Row-wise operations

dplyr, and R in general, are particularly well suited to performing operations over columns, and performing operations over rows is much harder. In this vignette, you'll learn dplyr's approach centred around the row-wise data frame created by `rowwise()`.

There are three common use cases that we discuss in this vignette:

- Row-wise aggregates (e.g. compute the mean of x, y, z).
- Calling a function multiple times with varying arguments.
- Working with list-columns.

These types of problems are often easily solved with a for loop, but it's nice to have a solution that fits naturally into a pipeline.

> Of course, someone has to write loops. It doesn't have to be you. — Jenny Bryan

```
library(dplyr, warn.conflicts = FALSE)
```

Creating

Row-wise operations require a special type of grouping where each group consists of a single row. You create this with `rowwise()`:

```
df <- tibble(x = 1:2, y = 3:4, z = 5:6)
df %>% rowwise()
#> # A tibble: 2 × 3
#> # Rowwise:
#>       x     y     z
#>   <int> <int> <int>
#> 1     1     3     5
#> 2     2     4     6
```

Like `group_by()`, `rowwise()` doesn't really do anything itself; it just changes how the other verbs work. For example, compare the results of `mutate()` in the following code:

```
df %>% mutate(m = mean(c(x, y, z)))
#> # A tibble: 2 × 4
#>       x     y     z     m
#>   <int> <int> <int> <dbl>
#> 1     1     3     5   3.5
#> 2     2     4     6   3.5
df %>% rowwise() %>% mutate(m = mean(c(x, y, z)))
#> # A tibble: 2 × 4
#> # Rowwise:
#>       x     y     z     m
#>   <int> <int> <int> <dbl>
```

```
#> 1     1     3     5     3
#> 2     2     4     6     4
```

If you use `mutate()` with a regular data frame, it computes the mean of `x`, `y`, and `z` across all rows. If you apply it to a row-wise data frame, it computes the mean for each row.

You can optionally supply "identifier" variables in your call to `rowwise()`. These variables are preserved when you call `summarise()`, so they behave somewhat similarly to the grouping variables passed to `group_by()`:

```
df <- tibble(name = c("Mara", "Hadley"), x = 1:2, y = 3:4, z = 5:6)

df %>%
  rowwise() %>%
  summarise(m = mean(c(x, y, z)))
#> # A tibble: 2 × 1
#>       m
#>   <dbl>
#> 1     3
#> 2     4

df %>%
  rowwise(name) %>%
  summarise(m = mean(c(x, y, z)))
#> `summarise()` has grouped output by 'name'. You can override using the
#> `.groups` argument.
#> # A tibble: 2 × 2
#> # Groups:   name [2]
#>   name       m
#>   <chr>  <dbl>
#> 1 Mara       3
#> 2 Hadley     4
```

`rowwise()` is just a special form of grouping, so if you want to remove it from a data frame, just call `ungroup()`.

Per row summary statistics

`dplyr::summarise()` makes it really easy to summarise values across rows within one column. When combined with `rowwise()` it also makes it easy to summarise values across columns within one row. To see how, we'll start by making a little dataset:

```
df <- tibble(id = 1:6, w = 10:15, x = 20:25, y = 30:35, z = 40:45)
df
#> # A tibble: 6 × 5
#>      id     w     x     y     z
#>   <int> <int> <int> <int> <int>
#> 1     1    10    20    30    40
#> 2     2    11    21    31    41
#> 3     3    12    22    32    42
```

```
#> 4     4    13    23    33    43
#> # ℹ 2 more rows
```

Let's say we want compute the sum of `w`, `x`, `y`, and `z` for each row. We start by making a row-wise data frame:

```
rf <- df %>% rowwise(id)
```

We can then use `mutate()` to add a new column to each row, or `summarise()` to return just that one summary:

```
rf %>% mutate(total = sum(c(w, x, y, z)))
#> # A tibble: 6 × 6
#> # Rowwise:  id
#>      id     w     x     y     z total
#>   <int> <int> <int> <int> <int> <int>
#> 1     1    10    20    30    40   100
#> 2     2    11    21    31    41   104
#> 3     3    12    22    32    42   108
#> 4     4    13    23    33    43   112
#> # ℹ 2 more rows
rf %>% summarise(total = sum(c(w, x, y, z)))
#> `summarise()` has grouped output by 'id'. You can override using the `.groups`
#> argument.
#> # A tibble: 6 × 2
#> # Groups:   id [6]
#>      id total
#>   <int> <int>
#> 1     1   100
#> 2     2   104
#> 3     3   108
#> 4     4   112
#> # ℹ 2 more rows
```

