IX Data wrangling

You may have noticed that the format of the `ablation` data frame is a bit peculiar. The Excel sheet you imported for the plotting exercise is probably not what you are used to getting from your colleagues, or working with yourself. It is, however, in the canonical format for storing and manipulating data that you should be using.

The hallmark of this canonical (tidy) format is that there is only one (set of) independently observed value(s) in each row. All of the other columns are identifying values. They explain exactly what was measured. This is also known as metadata in some circles.

More specifically, a tidy dataset is defined as one where:

- Each variable forms a column.
- Each observation forms a row.

When your data is in this format, it is straightfoward to subset, transform, and aggregate it by any combination of factors of the identifying variables. That is why, for example, the **ggplot** package essentially requires that your data is in tidy format.

The tidyverse that Hadley Wickham has been instrumental in creating has this format at its core, and his **tidyr** package includes functions to help coerce your data into this format. This section will also introduce another tidyverse package called **dplyr**, which is used to perform more complex manipulations on your data.

i. Going long

1. If you are given data in non-canonical format, you can use the `gather()` function to fix it. This will convert a data frame with several measurement columns (i.e., "fat" or "wide") into a "skinny" or "long" data frame which has one row for every observed (measured) value. The `gather()` function takes multiple columns that all have the same measurement type, and collapses them into key-value pairs, duplicating all other columns as needed.

   Let's start with a "fat" data frame that contains data about mouse weights.

   ```
   set.seed(1)
   mouse_weights_sim <- data.frame(
     time = seq(as.Date("2017/1/1"), by = "month", length.out = 12),
     mickey = rnorm(12, 20, 1),
     minnie = rnorm(12, 20, 2),
     mighty = rnorm(12, 20, 4)
   )
   ```

   This dataset consists of only one type of measurement - mouse weights - where each column in this dataset represents the weights of a given mouse over a year. The columns 'mickey', 'minnie' and 'mighty' are the names of each mouse, and each of the three columns contain weight data for that mouse. The tidy version of this data would have all the weight measurements in one column ("values") with another column detailing which mouse (or column) that measurement came from ("keys").

   ```
   mouse_weights <- gather(data = mouse_weights_sim, # data frame to be manipulated
     key = mouse,    # name of the future column storing the mouse names
     value = weight, # name of the future column storing the weight measurements
     mickey, minnie, mighty) # all the columns that contain the values
   ```



```
mouse_weights <- gather(data = mouse_weights_sim,
          key = mouse, value = weight, -time)
mouse_weights <- gather(data = mouse_weights_sim,
          key = mouse, value = weight, mickey:mighty)
```

After gathering our data, each variable forms a column. Our three variables are time, mouse, and weight. Each row is now an observation. Before tidying our data, each row represented three observations. *Note that the arguments to the key and value options become the names of the new columns.* Now that the data have been tidied, it is trivial to use as input to ggplot.

```
ggplot(mouse_weights, aes(x = mouse, y = weight)) +
   geom_boxplot(aes(fill = mouse))
   % mouse_weights$mouse <- factor(mouse_weights$mouse, levels = c("mickey", "minnie", "mighty
ggplot(mouse_weights, aes(x = time, y = weight)) +
   geom_boxplot(aes(group = time))
ggplot(mouse_weights, aes(x = time, y = weight)) +
   geom_boxplot(aes(group = time)) + geom_point(aes(color = mouse))
ggplot(mouse_weights, aes(x = time, y = weight)) +
   geom_point(aes(color = mouse)) + geom_line(aes(group = mouse, color = mouse))
```

2. Although you can only use the `gather()` function to tidy data structures such as data frames, you can always coerce other data structures into a format that can be used. For example, the USPersonalExpenditure dataset is a matrix, that we can coerce into a data frame, which can then be tidied as above.

```
uspe_df <- as.data.frame(USPersonalExpenditure)
uspe_df$Category <- rownames(USPersonalExpenditure)
uspe <- gather(uspe_df, Year, Amount, -Category)
```

Once tidied, the data can again be readily plotted with ggplot. Here we'll use stacked bar charts, showing the expenditure per year, colored by Category.

```
ggplot(uspe, aes(x = Year, y = Amount)) +
  geom_bar(stat = "identity", aes(fill = Category))

ggplot(uspe, aes(x = Year, y = Amount)) +
  geom_bar(stat = "identity", aes(fill = Category)) +
  theme(legend.justification = c(0,1),
        legend.position = "inside", legend.position.inside = c(0,1))

ggplot(uspe, aes(x = Year, y = Amount)) +
  geom_bar(stat = "identity", position = "dodge", aes(fill = Category)) +
  theme(legend.justification = c(0,1),
        legend.position = "inside", legend.position.inside = c(0,1))
```

