# Advanced R

# Session overview

1. Efficient iteration with purrr
2. Programming with data frames
3. Quarto and tables
4. Regular expressions
5. Going further

*"Don, we need to scream."*

# Efficient iteration

In Session 2, we introduced loops as a way of repeatedly manipulating an 'iterable' object, like a vector or a list.

```
age <- c(28, 55, 19, 73, 41)
age_sq <- vector(mode = "numeric",
                 length = 5)

for (i in 1:length(age)) {
  age_sq[i] <- age[i]^2
}
```

I also said that you should almost never use loops.

So, what other options are there?

# Vectorised functions

Many functions in R are 'vectorised'.

They take a vector as input, and return a vector (or single value) as the output. For example:

```
age <- c(28, 55, 19, 73, 41)

# Apply a function to each element
sqrt(age)

# Apply a function to the entire vector
mean(age)
```

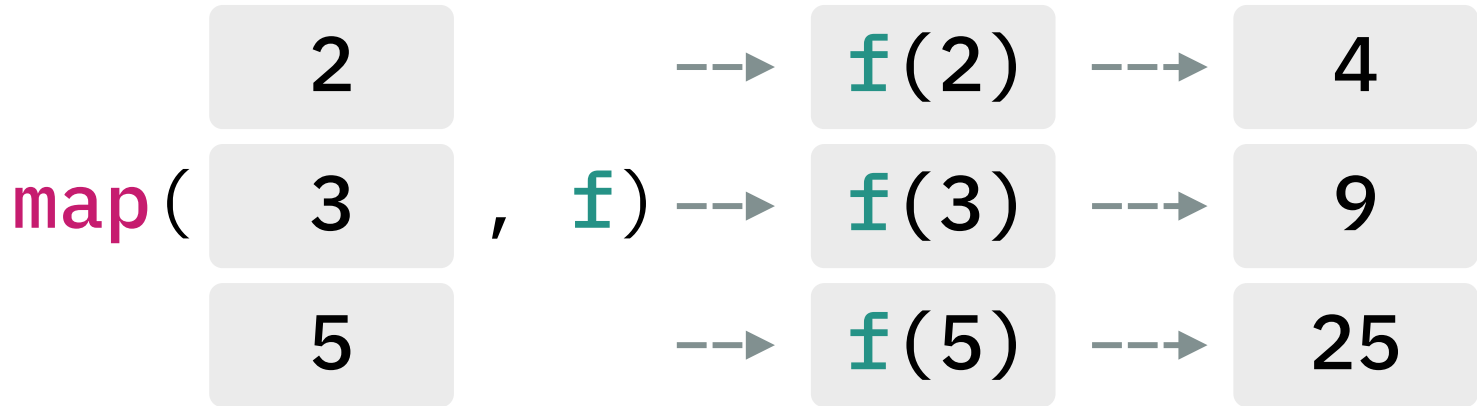
Where possible, use vectorised functions.

# Functional programming

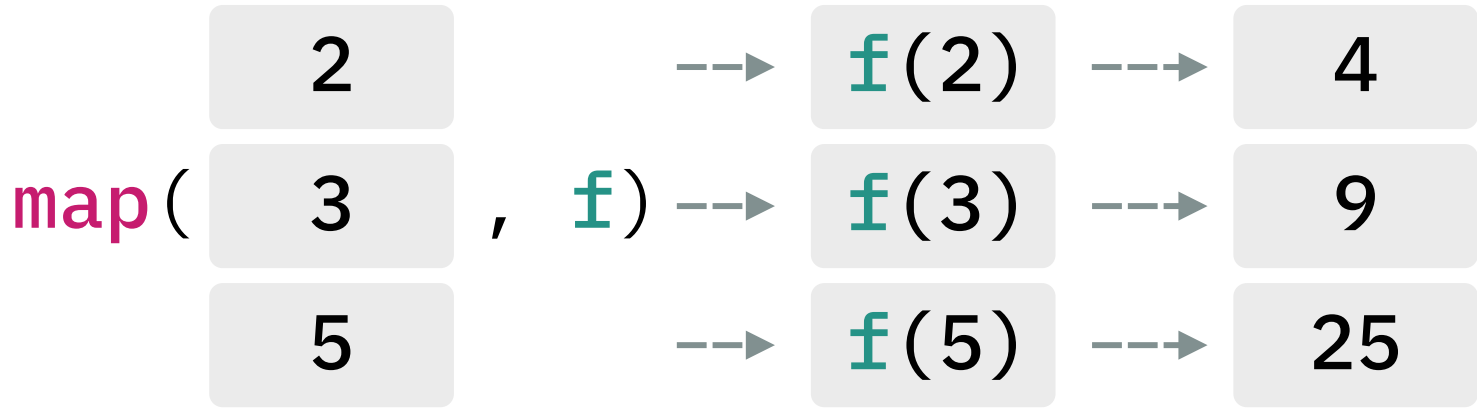
> “A functional is a function that takes a function as an input and returns a vector as an output.”
>
> https://adv-r.hadley.nz/functionals.html