Of course, if you have a lot of variables, it's going to be tedious to type in every variable name. Instead, you can use `c_across()` which uses tidy selection syntax so you can to succinctly select many variables:

```
rf %>% mutate(total = sum(c_across(w:z)))
#> # A tibble: 6 × 6
#> # Rowwise:  id
#>      id     w     x     y     z total
#>   <int> <int> <int> <int> <int> <int>
#> 1     1    10    20    30    40   100
#> 2     2    11    21    31    41   104
#> 3     3    12    22    32    42   108
#> 4     4    13    23    33    43   112
#> # ℹ 2 more rows
rf %>% mutate(total = sum(c_across(where(is.numeric))))
#> # A tibble: 6 × 6
#> # Rowwise:  id
#>      id     w     x     y     z total
#>   <int> <int> <int> <int> <int> <int>
```

```
#> 1     1    10    20    30    40   100
#> 2     2    11    21    31    41   104
#> 3     3    12    22    32    42   108
#> 4     4    13    23    33    43   112
#> # ℹ 2 more rows
```

You could combine this with column-wise operations (see `vignette("colwise")` for more details) to compute the proportion of the total for each column:

```
rf %>%
  mutate(total = sum(c_across(w:z))) %>%
  ungroup() %>%
  mutate(across(w:z, ~ . / total))
#> # A tibble: 6 × 6
#>      id     w     x     y     z total
#>   <int> <dbl> <dbl> <dbl> <dbl> <int>
#> 1     1 0.1   0.2   0.3   0.4     100
#> 2     2 0.106 0.202 0.298 0.394   104
#> 3     3 0.111 0.204 0.296 0.389   108
#> 4     4 0.116 0.205 0.295 0.384   112
#> # ℹ 2 more rows
```

Row-wise summary functions

The `rowwise()` approach will work for any summary function. But if you need greater speed, it's worth looking for a built-in row-wise variant of your summary function. These are more efficient because they operate on the data frame as whole; they don't split it into rows, compute the summary, and then join the results back together again.

```
df %>% mutate(total = rowSums(pick(where(is.numeric), -id)))
#> # A tibble: 6 × 6
#>      id     w     x     y     z total
#>   <int> <int> <int> <int> <int> <dbl>
#> 1     1    10    20    30    40   100
#> 2     2    11    21    31    41   104
#> 3     3    12    22    32    42   108
#> 4     4    13    23    33    43   112
#> # ℹ 2 more rows
df %>% mutate(mean = rowMeans(pick(where(is.numeric), -id)))
#> # A tibble: 6 × 6
#>      id     w     x     y     z  mean
#>   <int> <int> <int> <int> <int> <dbl>
#> 1     1    10    20    30    40    25
#> 2     2    11    21    31    41    26
#> 3     3    12    22    32    42    27
#> 4     4    13    23    33    43    28
#> # ℹ 2 more rows
```

NB: I use `df` (not `rf`) and `pick()` (not `c_across()`) here because `rowMeans()` and `rowSums()` take a multi-row data frame as input. Also note that `-id` is needed to avoid selecting `id` in `pick()`. This wasn't required with the rowwise data frame because we had specified `id` as an identifier in our original call to `rowwise()`, preventing it from being selected as a grouping column.

List-columns

`rowwise()` operations are a natural pairing when you have list-columns. They allow you to avoid explicit loops and/or functions from the `apply()` or `purrr::map()` families.

Motivation

Imagine you have this data frame, and you want to count the lengths of each element:

```
df <- tibble(
  x = list(1, 2:3, 4:6)
)
```

You might try calling `length()`:

```
df %>% mutate(l = length(x))
#> # A tibble: 3 × 2
#>   x             l
#>   <list>    <int>
#> 1 <dbl [1]>     3
#> 2 <int [2]>     3
#> 3 <int [3]>     3
```

But that returns the length of the column, not the length of the individual values. If you're an R documentation aficionado, you might know there's already a base R function just for this purpose:

```
df %>% mutate(l = lengths(x))
#> # A tibble: 3 × 2
#>   x             l
#>   <list>    <int>
#> 1 <dbl [1]>     1
#> 2 <int [2]>     2
#> 3 <int [3]>     3
```