By default, `geom_bar()` is set up to plot frequencies of categorical observations. Since we are plotting numerical values, we need to use the `stat = "identity"` option. In the second example, we have relocated the legend, and in the third, we demonstrated how we can draw the bars to be side-by-side, rather than stacked.

ii. Going wide

1. The complement of the `gather()` function is the `spread()` function. We can reshape our mouse weights to their original format.

   ```
   spread(data = mouse_weights, key = mouse, value = weight)
   ```

   Similarly, we can reshape our ablation dataset into a dataframe where there is one row per time point and one column per CellType.

   ```
   spread(ablation, key = CellType, value = Score)
   ```

   Note that all of the experimentally measured values in this table come from the original `Score` column; this is indicated by the `value` option in the above command.

2. It is also possible to have columns that are combinations of identifiers, but you will need to include an extra step of manually combining those columns first. Say we wanted a wide table where each of the measurement columns showed the value for a specific combination of Experiment and CellType. We would use another function from the **tidyr** package, `unite()`.

   ```
   abl_united <- unite(ablation, ExptCell, Experiment, CellType, sep = ".")
   spread(abl_united, ExptCell, Score)
   ```

   Here, `ExptCell` is the new column that we are defining, as a combination of Experiment and CellType, where the names of the identifiers will be separated by a period.

3. Finally, the opposite of the `unite()` function is `separate()`.

   ```
   separate(abl_united, ExptCell, c("Expt", "Cell"), sep = "\\.")
   ```

   Note that here, if the separator is a character string, it is interpreted as a regular expression, so we have to escape out the period character. The `separate()` function can be used to split any single column which captures multiple variables.

iii. Joining dataframes

1. It is usually a good idea to keep all of your data from a particular study or project in a very small number of canonical "skinny" data frames. Consider the ablation data we've been using; when new experiments are performed, you can add new rows to the `ablation` data frame with the `rbind` function. If the data for the new experiment is given to you in "fat" format (say via a new Excel workbook), you may need to `gather()` the new data first, and then `rbind()` it.

2. Sometimes, you will want to add new columns before doing this. For example, if all of the data thus far was collected by one tech, you probably did not bother to store that metadata. However, if, after some early success, your PI assigns another post-doc to the project, you may want to create a new data frame with this information and store it in the project environment.

   ```
   experiment_log <- data.frame(Experiment = c("E1909", "E1915", "E1921"),
                                Tech = c("Goneril", "Regan", "Cordelia"),
                                stringsAsFactors = TRUE)
   str(experiment_log)
   experiment_log
   ```

   When looking for technician-specific bias, you will need to merge the technician data with your main data. The **dplyr** package includes a number of functions to help with this.

   ```
   inner_join(ablation, experiment_log)
   ```


The `inner_join()` function merges two data frames based on common column values. By default, it looks for common column names, but these can be specified explicitly. The `inner_join()` function keeps only rows which have elements common to both data frames (it is similar to a database table inner join). You can force a left or right (or full) database-style join by using the `left_join()`, `right_join()` and `full_join()` functions, respectively.

The merged data frame contains redundant information; i.e., if you know the Experiment, you know the Tech (we say the data is "denormalized"). While the merged data frame may be convenient when investigating "tech effects", you probably don't want to store this data frame permanently. This becomes more important at scale, when the cost of storing the redundant information becomes a limiting factor (usually in terms of the memory needed by R).

Tip: Use `*_join()`! Don't depend on vectors being aligned unless you are absolutely, positively sure they are, and `*_join()` is not an option. Such assumptions are a very, very common source of errors in data analysis (not just in R – think about what you do when you paste a new column into an Excel sheet).

3. To save your work, use R's `save()` function. This will save it in an Rdata format, which can later be reloaded with the `load()` function. In the example below, we save a single object to a file; you can also pass a list of objects as the first argument to save the collection.

```
save(ablation, file = "ablation.Rdata")
load("ablation.Rdata")
```

iv. Subsetting with dplyr

The **dplyr** package has other functions to help you perform more complex manipulations, and a few others that will make your life easier. These include subsetting by columns (`select()`) and subsetting by rows (`filter()`). To some extent, these have the same functionality as indexing vectors, but especially as you start to chain together multiple operations, the **dplyr** functions will make the intent and readability of your code much clearer.

