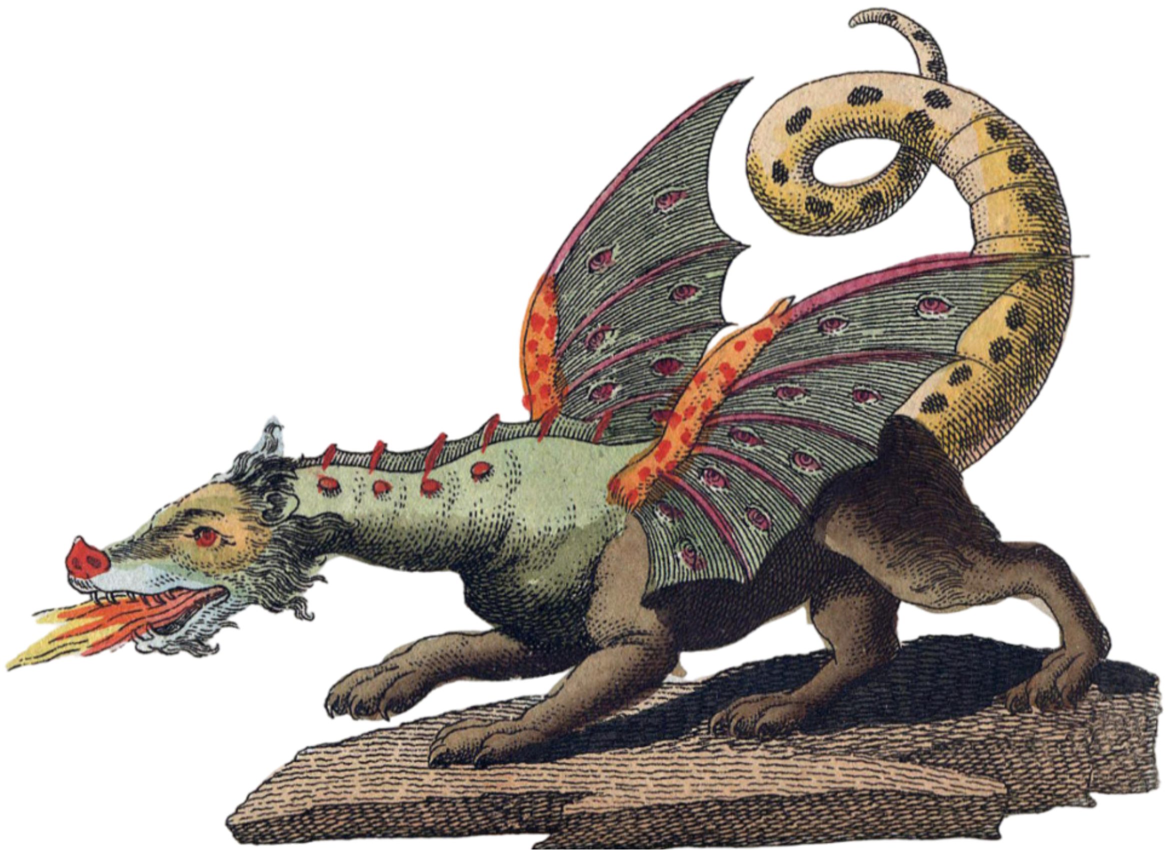
R Programming

Dr. Edgar J. Treischl

Last update: 2025-11-22



Press ⭕ or ➡️ to navigate

Agenda

01 Base R

02 Purrr

03 Purrr and friends


Base R is a Real Beauty 😄
GOT
HBO

R Basics

Functions for data exploration

```r
# View first rows / head of mtcars
head(mtcars)
```

```
##                    mpg cyl disp  hp drat
wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90
2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90
2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85
2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08
3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15
3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76
3.460 20.22  1  0    3    1
```

```r
# Structure of mtcars
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3
24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62
95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76
3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

R Basics II

Functions for data manipulation

```r
# Class of mtcars
class(mtcars)
```

```
## [1] "data.frame"
```

```r
# Create a vector/Subset the 2nd element
myvec <- c("This", "is", "awesome")
myvec[3]
```

