Recently I came across a question where someone was looking to take a bunch of CSV files, each of which contained numerical columns, and (a) get them into R, (b) calculate the mean and standard deviation of every column in every CSV file, and (c) calculate some overall summary like the mean of all the means and the mean of all the standard deviations.

I already know how to use `map_dfr()` to read a lot of CSVs with the same structure into a nice tidy tibble. (I'll show you below in a moment if you don't know how to do this.) It's a nice demonstration of the utility of iterating over a vector of filenames with a tidy result. But in thinking about the question I also wanted to provide a reproducible answer, which meant thinking about how to *create* a bunch of CSV files to read in by way of an example. So, here's one way to do that. Along the way we'll take advantage of a few small features of common tidyverse functions that I've found very helpful but whose existence is sometimes a little hard to discover. Or rather, their utility is sometimes hard to see when reading the help pages.

First let's make a bunch of CSVs in a folder called `tmpdat`. Each CSV will have five columns containing 100 normally-distributed observations with a mean of zero and a standard deviation of one.

Here's the code all at once:

```
library(tidyverse)

## Make a "tmpdat" folder in the working dir if one doesn't exist
ifelse(!dir.exists(file.path("tmpdat")), dir.create(file.path("tmpdat")), FALSE)

## Number of columns in each of our made-up data files.
nvars <- 5

## Create the CSV files
paste0("csv_", 1:100) %>%
  set_names() %>%
  map(~ replicate(n = nvars, rnorm(100, 0, 1))) %>%
  map_dfr(as_tibble, .id = "id", .name_repair = ~ paste0("v", 1:nvars)) %>%
  group_by(id) %>%
  nest() %>%
  pwalk(~ write_csv(x = .y, file = paste0("tmpdat/", .x, ".csv")))
```

The first few lines load the tidyverse and create the `tmpdir` folder if it doesn't already exist. We could do this in a full-on tidyverse way with the `fs` package, but here I just use the Base R equivalent.

Next we create `nvars` which is the number of columns each of our CSV files will have.

Now the fun starts. The first line creates a vector of 100 identifiers. The call to `set_names()` gives each element of the vector a name, which by default is the same as the value of that element. We do this so that the elements of the list we're about to create will have recognizable names as well.

```
paste0("csv_", 1:100) %>%
  set_names()
#>     csv_1     csv_2     csv_3     csv_4     csv_5     csv_6     csv_7
csv_8
#>   "csv_1"   "csv_2"   "csv_3"   "csv_4"   "csv_5"   "csv_6"   "csv_7"
"csv_8"
```

```
## (Cut off to save space)
```

Next, we use `map()` to feed each of our hundred elements to the `replicate()` function. We don't pass through any arguments to replicate at all (except for `nvars` which is the same for all of them). Instead, what happens is that `replicate()` creates a list of one hundred random matrices, each one of dimensions 100×5. One matrix lives inside each named list item, from `csv_1` to `csv_100`.

```
paste0("csv_", 1:100) %>%
  set_names() %>%
  map(~ replicate(n = nvars, rnorm(100, 0, 1)))
#> $csv_1
#>                [,1]         [,2]        [,3]        [,4]        [,5]
#>   [1,] -0.116601453 -2.339554484 -2.10008625  0.48430248  0.73549398
#>   [2,] -1.281593689  0.979867725  0.12217918  0.67191402 -2.10950592
#>   [3,] -0.131462279  2.025939426  1.01061386 -0.97849787 -0.86495635

## (Cut off to save space)
```

Step three, convert each matrix to a tibble and then bind them all together into one large tibble:

```
paste0("csv_", 1:100) %>%
  set_names() %>%
  map(~ replicate(n = nvars, rnorm(100, 0, 1))) %>%
  map_dfr(as_tibble, .id = "id", .name_repair = ~ paste0("v", 1:nvars))

#> # A tibble: 10,000 x 6
#>    id        v1     v2     v3     v4     v5
#>
#>  1 csv_1 -0.867  0.119  0.171  1.92   1.03
#>  2 csv_1  0.163  0.714 -1.31  -3.06   0.470
#>  3 csv_1  0.604 -0.282 -0.228  0.647  0.302
#>  4 csv_1 -0.775 -2.31  -0.522 -0.661 -0.160
#>  5 csv_1 -0.862  1.03  -0.781 -0.115 -1.61
#>  6 csv_1  0.943  1.12   0.251 -0.170 -0.356
#>  7 csv_1 -0.277 -1.03  -0.864 -1.62   1.70
#>  8 csv_1  0.613 -0.360 -0.491  1.01   0.436
#>  9 csv_1 -0.520  0.711 -2.77   2.10   0.450
#> 10 csv_1  1.01   0.657  0.519 -0.630 -0.927
#> # … with 9,990 more rows
```

It's starting to look a little tidier now. Like `map()`, `map_dfr()` feeds each element of the data—in this case, 100 list items, each of which contains a matrix—to the `as_tibble()` function. This converts each matrix to a tibble. Unlike `map()`, which always returns a list, `map_dfr()` will return a data frame or tibble (bound by row). Along the way we add an `id` column, to keep track of which imaginary dataset this will be from, and we also name the columns of the made-up data. The `.name_repair` argument has several useful pre-sets, but you can also send it a function. We do that here, to create the column names `v1`, `v2`, etc, all the way to however many columns there are. Tibbles require unique column names.

