Functional programming

Brooke Anderson

Advantages of writing functions

Do Not Repeat Yourself

Often, you'll want to do the same thing several times. For example, you might have data files from different locations or experiments, and you want to perform the same analysis with all of them.

Do Not Repeat Yourself

In this scenario, your first draft of code might include a lot of repeated code, places where you've cut-and-pasted, like:

```
bronx_calls <- bronx_calls %>%
  mutate(borough = "Bronx")
brooklyn_calls <- brooklyn_calls %>%
  mutate(borough = "Brooklyn")
manhattan_calls <- manhattan_calls %>%
  mutate(borough = "Manhattan")
queens_calls <- queens_calls %>%
  mutate(borough = "Queens")
staten_island_calls <- staten_island_calls %>%
  mutate(borough = "Staten Island")
```

Do Not Repeat Yourself

There are several limitations to code that repeats in this way, including:

- As you evolve the code to do something lengthy or complex, you waste a lot of time with copying-and-pasting
- If you want to change some of your code, you'll have to change it everywhere you copied-and-pasted it
- The code becomes very long, so it takes longer to read and understand it
- There is a limit to how far your code can scale (copy and paste 10 vs. 1,000 times)

Writing and using functions is one way to limit the amount of code you copy-and-paste.

Encapsulating an idea with a function

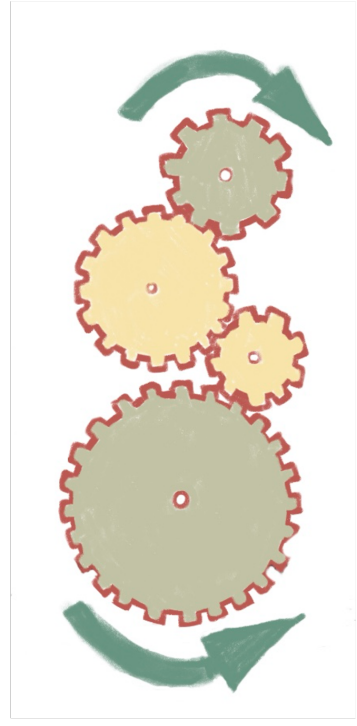
Another advantage of a function is that it separates the code that performs a subtask from the application of that subtask.

In practice, this lets you encapsulate the "idea" and implementation that you need to perform a subtask, and then you can think more abstractly about that subtask within the code script you develop for your data.

Encapsulating an idea with a function

You can think of the lines in a code script as a series of gears that work together to generate the final product, with the output of one feeding into the input of the next.

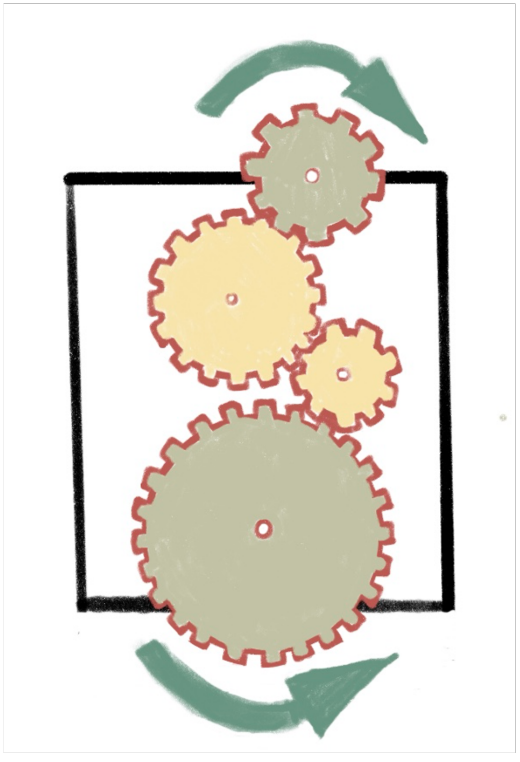
```
# Read in data and clean column names
man_animal_resp <- read_csv("../data/Manhattan_animal_response.csv") %>%
  clean_names() %>%
  rename(datetime_call = date_and_time_of_initial_call)

# Convert date-time column to appropriate data type
man_animal_resp <- man_animal_resp %>%
    mutate(datetime_call = mdy_hms(datetime_call))
```



Encapsulating an idea with a function

When you write a function, it's as if you enclose a series of these gears in a box. The gears at the start and end of the series will be exposed to the rest of the code, but the gears in between will be enclosed in the box.

```
read_and_clean_call_data <- function(file){

  # Read in data and clean column names
  call_data <- read_csv(file = file) %>%
    clean_names() %>%
    rename(datetime_call = date_and_time_of_initial_call)

  # Convert date-time column to appropriate data type
  call_data <- call_data %>%
    mutate(datetime_call = mdy_hms(datetime_call))

  return(call_data)
}
```