```
## [1] "awesome"
```

```r
# Subset by condition
head(mtcars[mtcars$cyl == 6, ])
```

```
##                 mpg cyl  disp  hp drat    wt
qsec vs am gear carb
## Mazda RX4      21.0   6 160.0 110 3.90 2.620
16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875
17.02  0  1    4    4
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215
19.44  1  0    3    1
## Valiant        18.1   6 225.0 105 2.76 3.460
20.22  1  0    3    1
## Merc 280       19.2   6 167.6 123 3.92 3.440
18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440
18.90  1  0    4    4
```

R Basics III

⬅️ Assignment Operator

```r
# AB(C) of the assignment operator
a <- 5
b <- 6

# The result
result <- a + b
result
```

```
## [1] 11
```

Assign like a Pro, press:

```
#> < Alt/Option > + < - >
#> (Windows/Unix)
```

🧰 Create your own functions

```r
# Create a function
randomize <- function(x) {
  sample_x <- sample(x, 1)
  return(sample_x)
}

# Call and feed the function
randomize(x = c(3, 2, 1, 5, 8, 12, 1))
```

```
## [1] 2
```

Snippet your way, in RStudio, type:

```
#> `fun` and press `<Tab>`
```

R Basics Summary

- **Data Exploration**
  - `head/tail(x)` – view first/last rows
  - `str(x)` – structure of an object
  - `names(x)` – column or element names
  - `class(x)` – object class
- **Indexing and Subsetting**
  - `x[i]` – subset vector
  - `x[i, j]` – subset matrix/data.frame
  - `x[condition]` – logical indexing
  - `subset(x, condition)` – filter rows
- **Vector & Sequence Operations**
  - `c()` – combine values into a vector
  - `seq()`, `rep()` – generate sequences/repeats
  - `sort(x)` – sort a vector
  - `rev(x)` – reverse elements
  - `length(x)` – length of a vector
- **Basic Math & Statistics**
  - `sum(x)`, `prod(x)` – sum/product
  - `mean(x)`, `median(x)`, `var(x)`, `sd(x)` – statistics
  - `min(x)`, `max(x)` – minimum/maximum
  - `round(x, digits)` – round numbers
- **Logical & Comparison**
  - `==`, `!=`, `<`, `>`, `<=`, `>=` – comparisons
  - `&`, `|`, `!` – logical AND, OR, NOT
  - `any()`, `all()` – check logical conditions over vectors
- **Object Manipulation**
  - `cbind()`, `rbind()` – combine matrices or data frames
  - `merge()` – join data frames
  - `dimnames()` – get/set row/column names
  - `replicate()` – repeat expressions

Base R: Functions

1️⃣ Functions as first-class objects:

You can assign functions to variables, pass them as arguments, and return them from other functions

```
# Assigning a function
f <- sum
f(1:5)
```

```
## [1] 15
```

```
# Passing a function as an argument
apply_fun <- function(x, fun) {
  fun(x)
}

apply_fun(1:5, mean)
```

```
## [1] 3
```

👻 Anonymous functions:

Functions without names. Anonymous functions are created on the fly using the `function` keyword and can be passed directly as arguments to other functions

```
# Anonymous functions
sapply(1:5, function(x) x^2)
```

```
## [1]  1  4  9 16 25
```

Base R: Functions II

🧽 Pure functions:

Functions without side effects (e.g. data export, printing), producing predictable results. *Impure functions* break predictability, reproducibility, testability, and parallel safety.

```r
# Impure function
x <- 5

add_to_x <- function(y) {
  x <<- x + y  # modifies a global variable
  return(x)
}

add_to_x(3)
```

```
## [1] 8
```

```r
add_to_x(2)
```

🌀 Closures/lexical scoping:

A closure is a function that captures (or “closes over”) variables from its surrounding environment — meaning it remembers the values that existed when it was created.

```r
# A closure
make_power <- function(n) {
  function(x) x^n
}

square <- make_power(2)
square(4)
```

```
## [1] 16
```

Base R: Interation

⚙️ Apply family

Function like `lapply()`, `sapply()`, `vapply()` enable iteration without loops. For example, sapply() applies a function to each element of a vector (matrix) and simplifies the output when possible.