1. We can use `select()` to select columns. We'll use a new dataset called msleep (which has data about mammalian sleep cycles) to demonstrate. The `select()` command below is exactly equivalent to a selection by column indexing vector.

```
head(select(msleep, name, sleep_total))
head(msleep[ , c("name", "sleep_total")])
```

2. Note that the data structure returned here looks a little different to what we are used to. The `msleep` dataset is called a tibble, which is essentially a data frame, with some small differences, which include the way that it is presented. When you print a tibble to the console, it will only display as many columns as will fit on the screen (while also listing the unseen columns at the bottom), displays the first 6 rows, and also tells you what data type each column consists of. You can use it exactly as you would a data frame, and functions that don't know about tibbles will use it as if it were a data frame (and in fact, it is).

```
class(msleep)
```

3. One of the paradigms in the tidyverse is readability of code, and a powerful tool that is introduced for this is a "pipe", or `%>%`. This is analogous to the pipe in Unix pipelines. The pipe will take the output from whatever is on the left hand side, and treat it as the first argument to the function on the right hand side. The `%>%` operator is loaded automatically once you load any of the **dplyr** packages, but comes specifically from a package called **magrittr**.


```
msleep %>% select(name, sleep_total) %>% head
msleep %>%
  select(name, sleep_total) %>%
  head
```

4. The `select()` function allows you to treat column names as their numeric position, so that anything you can do with numeric positions, you can do with the variable names. It is always a better idea to refer to variables by name, rather than position; it is much less error-prone, and you don't have to preserve order.

   Note also that, just as in `ggplot`, when referring to variables, we never have to refer to the data frame explicitly.

   ```
   head(msleep[ , -1])
   head(select(msleep, -name))
   head(select(msleep, -c(name, sleep_total)))
   msleep %>%
     select(-c(name, sleep_total)) %>%
     head
   ```

5. There are also a number of convenience functions that go along with `select()`. See `help(select)` or the dplyr cheatsheet for a complete list of other helper functions.

   ```
   msleep %>%
     select(starts_with("sl")) %>%
     head
   head(msleep[ , startsWith(names(msleep), "sl")])
   ```

   Exercise:

   a. Select all columns that have "wt" in their names.

   b. Select the name, genus and order variable columns.

6. The `filter()` function is used to select rows, similar to the row indexing vector.

   ```
   msleep[msleep$sleep_total >= 16, ]
   msleep %>%
     filter(sleep_total >= 16)

   msleep %>%
     filter(order %in% c("Perissodactyla", "Primates"))
   msleep[msleep$order %in% c("Perissodactyla", "Primates"), ]
   ```

7. By default, multiple arguments are chained together with logical AND.

   ```
   msleep %>%
     filter(sleep_total >= 16, bodywt >= 1)
   msleep %>%
     filter(sleep_total >= 16 & bodywt >= 1)
   msleep[msleep$sleep_total >= 16 & msleep$bodywt >=1, ]
   ```

   Exercise:

   a. Select all the rows where the order is Carnivora or Primates.

   b. Select all rows where `sleep_total` is between 10 and 15.

c. Select all rows where the `sleep_total` was more than 4 times as long as `sleep_rem`.

d. How would you filter out all rows where `brainwt` was unknown (*Hint: use* `is.na()`)?

8. Another useful function is `arrange()` which reorders rows by the values in one or more columns. The `desc()` function reverses the direction of the ordering.

```
msleep %>% arrange(order) %>% head

msleep %>%
  arrange(desc(order)) %>%
  head

msleep %>%
  select(name, order, sleep_total) %>%
  arrange(order, sleep_total) %>%
  head
```

9. It is common to use a column solely to drive ordering, but without actually seeing it.

```
msleep %>%
  arrange(order, sleep_total) %>%
  select(name, order) %>%
  head
```

Exercise:

a. Arrange the rows by bodyweight, from largest to smallest, showing only the `name` and `bodywt` columns.

b. Arrange the rows by the length of non-rem sleep.

v. Summarizing data by groups

1. When we were using `gather()` and `spread()` earlier we were only rearranging (and optionally subsetting) the raw data. In other words, every value in the new data frame could be found in the original data frame. The **dplyr** package has functions that allow us to summarize our data (this is also known as data aggregation).