Given a sequence of things, apply a function to each element.

```
f <- function(x) x^2
```



```
f <- function(x) x^2
```




We'll consider two approaches:

- The `apply` family (base R)
- The `purrr` package.

# The `apply` family

A group of functions for applying functions to vectors, lists, matrices and, arrays:

- `apply()`
- `lapply()`
- `sapply()`
- `vapply()`
- `mapply()`
- `rapply()`

(I forget what they all do).

# lapply

Apply a function to each element in a vector or list.

```
> x <- c(2, 5, 3, 9)
> lapply(x, sqrt)
[[1]]
[1] 1.414214

[[2]]
[1] 2.236068

[[3]]
[1] 1.732051

[[4]]
[1] 3
```

# Example: Convert strings to uppercase

```
> people <- list("Alice",
>                "Bob",
>                "Charlie")
> people_upper <- lapply(people, toupper)

> people_upper
[[1]]
[1] "ALICE"

[[2]]
[1] "BOB"

[[3]]
[1] "CHARLIE"
```

# Example: Read a folder of CSV files

```
> library(tidyverse)
> library(fs)
>
> files <- dir_ls(glob = "*.csv")
> lapply(files, read_csv)
>
> # Or with the pipe
> files |>
>  lapply(read_csv)
```

# apply

Apply a function to a matrix or data frame.

```
> m <- matrix(sample(1:100, 9), ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   63   72   88
[2,]   38   30   97
[3,]   21   35   10

> apply(m, 1, mean)
[1] 74.33333 55.00000 22.00000

> apply(m, 2, mean)
[1] 40.66667 45.66667 65.00000
```

The `MARGIN` argument controls whether to apply the function to the rows ( `1` ), columns ( `2` ) or both ( `c(1, 2)` ).

# purrr

A modern replacement for the `apply` family.


> “purrr enhances R’s functional programming toolkit by providing a complete and consistent set of tools for working with functions and vectors”.
>
> https://purrr.tidyverse.org

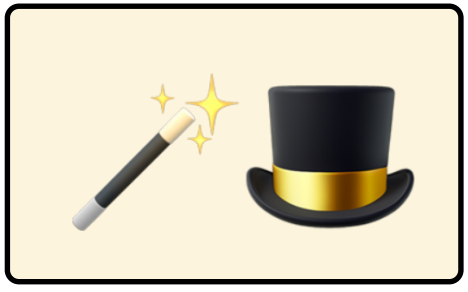
(It’s my favourite thing about R).

Your vector or list

1 2 3 4 5 6

+

Your function



Result

# `map`

`map` applies a function to each element in a vector or list.

```
map(list, function)
```

- `map` takes a vector or list as input.
- By default, `map` returns a list (like `lapply`), but we can changes this.

# Examples

```
> library(tidyverse)
> x <- c(5, 11, 21)
> map(x, sqrt)
[[1]]
[1] 2.236068

[[2]]
[1] 3.316625

[[3]]
[1] 4.582576
```

# Examples

```
> add_three <- function {
>   x + 3
> }
> x <- c(5, 11, 21)
> map(x, add_three)
[[1]]
[1] 8

[[2]]
[1] 14

[[3]]
[1] 24
```

# Three features of `map()`

1. We can return the results in different formats.
2. We can use temporary functions.
3. We can transform the results.

# 1. Returning results in different formats

`map()` returns a list, but we have other options.

| Syntax | Return format |
| --- | --- |
| `map()` | list |
| `map_lgl()` | logical vector |
| `map_int()` | integer vector |
| `map_dbl()` | double vector |
| `map_chr()` | character vector |