Or if you're an experienced R programmer, you might know how to apply a function to each element of a list using `sapply()`, `vapply()`, or one of the purrr `map()` functions:

```
df %>% mutate(l = sapply(x, length))
#> # A tibble: 3 × 2
#>   x             l
#>   <list>    <int>
#> 1 <dbl [1]>     1
#> 2 <int [2]>     2
```

```
#> 3 <int [3]>     3
df %>% mutate(l = purrr::map_int(x, length))
#> # A tibble: 3 × 2
#>   x             l
#>   <list>    <int>
#> 1 <dbl [1]>     1
#> 2 <int [2]>     2
#> 3 <int [3]>     3
```

But wouldn't it be nice if you could just write `length(x)` and dplyr would figure out that you wanted to compute the length of the element inside of `x`? Since you're here, you might already be guessing at the answer: this is just another application of the row-wise pattern.

```
df %>%
  rowwise() %>%
  mutate(l = length(x))
#> # A tibble: 3 × 2
#> # Rowwise:
#>   x             l
#>   <list>    <int>
#> 1 <dbl [1]>     1
#> 2 <int [2]>     2
#> 3 <int [3]>     3
```

Subsetting

Before we continue on, I wanted to briefly mention the magic that makes this work. This isn't something you'll generally need to think about (it'll just work), but it's useful to know about when something goes wrong.

There's an important difference between a grouped data frame where each group happens to have one row, and a row-wise data frame where every group always has one row. Take these two data frames:

```
df <- tibble(g = 1:2, y = list(1:3, "a"))
gf <- df %>% group_by(g)
rf <- df %>% rowwise(g)
```

If we compute some properties of `y`, you'll notice the results look different:

```
gf %>% mutate(type = typeof(y), length = length(y))
#> # A tibble: 2 × 4
#> # Groups:   g [2]
#>       g y         type   length
#>   <int> <list>    <chr>   <int>
#> 1     1 <int [3]> list        1
#> 2     2 <chr [1]> list        1
rf %>% mutate(type = typeof(y), length = length(y))
#> # A tibble: 2 × 4
#> # Rowwise:  g
#>       g y         type      length
#>   <int> <list>    <chr>      <int>
```

```
#> 1     1 <int [3]> integer        3
#> 2     2 <chr [1]> character      1
```

They key difference is that when `mutate()` slices up the columns to pass to `length(y)` the grouped mutate uses `[` and the row-wise mutate uses `[[`. The following code gives a flavour of the differences if you used a for loop:

```
# grouped
out1 <- integer(2)
for (i in 1:2) {
  out1[[i]] <- length(df$y[i])
}
out1
#> [1] 1 1

# rowwise
out2 <- integer(2)
for (i in 1:2) {
  out2[[i]] <- length(df$y[[i]])
}
out2
#> [1] 3 1
```

Note that this magic only applies when you're referring to existing columns, not when you're creating new rows. This is potentially confusing, but we're fairly confident it's the least worst solution, particularly given the hint in the error message.

```
gf %>% mutate(y2 = y)
#> # A tibble: 2 × 3
#> # Groups:   g [2]
#>       g y         y2
#>   <int> <list>    <list>
#> 1     1 <int [3]> <int [3]>
#> 2     2 <chr [1]> <chr [1]>
rf %>% mutate(y2 = y)
#> Error in `mutate()`:
#> ℹ In argument: `y2 = y`.
#> ℹ In row 1.
#> Caused by error:
#> ! `y2` must be size 1, not 3.
#> ℹ Did you mean: `y2 = list(y)` ?
rf %>% mutate(y2 = list(y))
#> # A tibble: 2 × 3
#> # Rowwise:  g
#>       g y         y2
#>   <int> <list>    <list>
#> 1     1 <int [3]> <int [3]>
#> 2     2 <chr [1]> <chr [1]>
```

Modelling

`rowwise()` data frames allow you to solve a variety of modelling problems in what I think is a particularly elegant way. We'll start by creating a nested data frame:

```
by_cyl <- mtcars %>% nest_by(cyl)
by_cyl
#> # A tibble: 3 × 2
#> # Rowwise:  cyl
#>     cyl data
#>   <dbl> <list>
#> 1     4 <tibble [11 × 12]>
#> 2     6 <tibble [7 × 12]>
#> 3     8 <tibble [14 × 12]>
```

This is a little different to the usual `group_by()` output: we have visibly changed the structure of the data. Now we have three rows (one for each group), and we have a list-col, `data`, that stores the data for that group. Also note that the output is `rowwise()`; this is important because it's going to make working with that list of data frames much easier.

