Lecture 2

Brooke Anderson

Encapsulating ideas with functions

Advantages of writing functions

- Fewer lines of code to do the task: Do not repeat yourself
- Separates the code from the logic of the function from the application of it
- More efficient across broader work: Functions can be reused
- Can be documented clearly and throughly
- A step on the path toward writing packages
- Gets you in the habit of breaking up a complex coding task into discrete, do-able steps

Do Not Repeat Yourself

Encapsulating an idea with a function

[Image—script with all the code]

Encapsulating an idea with a function

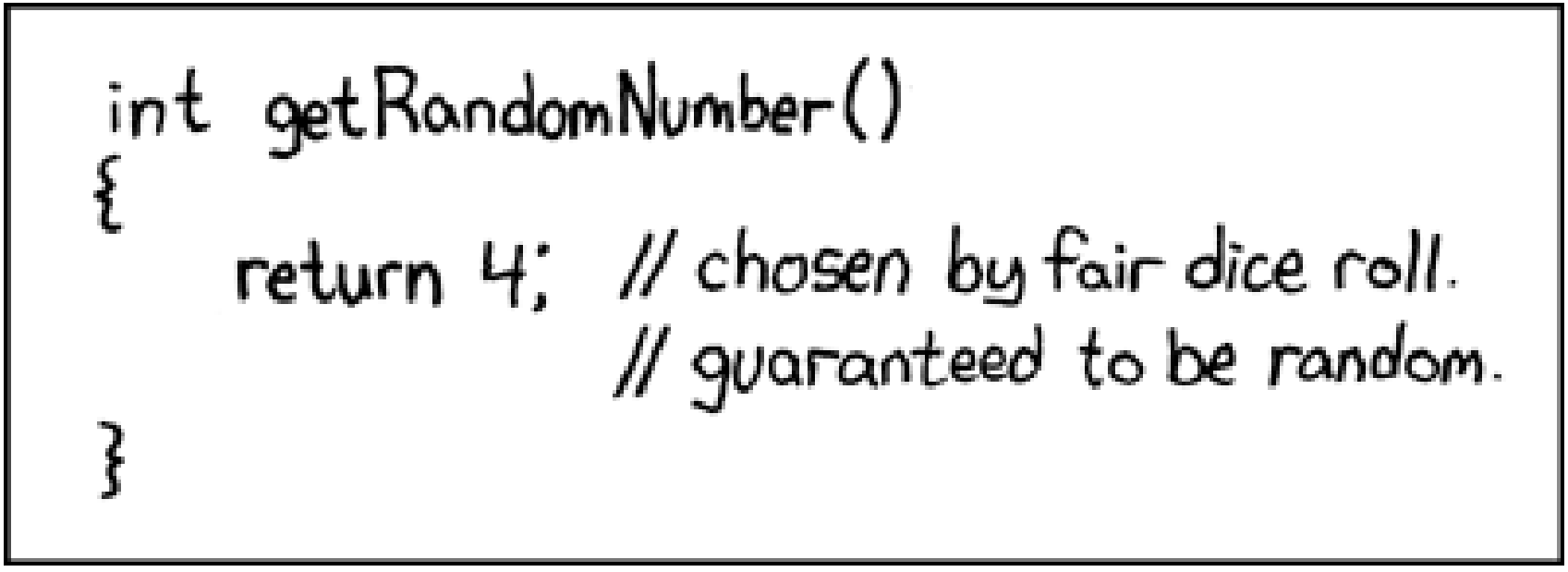
[Image–script with some parts encapsulated as functions]

Encapsulating an idea with a function

Functions can be simple, just encapsulating a small, simple step.

Writing functions

Example: Implementing an xkcd comic



Example: Implementing an xkcd comic

```
get_random_number <- function(dice_roll = 4){

  random_number <- dice_roll

  return(random_number)
}
```

What is a function

Anatomy of a function

Two key elements: API and innards

Anatomy of a function

API: what you put in and what you get out.

This is all someone needs to know to be able to use the function (although it's always nice if they have an idea of what happens in between).

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

Inputs are set within parentheses in the `function` call:

Anatomy of a function

The output is specified by `return`:

Anatomy of a function

- 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 what 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.

Anatomy of a function

Innards: What is run each time you call the function.

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.

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

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.

