Programming with data frames

I want to introduce two extensions to the ‘data manipulation’ material from Session 2:

1. Applying functions ‘across’ columns with `across` and `c_across`.

2. Programming with data frames using `tidyeval`.

In Session 2, we used `mutate` to create or modify columns:

```
mtcars |>
  mutate(wt_kg = (wt * 1000) * 0.454,
         hp_per_kg = hp / (wt_kg))
```

And `summarise` to summarise columns:

```
mtcars |>
  group_by(cyl) |>
  summarise(mpg = mean(mpg))
```

But what if we want to manipulate many columns at once?

We could do this with lots of typing:

```
data |>
  summarise(
    x1 = mean(x2),
    x2 = mean(x2),
    x3 = mean(x3),
    x4 = mean(x4),
    x5 = mean(x5)
  )
```

But there's a better way:

```
data |>
  summarise(across(c(x1, x2, x3), mean))
```

across

`across` is used inside `mutate` or `summarise` to apply a function to multiple columns.

```
across(.cols, .funs, .names)
```

It takes three arguments:

- `.cols` The **columns** we want to manipulate.
- `.fns` The **functions** we want to apply.
- `.names` The **names** of the new columns.

Let's see some examples...

Apply the mean function to `x1`, `x2`, and `x3`:

```
data |>
  summarise(across(c(x1, x2, x3), mean))
```

Note that we're selecting columns using `c()`.

Apply the mean function to `x1`, `x2`, and `x3`:

```
data |>
  summarise(across(c(x1, x2, x3), mean))
```

Note that we're selecting columns using `c()`.

Apply mean to all variables between `x1` and `x3`:

```
data |>
  summarise(across(x1:x3), mean))
```

Use `names` to check ordering.

Apply mean to columns starting with “x”:

```
data |>
  summarise(
    across(starts_with("x"), mean)
  )
```

Apply mean to columns starting with “x”:

```
data |>
  summarise(
    across(starts_with("x"), mean)
  )
```

Apply mean to columns containing “wk_”:

```
data |>
  summarise(
    across(matches("wk_"), mean)
  )
```

Apply mean to every column:

```
data |>
  summarise(across(everything(), mean))
```

To learn more, click here and here.

We can apply multiple functions too:

```
> data |>
>   summarise(
>     across(
>       .cols = x1:x3,
>       .fns = list(
>         mean = mean,
>         sd = sd,
>         min = min
>   )))
  x1_mean     x1_sd x1_min  x2_mean    x2_sd
1 3.21725 0.9784574  1.513 20.09062 6.026948

x2_min x3_mean    x3_sd x3_min
  10.4  6.1875 1.785922      4
```

Note that there are easier ways of making tables of summary statistics, covered later this session.

```
library(gtsummary)
data |> tbl_summary(include = x1:x3)
```

Characteristic	N = 32[1]
x1	3.33 (2.58, 3.61)
x2	19.2 (15.4, 22.8)
x3	17.71 (16.89, 18.90)

[1] Median (IQR)

See www.danieldsjoberg.com/gtsummary for details.

Applying functions conditionally

Use `where` to apply a function to all columns matching a condition.

Calculate the mean of all numeric columns:

```
data |>
  summarise(
    across(where(is.numeric), mean)
  )
```

Make all character columns uppercase:

```
data |>
  summarise(
    across(where(is.character), toupper)
  )
```

Rowwise calculations with `c_across`

We often need to do calculations across the rows. For example, calculating an individual's total score:

```
> scores
  participant item_1 item_2 item_3 item_4
1           A      2      2      2      1
2           B      2      1      0      0
3           C      2      1      3      1
4           D      1      1      1      2
```

We can use standard operators (e.g., `+`) but this becomes unwieldy with many columns:

```
scores |>
  mutate(
    total = item_1 + item_2 + item_3 + item_4
  )
```

We could try:

```
scores |>
  mutate(
    total = sum(c(item_1, item_2, item_3, item_4))
  )
```

But this doesn't work because it sums all the values at once, rather than row-wise:

```
  participant item_1 item_2 item_3 item_4 total
1           A      2      2      2      1    22
2           B      2      1      0      0    22
3           C      2      1      3      1    22
4           D      1      1      1      2    22
```

Instead, we need to calculate the summary by row.
To do this, we use `rowwise()`:

```
scores |>
  rowwise() |>
  mutate(
    total = sum(c(item_1, item_2, item_3, item_4))
  )
```

```
  participant item_1 item_2 item_3 item_4 total
  <chr>         <int>  <int>  <int>  <int> <int>
1 A                 2      2      2      1     7
2 B                 2      1      0      0     3
3 C                 2      1      3      1     7
4 D                 1      1      1      2     5
```

(This is just like `group_by`.)

But what if we have many columns? Use `c_across`:

```
scores |>
  rowwise() |>
  mutate(
    total = sum(c_across(starts_with("item_")))
  )
```

```
  participant item_1 item_2 item_3 item_4 total
  <chr>         <int>  <int>  <int>  <int> <int>
1 A                 2      2      2      1     7
2 B                 2      1      0      0     3
3 C                 2      1      3      1     7
4 D                 1      1      1      2     5
```

Using `c_across` allows us to use all the `tidyselect` tools (e.g., `starts_with`, `matches`, etc.

`across` is an advanced topic.

You don't need it right away.

Just remember it's there when you need it.

To learn more:

- `dplyr.tidyverse.org/articles/programming`
- `adv-r.hadley.nz/metaprogramming`

Programming with data frames

It's a good idea to write functions to automate parts of your data cleaning pipelines. However, it isn't so straightforward.

```
> my_summary <- function(d, var) {
>   d |>
>     group_by(var) |>
>     summarise(across(where(is.numeric), mean))
> }
>
> my_summary(penguins, species)
Error in `group_by()`:
! Must group by variables found in `.data`.
Column `var` is not found.
```

- `group_by(var)` treats `var` as a column name rather than the variable being passed.
- This results in `group_by()` searching for a column literally named "var", which doesn't exist in the dataset.

The embrace operator `{{}}`

To pass column names as arguments in tidyverse functions, we can use `{{}}` to 'wrap' function arguments.

```
> my_summary <- function(d, var) {
>   d |>
>     group_by({{var}}) |>
>     summarise(across(where(is.numeric),
>                      mean))
> }
> my_summary(penguins, species)
```

- `{{}}` tells `group_by()` to treat var as a column name rather than a literal string.
- This is an example of tidy evaluation, where function arguments are unquoted and evaluated in the correct context.

Using `:=` to dynamically create columns

Another useful tidyeval feature is `:=`, which allows dynamic column creation based on user input.

```
add_log <- function(d, .old, .new) {
  d |>
    mutate({{.new}} := log({{.old}}))
}

penguins |>
  add_log(bill_depth_mm, bill_depth_sqrt)
```

Here, `:=` is used to assign a dynamically named column within `mutate()`, allowing flexible function creation.

Note that you can construct new column names using quotation:

```
add_log <- function(d, .col) {
  d |>
    mutate("{{.col}}_sq" := {{.col}}^2)
}
```

Wrapping up

- tidyverse functions like `group_by` use non-standard evaluation (NSE). They expect unquoted column names rather than standard function arguments.
- Directly passing arguments causes errors because functions look for a literal column name rather than substituting the argument.
- The embrace operator `{{}}` allows function arguments to be correctly interpreted within tidyverse functions.

This is part of the tidy evaluation framework, powered by the rlang package. To learn more:

- The Metaprogramming section of Advanced R
- Programming with dplyr

Practical