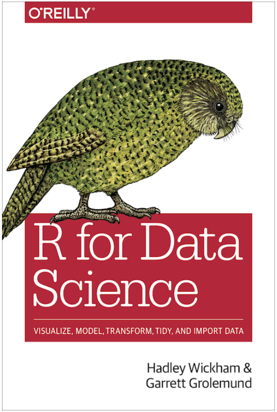
Socio-Informatics 348

Program Iteration

Dr Lisa Martin

Department of Information Science
Stellenbosch University

Today's Reading



R for Data Science, Chapter 26

Modifying Multiple Columns

- Problem: performing the same summary / transform on many columns
- Bad approach: copy & paste for each column

```
df |> summarize(
  n = n(),
  a = median(a),
  b = median(b),
  c = median(c),
  d = median(d),
)
#> # A tibble: 1 × 5
#>       n      a      b       c     d
#>   <int>  <dbl>  <dbl>   <dbl> <dbl>
#> 1    10 -0.246 -0.287 -0.0567 0.144
```

Modifying Multiple Columns

- Better: use `across()` in `summarize()` / `mutate()`

```
df |> summarize(
  n = n(),
  across(a:d, median),
)
#> # A tibble: 1 × 5
#>       n      a      b       c     d
#>   <int>  <dbl>  <dbl>   <dbl> <dbl>
#> 1    10 -0.246 -0.287 -0.0567 0.144
```

`across()`: Overview

- `across(.cols, .fns, .names)`
- `.cols` = which columns to modify
- `.fns` = function(s) to apply
  - `across(a:d, median)`
  - Note: no parentheses after function name
- `.names` = naming scheme for output columns

Selecting Columns with `.cols`

- `.cols` argument uses tidyselect semantics (e.g. `starts_with`, `everything`, `where`)

```
df <- tibble(
  grp = sample(2, 10, replace = TRUE),
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df |>
  group_by(grp) |>
  summarize(across(everything(), median))
#> # A tibble: 2 × 5
#>     grp       a       b     c     d
#>   <int>   <dbl>   <dbl> <dbl> <dbl>
#> 1     1 -0.0935 -0.0163 0.363 0.364
#> 2     2  0.312  -0.0576 0.208 0.565
```

Selecting Columns with `.cols`

- `where()` useful for selecting by column type
  - `where(is.numeric)` selects all numeric columns.
  - `where(is.character)` selects all string columns.
  - `where(is.Date)` selects all date columns.
  - `where(is.POSIXct)` selects all date-time columns.
  - `where(is.logical)` selects all logical columns.
- Can be combined with other selectors

  `starts_with("a") & where(is.logical)`

Calling Functions

- To call a function, simply provide its name (no parentheses)
- But what if you want to pass arguments to the function?
- We create a new 'anonymous' function in-line:

```r
df_miss |>
  summarize(
    across(a:d, function(x) median(x, na.rm = TRUE)),
    n = n()
  )
```

```r
df_miss |>
  summarize(
    across(a:d, \(x) median(x, na.rm = TRUE)),
    n = n()
  )
```

Calling Functions

- To call multiple functions, provide a list of functions:

```r
df_miss |>
  summarize(
    a = median(a, na.rm = TRUE),
    b = median(b, na.rm = TRUE),
    c = median(c, na.rm = TRUE),
    d = median(d, na.rm = TRUE),
    n = n()
  )
```

```r
df_miss |>
  summarize(
    across(a:d, list(
      median = \(x) median(x, na.rm = TRUE),
      n_miss = \(x) sum(is.na(x))
    )),
    n = n()
  )
#> # A tibble: 1 × 9
#>   a_median a_n_miss b_median b_n_miss c_median c_n_miss d_median d_n_miss
#>      <dbl>    <int>    <dbl>    <int>    <dbl>    <int>    <dbl>    <int>
#> 1    0.139        1    -1.11        1   -0.387        2     1.15        0
#> # ℹ 1 more variable: n <int>
```