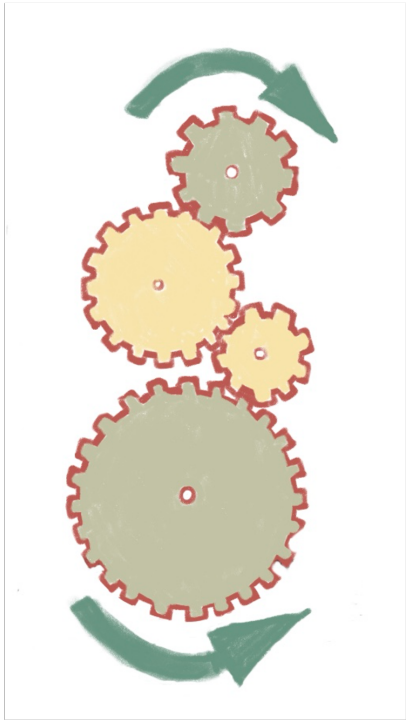


Encapsulating an idea with a function

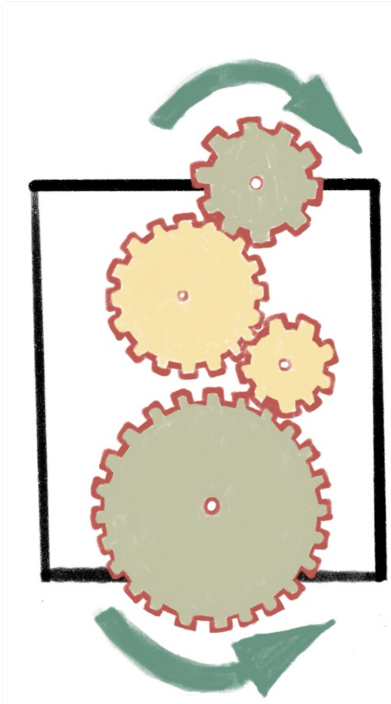
Once you've enclosed the interior gears, you can think about the function in terms of the task it does. When you apply the function, you can focus on this more abstract idea, while you can work with the details of the code when you edit the code that defines the function.

```
# Read in data and clean data
man_animal_resp <- read_and_clean_call_data(file = "../data/Manhattan_animal_response.csv",
                                            remove_partial_years = TRUE)

# Filter to data on reptiles and amphibians in Central Park
man_animal_resp <- man_animal_resp %>%
  filter(property == "Central Park") %>%
  filter(animal_class == "Terrestrial Reptile or Amphibian")

# Count number of response calls per month
monthly_man_animal_response <- count_monthly_calls(man_animal_resp)
monthly_man_animal_response
```

Encapsulating an idea with a function



You can think of lines in a code script as gears that work together.



You can encapsulate a subtask that involves one or more lines in the script as a function.

You can now abstract the details of the code to perform that subtask in the code script.

Encapsulating an idea with a function

Encapsulation and abstraction are excellent tools for solving a complex problem by:

- breaking the problem into smaller steps
- tackling each of those small steps with attention to detail
- then combining the solutions for the small steps in a way that focuses on the big picture of the overall task

Advantages of writing functions

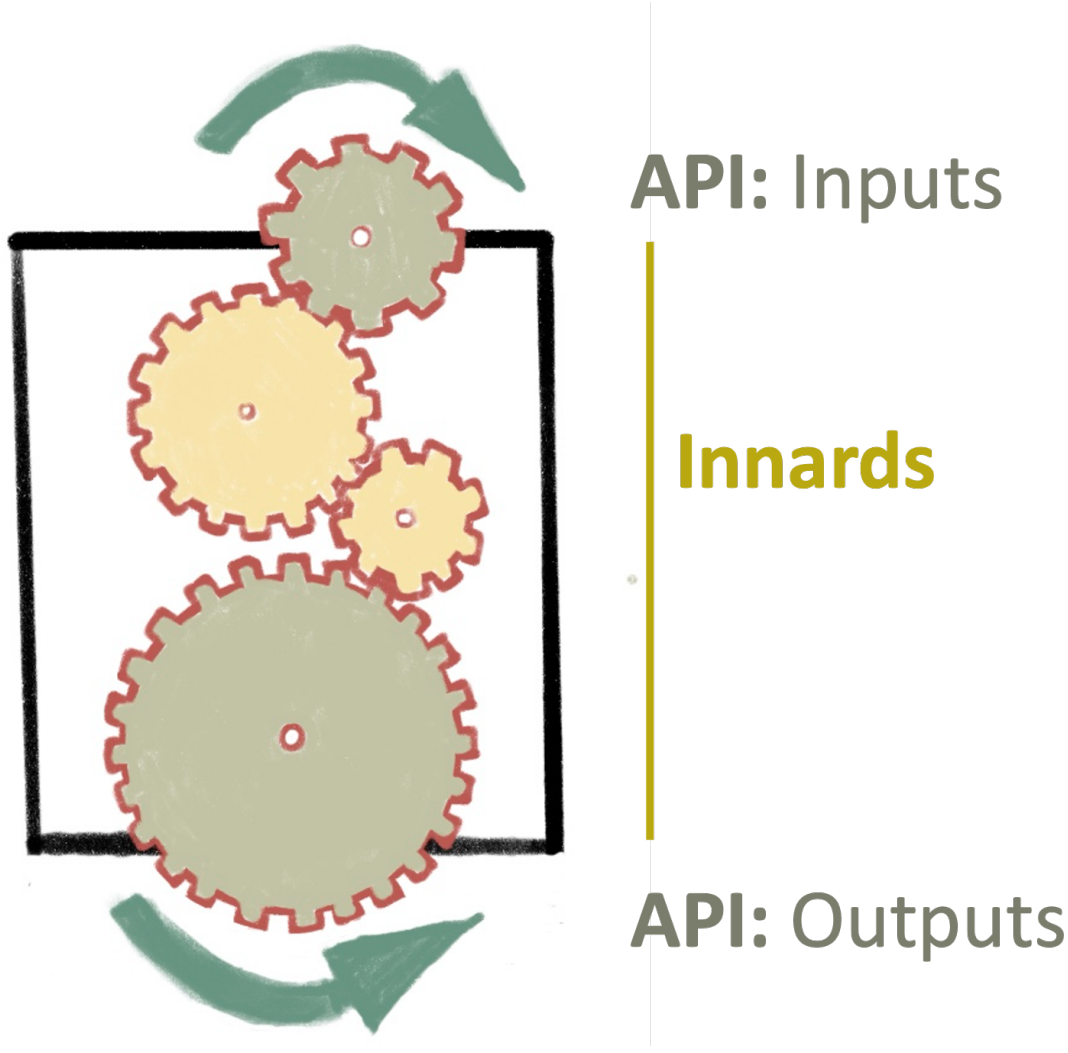
There are also some other advantages of writing functions:

- More efficient across broader work, since functions can be reused
- Each function can be documented clearly and thoroughly
- Writing functions is a step on the path toward writing packages and facilitates reproducibility within a team or larger community
- Gets you in the habit of breaking up a complex coding task into discrete, do-able steps

Writing functions

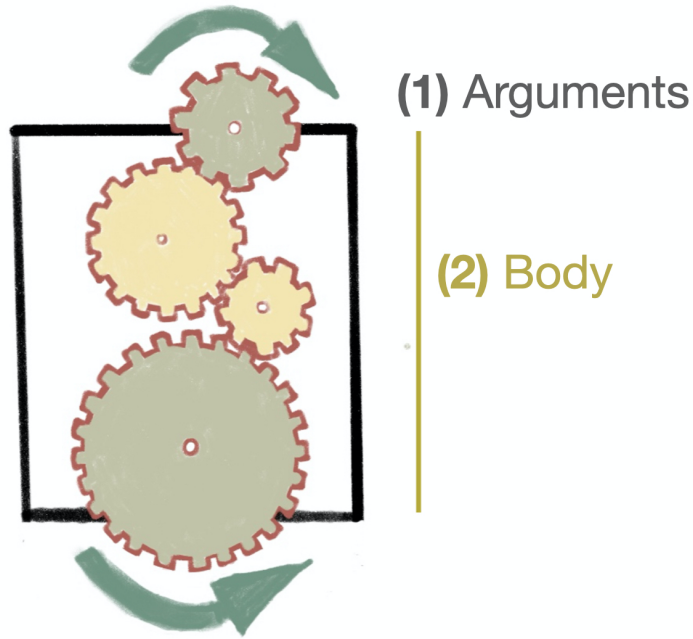
Anatomy of a function

There are two key elements in the anatomy of a function: the API and the innards.



Anatomy of a function

Another way to think about function anatomy is in terms of (1) the arguments, (2) the body, and (3) the environment. Arguments and body are both defined explicitly; the function's environment is implicit based on where the function is defined. Functions are objects:



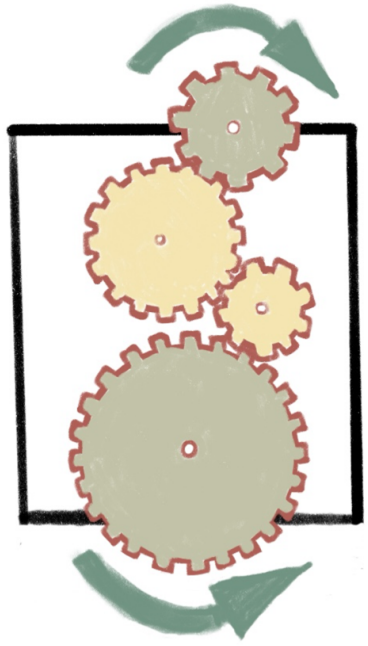
Anatomy of a function

The Application Programming Interface (API) of the function is the interface that you'll use when you apply the function—what do you put into and get out of the function?

(1) Arguments

```
do_my_task <- function(.data, keep_all = TRUE){

  [Function code]

  return(out)
}
```

(2) Body



Inputs are set within parentheses in the `function` call, while the output is specified by `return`.

