

Tidyverse Updates

There is no doubt that the `tidyverse` opinionated collection of R packages offers attractive, intuitive ways of wrangling data for data science. In earlier versions of `tidyverse` some elements of user control were sacrificed in favor of simplifying functions that could be picked up and easily used by rookies. In the 2020 updates to `dplyr` and `tidyr` there has been progress to restoring some finer control.



This means that there are new methods available in the `tidyverse` that some may not be aware of. The methods allow you to better transform your data directly to the way you want and to perform operations more flexibly. They also provide new ways to perform common tasks like nesting, modeling and graphing in ways where the code is more readable. Often users are only just scratching the surface of what can be done with the latest updates to this important set of packages.

It's incumbent on any analyst to stay up to date with new methods. This post covers ten examples of approaches to common data tasks that are better served by the latest `tidyverse` updates. We will use the new Palmer Penguins dataset, a great all round dataset for illustrating data wrangling.

First let's load our `tidyverse` packages and the Palmer Penguins dataset and take a quick look at it. Please be sure to install the latest versions of these packages before trying to replicate the work here.

```
library(tidyverse)
library(palmerpenguins)

penguins <- palmerpenguins::penguins %>%
  filter(!is.na(bill_length_mm))

penguins

## # A tibble: 342 x 8
##   species island bill_length_mm bill_depth_mm
## flipper_length_mm body_mass_g
##
## 1 Adelie Torge~          39.1          18.7
## 181          3750
## 2 Adelie Torge~          39.5          17.4
## 186          3800
## 3 Adelie Torge~          40.3           18
## 195          3250
## 4 Adelie Torge~          36.7          19.3
## 193          3450
```

```
## 5 Adelie Torge~      39.3      20.6
190      3650
## 6 Adelie Torge~      38.9      17.8
181      3625
## 7 Adelie Torge~      39.2      19.6
195      4675
## 8 Adelie Torge~      34.1      18.1
193      3475
## 9 Adelie Torge~      42      20.2
190      4250
## 10 Adelie Torge~      37.8      17.1
186      3300
## # ... with 332 more rows, and 2 more variables: sex , year
```

The dataset presents several observations of anatomical parts of penguins of different species, sexes and locations, and the year that the measurements were taken.

1. Selecting columns

`tidyselect` helper functions are now built in to allow you to save time by selecting columns using `dplyr::select()` based on common conditions. In this case, if we want to reduce the dataset to just bill measurements we can use this, noting that all measurement columns contain an underscore:

```
penguins %>%
  dplyr::select(!contains("_"), starts_with("bill"))

## # A tibble: 342 x 6
##   species island    sex    year bill_length_mm
bill_depth_mm
##
## 1 Adelie  Torgersen male    2007      39.1
18.7
## 2 Adelie  Torgersen female  2007      39.5
17.4
## 3 Adelie  Torgersen female  2007      40.3
18
## 4 Adelie  Torgersen female  2007      36.7
19.3
## 5 Adelie  Torgersen male    2007      39.3
20.6
## 6 Adelie  Torgersen female  2007      38.9
17.8
## 7 Adelie  Torgersen male    2007      39.2
19.6
## 8 Adelie  Torgersen      2007      34.1      18.1
## 9 Adelie  Torgersen      2007      42      20.2
## 10 Adelie Torgersen      2007      37.8      17.1
## # ... with 332 more rows
```

A full set of `tidyselect` helper functions can be found in the documentation [here](#).