Onward:

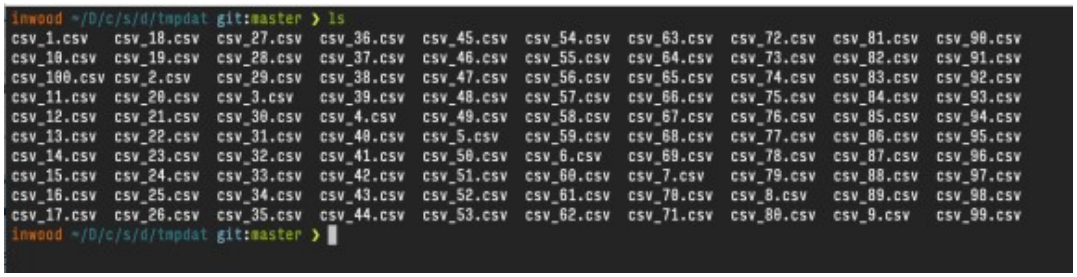
```
paste0("csv_", 1:100) %>%
  set_names() %>%
  map(~ replicate(n = nvars, rnorm(100, 0, 1))) %>%
  map_dfr(as_tibble, .id = "id", .name_repair = ~ paste0("v", 1:nvars)) %>%
  group_by(id) %>%
  nest()

#> # A tibble: 100 x 2
#> # Groups:   id [100]
#>    id     data
#>
#>  1 csv_1
#>  2 csv_2
#>  3 csv_3
#>  4 csv_4
#>  5 csv_5
#>  6 csv_6
#>  7 csv_7
#>  8 csv_8
#>  9 csv_9
#> 10 csv_10
#> # … with 90 more rows
```

Having created our data, we `group_by()` the `id` column and use `nest()` to create a list column of datasets, which are now all tibbles and all have consistent column names. Finally, we write each one out to a file:

```
paste0("csv_", 1:100) %>%
  set_names() %>%
  map(~ replicate(n = nvars, rnorm(100, 0, 1))) %>%
  map_dfr(as_tibble, .id = "id", .name_repair = ~ paste0("v", 1:nvars)) %>%
  group_by(id) %>%
  nest() %>%
  pwalk(~ write_csv(x = .y, file = paste0("tmpdat/", .x, ".csv")))
```

We use `walk()` when we want to iterate over a list, just as with `map()`. But `walk()` is for those times when the result of whatever we do is not an object we'll use further in R, but rather a "side effect", like a file or a plot. The `pwalk()` function is a special case of `walk()` designed for tibbles and data frames. It iterates over each row of the tibble. Here it takes the second argument of the tibble, the `data` list-column, and writes out its contents to a file whose name is constructed from the `id` column.

Now we have conjured up one hundred CSVs of made-up data. Perhaps a career in Social Psychology awaits us.



All the data

With our data now on disk, we can read it all back in and do our calculations. Once again, `map_dfr()` is our friend. We feed it a vector, `filenames`, which is just the full path to each CSV file in `tmpdat`.

```
filenames %>%
  map_dfr(read_csv, .id = "id", col_types = cols()) %>%
  group_by(id) %>%
  summarize(across(everything(),
                   list(mean = mean, sd = sd))) %>%
  pivot_longer(-id,
               names_to = c("col", ".value"), names_sep= "_") %>%
  group_by(col) %>%
  summarize(avg_mean = mean(mean),
            avg_sd = mean(sd))
#> # A tibble: 5 x 3
#>   col    avg_mean avg_sd
#>
#> 1 v1      0.0119    0.993
#> 2 v2      0.000967  0.992
#> 3 v3     -0.000190  0.991
#> 4 v4     -0.00533   0.994
#> 5 v5     -0.0172    0.986
```