Column Naming and .names

With `summarize()`, default names are `<col>_<fn>`, but can be customised with `.names`:

```
df_miss |>
  summarize(
    across(
      a:d,
      list(
        median = \(x) median(x, na.rm = TRUE),
        n_miss = \(x) sum(is.na(x))
      ),
      .names = "{.fn}_{.col}"
    ),
    n = n(),
  )
#> # A tibble: 1 × 9
#>   median_a n_miss_a median_b n_miss_b median_c n_miss_c median_d n_miss_d
#>      <dbl>    <int>    <dbl>    <int>    <dbl>    <int>    <dbl>    <int>
#> 1    0.139        1    -1.11        1   -0.387        2     1.15        0
#> # ℹ 1 more variable: n <int>
```

Column Naming and `.names`

With `mutate()`, default names are the same as input columns.
Problem: overwriting input columns.

```
df_miss |>
  mutate(
    across(a:d, \(x) coalesce(x, 0))
  )
#> # A tibble: 5 × 4
#>        a      b      c     d
#>    <dbl>  <dbl>  <dbl> <dbl>
#> 1  0.434 -1.25   0     1.60
#> 2  0     -1.43  -0.297 0.776
#> 3 -0.156 -0.980  0     1.15
#> 4 -2.61  -0.683 -0.785 2.13
#> 5  1.11   0     -0.387 0.704
```

Column Naming and `.names`

Solution: Customise with `.names`:

```
df_miss |>
  mutate(
    across(a:d, \(x) coalesce(x, 0), .names = "{.col}_na_zero")
  )
#> # A tibble: 5 × 8
#>        a      b      c     d a_na_zero b_na_zero c_na_zero d_na_zero
#>    <dbl>  <dbl>  <dbl> <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
#> 1  0.434 -1.25  NA     1.60      0.434    -1.25      0         1.60
#> 2 NA     -1.43  -0.297 0.776     0        -1.43     -0.297     0.776
#> 3 -0.156 -0.980 NA     1.15     -0.156    -0.980     0         1.15
#> 4 -2.61  -0.683 -0.785 2.13     -2.61     -0.683    -0.785     2.13
#> 5  1.11  NA     -0.387 0.704     1.11      0        -0.387     0.704
```

Filtering with `if_any()`, `if_all()`

`if_any()` & `if_all()` are variants of `across()`:

- `across()` less suited for filter conditions
- Instead, use `if_any()` or `if_all()` inside `filter()`

```
# same as df_miss |> filter(is.na(a) | is.na(b) | is.na(c) | is.na(d))
df_miss |> filter(if_any(a:d, is.na))
#> # A tibble: 4 × 4
#>        a      b      c     d
#>    <dbl>  <dbl>  <dbl> <dbl>
#> 1  0.434 -1.25  NA     1.60
#> 2 NA     -1.43  -0.297 0.776
#> 3 -0.156 -0.980 NA     1.15
#> 4  1.11  NA     -0.387 0.704

# same as df_miss |> filter(is.na(a) & is.na(b) & is.na(c) & is.na(d))
df_miss |> filter(if_all(a:d, is.na))
#> # A tibble: 0 × 4
#> # ℹ 4 variables: a <dbl>, b <dbl>, c <dbl>, d <dbl>
```

Using across() Inside Functions

- You can embed across() inside your own functions for reusability
- Example from Jacob Scott

```r
expand_dates <- function(df) {
  df |>
    mutate(
      across(where(is.Date), list(year = year, month = month, day = mday))
    )
}

df_date <- tibble(
  name = c("Amy", "Bob"),
  date = ymd(c("2009-08-03", "2010-01-16"))
)

df_date |>
  expand_dates()
#> # A tibble: 2 × 5
#>   name  date       date_year date_month date_day
#>   <chr> <date>         <dbl>      <dbl>    <int>
#> 1 Amy   2009-08-03      2009          8        3
#> 2 Bob   2010-01-16      2010          1       16
```

Reading Multiple Files

```r
data2019 <- readxl::read_excel("data/y2019.xlsx")
data2020 <- readxl::read_excel("data/y2020.xlsx")
data2021 <- readxl::read_excel("data/y2021.xlsx")
data2022 <- readxl::read_excel("data/y2022.xlsx")
```

```r
data <- bind_rows(data2019, data2020, data2021, data2022)
```

- What if you have many more files to read?
- How do we avoid copy & paste?