2. Reordering columns

`dplyr::relocate()` allows a new way to reorder specific columns or sets of columns. For example, if we want to make sure that all of the measurement columns are at the end of the dataset, we can use this, noting that my last column is year:

```
penguins <- penguins %>%
  dplyr::relocate(contains("_"), .after = year)

penguins

## # A tibble: 342 x 8
##   species island sex      year bill_length_mm bill_depth_mm
## flipper_length_~
##
## 1 Adelie   Torge~ male    2007          39.1          18.7
181
## 2 Adelie   Torge~ fema~    2007          39.5          17.4
186
## 3 Adelie   Torge~ fema~    2007          40.3           18
195
## 4 Adelie   Torge~ fema~    2007          36.7          19.3
193
## 5 Adelie   Torge~ male    2007          39.3          20.6
190
## 6 Adelie   Torge~ fema~    2007          38.9          17.8
181
## 7 Adelie   Torge~ male    2007          39.2          19.6
195
## 8 Adelie   Torge~         2007          34.1          18.1
193
## 9 Adelie   Torge~         2007           42          20.2
190
## 10 Adelie  Torge~         2007          37.8          17.1
186
## # ... with 332 more rows, and 1 more variable: body_mass_g
```

Similar to `.after` you can also use `.before` as an argument here.

3. Controlling mutated column locations

Note in the `penguins` dataset that there are no unique identifiers for each study group. This can be problematic when we have multiple penguins of the same species, island, sex and year in the dataset. To address this and prepare for later examples, let's add a unique identifier using `dplyr::mutate()`, and here we can illustrate how `mutate()` now allows us to position our new column in a similar way to `relocate()`:

```
penguins_id <- penguins %>%
  dplyr::group_by(species, island, sex, year) %>%
  dplyr::mutate(studygroupid = row_number(), .before =
contains("_"))

penguins_id

## # A tibble: 342 x 9
```

```
## # Groups:   species, island, sex, year [35]
##   species island sex      year studygroupid bill_length_mm
bill_depth_mm
##
## 1 Adelie   Torge~ male    2007              1          39.1
18.7
## 2 Adelie   Torge~ fema~  2007              1          39.5
17.4
## 3 Adelie   Torge~ fema~  2007              2          40.3
18
## 4 Adelie   Torge~ fema~  2007              3          36.7
19.3
## 5 Adelie   Torge~ male    2007              2          39.3
20.6
## 6 Adelie   Torge~ fema~  2007              4          38.9
17.8
## 7 Adelie   Torge~ male    2007              3          39.2
19.6
## 8 Adelie   Torge~        2007              1          34.1
18.1
## 9 Adelie   Torge~        2007              2          42
20.2
## 10 Adelie  Torge~        2007              3          37.8
17.1
## # ... with 332 more rows, and 2 more variables:
flipper_length_mm ,
## #   body_mass_g
```

4. Transforming from wide to long

The `penguins` dataset is clearly in a wide form, as it gives multiple observations across the columns. For many reasons we may want to transform data from wide to long. In long data, each observation has its own row. The older function `gather()` in `tidyr` was popular for this sort of task but its new version `pivot_longer()` is even more powerful. In this case we have different body parts, measures and units inside these column names, but we can break them out very simply like this:

```
penguins_long <- penguins_id %>%
  tidyr::pivot_longer(contains("_"), # break out the
    measurement cols
                        names_to = c("part", "measure",
    "unit"), # break them into these three columns
                        names_sep = "_") # use the underscore
    to separate

penguins_long

## # A tibble: 1,368 x 9
## # Groups:   species, island, sex, year [35]
##   species island sex      year studygroupid part
measure unit  value
##
```

```
## 1 Adelie Torgersen male 2007 1 bill
length mm 39.1
## 2 Adelie Torgersen male 2007 1 bill
depth mm 18.7
## 3 Adelie Torgersen male 2007 1 flipper
length mm 181
## 4 Adelie Torgersen male 2007 1 body
mass g 3750
## 5 Adelie Torgersen female 2007 1 bill
length mm 39.5
## 6 Adelie Torgersen female 2007 1 bill
depth mm 17.4
## 7 Adelie Torgersen female 2007 1 flipper
length mm 186
## 8 Adelie Torgersen female 2007 1 body
mass g 3800
## 9 Adelie Torgersen female 2007 2 bill
length mm 40.3
## 10 Adelie Torgersen female 2007 2 bill
depth mm 18
## # ... with 1,358 more rows
```