2. Let's use a smaller dataset to explore these capabilities. The first function we look at is `summarize()`. This will run a summary function (like `mean()`) on a column and return the result in a new dataframe.

```
ToothGrowth %>%
  summarize(meanLen = mean(len))
```

3. On its own, it will always return a data frame with a single row. This is not terribly useful, though! We already know other ways of getting this information. Where this becomes extremely useful is in combination with the `group_by()` function. This allows you to subset your dataset by a set of one or more "grouping" variables, and run the summary functions per group.

```
ToothGrowth %>%
  group_by(supp) %>%
  summarize(meanLen = mean(len))
```

The `group_by()` function allows you to define your unit of interest, here the supplement, and evaluate one or more expressions *in the context of the group*. Running the `group_by()` function on its own will

return the entire dataset, but rearranged into the required groupings. The resultant tibble knows how many groups there are, and how many observations are in each.

4. You can use combinations of variables to subset your data into groups.

```
ToothGrowth %>%
  group_by(supp, dose) %>%
  summarize(meanLen = mean(len), n = n())
```

Note that we included two summary functions here. The `n()` function returns the number of observations defined by the current grouping. It is generally a good idea to include this information, to get a sense of how robust the description of that group is, and perhaps later filter by the minimum number of observations.

You can use your own functions as arguments to the `summarize()` function, but at this time, you are restricted to only returning a single value from the summarizing function. There are ways around this, but they are outside the scope of this class.

5. The final **dplyr** tool is the `mutate()` function. Unlike `summarize()`, which results in a new data frame, `mutate()` adds a new column to the input data frame, and computes a value for each row. Like `summarize()`, `mutate()` can output multiple new columns.

```
ToothGrowth %>%
  group_by(supp, dose) %>%
  mutate(norm.len = (len - mean(len))/sd(len), max = max(len)) %>%
  print(n = 60)
```

Here, `mean()` and `sd()` are computed on the lengths defined by each group, not the lengths of the entire dataset.

Exercise:

a. Repeat the previous exercise with the msleep dataset, but this time, also add a new variable with the length of non-REM sleep.

b. How would you check if the sleep_total and awake columns for each organism added up to 24 hours?

6. Let's use our new functions to further explore the ablation dataset. Some recap first:

Exercise:

a. Reshape the ablation dataset so that there is a column for each CellType, and removing the Direction column.

b. Reshape the ablation dataset so that every unique combination of Time and CellType has its own column. Again, remove the Direction column.

7. We can chain many operations together. What does this do?

```
ablation %>%
  select(Time, Measurement, CellType, Score) %>%
  group_by(Time, Measurement, CellType) %>%
  summarize(mean_score = mean(Score)) %>%
  spread(CellType, mean_score)
```

Note that the `spread()` function refers to a variable newly created by the function directly before.

8. Other examples of useful summary functions include `min()`, `max()`.


```
ablation %>%
  select(Time, Measurement, CellType, Score) %>%
  group_by(Time, Measurement, CellType) %>%
  summarize(min = min(Score), max = max(Score))
```

9. We can compute the mean and standard deviation within groups too. If we assign these results to a new data frame, we can use them as input to `ggplot`.

```
ablation_mean_sd <- ablation %>%
  select(Time, Measurement, CellType, Score) %>%
  group_by(Time, Measurement, CellType) %>%
  summarize(mean = mean(Score), sd = sd(Score))

ggplot(ablation_mean_sd, aes(x = Time, y = mean)) +
  geom_point(size = 4) +
  geom_errorbar(aes(ymin = mean - sd, ymax = mean + sd), width = 0.4) +
  facet_grid(Measurement ~ CellType) +
  geom_line() +
  geom_point(data = ablation, aes(y = Score), color = "pink", shape = 1) +
  labs(title = "+/- 1 SD")
```

   In the above plot, we used the `geom_errorbar()` function which requires a unique aesthetic that binds `ymax` and `ymin` to the upper and lower bounds of the error bars.

