Tidyr

Recap

- `tidyverse` is an *opinionated* collection of pacakges
- All packages within it's ecosystem use the same syntax:
    - `%>%` pipe operators at the end of the line read as *"and then"*

> "I took my original data.frame %>%
>
> I kept only 5 out of the original 20 columns %>%
>
> I added a new column that was based on the 2nd column %>%
>
> I grouped the data based on a categorical column %>%
>
> I got descriptive statistics per level of the categorical var"

Recap

- `tidyverse` is an *opinionated* collection of pacakges
- All packages within it's ecosystem use the same syntax:
  - `%>%` pipe operators at the end of the line read as *"and then"*

```
originalData %>%

select(1:5) %>%

mutate(newVar = sqrt(var1)) %>%

group_by(factorVar) %>%

summarize(meanVar = mean(var))
```

This time

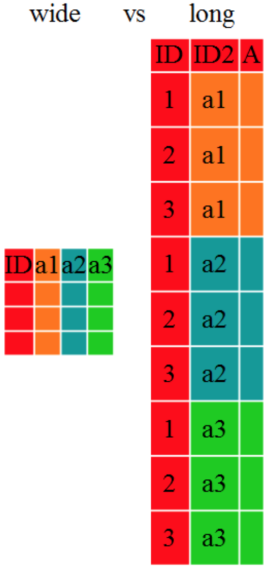
Functions from the `tidyr` package (but DON'T memorize which functions come from which packages!)

- Go from long to wide format
- Split columns and combine them
- Missing data

Long vs. Wide data

Long data - Each column is a variable and each row is an observation. Each row does NOT need to be a unique participant.

Wide data - Each row is a particular participant, and columns can contain multiple observations for the same data.



https://swcarpentry.github.io/r-novice-gapminder/14-tidyr/

```
##   Subject Time1 Time2 Time3
## 1       1   0.2   0.4   0.3
## 2       2   0.8   0.9   0.7
## 3       3   1.3   1.0   1.1
```

```
##   Subject TimePoint Score
## 1       1         1   0.2
## 2       2         1   0.8
## 3       3         1   1.3
## 4       1         2   0.4
## 5       2         2   0.9
## 6       3         2   1.0
## 7       1         3   0.3
## 8       2         3   0.7
## 9       3         3   1.1
```

Long vs. Wide data

For the most part, you want your data to be in the **long** format

- Especially for plotting in `ggplot2`!
- (some analyses, like reliability, require the wide format, but most stick with long)

However, we often receive data in the wide format. It is useful to be able to go between the two. `tidyr` makes this easy with:

- `pivot_wider()` to go from long to wide
- `pivot_longer()` to go from wide to long

`pivot_wider()` function

This function takes in long data and makes it wide. Important arguments:

- `names_from =` which columns to get the *name* of the output column.
- `values_from =` which columns to get the *value* of the output column.

`pivot_wider()` function

Let's take the example data.frame I showed earlier. Since it's completely arbitrary, I'm going to call it `generic`

```
generic
```

```
##   Subject TimePoint Score
## 1       1         1   0.2
## 2       2         1   0.8
## 3       3         1   1.3
## 4       1         2   0.4
## 5       2         2   0.9
## 6       3         2   1.0
## 7       1         3   0.3
## 8       2         3   0.7
## 9       3         3   1.1
```

`pivot_wider()` function

This `generic` data.frame is in the **long** format. To make it into the **wide** format, let's use `pivot_wider()`

```
wideGeneric <- generic %>%
  pivot_wider(names_from = TimePoint,
              values_from = Score)

wideGeneric
```

```
## # A tibble: 3 x 4
##   Subject   `1`   `2`   `3`
##     <dbl> <dbl> <dbl> <dbl>
## 1       1   0.2   0.4   0.3
## 2       2   0.8   0.9   0.7
## 3       3   1.3   1     1.1
```

`pivot_wider()` function

Sometimes, it's a bit more complicated. Let's add some more variables to `generic` to test this out.

- `hairColor` factor with 2 levels (brown & blonde)
- `happiness` scale of 1 to 10 measured at each time point

```
generic <- generic %>%
  mutate(hairColor = rep(c("brown", "blonde", "blonde"), times = 3),
         happiness = c(10, 2, 6, 9, 2, 5, 10, 3, 4))

generic
```

```
##   Subject TimePoint Score hairColor happiness
## 1       1         1   0.2     brown        10
## 2       2         1   0.8    blonde         2
## 3       3         1   1.3    blonde         6
## 4       1         2   0.4     brown         9
## 5       2         2   0.9    blonde         2
## 6       3         2   1.0    blonde         5
## 7       1         3   0.3     brown        10
## 8       2         3   0.7    blonde         3
```

`pivot_wider()` function

Now, let's say we want each time point's `Score` and `happiness` variables in the wide format...

