R Tooklit Digest

...dose of hidden gems, packages, and practical R tricks...

Do More with Less Code in the Tidyverse

by Roy Mwavita
#RToolkitDigest

Tidyverse Tools That Do Less

Modern data work is not about writing more code.

It is about:

- Writing **less repetitive code**
- Making pipelines easier to read
- Scaling transformations across many variables
- Reducing human error

Today we focus on three tools:

- `across()`
- `map()`
- `reduce()`

The Problem with Repetition

Consider this common pattern in a dataset:

```
df %>%
  mutate(
  date_of_birth = as.Date(date_of_birth),
  date_enrolled = as.Date(date_enrolled),
  date_completed = as.Date(date_completed)
         )
```

Problems:

- Repetitive code
- Hard to scale
- Easy to make mistakes

1: `across()` as the solution

What it does:

Applies a function to multiple columns at once.

```
df %>%
  mutate(across(c(var1, var2, var3), mean, na.rm = TRUE))

```

Even better (pattern-based selection):

```
df %>%
  mutate(
    across(
      starts_with("date"), # other helpers like `ends_with`, `contains` can be used.
      as.Date
    )
  )
```

`across()` with Multiple Functions

You can apply multiple transformations at once:

Code:

```
library(dplyr)

# Using the built-in mtcars dataset to illustrate

mtcars %>%
  summarise(
    across(
      c(mpg, hp, wt),  # Selecting columns of interest
      list(
        mean = ~mean(.x, na.rm = TRUE),
        sd   = ~sd(.x, na.rm = TRUE),
        min  = ~min(.x, na.rm = TRUE),
        max  = ~max(.x, na.rm = TRUE)
      )
    )
  )
```

Output:

mpg_mean	mpg_sd	mpg_min	mpg_max	hp_mean	hp_sd	hp_min	hp_max	wt_mean	wt_sd	wt_min	wt_max
20.09062	6.026948	10.4	33.9	146.6875	68.56287	52	335	3.21725	0.9784574	1.513	5.424

2: Repeat Tasks Without Loops : `map()`

What it does:

Applies the same operation to many objects.

Instead of writing:

```
summary(df1)
summary(df2)
summary(df3)
summary(df4)
```

Use:

```
library(purrr)

list(df1, df2, df3, df4) %>%
  map(summary)
```

Benefits:

`map()` with Real Data Wrangling

Suppose you receive monthly beneficiary datasets that need the same cleaning steps.

Instead of:

```r
jan_clean <- jan %>%
  filter(!is.na(unique_id)) %>%
  mutate(sex = str_to_title(sex))

feb_clean <- feb %>%
  filter(!is.na(unique_id)) %>%
  mutate(sex = str_to_title(sex))

mar_clean <- mar %>%
  filter(!is.na(unique_id)) %>%
  mutate(sex = str_to_title(sex))

apr_clean <- apr %>%
  filter(!is.na(unique_id)) %>%
  mutate(sex = str_to_title(sex))
```

Use:

```r
monthly_data <- list(
  jan,
  feb,
  mar,
  apr
)

monthly_data %>%
  map(
    ~ .x %>%
      filter(!is.na(unique_id)) %>%
      mutate(sex = str_to_title(sex))
  )
```

Benefits:

- Apply the same cleaning logic consistently
- Avoid duplicating code
- Scale easily to many datasets

`map()` for Data Quality Checks

```r
datasets <- list(
  households,
  children,
  caregivers
)

datasets %>%
  map(~ summarise(
    .x,
    missing_ids = sum(is.na(unique_id)),
    records = n()
  ))
```

This produces the same quality checks for every dataset with a single pipeline.

Understanding .x in map()

When using `map()`, **.x** represents the current item being processed.

Think of it as a placeholder for each element in a list.

Example:

```
numbers <- list(1, 2, 3)

numbers %>%
  map(~ .x * 2)
```

Output:

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

[[2]]
[1] 4

[[3]]
[1] 6
```

Here:

- First iteration: .x = 1
- Second iteration: .x = 2
- Third iteration: .x = 3

In Our Data Wrangling Example

Code:

```
monthly_data %>%
  map(
    ~ .x %>%
      filter(!is.na(unique_id)) %>%
      mutate(sex = str_to_title(sex))
  )
```

Output::

Iteration	`.x` Represents
1	`jan`
2	`feb`
3	`mar`
4	`apr`

So map() effectively runs:

```
jan %>%
  filter(!is.na(unique_id)) %>%
  mutate(sex = str_to_title(sex))

feb %>%
  filter(!is.na(unique_id)) %>%
  mutate(sex = str_to_title(sex))

mar %>%
  filter(!is.na(unique_id)) %>%
  mutate(sex = str_to_title(sex))

apr %>%
  filter(!is.na(unique_id)) %>%
  mutate(sex = str_to_title(sex))
```

automatically.

3: `reduce()`: When You Have Many Tables

Problem: multiple left joins

```
df1 %>%
  left_join(df2) %>%
  left_join(df3) %>%
  left_join(df4)
```

Solution: purrr::reduce()

```
library(purrr)

list(df1, df2, df3, df4) %>%
  reduce(left_join, by = "id")
```

Benefits:

- Scales to many tables
- Cleaner pipelines
- Easier automation

Real-World Use Case

Imagine program data:

- Household table
- Child table
- Service table
- Location table

Instead of chaining joins manually, reduce works in this way:

```
reduce(list(hh, child, service, location),
       left_join, by = "unique_id")
```

When to Use Each Tool

Tool	Purpose
`across()`	Apply functions to many columns.
`map()`	Apply functions to many objects, lists, e.g. reading multiple files.
`reduce()`	Combine many objects into one, avoiding repeated joins.

Together they help you write code that is:

- Shorter
- More scalable
- Easier to maintain
- Less error-prone

Further Reading

- Tidyverse documentation: Column-wise operations with across()
- R for Data Science: Data Transformation
- purrr documentation: map() Family
- R for Data Science: Iteration
- purrr documentation: reduce()
- Advanced R: Functionals