```
# Apply family: Sapply
sapply(iris[1:4], mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length
Petal.Width
##     5.843333     3.057333     3.758000
1.199333
```

```
# Apply family: Mapply
mapply(sum, 1:3, 4:6)
```

```
## [1] 5 7 9
```

👉 Sapply makes it hard to predict the output

```
mylist <- list(1:3, 4:6)
class(mylist)
```

```
## [1] "list"
```

```
#> 01
mylist <- sapply(mylist, sum)
class(mylist)
```

```
## [1] "integer"
```

```
#> 02
mylist <- sapply(mylist, range)
class(mylist)
```

```
## [1] "matrix" "array"
```

Base R: Loops

🔁 While-loop

Repeat operations *as long as a condition is TRUE*.

```r
# While-loop
count <- 1
result <- c()

while(count <= 5) {
  result <- c(result, count^2)
  count <- count + 1
}

result
```

```
## [1]  1  4  9 16 25
```

Useful when the number of iterations is not known in advance, or when looping depends on a changing condition.

🔂 For-loop

Repeat operations over a sequence of values.

```r
# For-loop
squares <- numeric(5)

for(i in 1:5) {
  squares[i] <- i^2
}

squares
```

```
## [1]  1  4  9 16 25
```

Useful for iterative tasks, but vectorized alternatives (like `sapply`) are often more concise and faster.

Base R (4/4)

🔢 Vectorization

It enables performing operations on entire vectors at once, avoiding explicit loops. Leads to **faster, simpler, and more readable code**.

```r
x <- 1:5

# Loop version (slower)
squares_loop <- numeric(length(x))
for(i in seq_along(x)) {
  squares_loop[i] <- x[i]^2
}

# Vectorized version (faster)
squares_vec <- x^2
squares_vec
```

```
## [1]  1  4  9 16 25
```

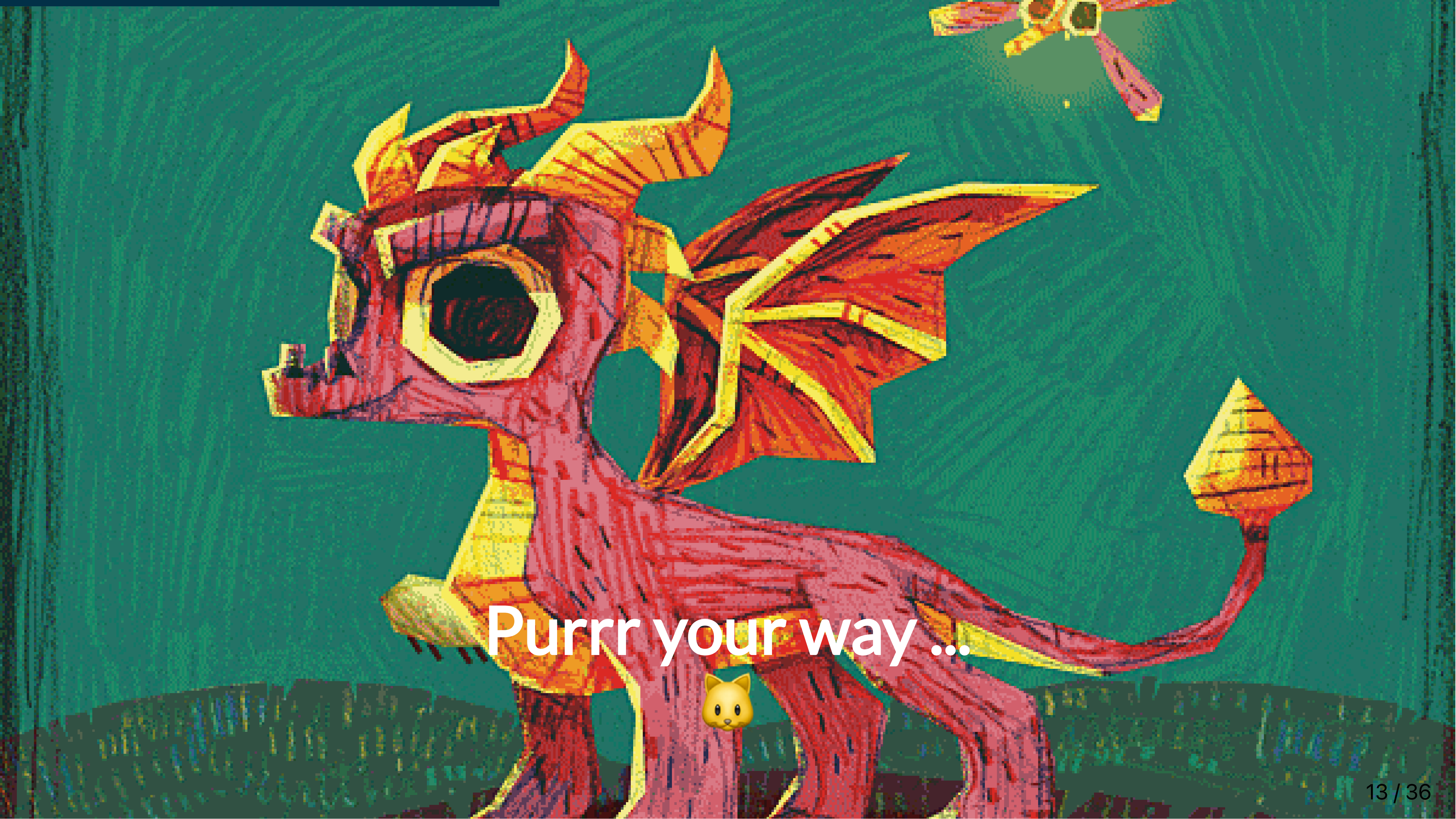
⛓️ Pipe operator (%>%)

It passes the output of one expression as the input to the next, enabling **clear, readable, and sequential transformations**.