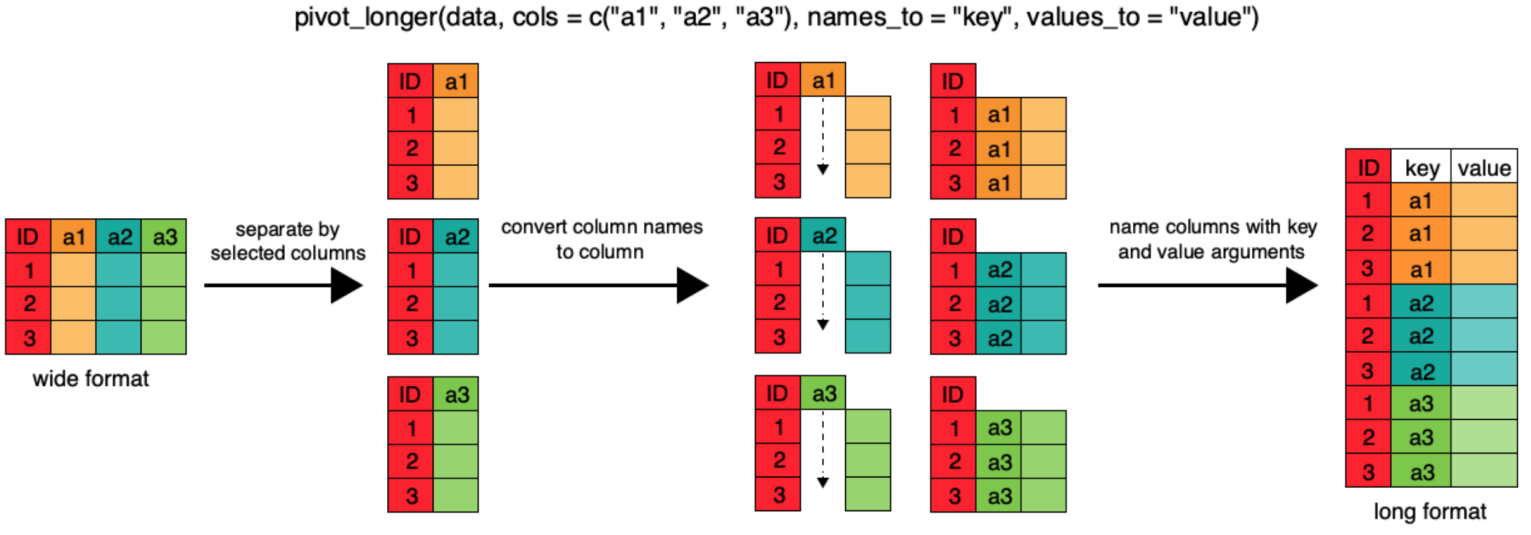
```
wideGenericMore <- generic %>%
  pivot_wider(names_from = TimePoint,
              values_from = c(Score, happiness))

wideGenericMore
```

```
## # A tibble: 3 x 8
##   Subject hairColor Score_1 Score_2 Score_3 happiness_1 happiness_2 happine
##     <dbl> <chr>       <dbl>   <dbl>   <dbl>       <dbl>       <dbl>       <
## 1       1 brown         0.2     0.4     0.3          10           9
## 2       2 blonde        0.8     0.9     0.7           2           2
## 3       3 blonde        1.3     1       1.1           6           5
```

`pivot_longer()` function

The exact opposite of `pivot_wider()` is `pivot_longer`. This takes a wide data.frame and makes it into a **long** data.frame. Arguments are now `names_to =` and `values_to =`. You also need to include a `cols =` argument to say which columns you want into the longer format.