Anatomy of a function

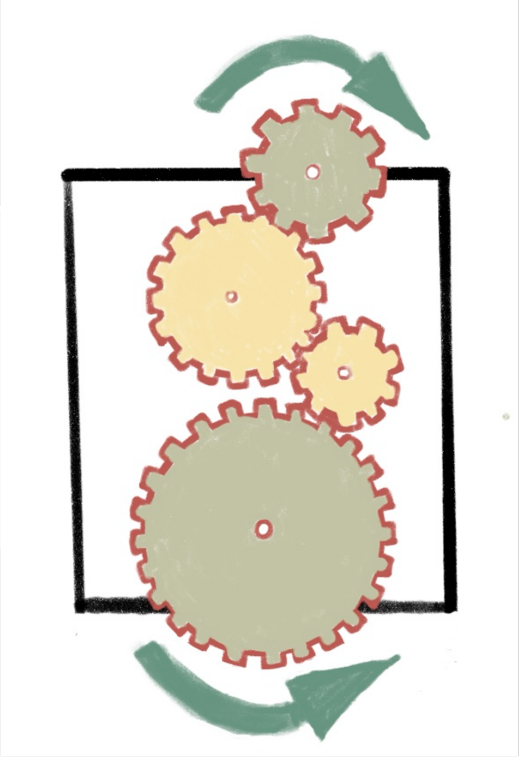
For the function's API, it's help to know that:

- You can only output one thing from the function. If you have a lot you need to get out, you can pack it into a list object to do so (this is why a lot of statistical tests and models return a list object)
- There are ways to avoid using `return`, but it's helpful to use it as your start writing functions, to make sure you're clear on what you're sending out of the function.
- If the API stays stable, the function should work robustly in earlier code you wrote with it, even if you change some of the details of how the code within the function runs.

Anatomy of a function

The "innards" of the function are the code that is run each time you call the function.

```
do_my_task <- function(.data, keep_all = TRUE){

  [Function code]

  return(out)
}
```



Anatomy of a function

For the innards of the function, it's helpful to know that:

- R will move to a different R environment when the function runs, so the objects created within the function code won't show up in your working environment.
- The environment determines how the function finds the values associated with names
- If the innards include more than one function call, then group them together with curly brackets ({ to start the code for the function, } to end it).

How to build a function

Writing functions

Functions can be simple, just encapsulating a single, simple (at least to simple to describe) step.

- `mutate`: Change or add a column in a dataframe
- `rename`: Rename a column
- `read_csv`: Read data from a comma-separated file into R
- `str_to_lower`: Convert all characters in a character string to lowercase

Example: Getting monthly counts of animal response calls

As an example, we'll build a function for one of the tasks in the animal response call script we edited in the last lab.

One of the tasks we did was to count the number of calls per month. We'll see if we can build a function to do this task.

Dataset with one row per animal response call

	species_description	datetime_call	call_month
	<chr>	<dttm>	<ord>
1	Corn Snake	2019-04-04 15:30:00	April
2	Raccoon	2019-02-09 13:00:00	February
3	Black-crowned Night Heron	2020-07-17 11:52:00	July
4	Mallard Duck	2020-07-15 09:15:00	July
5	Groundhog	2020-11-01 10:00:00	November
6	Striped Skunk	2019-07-05 14:30:00	July
7	Blue Jay	2020-06-22 13:30:00	June
8	Red-tailed Hawk	2019-02-16 13:00:00	February
9	Raccoon	2020-02-02 09:09:00	February
10	Cat	2019-11-19 13:30:00	November
11	Groundhog	2020-11-04 08:30:00	November
12	Mallard Duck	2020-07-05 16:15:00	July
13	Eastern Gray Squirrel	2020-05-28 15:00:00	May
14	Red-tailed Hawk	2019-09-26 16:30:00	September
15	Eastern Gray Squirrel	2019-07-16 15:00:00	July

count_monthly_calls

We want to create a function called `count_monthly_calls` that will input a dataframe with animal calls and output the number of calls per month.

Dataset with summary of calls by month

	call_month	n
	<chr>	<int>
1	February	3
2	April	1
3	May	1
4	June	1
5	July	5
6	September	1
7	November	3

How to name a function

The first step is to pick a name for the function.

Some general tips for naming a function include:

- Make the name a verb
- Make the name self-documenting (so it means something related to what the function does)
- Use the same prefix for similar functions (tab completion can jog your memory)
- Make sure it's not too long or too short, all lowercase, use underscores to help make them more readable

Identify relevant code in script

Next identify the section of code that you want to encapsulate in a function.

If we look at the final script from the last lab, we can find the section where we calculate the number of response calls per month:

```
# Count number of response calls per month
monthly_man_animal_response <- man_animal_resp %>%
  count(.wt = call_month)
```

Identify function elements in the code

Next, we need to identify how this code can be divided into a function, specifically into the **API** versus the **innards** of the function.

We can pick out those parts in the code and highlight them:

Orange: API

Yellow: innards

```
# Count number of response calls per month
monthly_man_animal_response <- man_animal_resp %>%
  count(.wt = call_month)
```

Identify function elements in the code

In this example the **argument** is the object that's being fed into the **body** of the function:

Orange: API

Yellow: innards

```
# Count number of response calls per month
monthly_man_animal_response <- man_animal_resp %>%
  count(.wt = call_month)
```

(1) Argument

(2) Body

Convert the code to a function

Now that we've identified the parts, we can build the function from that.

```
count_monthly_calls <- function(man_animal_resp) {

  monthly_man_animal_response <- man_animal_resp %>%
    count(.wt = call_month)

  return(monthly_man_animal_response)
}
```

Convert the code to a function

Name

(1) Argument

```
count_monthly_calls <- function(man_animal_resp) {

  monthly_man_animal_response <- man_animal_resp %>%
    count(.wt = call_month)

  return(monthly_man_animal_response)
}
```

(2) Body

Convert the code to a function

Again, let's use highlighting to think about where the elements from the script ended up in the function:

Orange: API

Yellow: innards

```
count_monthly_calls <- function(man_animal_resp) {

  monthly_man_animal_response <- man_animal_resp %>%
    count(.wt = call_month)

  return(monthly_man_animal_response)
}
```

Edit the function

We now have a working function.