5. Transforming from long to wide

It's just as easy to move back from long to wide. `pivot_wider()` gives much more flexibility compared to the older `spread()`:

```
penguins_wide <- penguins_long %>%
  tidyr::pivot_wider(names_from = c("part", "measure",
    "unit"), # pivot these columns
    values_from = "value", # take the values
    from here
    names_sep = "_") # combine col names
  using an underscore

penguins_wide

## # A tibble: 342 x 9
## # Groups:   species, island, sex, year [35]
##   species island sex    year studygroupid bill_length_mm
bill_depth_mm
##
## 1 Adelie Torge~ male 2007 1 39.1
18.7
## 2 Adelie Torge~ fema~ 2007 1 39.5
17.4
## 3 Adelie Torge~ fema~ 2007 2 40.3
18
## 4 Adelie Torge~ fema~ 2007 3 36.7
19.3
## 5 Adelie Torge~ male 2007 2 39.3
20.6
```

```
## 6 Adelie Torge~ fema~ 2007 4 38.9
17.8
## 7 Adelie Torge~ male 2007 3 39.2
19.6
## 8 Adelie Torge~ 2007 1 34.1
18.1
## 9 Adelie Torge~ 2007 2 42
20.2
## 10 Adelie Torge~ 2007 3 37.8
17.1
## # ... with 332 more rows, and 2 more variables:
flipper_length_mm ,
## # body_mass_g
```

6. Running group statistics across multiple columns

`dplyr` can now apply multiple summary functions to grouped data using the `across` adverb, helping you be more efficient. If we wanted to summarize all bill and flipper measurements in our penguins we would do this:

```
penguin_stats <- penguins %>%
  dplyr::group_by(species) %>%
  dplyr::summarize(across(ends_with("mm"), # do this for
    columns ending in mm
    list(~mean(.x, na.rm = TRUE),
         ~sd(.x, na.rm = TRUE)))) #
calculate a mean and sd

penguin_stats

## # A tibble: 3 x 7
##   species bill_length_mm_1 bill_length_mm_2
bill_depth_mm_1 bill_depth_mm_2
##
## 1 Adelie 38.8 2.66
18.3 1.22
## 2 Chinstrap 48.8 3.34
18.4 1.14
## 3 Gentoo 47.5 3.08
15.0 0.981
## # ... with 2 more variables: flipper_length_mm_1 ,
## # flipper_length_mm_2
```

7. Control output column names when summarising columns

The columns in `penguin_stats` have been given default names which are not that intuitive. If we name our summary functions, we can then use the `.names` argument to control precisely how we want these columns named. This uses `glue` notation. For example, here we want to construct the new column names by taking the existing column names, removing any underscores or 'mm' metrics, and pasting to the summary function name using an underscore:

```

penguin_stats <- penguins %>%
  dplyr::group_by(species) %>%
  dplyr::summarize(across(ends_with("mm"),
                          list(mean = ~mean(.x, na.rm =
TRUE),
                              sd = ~sd(.x, na.rm = TRUE))), #
name summary functions
                          .names = "{gsub('_', '_mm', '',
col)}}_{fn}") # column names structure

penguin_stats

## # A tibble: 3 x 7
##   species billlength_mean billlength_sd billdepth_mean
billdepth_sd
##
## 1 Adelie             38.8           2.66           18.3
1.22
## 2 Chinstrap          48.8           3.34           18.4
1.14
## 3 Gentoo            47.5           3.08           15.0
0.981
## # ... with 2 more variables: flipperlength_mean ,
flipperlength_sd

```

8. Running models across subsets

The output of `summarize()` can now be literally anything, because `dplyr` now allows different column types. We can generate summary vectors, dataframes or other objects like models or graphs.