Working with Lists

- Read multiple files into a list using `list.files()`
  - First argument: directory path
  - Second argument: pattern (e.g. file extension)
  - Third argument: `full.names = TRUE` to get full paths

```
paths <- list.files("data/gapminder", pattern = "[.]xlsx$", full.names = TRUE)
paths
#>  [1] "data/gapminder/1952.xlsx" "data/gapminder/1957.xlsx"
#>  [3] "data/gapminder/1962.xlsx" "data/gapminder/1967.xlsx"
#>  [5] "data/gapminder/1972.xlsx" "data/gapminder/1977.xlsx"
#>  [7] "data/gapminder/1982.xlsx" "data/gapminder/1987.xlsx"
#>  [9] "data/gapminder/1992.xlsx" "data/gapminder/1997.xlsx"
#> [11] "data/gapminder/2002.xlsx" "data/gapminder/2007.xlsx"
```

Working with Lists

We can now use the list of file paths to read each file.

```
gapminder_1952 <- readxl::read_excel("data/gapminder/1952.xlsx")
gapminder_1957 <- readxl::read_excel("data/gapminder/1957.xlsx")
gapminder_1962 <- readxl::read_excel("data/gapminder/1962.xlsx")
 ...,
gapminder_2007 <- readxl::read_excel("data/gapminder/2007.xlsx")
```

To make things easier down the line, we want to combine all the data frames into a single object.

```
files <- list(
  readxl::read_excel("data/gapminder/1952.xlsx"),
  readxl::read_excel("data/gapminder/1957.xlsx"),
  readxl::read_excel("data/gapminder/1962.xlsx"),
  ...,
  readxl::read_excel("data/gapminder/2007.xlsx")
)
```

Working with Lists

Now we can easily refer to each file within the list:

```
files[[3]]
#> # A tibble: 142 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         32.0 10267083      853.
#> 2 Albania     Europe       64.8  1728137     2313.
#> 3 Algeria     Africa       48.3 11000948     2551.
#> 4 Angola      Africa       34    4826015     4269.
#> 5 Argentina   Americas     65.1 21283783     7133.
#> 6 Australia   Oceania      70.9 10794968    12217.
#> # ℹ 136 more rows
```

Using `purrr::map()`

- But… The code to read each file is still repetitive!
- We can use iteration to apply the same function to each element of a list
- The `purrr` package provides the `map()` family of functions for this purpose

```
files <- map(paths, readxl::read_excel)
length(files)
#> [1] 12

files[[1]]
#> # A tibble: 142 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         28.8  8425333      779.
#> 2 Albania     Europe       55.2  1282697     1601.
#> 3 Algeria     Africa       43.1  9279525     2449.
#> 4 Angola      Africa       30.0  4232095     3521.
#> 5 Argentina   Americas     62.5 17876956     5911.
#> 6 Australia   Oceania      69.1  8691212    10040.
#> # ℹ 136 more rows
```

Using purrr::map() + list_rbind()

We can bind the list objects into a single data frame using `list_rbind()`

```
list_rbind(files)
#> # A tibble: 1,704 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         28.8  8425333      779.
#> 2 Albania     Europe       55.2  1282697     1601.
#> 3 Algeria     Africa       43.1  9279525     2449.
#> 4 Angola      Africa       30.0  4232095     3521.
#> 5 Argentina   Americas     62.5 17876956     5911.
#> 6 Australia   Oceania      69.1  8691212    10040.
#> # ℹ 1,698 more rows
```