```r
#> Make the native pipe (|>) the default
#> Cmd + Shift + M

mtcars |>
  filter(cyl == 6) |>
  mutate(power_to_weight = hp / wt) |>
  summarise(avg_ptw = mean(power_to_weight))
```

```
##    avg_ptw
## 1 39.92794
```

Pipelines are great for creating linear, readable code, but when a single chain becomes too long or complex, it's better to break it into smaller, named steps for clarity.



Purrr your way ...

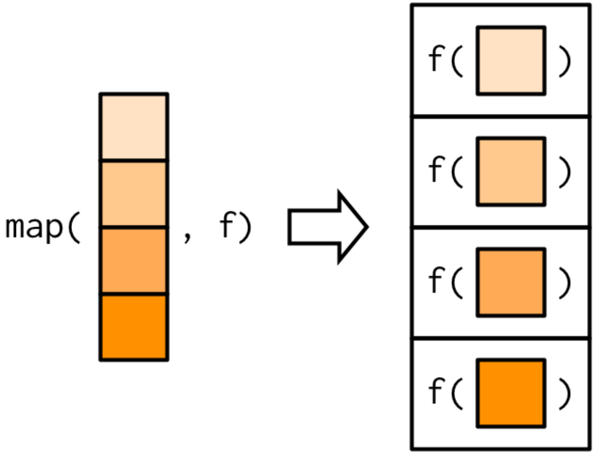
🐱

Hello Purrr

The purrr package helps you avoid repetitive code (Wickham and Henry 2025). The package is part of the tidyverse and provides a consistent, readable way to perform iteration in R. Instead of writing loops or repeating function calls manually, purrr lets you apply functions over lists, vectors, or data frames using a family of tools like map, map2, and pmap.

map

The heart of `purrr` is its **mapping functions**, which let you apply a function systematically across a list, vector, or multiple inputs.



Artwork: Hadley Wickham

The `map()` function applies a function to each element of a list or vector and **always returns a list**.

```r
# Create a list of numeric vectors
num_list <- list(a = 1:5,
                 b = 6:10,
                 c = 11:15)

# Calculate the mean of each vector
map(num_list, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 8
##
## $c
## [1] 13
```

map

Each result is a list element, keeping the structure consistent. If you want a specific type of output, `purrr` provides **type-specific variants** that guarantee the output type, making your code more predictable and reducing subtle bugs.

- `map_dbl()` – returns a numeric vector
- `map_chr()` – returns a character vector
- `map_int()` – returns an integer vector
- `map_lgl()` – returns a logical vector
- `map_df()` – returns a data frame