If we wanted to run a model for each species you could do it like this:

```

penguin_models <- penguins %>%
  dplyr::group_by(species) %>%
  dplyr::summarize(model = list(lm(body_mass_g ~
flipper_length_mm + bill_length_mm + bill_depth_mm))) #
store models in a list column

penguin_models

## # A tibble: 3 x 2
##   species    model
##
## 1 Adelie
## 2 Chinstrap
## 3 Gentoo

```

It's not usually that useful to keep model objects in a dataframe, but we could use other tidy-oriented packages to summarize the statistics of the models and return them all as nicely integrated dataframes:

```

library(broom)

penguin_models <- penguins %>%
  dplyr::group_by(species) %>%
  dplyr::summarize(broom::glance(lm(body_mass_g ~
    flipper_length_mm + bill_length_mm + bill_depth_mm))) #
summarize model stats

penguin_models

## # A tibble: 3 x 13
##   species r.squared adj.r.squared sigma statistic p.value
##   df logLik   AIC
##
## 1 Adelie      0.508      0.498   325.      50.6 1.55e-22
## 3 -1086. 2181.
## 2 Chinstrap  0.504      0.481   277.      21.7 8.48e-10
## 3 -477. 964.
## 3 Gentoo     0.625      0.615   313.      66.0 3.39e-25
## 3 -879. 1768.
## # ... with 4 more variables: BIC , deviance , df.residual
## # ,
## # nobs

```

9. Nesting data

Often we have to work with subsets, and it can be useful to apply a common function across all subsets of the data. For example, maybe we want to take a look at our different species of penguins and make some different graphs of them. Grouping based on subsets would previously be achieved by the following somewhat awkward combination of `tidyverse` functions.

```

penguins %>%
  dplyr::group_by(species) %>%
  tidyr::nest() %>%
  dplyr::rowwise()

## # A tibble: 3 x 2
## # Rowwise: species
##   species data
##
## 1 Adelie
## 2 Gentoo
## 3 Chinstrap

```

The new function `nest_by()` provides a more intuitive way to do the same thing:

```

penguins %>%
  nest_by(species)

## # A tibble: 3 x 2
## # Rowwise: species
##   species data

```



```
##           >
## 1 Adelie           [151 x 7]
## 2 Chinstrap       [68 x 7]
## 3 Gentoo          [123 x 7]
```

The nested data will be stored in a column called `data` unless we specify otherwise using a `.key` argument.

10. Graphing across subsets

Armed with `nest_by()` and the fact that we can summarize or mutate virtually any type of object now, this allows us to generate graphs across subsets and store them in a dataframe for later use. Let's scatter plot bill length and depth for our three penguin species:

```
# generic function for generating a simple scatter plot in
ggplot2
scatter_fn <- function(df, col1, col2, title) {
  df %>%
    ggplot2::ggplot(aes(x = , y = )) +
    ggplot2::geom_point() +
    ggplot2::geom_smooth(method = "loess", formula = "y ~ x")
  +
    ggplot2::labs(title = title)
}

# run function across species and store plots in a list
column
penguin_scatters <- penguins %>%
  dplyr::nest_by(species) %>%
  dplyr::mutate(plot = list(scatter_fn(data, bill_length_mm,
bill_depth_mm, species)))

penguin_scatters

## # A tibble: 3 x 3
## # Rowwise:  species
##   species           data plot
##       >
## 1 Adelie           [151 x 7]
## 2 Chinstrap       [68 x 7]
## 3 Gentoo          [123 x 7]
```

Now we can easily display the different scatter plots to show, for example, that our penguins exemplify [Simpson's Paradox](#):

```
library(patchwork)

# generate scatter for entire dataset
p_all <- scatter_fn(penguins, bill_length_mm, bill_depth_mm,
"All Species")

# get species scatters from penguin_scatters dataframe
for (i in 1:3) {
```

```

assign(paste("p", i, sep = "_"),
      penguin_scatters$plot[i][[1]])
}

# display nicely using patchwork in R Markdown
p_all /
(p_1 | p_2 | p_3) +
  plot_annotation(caption = "{palmerpenguins} dataset")

```