Full pipeline:

```
paths |>
  map(readxl::read_excel) |>
  list_rbind()
```

Using purrr::map() + list_rbind()

How do we add arguments to the function we are mapping?
The same as we did for across() – create an anonymous function

```
paths |>
  map(\(path) readxl::read_excel(path, n_max = 1)) |>
  list_rbind()
#> # A tibble: 12 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         28.8  8425333      779.
#> 2 Afghanistan Asia         30.3  9240934      821.
#> 3 Afghanistan Asia         32.0 10267083      853.
#> 4 Afghanistan Asia         34.0 11537966      836.
#> 5 Afghanistan Asia         36.1 13079460      740.
#> 6 Afghanistan Asia         38.4 14880372      786.
#> # ℹ 6 more rows
```

Extracting Data from File Paths

- Often file names encode metadata (e.g. year)
- Use `set_names(basename)` to label list elements

```
paths |> set_names(basename)
#>                   1952.xlsx                   1957.xlsx
#> "data/gapminder/1952.xlsx" "data/gapminder/1957.xlsx"
#>                   1962.xlsx                   1967.xlsx
#> "data/gapminder/1962.xlsx" "data/gapminder/1967.xlsx"
#>                   1972.xlsx                   1977.xlsx
#> "data/gapminder/1972.xlsx" "data/gapminder/1977.xlsx"
#>                   1982.xlsx                   1987.xlsx
#> "data/gapminder/1982.xlsx" "data/gapminder/1987.xlsx"
#>                   1992.xlsx                   1997.xlsx
#> "data/gapminder/1992.xlsx" "data/gapminder/1997.xlsx"
#>                   2002.xlsx                   2007.xlsx
#> "data/gapminder/2002.xlsx" "data/gapminder/2007.xlsx"
```

Extracting Data from File Paths

- Use `names_to` in `list_rbind()` to turn names into a column

```
paths |>
  set_names(basename) |>
  map(readxl::read_excel) |>
  list_rbind(names_to = "year") |>
  mutate(year = parse_number(year))
#> # A tibble: 1,704 × 6
#>    year country     continent lifeExp      pop gdpPercap
#>   <dbl> <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1  1952 Afghanistan Asia         28.8  8425333      779.
#> 2  1952 Albania     Europe       55.2  1282697     1601.
#> 3  1952 Algeria     Africa       43.1  9279525     2449.
#> 4  1952 Angola      Africa       30.0  4232095     3521.
#> 5  1952 Argentina   Americas     62.5 17876956     5911.
#> 6  1952 Australia   Oceania      69.1  8691212    10040.
#> # ℹ 1,698 more rows
```

Saving Your Newly Combined Work

- After combining, write out to a single file (e.g. ‘write_csv()‘)
- Future work can start from this cleaned file

Saving Multiple Files

- After generating multiple results (e.g. tables)
- Use iteration (map, loops) to save each object (e.g.`write_csv()`, `ggsave()`, `DB write()`)
- Can embed naming logic and file paths programmatically

Saving Multiple Files

Example: We have a list of data frames to save:

```
by_clarity <- diamonds |>
  group_nest(clarity)

by_clarity
#> # A tibble: 8 × 2
#>   clarity                 data
#>   <ord>   <list<tibble[,9]>>
#> 1 I1                [741 × 9]
#> 2 SI2             [9,194 × 9]
#> 3 SI1            [13,065 × 9]
#> 4 VS2            [12,258 × 9]
#> 5 VS1             [8,171 × 9]
#> 6 VVS2            [5,066 × 9]
#> # ℹ 2 more rows
```

