr` cheatsheet here: https://www.rstudio.com/resources/cheatsheets/
- We will mostly use the `map()` function



- The library `purrr` makes functional programming with R easier
- Find the `purr` cheatsheet here: https://www.rstudio.com/resources/cheatsheets/
- We will mostly use the `map()` function



map()

- `map()` example

```
# vector of values
words <- c("alphabet", "bunny", "cathedral")

# iterate the function over values
# return a list of number of characters per word
map(words, nchar)
```

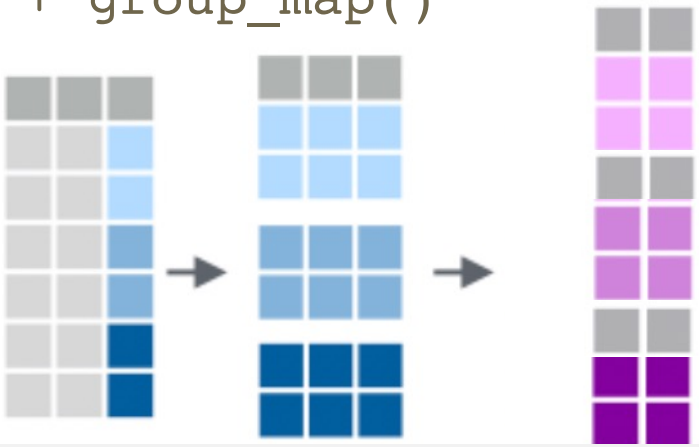
```
## [[1]]
## [1] 8
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 9
```

```
# iterate the function over values
# return a vector of number of characters per word
map_int(words, nchar)
```

```
## [1] 8 5 9
```

Mapping with and grouping

- We can use `group_by()` and apply different mapping functions to groups within our datasets
- Ex. `group_by() + group_map()`

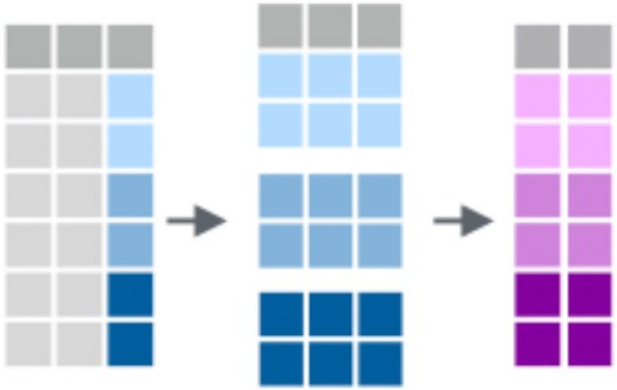


```
mtcars %>%
    group_by(cyl) %>% # creates a *list* of data frames!
    group_map(head, n = 2)
```

```
[[1]]
# A tibble: 2 x 10
    mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  22.8  108     93  3.85  2.32  18.6     1     1     4     1
2  24.4  147.    62  3.69  3.19  20       1     0     4     2

[[2]]
# A tibble: 2 x 10
    mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1    21   160   110  3.9  2.62  16.5     0     1     4     4
```

Mapping with and grouping

- We can use `group_by()` and apply different mapping functions to groups within our datasets
- Ex. `group_split() + map_dfr()`



```
mtcars %>%
    group_split(cyl) %>% # creates a grouped data frame
    map_dfr(head, n = 2)
```

```
## # A tibble: 6 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
## 2  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
## 3  21       6  160    110  3.9   2.62  16.5     0     1     4     4
## 4  21       6  160    110  3.9   2.88  17.0     0     1     4     4
## 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
## 6  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
```