Next, we'll want to make some edits to it to make it easier to understand, debug, and use to reproduce analysis.

Edit to improve object names

We can edit the function to use better, more generic, names. Originally, the input and output object names were specific to our code script.

We'll change the input parameter `man_animal_resp` to `call_data` and the output object from `monthly_man_animal_response` to `monthly_calls`:

```
count_monthly_calls <- function(call_data) {

  monthly_calls <- call_data %>%
    count(.wt = call_month)

  return(monthly_calls)
}
```

How to name function arguments

Here are a couple of things to keep in mind as you decide the names to use for function arguments and define them in the function code:

- Arguments can be either optional (`n = 6`) or required (`.data`)
- Consider using common parameter names (easier for user to figure out what each does and to remember them)

Adding package::function notation

Next, we'll update the function to add something that will make the function more robust when you re-use it or others use it.

Often, within function code you'll use functions from packages rather than just base R. There are a few ways to call those functions. In your scripts, you probably load the library and the call the functions in it by name, doing something like:

```
library(pkg)
pkg_function(foo)
```

Adding package::function notation

However, you can use a different notation to call a function from a package:

```
pkg::pkg_function(foo)
```

When you use this method, you must have the package installed, but you don't need to have it loaded in your current session.

Also, your code will explicitly get the function from the specified package. If you're using a function that's included in many packages, this notation will guarantee which one R uses.

```
stats::filter()
dplyr::filter()
```

Adding package::function notation

Let's make that change to our function. The `count` function comes from the `dplyr` package, so we can use the notation `dplyr::count`.

```
count_monthly_calls <- function(call_data) {

  monthly_calls <- call_data %>%
    dplyr::count(.wt = call_month)

  return(monthly_calls)
}
```

Adding help documentation

Since you are using the function to encapsulate and extract a subtask, you will find it helpful to include some documentation to describe the details of the function, and you can include that in the code where you define the function.

Documentation you might want to include:

- Describe what the function is meant to do
- Document each parameter: What it does, what type of value it can take
- Describe what the output from the function will be
- Consider adding an example of how to use it (ideally with a dataset that comes with R or an R package)

Adding help documentation

Here's an example of documentation we might want to add to the code that defines our function:

```
# This funtion inputs a dataframe of animal call
# response data (`call_data`) and returns a
# dataframe with the counts of animal calls per
# month. The input dataframe should include a column
# named `call_month` with the month of the call.
# To run this function, you will need to have the
# following packages installed: dplyr
count_monthly_calls <- function(call_data) {

  monthly_calls <- call_data %>%
    dplyr::count(.wt = call_month)

  return(monthly_calls)
}
```

How to check the function code as you build it

You'll want to try the code out as you work on the function. There are different ways to do that, but here's a simple method as you learn to write functions:

- Create example objects for each of your parameters
- Walk through the code inside the function line-by-line

Getting used to this workflow will also help with debugging.