Saving Multiple Files

Each data frame can be found in the ‘data‘ column:

```
by_clarity$data[[1]]
#> # A tibble: 741 × 9
#>   carat cut       color depth table price     x     y     z
#>   <dbl> <ord>     <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#> 1  0.32 Premium   E      60.9    58   345  4.38  4.42  2.68
#> 2  1.17 Very Good J      60.2    61  2774  6.83  6.9   4.13
#> 3  1.01 Premium   F      61.8    60  2781  6.39  6.36  3.94
#> 4  1.01 Fair      E      64.5    58  2788  6.29  6.21  4.03
#> 5  0.96 Ideal     F      60.7    55  2801  6.37  6.41  3.88
#> 6  1.04 Premium   G      62.2    58  2801  6.46  6.41  4
#> # ℹ 735 more rows
```

Saving Multiple Files

Use `str_glue()` to create custom file paths:

```
by_clarity <- by_clarity |>
  mutate(path = str_glue("diamonds-{clarity}.csv"))

by_clarity
#> # A tibble: 8 × 3
#>   clarity               data path
#>   <ord>   <list<tibble[,9]>> <glue>
#> 1 I1               [741 × 9] diamonds-I1.csv
#> 2 SI2            [9,194 × 9] diamonds-SI2.csv
#> 3 SI1           [13,065 × 9] diamonds-SI1.csv
#> 4 VS2           [12,258 × 9] diamonds-VS2.csv
#> 5 VS1            [8,171 × 9] diamonds-VS1.csv
#> 6 VVS2           [5,066 × 9] diamonds-VVS2.csv
#> # ℹ 2 more rows
```

Saving Multiple Files

So this is what we have now:

clarity	data	path
6 clarity categories	6 data frames	path.csv

We could save these data frames by hand...

```
write_csv(by_clarity$data[[1]], by_clarity$path[[1]])
write_csv(by_clarity$data[[2]], by_clarity$path[[2]])
write_csv(by_clarity$data[[3]], by_clarity$path[[3]])
...
write_csv(by_clarity$by_clarity[[8]], by_clarity$path[[8]])
```

But we can use iteration to speed this up!

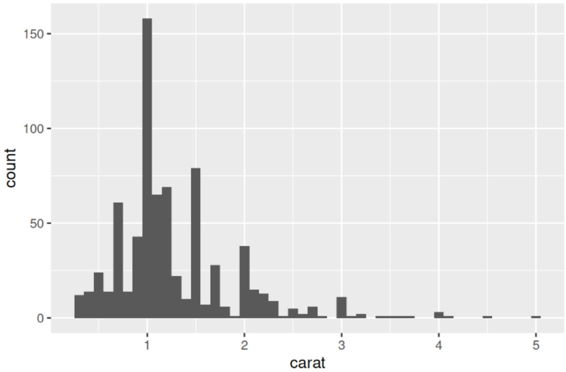
```
walk2(by_clarity$data, by_clarity$path, write_csv)
```

Saving Multiple Plots

What if we wanted to create and save a plot for each data frame?

```
carat_histogram <- function(df) {
  ggplot(df, aes(x = carat)) + geom_histogram(binwidth = 0.1)
}

carat_histogram(by_clarity$data[[1]])
```



Saving Multiple Plots

```
by_clarity <- by_clarity |>
  mutate(
    plot = map(data, carat_histogram),
    path = str_glue("clarity-{clarity}.png")
  )
```

The path gets overwritten and a plot is saved for each data frame:

clarity	data	path	plot
6 clarity categories	6 data frames	was path.csv overwritten: path.png	6 plots

Saving Multiple Plots

Use `walk2()`

```
walk2(
  by_clarity$path,
  by_clarity$plot,
  \(path, plot) ggsave(path, plot, width = 6, height = 6)
)
```

This is shorthand for:

```
ggsave(by_clarity$path[[1]], by_clarity$plot[[1]], width = 6, height = 6)
ggsave(by_clarity$path[[2]], by_clarity$plot[[2]], width = 6, height = 6)
ggsave(by_clarity$path[[3]], by_clarity$plot[[3]], width = 6, height = 6)
...
ggsave(by_clarity$path[[8]], by_clarity$plot[[8]], width = 6, height = 6)
```