```
# Using type-specific variants
map_df(mtcars, mean)
```

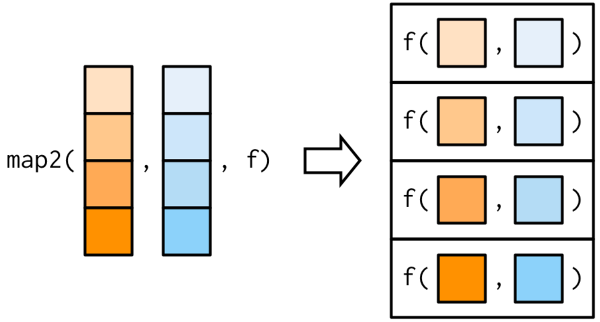
```
## # A tibble: 1 × 11
##     mpg   cyl  disp    hp  drat    wt  qsec
vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
<dbl> <dbl> <dbl> <dbl>
## 1  20.1  6.19  231.  147.  3.60  3.22  17.8
0.438 0.406  3.69  2.81
```

```
map_chr(mtcars, class)
```

```
##       mpg       cyl      disp        hp
drat        wt      qsec        vs
## "numeric" "numeric" "numeric" "numeric"
"numeric" "numeric" "numeric" "numeric"
##        am      gear      carb
## "numeric" "numeric" "numeric"
```

map2 – iterating over two inputs

`map2()` is used when you want to **apply a function to two vectors or lists at the same time**, element by element.



Artwork: Hadley Wickham

So, `map2()` takes three main arguments:

1. The first input vector or list (`.x`)
2. The second input vector or list (`.y`)
3. The function to apply to each pair of elements

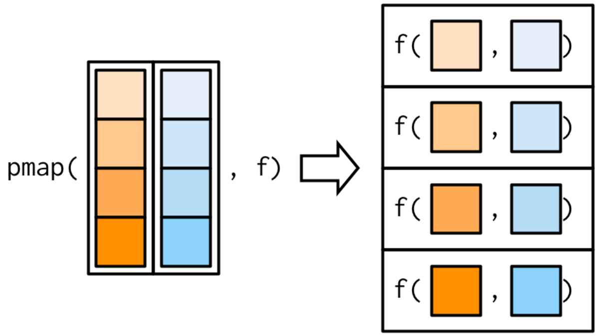
```
column_names <- names(mtcars)
column_means <- map_dbl(mtcars, mean)

map2result <- map2_chr(
  column_names,
  column_means, ~ paste(
    .x, "mean =",
    round(.y, 1)
  )
)
map2result[1:3]
```

```
## [1] "mpg mean = 20.1"   "cyl mean = 6.2"
"disp mean = 230.7"
```

Here, `.x` refers to an element from the first vector (`column_names`), `.y` refers to the corresponding element from the second vector (`column_means`); the tilde (`~`) creates an anonymous function.

pmap – iterating over multiple inputs

When you have **more than two inputs**, `pmap()` generalizes this idea. It takes a **list of vectors or lists** and applies a function to corresponding elements across all inputs.



Artwork: Hadley Wickham

Imagine we want to combine column names, means, and standard deviations:

```
column_sds <- map_dbl(mtcars, sd)
inputs <- list(column_names, column_means,
column_sds)

pmap_result <- pmap(
  inputs,
  ~ paste(
    ..1, " with a mean =", round(..2, 1),
    " and a sd =", round(..3, 1)
  )
)
```

Here, `..1`, `..2`, `..3` refer to elements from the first, second, and third lists, respectively.

pmap II

```r
# Show the first 3 results of the list
pmap_result[1:3]
```

```
## [[1]]
## [1] "mpg  with a mean = 20.1  and a sd = 6"
##
## [[2]]
## [1] "cyl  with a mean = 6.2  and a sd = 1.8"
##
## [[3]]
## [1] "disp  with a mean = 230.7  and a sd = 123.9"
```

👉 **`pmap()` allows you to work cleanly with any number of parallel inputs, keeping your code short and readable while avoiding nested loops.**

FP in Action ✊

Example: Reading and Cleaning Data

Imagine, a folder full of CSV files, all with the same structure, and we need to import them and clean the column names.

The manual ✋ approach