How to name a function

- Verb
- Self-documenting: name means something
- Use the same prefix for similar functions (tab completion can jog your memory)
- Similar to naming objects: Not too long or too short, all lowercase, use underscores to help make them more readable

Identify relevant code in script

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)
```

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

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 to improve object names

The input and output object names were specific to our code script. We can edit the function to use better names.

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

- Optional versus required arguments
- Use common parameter names (easier for user to figure out what each does and to remember them)

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

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
```

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, you can guarantee with package the code gets the function from. If you're using a function that's included in many packages, this notation will guarantee which one R uses.

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

- 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)

Notice that this is getting similar to a helpfile for a function in a 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)
}
```

Improving the function

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

Improve the function to make sure all months are included, even if the count is zero.

Dataset with summary of calls by month

	call_month	n
	<chr>	<int>
1	January	0
2	February	3
3	March	0
4	April	1
5	May	1
6	June	1
7	July	5
8	August	0
9	September	1
10	October	0
11	November	3
12	December	0

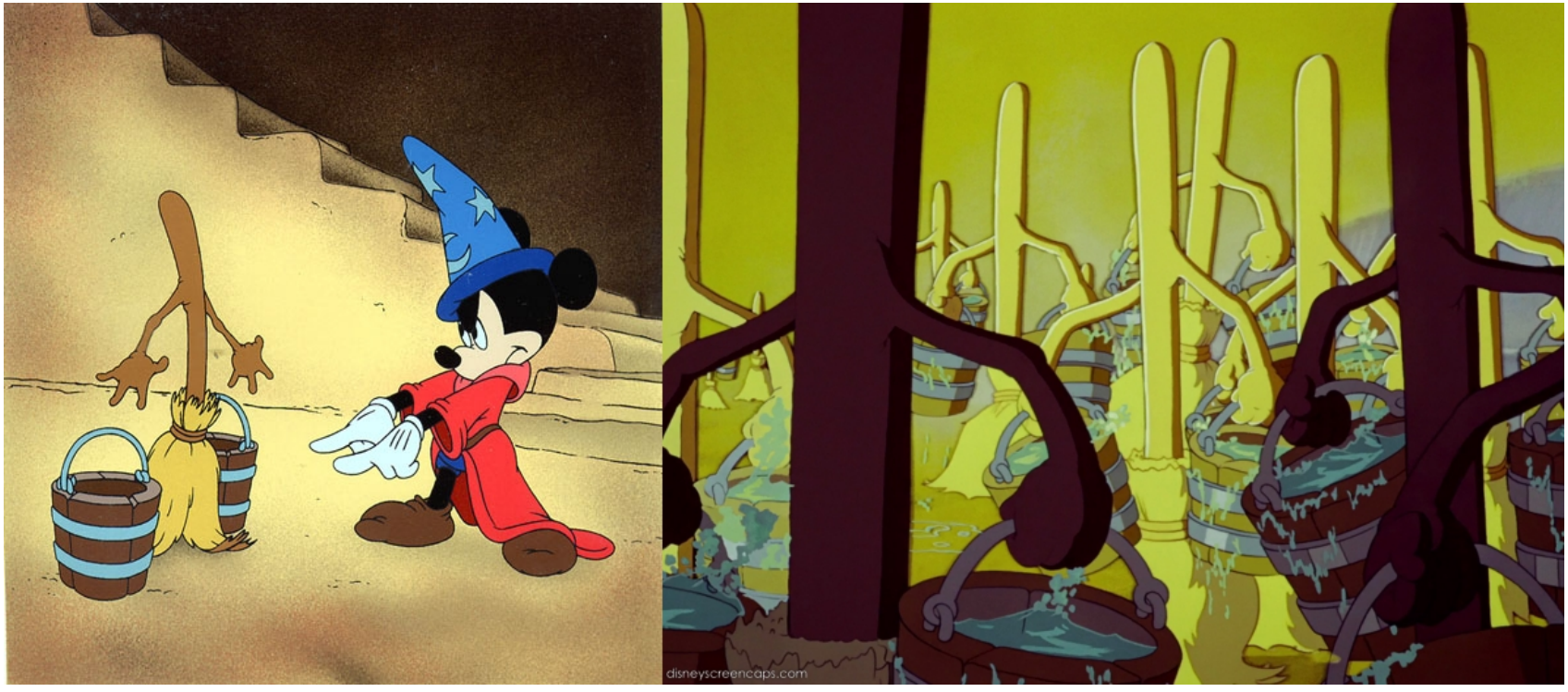
Improving the function

```r
# 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

Mapping a function across values in a list / vector



Nesting and mapping

Some applications of mapping

- 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