How to check the function code as you build it

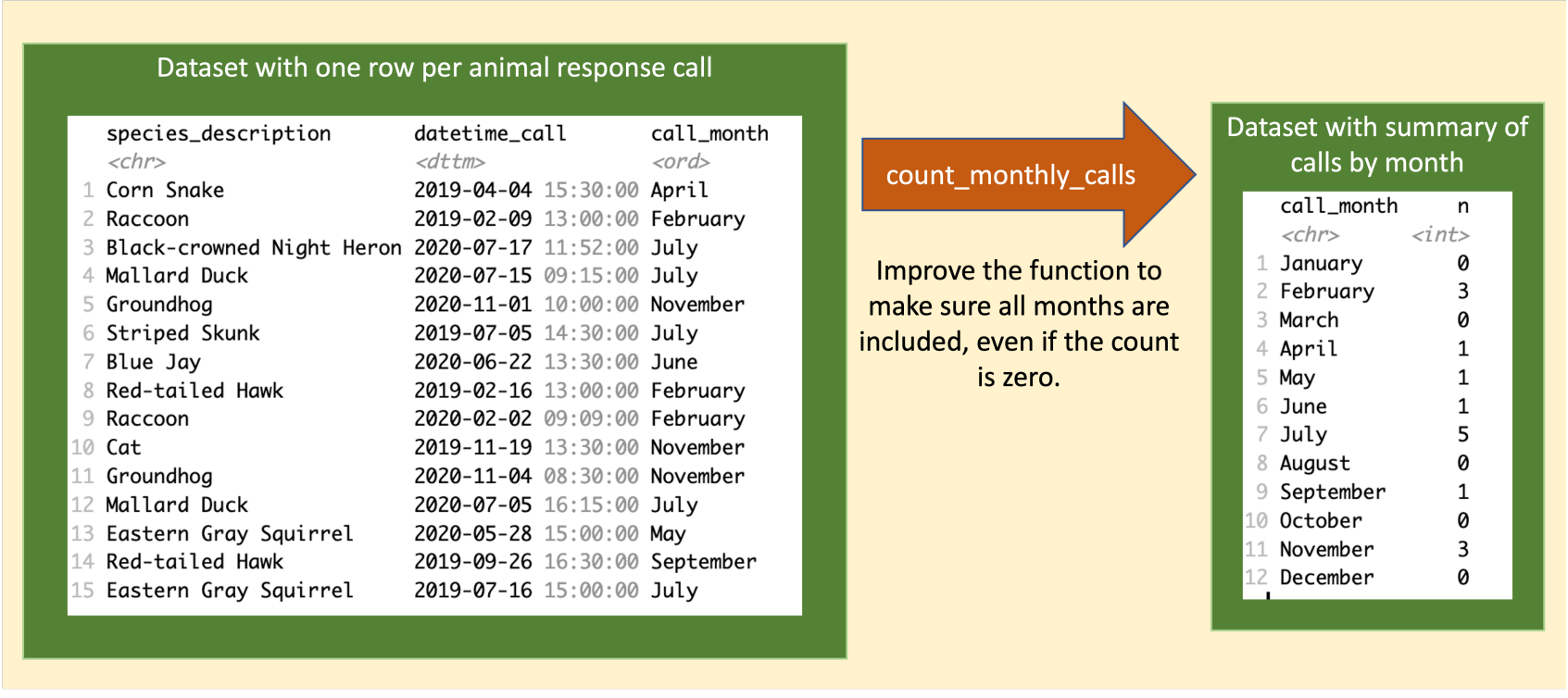
Let's try that with our example code:

```
# Set an example for function input parameter(s)
call_data <- man_animal_resp

# Walk through code inside the function
monthly_calls <- call_data %>%
    count(.wt = call_month)

# Check out the code results as you go
head(monthly_calls, 2)
```

```
## # A tibble: 2 x 2
##   .wt           n
##   <ord>     <int>
## 1 February      1
## 2 May           6
```

Improving the function

In the lab, we'll work with a version of the function that we've improved, to make sure the output includes all months, even those with zero animal response calls.



Improving the function

Here is our first version of that improved function code. In a later lab, we'll see that this is a bit of a "kluge", and we'll work to improve this function even more.

```
# To run this function, you will need to have the following packages installed:
# dplyr, tibble
count_monthly_calls <- function(call_data) {
  monthly_calls <- call_data %>%
    dplyr::count(.wt = call_month) %>%
    dplyr::rename(month = .wt)

  all_months <- tibble::tibble(month = c("January", "February", "March", "April",
                                         "May", "June", "July", "August", "September",
                                         "October", "November", "December"),
                               order = 1:12)

  monthly_calls <- dplyr::full_join(monthly_calls, all_months, by = "month") %>%
    dplyr::mutate(n = if_else(is.na(n), 0, n)) %>%
    dplyr::arrange(order) %>%
    dplyr::select(-order)

  return(monthly_calls)
}
```