Once we have one data frame per row, it's straightforward to make one model per row:

```
mods <- by_cyl %>% mutate(mod = list(lm(mpg ~ wt, data = data)))
mods
#> # A tibble: 3 × 3
#> # Rowwise:  cyl
#>     cyl data               mod
#>   <dbl> <list>             <list>
#> 1     4 <tibble [11 × 12]> <lm>
#> 2     6 <tibble [7 × 12]>  <lm>
#> 3     8 <tibble [14 × 12]> <lm>
```

And supplement that with one set of predictions per row:

```
mods <- mods %>% mutate(pred = list(predict(mod, data)))
mods
#> # A tibble: 3 × 4
#> # Rowwise:  cyl
#>     cyl data               mod    pred
#>   <dbl> <list>             <list> <list>
#> 1     4 <tibble [11 × 12]> <lm>   <dbl [11]>
#> 2     6 <tibble [7 × 12]>  <lm>   <dbl [7]>
#> 3     8 <tibble [14 × 12]> <lm>   <dbl [14]>
```

You could then summarise the model in a variety of ways:

```
mods %>% summarise(rmse = sqrt(mean((pred - data$mpg) ^ 2)))
#> `summarise()` has grouped output by 'cyl'. You can override using the `.groups`
#> argument.
#> # A tibble: 3 × 2
#> # Groups:   cyl [3]
#>     cyl  rmse
```

```
#>   <dbl> <dbl>
#> 1     4 3.01
#> 2     6 0.985
#> 3     8 1.87
mods %>% summarise(rsq = summary(mod)$r.squared)
#> `summarise()` has grouped output by 'cyl'. You can override using the `.groups`
#> argument.
#> # A tibble: 3 × 2
#> # Groups:   cyl [3]
#>     cyl   rsq
#>   <dbl> <dbl>
#> 1     4 0.509
#> 2     6 0.465
#> 3     8 0.423
mods %>% summarise(broom::glance(mod))
#> `summarise()` has grouped output by 'cyl'. You can override using the `.groups`
#> argument.
#> # A tibble: 3 × 13
#> # Groups:   cyl [3]
#>     cyl r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
#>   <dbl>     <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>
#> 1     4     0.509         0.454  3.33      9.32  0.0137     1 -27.7   61.5  62.7
#> 2     6     0.465         0.357  1.17      4.34  0.0918     1  -9.83  25.7  25.5
#> 3     8     0.423         0.375  2.02      8.80  0.0118     1 -28.7   63.3  65.2
#> # ℹ 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Or easily access the parameters of each model:

```
mods %>% reframe(broom::tidy(mod))
#> # A tibble: 6 × 6
#>     cyl term        estimate std.error statistic    p.value
#>   <dbl> <chr>          <dbl>     <dbl>     <dbl>      <dbl>
#> 1     4 (Intercept)    39.6       4.35      9.10 0.00000777
#> 2     4 wt             -5.65      1.85     -3.05 0.0137
#> 3     6 (Intercept)    28.4       4.18      6.79 0.00105
#> 4     6 wt             -2.78      1.33     -2.08 0.0918
#> # ℹ 2 more rows
```

Repeated function calls

`rowwise()` doesn't just work with functions that return a length-1 vector (aka summary functions); it can work with any function if the result is a list. This means that `rowwise()` and `mutate()` provide an elegant way to call a function many times with varying arguments, storing the outputs alongside the inputs.

Simulations

I think this is a particularly elegant way to perform simulations, because it lets you store simulated values along with the parameters that generated them. For example, imagine you have the following data frame that describes the properties of 3 samples from the uniform distribution:

```
df <- tribble(
  ~ n, ~ min, ~ max,
    1,     0,     1,
    2,    10,   100,
    3,   100,  1000,
)
```

You can supply these parameters to `runif()` by using `rowwise()` and `mutate()`:

```
df %>%
  rowwise() %>%
  mutate(data = list(runif(n, min, max)))
#> # A tibble: 3 × 4
#> # Rowwise:
#>       n   min   max data
#>   <dbl> <dbl> <dbl> <list>
#> 1     1     0     1 <dbl [1]>
#> 2     2    10   100 <dbl [2]>
#> 3     3   100  1000 <dbl [3]>
```

Note the use of `list()` here - `runif()` returns multiple values and a `mutate()` expression has to return something of length 1. `list()` means that we'll get a list column where each row is a list containing multiple values. If you forget to use `list()`, dplyr will give you a hint:

```
df %>%
  rowwise() %>%
  mutate(data = runif(n, min, max))
#> Error in `mutate()`:
#> ℹ In argument: `data = runif(n, min, max)`.
#> ℹ In row 2.
#> Caused by error:
#> ! `data` must be size 1, not 2.
#> ℹ Did you mean: `data = list(runif(n, min, max))` ?
```

Multiple combinations

What if you want to call a function for every combination of inputs? You can use `expand.grid()` (or `tidyr::expand_grid()`) to generate the data frame and then repeat the same pattern as above:

```
df <- expand.grid(mean = c(-1, 0, 1), sd = c(1, 10, 100))

df %>%
  rowwise() %>%
  mutate(data = list(rnorm(10, mean, sd)))
#> # A tibble: 9 × 3
#> # Rowwise:
#>    mean    sd data
#>   <dbl> <dbl> <list>
```

```
#> 1    -1     1 <dbl [10]>
#> 2     0     1 <dbl [10]>
#> 3     1     1 <dbl [10]>
#> 4    -1    10 <dbl [10]>
#> # ℹ 5 more rows
```

Varying functions

In more complicated problems, you might also want to vary the function being called. This tends to be a bit more of an awkward fit with this approach because the columns in the input tibble will be less regular. But it's still possible, and it's a natural place to use `do.call()`:

```
df <- tribble(
   ~rng,     ~params,
   "runif",  list(n = 10),
   "rnorm",  list(n = 20),
   "rpois",  list(n = 10, lambda = 5),
) %>%
  rowwise()

df %>%
  mutate(data = list(do.call(rng, params)))
#> # A tibble: 3 × 3
#> # Rowwise:
#>   rng   params           data
#>   <chr> <list>           <list>
#> 1 runif <named list [1]> <dbl [10]>
#> 2 rnorm <named list [1]> <dbl [20]>
#> 3 rpois <named list [2]> <int [10]>
```

Previously

`rowwise()`

`rowwise()` was also questioning for quite some time, partly because I didn't appreciate how many people needed the native ability to compute summaries across multiple variables for each row. As an alternative, we recommended performing row-wise operations with the purrr `map()` functions. However, this was challenging because you needed to pick a map function based on the number of arguments that were varying and the type of result, which required quite some knowledge of purrr functions.

I was also resistant to `rowwise()` because I felt like automatically switching between `[` to `[[` was too magical in the same way that automatically `list()`-ing results made `do()` too magical. I've now persuaded myself that the row-wise magic is good magic partly because most people find the distinction between `[` and `[[` mystifying and `rowwise()` means that you don't need to think about it.

Since `rowwise()` clearly is useful it is not longer questioning, and we expect it to be around for the long term.

`do()`

We've questioned the need for `do()` for quite some time, because it never felt very similar to the other dplyr verbs. It had two main modes of operation:

- Without argument names: you could call functions that input and output data frames using `.` to refer to the "current" group. For example, the following code gets the first row of each group:

```
mtcars %>%
  group_by(cyl) %>%
  do(head(., 1))
#> # A tibble: 3 × 13
#> # Groups:   cyl [3]
#>     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  cyl2  cyl4
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1     8    16
#> 2  21       6   160   110  3.9   2.62  16.5     0     1     4     4    12    24
#> 3  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2    16    32
```

  This has been superseded by `pick()` plus `reframe()`, a variant of `summarise()` that can create multiple rows and columns per group.

```
mtcars %>%
  group_by(cyl) %>%
  reframe(head(pick(everything()), 1))
#> # A tibble: 3 × 13
#>     cyl   mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb  cyl2  cyl4
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1     4  22.8   108    93  3.85  2.32  18.6     1     1     4     1     8    16
#> 2     6  21     160   110  3.9   2.62  16.5     0     1     4     4    12    24
#> 3     8  18.7   360   175  3.15  3.44  17.0     0     0     3     2    16    32
```

- With arguments: it worked like `mutate()` but automatically wrapped every element in a list:

```
mtcars %>%
  group_by(cyl) %>%
  do(nrows = nrow(.))
#> # A tibble: 3 × 2
#> # Rowwise:
#>     cyl nrows
#>   <dbl> <list>
#> 1     4 <int [1]>
#> 2     6 <int [1]>
#> 3     8 <int [1]>
```

  I now believe that behaviour is both too magical and not very useful, and it can be replaced by `summarise()` and `pick()`.

```
mtcars %>%
  group_by(cyl) %>%
  summarise(nrows = nrow(pick(everything())))
#> # A tibble: 3 × 2
```

```
#>     cyl nrows
#>   <dbl> <int>
#> 1     4    11
#> 2     6     7
#> 3     8    14
```

If needed (unlike here), you can wrap the results in a list yourself.

The addition of `pick()`/`across()` and the increased scope of `summarise()`/`reframe()` means that `do()` is no longer needed, so it is now superseded.