```
df1 <- read.csv("df1.csv")
df1 <- janitor::clean_names(df1)
...
df101 <- read.csv("df101.csv")
```

Don't: This is tedious and error-prone!

The ➿ loooop

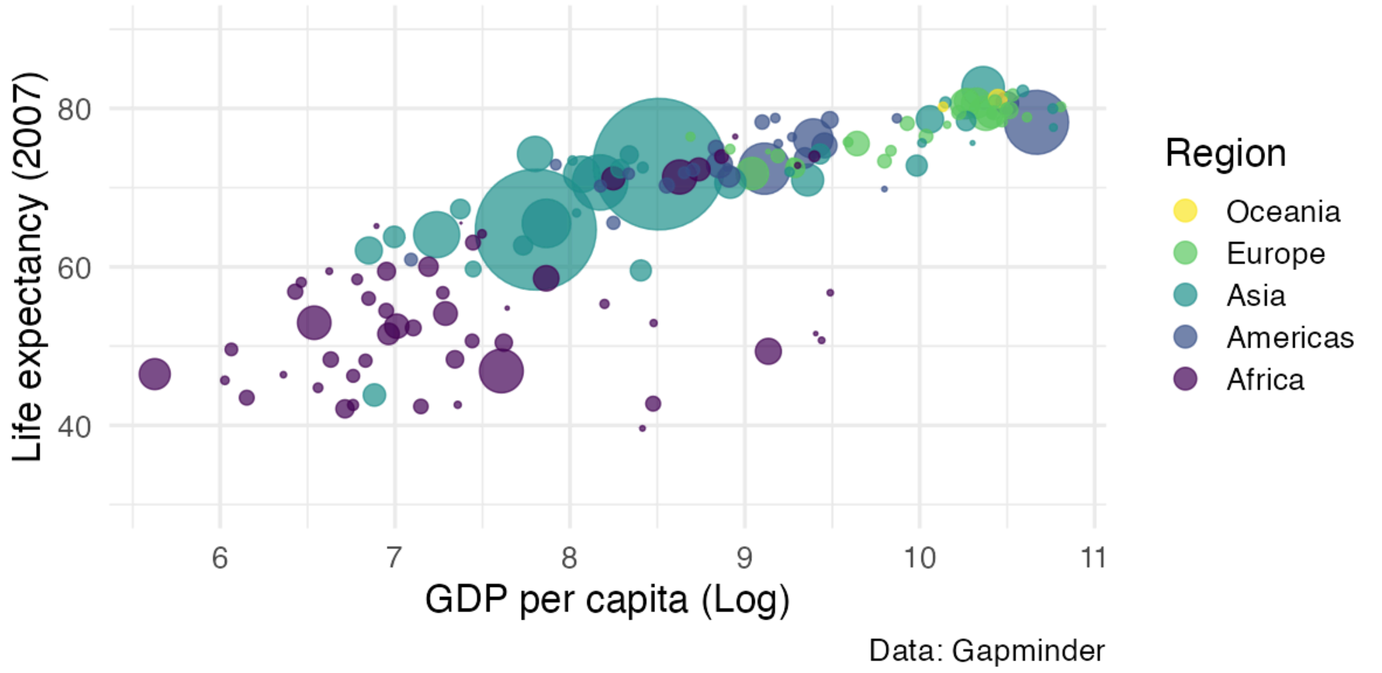
```
files <- list.files("data", pattern =
"*.csv", full.names = TRUE)
dfs <- list()

for (i in seq_along(files)) {
  df <- read.csv(files[i])
  df <- janitor::clean_names(df)
  dfs[[i]] <- df
}
```

The loop works, better than a manual approch, but there are some downsides:

- You have to manually manage the **index** `i` and the `dfs` list.
- The code is **verbose**, with variables that can clutter your workspace.
- It's **less composable**, making it harder to fit into a pipe-based workflow or reuse as a function.

Gapminder

- The Gapminder data contains country-level statistics on life expectancy, GDP per capita, and population over time.



- I splitted the original dataset into separate files by country and year.
- Your task is to read all CSV files and combine them into a single cleaned data frame.

Example: Reading and Cleaning Data

Function programming with purrr is often cleaner, safer, and easier to maintain:

- **No manual indexing** — `map()` takes care of iterating over each file automatically. No need to track `i` or worry about list positions.
- **Clean and readable** — the workflow reads top-down: read the file, clean the names, done. No clutter from intermediate variables.
- **Predictable and type-safe** — `map()` always returns a list, so `dfs` is consistent and reliable.
- **Easily reusable** — wrap this logic in a function, and you can apply it to any folder of files without rewriting the loop.

Hallo 🐱

```r
library(purrr)
library(janitor)

files <- list.files("data", pattern = "*.csv", full.names = TRUE)

dfs <- files |>
  map(read.csv) |>
  map(clean_names)
```

imap

`imap()` is a variant of `map()` that supplies **both the element and its name (or index)** to the function.
This makes it useful when you need to work with **values + names** at the same time.

`imap()` applies a function to each element of a list or vector and passes two arguments:

- `.x` = the value
- `.y` = the name (or index)

It **always returns a list**, like `map()`.