Three things here. First, sending `filenames` down the pipe results in each element of it (i.e. each filename) getting fed one at a time to `read_csv()`. Because we're using `map_dfr()` to do the feeding, we get a tibble back. Second, we know we're pivoting all the columns except `id`, so instead of naming the column range we're including, we drop the single column that's not part of the pivot, using the shorthand of putting a minus sign in front of its name, like this: `-id`. Third, given that we know what we're dealing with inside each CSV, we use `col_types = cols()` to suppress the column specification message that would otherwise be displayed at the console for each file as it's read in.

```
filenames %>%
  map_dfr(read_csv, .id = "id", col_types = cols())
#> # A tibble: 10,000 x 6
#>    id          v1     v2     v3      v4     v5
#>
#>  1 1      -2.72    1.44  -0.883  0.603  -0.739
#>  2 1      -1.31   -0.516  0.701  0.0594  1.56
#>  3 1       0.834   1.00  -2.13   1.70   -0.591
#>  4 1      -0.139   0.601 -0.356 -1.12    0.167
#>  5 1      -1.43    0.194  1.20  -0.284   0.457
#>  6 1      -0.0937  0.116 -0.725 -0.521  -0.677
#>  7 1       0.556  -1.87   1.20  -0.449   1.92
```

```
#>  8 1      0.609   0.792 -0.844  0.550   0.587
#>  9 1      0.231  -1.04  -1.12   1.01    0.599
#> 10 1     -0.553   0.185 -0.172 -0.0263  0.281
#> # … with 9,990 more rows
```

Next we group by `id` (i.e., by CSV file) and use `across()` to get the means and standard deviations for everything. There's no missing data, so we don't need to add `na.rm = TRUE` in the summarize statement.

```
filenames %>%
  map_dfr(read_csv, .id = "id", col_types = cols()) %>%
  group_by(id) %>%
  summarize(across(everything(),
                   list(mean = mean, sd = sd)))
```

```
#> # A tibble: 100 x 11
#>    id     v1_mean v1_sd v2_mean v2_sd v3_mean v3_sd  v4_mean v4_sd  v5_mean
v5_sd
#>
#>  1 1      0.0649  1.06   0.0335 0.991  0.0182 1.06  -0.131   1.07  -0.0231
1.05
#>  2 10     0.00919 0.917  0.0283 1.04  -0.0184 0.970  0.0950  1.04   0.0243
0.973
#>  3 100    0.0760  1.08  -0.172  1.13   0.0303 1.05   0.00903 1.01   0.148
1.03
#>  4 11     0.0506  0.949  0.0667 1.00   0.0904 0.884 -0.0381  1.10   0.0538
0.954
#>  5 12     0.0938  1.00   0.117  1.05  -0.105  0.994 -0.0474  0.926  0.0981
0.992
#>  6 13    -0.112   0.933 -0.178  1.04   0.162  0.934 -0.0837  0.990  0.00821
1.01
#>  7 14     0.00121 1.04  -0.0464 1.01   0.101  0.886  0.0834  0.854 -0.0728
1.09
#>  8 15    -0.103   1.13   0.0927 0.994  0.186  1.02  -0.166   1.07  -0.110
0.989
#>  9 16     0.0828  1.01  -0.0375 0.970 -0.113  0.879 -0.0154  0.985 -0.0130
0.830
#> 10 17    -0.278   1.09  -0.115  1.06   0.0784 1.16   0.0556  0.979 -0.0731
0.992
#> # … with 90 more rows
```

Now, what we'd like next is to end up with a tibble with four columns that looks like this:

```
# A tibble: 500 x 4
   id    col       mean    sd

 1 1     v1      0.0649  1.06
 2 1     v2      0.0335  0.991
 3 1     v3      0.0182  1.06
 4 1     v4     -0.131   1.07
```

```
 5 1     v5    -0.0231  1.05
 6 10    v1     0.00919 0.917
 7 10    v2     0.0283  1.04
 8 10    v3    -0.0184  0.970
 9 10    v4     0.0950  1.04
10 10    v5     0.0243  0.973
# … with 490 more rows
```

That is, a tidy or long-form summary, where the first column is the CSV id, the second is which variable we're talking about, and the third and fourth columns are the summary statistics for that variable in that CSV.

If you use `pivot_longer()` to do this in the default way, you will not get quite what you want:

```
filenames %>%
  map_dfr(read_csv, .id = "id", col_types = cols()) %>%
  group_by(id) %>%
  summarize(across(everything(),
                   list(mean = mean, sd = sd))) %>%
  pivot_longer(-id)
#> # A tibble: 1,000 x 3
#>    id    name      value
#>
#>  1 1     v1_mean  0.0649
#>  2 1     v1_sd    1.06
#>  3 1     v2_mean  0.0335
#>  4 1     v2_sd    0.991
#>  5 1     v3_mean  0.0182
#>  6 1     v3_sd    1.06
#>  7 1     v4_mean -0.131
#>  8 1     v4_sd    1.07
#>  9 1     v5_mean -0.0231
#> 10 1     v5_sd    1.05
#> # … with 990 more rows
```