Before doing this with code, here's a schematic that might be helpful:



`pivot_longer()` function

Let's keep going with our current example, starting from `wideGenericMore`

```
longGeneric <- wideGenericMore %>%
  pivot_longer(cols = 3:8,
               names_to = "valueType",
               values_to = "allScores")

longGeneric
```

```
## # A tibble: 18 x 4
##    Subject hairColor valueType   allScores
##      <dbl> <chr>     <chr>           <dbl>
##  1       1 brown     Score_1           0.2
##  2       1 brown     Score_2           0.4
##  3       1 brown     Score_3           0.3
##  4       1 brown     happiness_1      10
##  5       1 brown     happiness_2       9
##  6       1 brown     happiness_3      10
##  7       2 blonde    Score_1           0.8
##  8       2 blonde    Score_2           0.9
##  9       2 blonde    Score_3           0.7
```

`pivot_longer()` function

For both of these `pivot` functions, you can use the `-` (minus) sign to say "everything except this column". For example:

```
longGeneric <- wideGenericMore %>%
  pivot_longer(cols = c(-hairColor, -Subject),
               names_to = "valueType",
               values_to = "allScores")

longGeneric
```

```
## # A tibble: 18 x 4
##    Subject hairColor valueType   allScores
##      <dbl> <chr>     <chr>           <dbl>
##  1       1 brown     Score_1           0.2
##  2       1 brown     Score_2           0.4
##  3       1 brown     Score_3           0.3
##  4       1 brown     happiness_1      10
##  5       1 brown     happiness_2       9
##  6       1 brown     happiness_3      10
##  7       2 blonde    Score_1           0.8
##  8       2 blonde    Score_2           0.9
##  9       2 blonde    Score_3           0.7
```

The `pivot` functions

Some things to notice:

- In `pivot_longer`, the arguments take in strings (aka, need quotations!). That's because you need to tell R what to name something.
- In `pivot_wider`, the arguments take in variable names that already exist. So you do not need to wrap those in quotation marks.
- These are the types of functions that I mess up ALL. THE. TIME. Use your History tab!

`separate()` function

In our latest iteration, `longGeneric`, we have a column called `valueType` where it is a name, then an underscore (_), and a number, ex: `Score_1`.

We can use `separate()` to make split `valueType` into 2 separate columns...1 for the `Score` and another for the `1`.

```
longGeneric %>%
  separate(col = valueType,
           into = c("variableName", "timePoint"))
```

```
## # A tibble: 18 x 5
##    Subject hairColor variableName timePoint allScores
##      <dbl> <chr>     <chr>        <chr>          <dbl>
##  1       1 brown     Score        1                0.2
##  2       1 brown     Score        2                0.4
##  3       1 brown     Score        3                0.3
##  4       1 brown     happiness    1               10
##  5       1 brown     happiness    2                9
##  6       1 brown     happiness    3               10
##  7       2 blonde    Score        1                0.8
##  8       2 blonde    Score        2                0.9
```

`separate()` function

Note that I did not specify that I wanted to separate based on the underscore.

- When it is simple like this, R can automatically detect it.
- But if it's a bit trickier, you can specify how to separate in the `sep =` argument.
  - For example, `sep = ": "` if you want to separate on a colon + space.

`unite()` function

The opposite of separate is `unite()`. For instance, let's say we want to create a variable called `bogus` that looks something like `brown: Score` or `blonde: happiness`. The separator is a colon + space.

```
longGeneric %>%
  unite(col = "bogus",
        hairColor, valueType,
        sep = ": ")
```

```
## # A tibble: 18 x 3
##    Subject bogus                allScores
##      <dbl> <chr>                    <dbl>
##  1       1 brown: Score_1             0.2
##  2       1 brown: Score_2             0.4
##  3       1 brown: Score_3             0.3
##  4       1 brown: happiness_1        10
##  5       1 brown: happiness_2         9
##  6       1 brown: happiness_3        10
##  7       2 blonde: Score_1            0.8
##  8       2 blonde: Score_2            0.9
##  9       2 blonde: Score_3            0.7
```

Missing values in `tidyverse`

- Like `base R` and others, many `tidyverse` functions have an argument for `na.rm =`.
- You can add a `drop_na()` function to your `tidyverse` chunk. This function is part of `tidyr` and it will get rid of any rows that contain missing values. It's the equivalent of `na.omit()`
- Do everything in your power to make sure missing values are treated as `NA` and *not* something else. Ex:
  - `999` -- Many measurements can have a value of 999...
  - `" "` -- Spaces are treated as a character string, not truly missing! Remember, the class of your object is based on the least specific object. So if you have a vector of integers, but one missing value that is `" "`, the class of your vector will be a character! Same thing goes for `.` (periods).
- If you have something like `999` and you want to replace that with an `NA`, either of the following will work:
  - `data[data == 999] <- NA` (for the entire dataset)
  - `data$column[data$column == 999] <- NA` (for a single column)
  - `data <- gsub(pattern = 999, replacement = NA, x = data)` (but this will find anything with 999, so be careful!)