```r
# A named list
num_list <- list(a = 1:3,
                 b = 4:6,
                 c = 7:9)

# Use imap() to create labeled summaries
imap(num_list, ~ paste0("Mean of ", .y, ": ",
mean(.x)))
```

```
## $a
## [1] "Mean of a: 2"
##
## $b
## [1] "Mean of b: 5"
##
## $c
## [1] "Mean of c: 8"
```

Example: Creating Multiple Plots

- `imap()` iterates over both the **values** and **names** of each column in `mtcars`.
- For every column (`.x`), it checks if it's numeric using `is.numeric(.x)`.
- If it is, a `ggplot` histogram is created.
- The column name (`.y`) is used dynamically in the plot title.
- All resulting plots are stored neatly in a **list object** called `plots`, ready for display or export.

```
library(ggplot2)
library(purrr)

# Create a histogram for each numeric column
plots <- imap(mtcars, ~ {
  if(is.numeric(.x)) {
    ggplot(mtcars, aes(x = .x)) +
      geom_histogram(bins = 10, fill =
"steelblue") +
      ggtitle(paste("Histogram of", .y))
  }
})

# A list of plots
class(plots)
```

```
## [1] "list"
```

When iterating over many elements, not every operation will go smoothly — some columns might be non-numeric, a plot might fail to render, or a transformation could throw an error.

Purrr and friends to slay the 🐉

dplyr::across

Across allows you to apply one or more functions to multiple columns inside `mutate`, `summarise`, `filter`, etc.

- `.cols`: which columns to target (by name, tidyselect helpers, or predicates)
- `.fns`: the function(s) to apply (formula: `~ .x`, lambda: `\(x) x + 1`, or a list of functions)
- `.names`: optional names for new columns

```
# Basic syntax
across(.cols, .fns, ..., .names = NULL)
```

👉 **Think of it as map over columns — the *column-wise equivalent* of `map()`.**

```
# Neah ....
mtcars |> summarise(
  mpg = mean(mpg),
  cyl = mean(cyl),
  disp = mean(disp)
  )
```

```
##        mpg    cyl     disp
## 1 20.09062 6.1875 230.7219
```

```
# Yah ...
mtcars |> summarise(
  across(c(mpg, cyl, disp), mean)
)
```

```
##        mpg    cyl     disp
## 1 20.09062 6.1875 230.7219
```

Across in Action

🔍 Where ...

allows you to select columns based on their type (`is.numeric`, `is.character`, `is.Date`, etc.)

```
summarise(mtcars,
    across(where(is.numeric), mean)
  )
```

```
##          mpg     cyl     disp        hp      drat
wt      qsec      vs       am
## 1 20.09062 6.1875 230.7219 146.6875 3.596563
3.21725 17.84875 0.4375 0.40625
##      gear   carb
## 1 3.6875 2.8125
```

👉 You can select columns using other **tidyselect** helpers, such as: `starts_with`, `ends_with`, `contains`, `matches` `everything`.

📝 Contains ...

... selects columns whose names include a string; `matches` selects columns whose names match a regex.