And if you read in the docs for `pivot_longer()` you might also try giving it a regular expression to remove the variable prefixes and get them in to their own column:

```
filenames %>%
  map_dfr(read_csv, .id = "id", col_types = cols()) %>%
  group_by(id) %>%
  summarize(across(everything(),
                   list(mean = mean, sd = sd))) %>%
  pivot_longer(-id,
               names_pattern = "(v\\d?)_(.*)",
               names_to = c("mean", "sd")) #< this is wrong

#> # A tibble: 1,000 x 4
#>    id    mean  sd      value
#>
```

```
#>  1 1     v1    mean   0.0649
#>  2 1     v1    sd     1.06
#>  3 1     v2    mean   0.0335
#>  4 1     v2    sd     0.991
#>  5 1     v3    mean   0.0182
#>  6 1     v3    sd     1.06
#>  7 1     v4    mean  -0.131
#>  8 1     v4    sd     1.07
#>  9 1     v5    mean  -0.0231
#> 10 1     v5    sd     1.05
#> # … with 990 more rows
```

Whoops, that's not right either. And even if we clean up the column headers we would still be left wanting to pivot wider again to get `mean` and `sd` in their own columns. But *that* would create a tibble with alternating `NA`s on each row for `mean` and `sd`. All we want is for the `mean` and `sd` parts to become the column names, and get their respective `value`. We should be able to do this in one step.

We can:

```
filenames %>%
  map_dfr(read_csv, .id = "id", col_types = cols()) %>%
  group_by(id) %>%
  summarize(across(everything(),
                   list(mean = mean, sd = sd))) %>%
  pivot_longer(-id,
               names_sep= "_",
               names_to = c("col", ".value"))
```

```
#> # A tibble: 500 x 4
#>    id    col       mean    sd
#>
#>  1 1     v1      0.0649  1.06
#>  2 1     v2      0.0335  0.991
#>  3 1     v3      0.0182  1.06
#>  4 1     v4     -0.131   1.07
#>  5 1     v5     -0.0231  1.05
#>  6 10    v1      0.00919 0.917
#>  7 10    v2      0.0283  1.04
#>  8 10    v3     -0.0184  0.970
#>  9 10    v4      0.0950  1.04
#> 10 10    v5      0.0243  0.973
#> # … with 490 more rows
```

We create the names of the new columns by splitting the existing names (`v1_mean`, `v1_sd` etc) on the `_` character. In the first place the split gives us `v1`, `v2`, `v3`, etc, which we put into a column named `col`. This leaves us with `mean` and `sd` names, each with its own particular `value`. Now, we *don't* want to put alternating `mean` and `sd` names in a single column named, say, "measure", with their values in a `value` column, as is the default. We want a single column of mean values and a single column of sd values. The trick is the special `.value` sentinel in the `names_to` argument. As noted in the help, the `names_to` argument is "a string specifying the name of the column to create from the data stored in the column

names of `data`." This can be a character vector, thus enabling the pivoting of multiple columns. And in addition, the help goes on to note,

> `.value` indicates that component of the name defines the name of the column containing the cell values, overriding `values_to`.

It might not jump out at you how handy this is. What it means is that we just take whatever unique name elements are left over when we split the original column names, and we use those as the names of the new columns. The corresponding values get inserted in a column with that name. This is a *very* useful option, because we find ourselves wanting to pivot out summary statistics quite a lot. The basic action of `summarize()` in conjunction with `group_by()` will do a lot for us a lot of the time. But sometimes we want to change the shape of the data we have in just this way. In those cases, being aware of `".value"` in `pivot_longer()` is your friend.

From there we can get the overall statistics we originally wanted, grouping by `col` to return the mean of all means and the mean of all sds per column:

```
 1
 2 filenames %>%
 3   map_dfr(read_csv, .id = "id", col_types = cols()) %>%
 4   group_by(id) %>%
 5   summarize(across(everything(),
 6                    list(mean = mean, sd = sd))) %>%
 7   pivot_longer(-id,
 8                names_to = c("col", ".value"), names_sep= "_") %>%
 9   group_by(col) %>%
10   summarize(avg_mean = mean(mean),
11             avg_sd = mean(sd))
12
13 #> # A tibble: 5 x 3
14 #>   col    avg_mean avg_sd
15 #>
16 #> 1 v1     0.0119    0.993
17 #> 2 v2     0.000967  0.992
18 #> 3 v3    -0.000190  0.991
19 #> 4 v4    -0.00533   0.994
20 #> 5 v5    -0.0172    0.986…
21
```