Applying functions with `purrr`

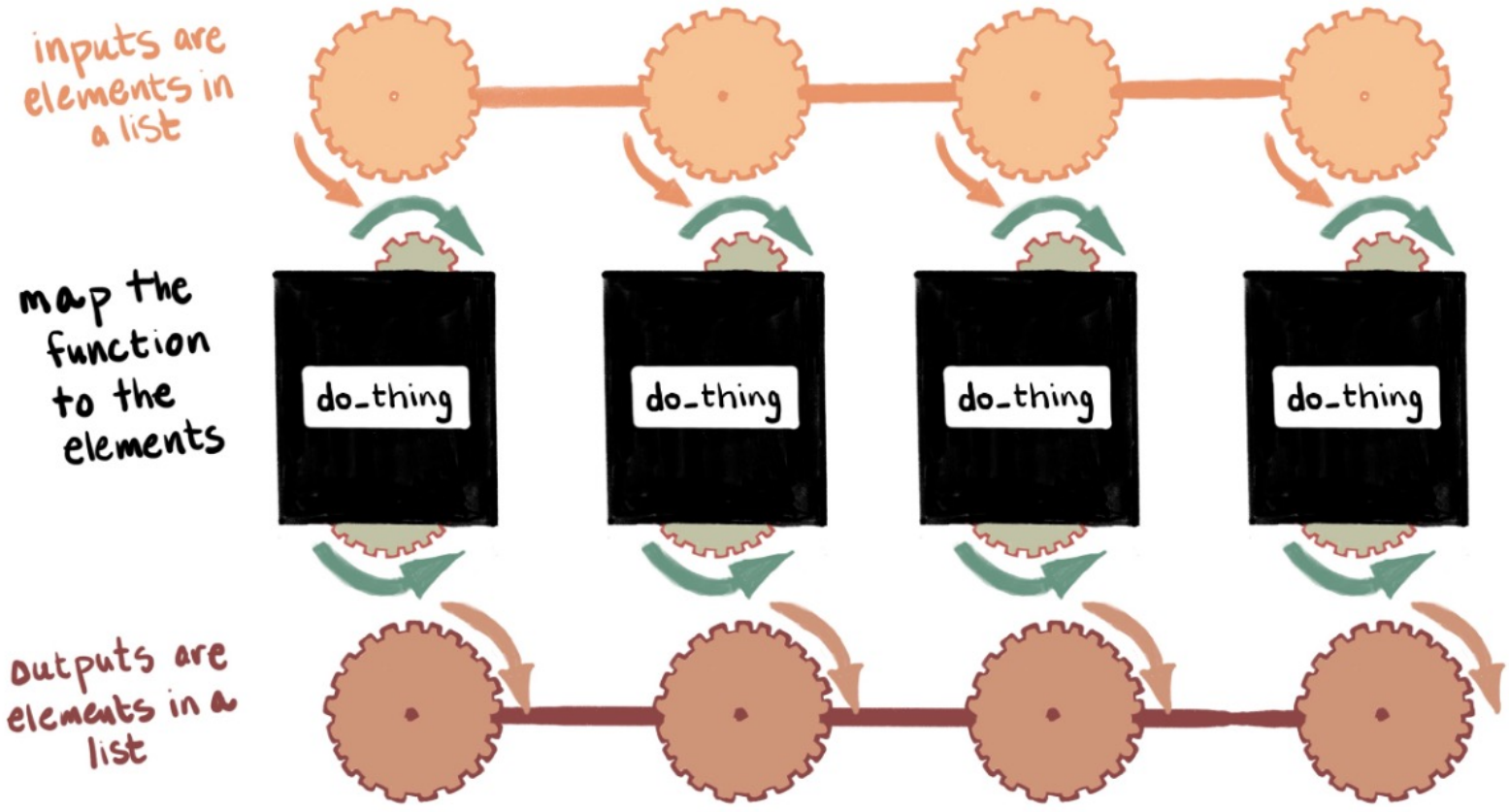
Mapping a function across values in a list / vector

Even with your own functions, you'll find times when you're repeating code. For example, if you write a function to read in data from a file in a certain format, and you have many files in that format, you may end up with code like this:

```
bronx_calls <- read_and_clean_call_data("../data/Bronx_animal_response.csv",
                                        remove_partial_years = TRUE)
brooklyn_calls <- read_and_clean_call_data("../data/Brooklyn_animal_response.csv",
                                        remove_partial_years = TRUE)
manhattan_calls <- read_and_clean_call_data("../data/Manhattan_animal_response.csv",
                                        remove_partial_years = TRUE)
queens_calls <- read_and_clean_call_data("../data/Queens_animal_response.csv",
                                        remove_partial_years = TRUE)
staten_island_calls <- read_and_clean_call_data("../data/Staten_Island_animal_response.csv",
                                        remove_partial_years = TRUE)
```

Mapping a function across values in a list / vector

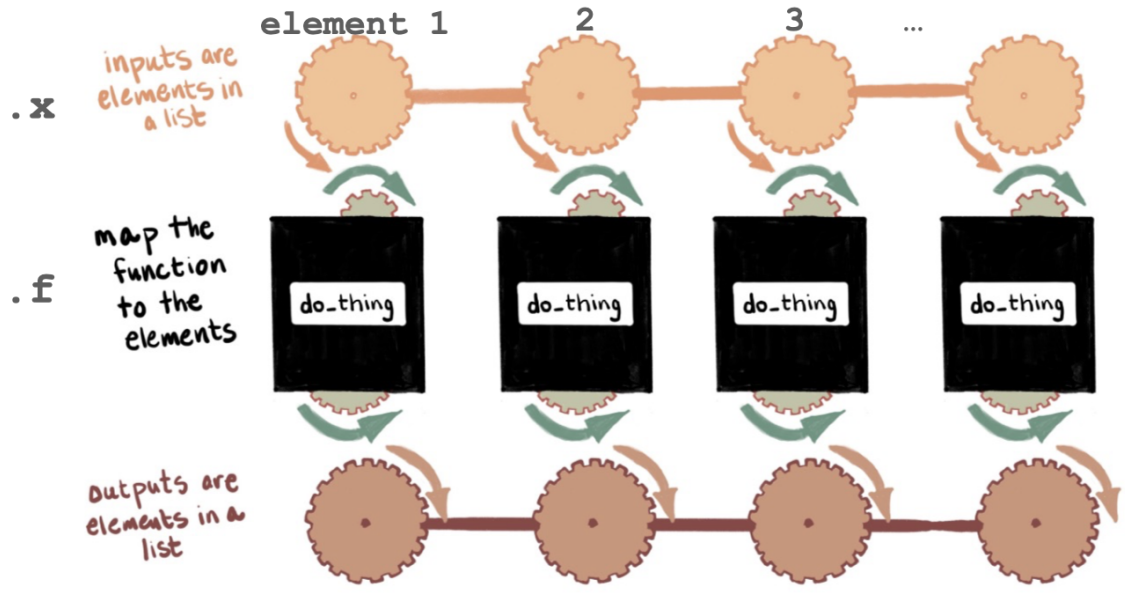
The `purrr` package has a collection of `map` functions that allow you to apply a function to all the elements of list or vector at once.