```
summarise(mtcars,
    across(contains("m"), mean)
  )
```

```
##          mpg      am
## 1 20.09062 0.40625
```

🧩 Matches

```
 summarise(mtcars,
    across(matches("^m"), mean)
  )
```

```
##          mpg
## 1 20.09062
```

safely

`safely()` **catches errors** and returns them as part of the output, rather than halting execution. It transforms a function that always returns a **list** with two elements:

- `result`: the output (or `NULL` if an error occurred)
- `error`: the error message (or `NULL` if everything worked)

```
# Some test inputs
data_list <- list(
  c("a", "b", "c"),
  c(1, 2, 3)
)

mean(data_list[[1]])
```

```
## Warning in mean.default(data_list[[1]]):
argument is not numeric or logical:
## returning NA
```

```
## [1] NA
```

```
# Wrap mean() safely
safe_mean <- safely(mean)

# Apply safely-wrapped mean() to each element
results <- map(data_list, safe_mean)

# Extract via map
numeric_results <- map(results, "result")
errors <- map(results, "error")

# Keep only successful numeric results and
simplify to a vector
flatten_dbl(compact(numeric_results))
```

```
## [1] NA  2
```

possibly

`possibly()` **replaces errors with a default value** rather than halting execution. It transforms a function so that if an error occurs, it returns a **fallback value** you specify (via `.otherwise`).

This is useful when you want to keep going but don't need the error details.

```r
# Function
do <- function(x) {
  if (!is.numeric(x)) stop("Not numeric!")
  mean(x)
}

# Test inputs
data_list <- list(
  "a",
  c(1, 2, 3)
)
```

```r
# Wrap with possibly()
safe_test2 <- possibly(do, otherwise =
NA_real_)
map_dbl(data_list, safe_test2)
```

```
## [1] NA  2
```

Here, instead of stopping (or returning lists of errors), possibly() returns the numeric fallback NA for the failed call and proceeds smoothly.

walk

- The `walk()` function is perfect for operations that have **side effects**, such as saving files, printing messages, or generating plots.
- It behaves just like `map()`, but it **returns nothing**, keeping your workflow clean when you only care about the action being performed.
- Plus: The `.progress = TRUE` option automatically shows a progress bar while mapping, which is invaluable for long-running tasks.

For example, after creating your list of plots, you can easily export them all at once:

```r
library(purrr)
library(ggplot2)

# Save each plot to file
walk(names(plots), ~ {
  ggsave(
    filename = paste0(.x, "_plot.png"),
    plot = plots[[.x]],
    width = 6,
    height = 4
  )
})
```

quietly

`quietly()` **captures all output** generated by a function. It returns a **list** with four elements:

- `result`: the function output (or `NULL`)
- `output`: captured printed output
- `warnings`: any warnings generated
- `messages`: any messages generated

This is useful when you want to **run a function silently** but still inspect what happened.

```r
# Example function
example_fun <- function(x) {
  message("A message")
  warning("A warning")
  print(x)
  x + 1
}
```

]

```r
library(purrr)

# Wrap the function quietly
quiet_fun <- quietly(example_fun)

# Apply to a value
result <- quiet_fun(5)

# Inspect the result/output/etc.
result$result
```

```
## [1] 6
```

All output, warnings, and messages are captured in the list elements for inspection.

Parallel Processing

- When performance is important — say you're generating hundreds of plots or running computationally-heavy tasks — the parallel capabilities of `purrr` come into play.
- Starting with version 1.2.0, `purrr` supports parallel execution via the `in_parallel()` "adverb".
- Parallelism can significantly speed up operations by distributing tasks across multiple CPU cores. The latter is also a advanced topic which is why we only touch it briefly here.

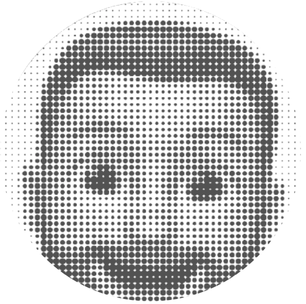
```r
# Prepare inputs
column_names <- names(mtcars)
column_data  <- mtcars
# Ensure daemons are running
mirai::daemons(4)
mirai::require_daemons()
```

Parallel Processing II

Now we can export the list of plots in parallel:

```r
# Export histograms in parallel
plots_parallel <- map2(
  column_names, column_data,
  ~ in_parallel(
      \(col_name, data) {
        ggplot2::ggplot(data, ggplot2::aes(x = .data[[col_name]])) +
          ggplot2::geom_histogram(bins = 10, fill = "steelblue") +
          ggplot2::ggtitle(paste("Histogram of", col_name))
      },
      col_name  = .x,
      data      = .y
    )
)
```

Thank you for your attention!



www.edgar-treischl.de

edgar-treischl

edgar.treischl@isb.bayern.de

Licence