10. Confidence intervals computed from a t-test are often used as the limits of the error bars, but including those in a similar figure is a little less elegant, because the summarizing function currently cannot return more than one value.

```
ablation_mean_ci <- ablation %>%
  select(Time, Measurement, CellType, Score) %>%
  group_by(Time, Measurement, CellType) %>%
  summarize(mean = mean(Score),
            lower_limit = t.test(Score)$conf.int[1],
            upper_limit = t.test(Score)$conf.int[2])
```

11. Let's use the `mutate()` function to add another column to our data frame, calculating the rate of ablation. Note that at Time 0, the rate cannot be calculated and is therefore unknown.

```
ablation %>% mutate(rate = ifelse(Time > 0, Score / Time, NA))
```

   Note the use of the `ifelse()` function. This is a vector operation that tests the expression given as the first argument for every element in a vector. If the expression evaluates to TRUE, the second argument is the result, otherwise the third one is. The `mutate()` function adds a column to the ablation data frame with the computed result.

12. When coupled with `group_by()`, `mutate()` can compute a value for every line that is a function of some grouping. In the following example, we use `mutate()` with `group_by()` to determine whether a Score is an outlier within a group of experiments (here we define an outlier as being outside of ± 1 SD of the mean.

```
ggplot(ablation_mean_sd, aes(x = Time, y = mean)) +
  geom_point(size = 2) +
  geom_errorbar(aes(ymin = mean - sd,
                    ymax = mean + sd), width = 0.4) +
  facet_grid(Measurement ~ CellType) + geom_line() +
```



```
  geom_point(data = ablation %>%
                    group_by(Measurement, CellType, Time) %>%
                    mutate(outlier = abs((Score - mean(Score)) / sd(Score)) > 1),
             aes(y = Score, color = outlier), size = 4, shape = 1) +
  labs(title = "+/- 1 SD", y = "Mean") +
  scale_colour_discrete(name = "Outlier Status",
                        labels = c("Within 1 SD", "Outside 1 SD"))
```

Here, the data argument to the second `geom_point()` function is an inline call to **dplyr** functions. This is not recommended in practice, but is shown to give you an idea of what is possible. Also, note that a more appropriate cutoff for outliers is ± 3 SD.

X Reproducible analysis

To facilitate reproducible analysis, it is a best practice to write a script that loads your raw data, runs your entire analysis, and produces appropriate plots and output without any intervention. Keeping the script open in the Source panel in RStudio and checking the Source on Save option can be helpful as you develop your script.

An example based on the material we have covered in this workshop is shown below.

```
library(tidyverse) # using ggplot2, dplyr, tidyr packages

analyze.all <- function(save_plots = TRUE) {

  # Load data
  ablation <- read.csv(file = "Ablation.csv",
                       header = TRUE,
                       stringsAsFactors = TRUE)
  names(ablation)[names(ablation) == "SCORE"] <- "Score"
  print(ablation)

  ablation_means <- ablation %>%
    group_by(CellType, Measurement, Time) %>%
    summarize(mean = mean(Score), n = n())
  print(ablation_means)

  # Set up plotting
  if (save_plots) {
    pdf(file = "plot.pdf")
  }

  # Plot all data
  g <- ggplot(ablation, aes(x = Time, y = Score)) +
    geom_point() +
    geom_line(aes(color = Experiment)) +
    facet_grid(Measurement ~ CellType) +
    theme_bw()
  print(g)

  # Plot averages over experiments
  g <- ggplot(ablation_means, aes(x = Time, y = mean)) +
    geom_point() +
    geom_line(aes(color = CellType)) +
    facet_wrap(~ Measurement) +
    theme_bw()
  print(g)

  # Separate plots of averages over experiments
  for (measurement in levels(ablation.means$Measurement)) {
    g <- ggplot(ablation_means %>%
                filter(Measurement == measurement), aes(x = Time, y = mean)) +
      geom_point() +
```



```
      geom_line(aes(color = CellType)) +
      labs(title = measurement) +
      theme_bw()
    print(g)
  }

  # Close plotting device
  if (save_plots) {
    dev.off()
  }
}

analyze.all(FALSE)
```

Note the use of `for` loops and `if` blocks. Control structures such as these are often needed to ensure your script can run autonomously. Here we use an `if` block to control whether plots are saved to a PDF or viewed in RStudio, a technique that can be handy when developing your script.

Also note that the script works when invoked with the appropriate working directory and with an empty environment. It is important that you test this to ensure that you are not dependent on some objects left in your workspace from interactive sessions.

We close by noting that some journals, such as PLoS One, are now requiring scripts such as these to address the problem of imprecise or incomplete descriptions of analysis methods.