Mapping a function across values in a list / vector

```
purrr::map(.x, .f, ...)
```

For every element of `.x`, apply `.f`



Mapping a function across values in a list / vector

```
purrr::map(stickers, turn_over)
```

For every element of `.x`, apply `.f`

Mapping a function across values in a list / vector

```
purrr::map(dr_pepper, add_sunglasses)
```

For every element of `.x`, apply `.f`

`element` 1 2 3

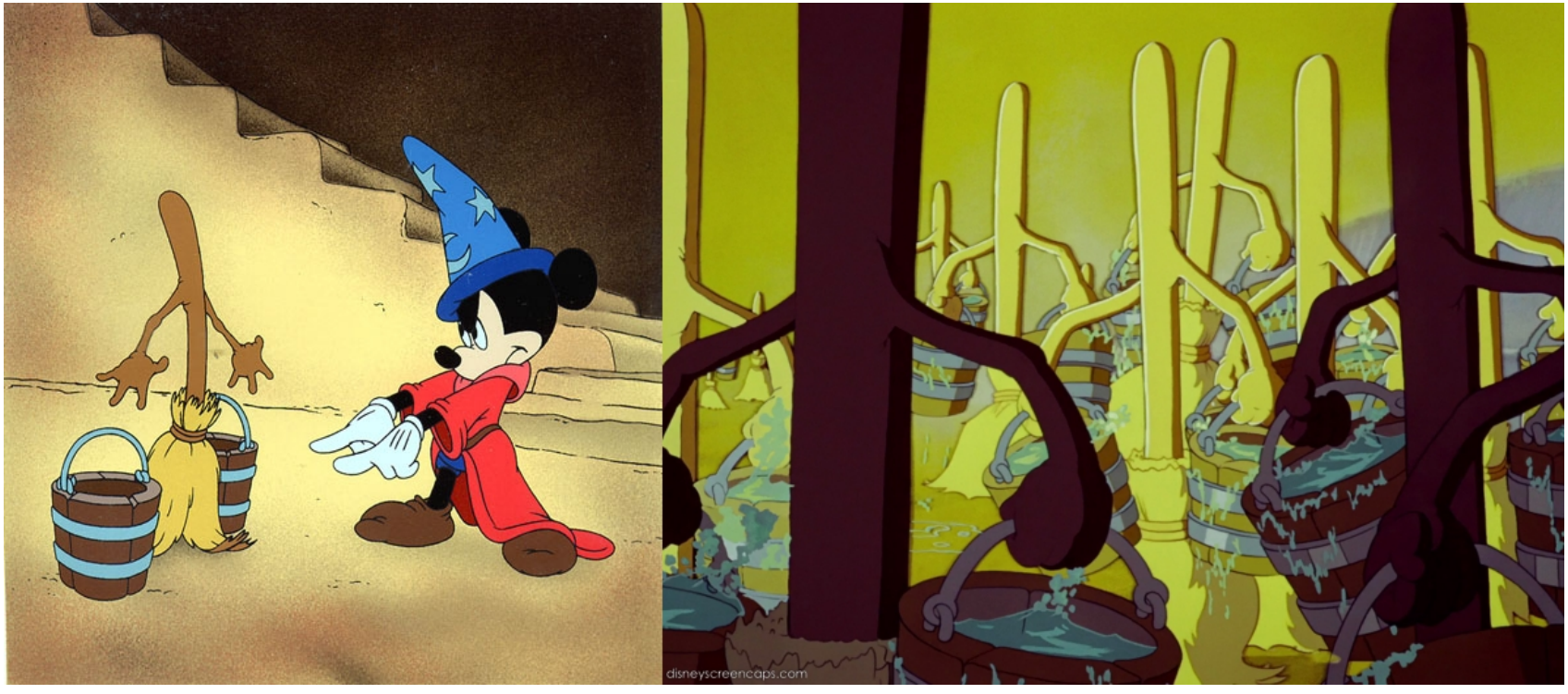
`.x`

`.f` `add_sunglasses` `add_sunglasses` `add_sunglasses`

`Outputs`

Mapping a function across values in a list / vector

Once you've mastered this approach, you'll feel similar to Mickey Mouse when he has dozens of brooms working for him in Fantasia (before things get out of control).



Some applications of mapping

There are many applications of the `map` functions from `purrr`. Two that I see often among researchers from a variety of fields are:

- Reading in lots of files from a directory and joining them into a single dataframe: Applying the function along a vector of filenames
- Applying a statistical test or model to many subsets of the data: Nesting the data by a grouping variable and applying the function to values in a list column for each group

Using `purrr` for functional programming

When you use these `map` functions to apply a function efficiently, it can feel at first like playing three-dimensional chess.

The best way to start to learn this approach is to try it out with some real applications, which we'll do in a later lab.