# Examples

```
> add_three <- function {
>   x + 3
> }
> x <- c(5, 11, 21)
> map_dbl(x, add_three)
[1]  8 14 24
```

# 2. Temporary functions

`map` can be used with existing functions:

```
> cube <- function(x) x^3
> map_dbl(1:5, cube)
[1]   1   8  27  64 125
```

# 2. Temporary functions

`map` can be used with existing functions:

```
> cube <- function(x) x^3
> map_dbl(1:5, cube)
[1]   1   8  27  64 125
```

But we can also use temporary functions:

```
> map_dbl(1:5, \(x) x^3)
[1]   1   8  27  64 125

> map_chr(c("morning", "evening"),
>         \(x) paste("Good", x))
[1] "Good morning"
[2] "Good evening"
```

Syntax for temporary functions:

```
map(list, \(x) x + 1)
map(list, \(thing) thing * 2)
```

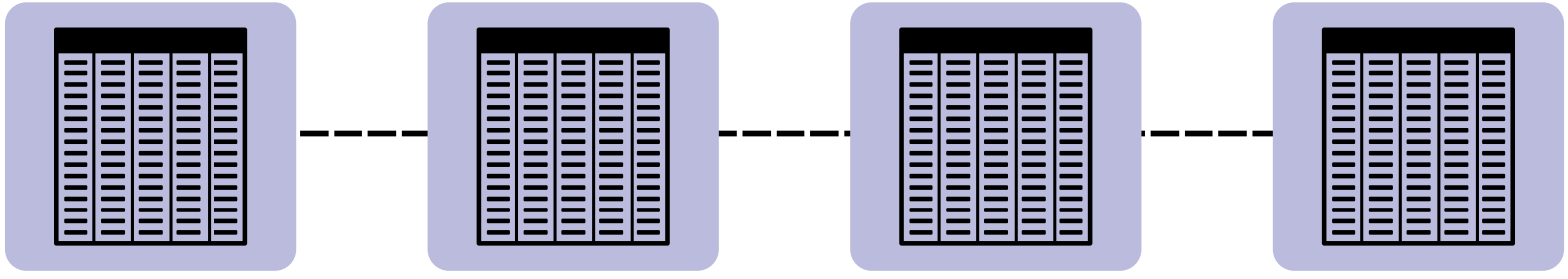
Or for a function spanning multiple lines:

```
map(list, \(x) {
    if (!is.numeric(x)) {
        stop("Input must be numeric")
    }
    return(x^2)
})
```

# 3. Transforming the results

We often need to work with lists of data frames, e.g.,

```
# List all CSV files in the current directory
dir_ls(glob = "*.csv") |>
  map(read_csv)   # Import each file as a
                  # data frame
```



But then what? Suppose we want to combine the list of data frames into a single data frame.

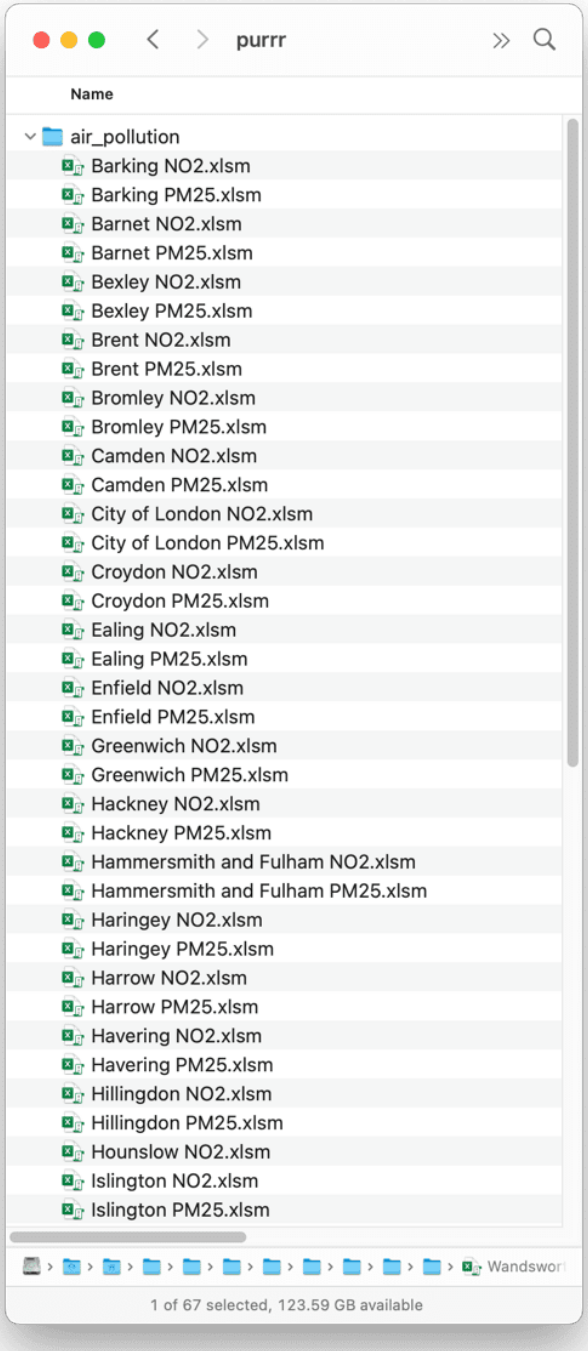
For this we can use:

| | |
|---|---|
| `list_rbind` | Combines elements into a data frame by row-binding them together. |
| `list_cbind` | Combines elements into a data frame by column-binding them together. |

```
> dir_ls() |>
>   map(read_csv) |>
>   list_rbind()
```

Use the `names_to` argument to retain the element names.

## Demonstration




Our tasks:

- ❏ Import and clean each spreadsheet.
- ❏ Combine 67 sheets into a single data frame.
- ❏ From the filename, extract “Borough” and “measure” (i.e., PM25 or NO2).

# Extensions to `map()`

You won't need these immediately, but useful to know.

| | |
|---|---|
| `map2()` | Apply a function to two parallel inputs<br>Example: BMI calculation<br>`map2(weight, height, \(x, y) x / (y^2))` |
| `walk()` | Similar to `map()`, but used for side effects<br>Example: Export a CSV for each patient<br>`walk(patient_ids, write_csv)` |
| `map_if()` | Apply a function conditionally<br>Example: Square root for numbers only<br>`map_if(values, is.numeric, sqrt)` |

See the package index for more:

https://purrr.tidyverse.org/reference/index.html

# Where to learn more?

1. Read the “Iteration” chapter in “R for Data Science”

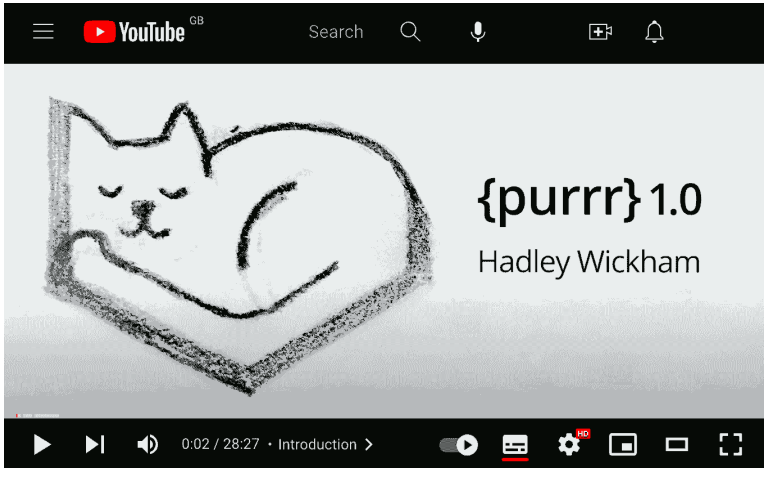
   `https://r4ds.hadley.nz/iteration.html`

2. Read “Chapter 9: Functions” from Advanced R

   `https://adv-r.hadley.nz/functionals.html`

3. Watch this video

   `https://www.youtube.com/watch?v=EGAs7zuRutY`



# Wrapping up

- purrr is a powerful tool for manipulating lists and vectors.
- It's an advanced topic; you don't need it.
- It has redefined how I use R.

Everything is a list.

```r
library(tidyverse)
library(broom)
library(palmerpenguins)
library(tinytable)

penguins |>
  group_split(species) |>
  map(\(d) {
    lm(body_mass_g ~ bill_length_mm,
       data = d) |>
      tidy(conf.int = TRUE) |>
      mutate(group = d$species[1])
  }) |>
  list_rbind() |>
  tt()
```

1. Split data frame by ‘species’ to produce a list of data frames
2. Fit a model for each group
3. Extract the coefficients from each model into list of data frames
4. Combine the coefficients from all models into a single data frame
5. Create a table with the coefficients

# Practical