Stop repeating yourself: functions and iteration in R

Scientific workflows: Tools and Tips 🛠️

2026-06-18

What is this lecture series?

Scientific workflows: Tools and Tips 🛠️

🗓️ Every 3rd Thursday 🕓 4-5 p.m. 📍 Webex

- One topic from the world of scientific workflows
- Material provided online
- If you don't want to miss a lecture
  - Subscribe to the mailing list
- For credit points: Send me a short message (Email or Webex)

Today: Functions and iterations

- How to write your own **functions**
- How to **iterate** using the `purrr` package from the `tidyverse`

How it works

- There's a **GitHub repo** you download and open in your IDE
- I demonstrate each topic: **follow along or just watch**
- Each module has a short **exercise** to try it yourself
- **Questions** anytime: in the chat or just unmute

Get set up

1. Click the green **Code** button → **Download ZIP**
2. Unzip it somewhere you can find it
3. Double-click the `.Rproj` file → opens in RStudio
    - *(Positron/VS code users: open the folder)*
4. Install today's packages (if you don't have them yet)

```
install.packages(c("tidyverse", "palmerpenguins"))
```

Repo: https://github.com/selinaZitrone/functions-iteration-workshop

The problem: copy-paste code

- Copy paste code violates the **DRY** (do not repeat yourself) principle
- Fragile and hard to maintain
  - Copy-paste errors
  - Change logic in multiple places

Solution: Write a **function** to capture the logic, then **iterate** over different inputs

```
library(tidyverse)
library(palmerpenguins)

# Adelie species
adelie <- filter(penguins, species == "Adelie")
ggplot(adelie, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  geom_smooth(method = "lm") +
  theme_minimal()

# Chinstrap species
chinstrap <- filter(penguins, species == "Chinstrap")
ggplot(chinstrap, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  geom_smooth(method = "lm") +
  theme_minimal()

# Gentoo species
gentoo <- filter(penguins, species == "Chinstrap")
ggplot(gentoo, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
```

Writing your own functions in R

Anatomy of a function

```
name <- function(arguments) {
  body
}
```

A function has

- a **name**: Choose something descriptive that starts with a verb
- **arguments** (its inputs)
- a **body**: All the code that is run using the input
- a **return value**: the output of the function (the last line of the body)

A simple example

A function has

```
name <- function(arguments) {
  body
}
```

```
add_numbers <- function(x, y) {
  x + y
}

add_numbers(1, 1)
```

```
[1] 2
```

```
add_numbers(x = 1, y = 1)
```

```
[1] 2
```

Turn code into functions

- What **changes** becomes an **argument** (the data)
- What **stays the same** becomes the **body** (the ggplot)

```
adelie <- filter(penguins, species == "Adelie")
ggplot(adelie, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  geom_smooth(method = "lm") +
  theme_minimal()
```

The function version:

```
# generic penguin function
plot_penguins <- function(data) {
  ggplot(data, aes(flipper_length_mm, body_mass_g)) +
    geom_point() +
    geom_smooth(method = "lm") +
    theme_minimal()
}
# apply the function to the Adelie data
adelie <- filter(penguins, species == "Adelie")
plot_penguins(adelie)
```

Default arguments

- Arguments with default values don't have to be specified when calling the function
- They can be used for things that usually stay the same, but sometimes are changed
- Default arguments are defined in the function body with `arg = default`

```
# Define a default color for penguin data points
plot_penguins <- function(data, color = "black") {
  ggplot(data, aes(flipper_length_mm, body_mass_g)) +
    geom_point(color = color) +
    geom_smooth(method = "lm") +
    theme_minimal()
}
# Call the function with the default black color
plot_penguins(adelie)
# Call the function with a custom color
plot_penguins(adelie, color = "red")
```

Live demo

Build `plot_penguins()`

`demo/01_functions.R`

Your turn

> ✏️ Exercise 1 (~7 min):
>
> Open `exercises/01_functions_exercise.R`

Iteration with the `purrr` package

Base R iteration: for-loops and `apply()`

Run the same function for multiple inputs, e.g.

```
sqrt(1)
sqrt(2)
sqrt(3)
```

Base R iteration options:

```
# for loop
for (x in 1:3) {
  sqrt(x)
}

# Functions from the apply family
lapply(1:3, sqrt) # a list
sapply(1:3, sqrt) # a vector
```

The `purrr` package

- Part of the `tidyverse` packages
- Provides functions for functional programming and iteration
- Why use it?
  - More consistent than base R `apply` functions
  - More flexible than base R iteration
  - Integrates well in a tidyverse workflow

The main function: map

`map(x, f)` calls `f` **once per element** of `x` and gives back **one result per element**.

```
# x: a list or vector to iterate over
# f: a function to apply to each element of x
map(x, f)
```

```
# The map function always returns its result as a list
library(purrr)
map(1:3, sqrt)
```

```
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051
```

The `map_*()` variants

The suffix says what is the output of the function:

- `map()` returns a list
- `map_dbl()` returns a numeric vector
- `map_chr()` returns a character vector
- `map_lgl()` returns a logical vector
- `map_int()` returns an integer vector

For example:

```
# return result as list (always works)
map(1:3, sqrt)
# return results as double vector (only works if results are single numbers)
map_dbl(1:3, sqrt)
```

More complex situations

What if it's not so simple, e.g. my iterator is not the first argument?

```
# round the number pi to a different number of digits
round(pi, digits = 1)
round(pi, digits = 2)
round(pi, digits = 3)
```

Just using `map` with `round` does not work -> Where does `pi` go?

```
# iterator is passed to the first argument not as `digits`
map(1:3, round)
```

Solution: Anonymous function

Solution: Write an anonymous function on the spot:

```
map(1:3, \(x) round(pi, digits = x))
# same as
map(1:3, function(x) round(pi, digits = x))
```

- `\(x)` syntax is a shortcut for `function(x)`
- `x` is the argument that takes the value of each element of the iterator

Live demo

Map over columns, write an anonymous function, and read every CSV in a folder.

`demo/02_map.R`

Your turn

> ✏️ Exercise 2 (~7 min):
>
> Open `exercises/02_map_exercise.R`

`map()` with your own functions

Put the two skills together

We can plot one species with our function from earlier:

```
plot_penguins(filter(penguins, species == "Adelie"))
```

But how do we do this for all three species? -> Map needs a list of data frames, one per species.

`split()` into a named list

The base R function `split()` does exactly that:

```
# split the penguins data into a named list of data frames, one per species
by_species <- split(penguins, penguins$species)

names(by_species)        # "Adelie" "Chinstrap" "Gentoo"
```

```
[1] "Adelie"    "Chinstrap" "Gentoo"
```

```
by_species[["Adelie"]]   # each element is a data frame
```

```
# A tibble: 152 × 8
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# ℹ 142 more rows
# ℹ 2 more variables: sex <fct>, year <int>
```

`map()` over the list of data frames

```
plots <- map(by_species, plot_penguins)
# Look at one individual plot
plots[["Adelie"]]
```

`map` works very well with the pipe operator, so we can do everything in one chain:

```
plots <- penguins |>
  split(penguins$species) |>
  map(plot_penguins)
```

The same structure, anything per group

Run a linear model for each species and pull out the R^2 values:

```
penguins |>
  split(penguins$species) |>
  map(\(df) lm(body_mass_g ~ flipper_length_mm, data = df)) |>
  map(summary) |>
  map_dbl(\(x) x$r.squared)
```

Live demo

`demo/03_map_functions.R`

Your turn

> ✏️ Exercise 3 (~7 min):
>
> Open `exercises/03_map_functions_exercise.R`

Saving results with `walk()`

`map` vs `walk`

Both functions work in the same way, but they have different purposes:

- `map()` → you want the **results** back
- `walk()` → you want a **side effect** (print a plot, save a file)

Example: print all the plots

```
# Create all plots and save them in a list -> results we want back
plots <- penguins |>
  split(penguins$species) |>
  map(plot_penguins)

# Take the list of plots and print them -> side effect we want (the printing)
walk(plots, print)              # draw each plot
```

Example: save all the plots

We can also `walk()` over two inputs at the same time with `walk2()`: the list of plots (x) and their names (y).

```
# Walk 2 iterates over 2 inputs (x, y) in parallel
walk2(plots, names(plots), function(x, y) {ggsave(
  paste0("plots/", y, ".png"), x
)})
```

Live demo

`demo/04_walk.R`

Your turn

> ✏️ Exercise 4 (~7 min, optional / take-home):
>
> Open `exercises/04_walk_exercise.R`

Wrap-up

Function

- Functions are reusable pieces of code that take inputs and return outputs
- Default arguments are defined with `arg = value` in the function body

For example

```
# y is optional and defaults to 1
add_numbers <- function(x, y = 1) {
  x + y
}
```

Iteration with the purrr package

`map(x, f)` applies function `f` to each element of `x` and returns a list

```
map(1:3, sqrt) # returns a list of square roots
map_dbl(1:3, sqrt) # returns a numeric vector of square roots
```

You can use `map` with your own functions, and you can write anonymous functions on the spot if needed.

```
map(1:3, \(x) round(pi, digits = x))
```

The `walk()` function is the same as map but for side effects (printing, saving files) instead of returning results.

```
walk(plots, print) # prints each plot
```

Where next

- More inputs at once: `map2()`, `pmap()`
- When some iterations fail: `safely()`, `possibly()`
- Going parallel: purrr's `in_parallel()` or the `furrr` package

Resources: purrr cheatsheet and purrr documentation

Next lecture/workshop

Topic: Code sharing and reproducible repositories

🗓️ 16.07.2026 🕓 4-5 p.m. 📍 Webex

🔔 Subscribe to the mailing list

📧 For topic suggestions and/or feedback send me an email

The end :)

Questions?