VII CRAN and Libraries

One of the major advantages of using R for data analysis is the rich and active community that surrounds it. There is a rich ecosystem of extensions (also known as libraries or packages) to the base R system. Some of these provide general functionality while others address very specific tasks.

The main hub for this ecosystem is known as CRAN (Comprehensive R Archive Network). CRAN can be accessed from `https://cran.r-project.org/`. This is also where you go to download the R software.

Follow the `Packages` link to browse the 5000+ packages currently available.

Because `R` is a not a very specific search term, often when doing a web search, the term `CRAN` is used.

In the next section, we will learn how to use the ggplot2 package for preparing publication-quality figures. Here we will download and install the tidyverse package, which includes ggplot2, as well as dplyr and tidyr which we will be using shortly. This couldn't be easier, because R knows all about CRAN.

```
install.packages("tidyverse") # Library name is given as a string
```

If this is the first time a package is being installed on your computer, R may ask you to select a CRAN mirror. Pick something geographically close by. Note that you only have to install a package once per R install.

Depending on how your computer (and R installation) is set up, you may receive a message indicating that the central location for packages is not writable; in this case R will ask if you want to use a personalized collection of packages stored in your home directory.

Installing a package does not make it ready for use in your current R session. To do this, use the `library()` function.

```
library(tidyverse)              # Library name is an object (not a string)
```

You need to do this in every session or script that will use functions from this library.

VIII Plotting

Although R has some basic plotting functionality which we have seen hints of, the **ggplot2** package is more comprehensive and consistent. We'll use ggplot2 for plotting for the rest of this workshop.

ggplot2 is written by Hadley Wickham (`https://hadley.nz/`). He maintains a number of other libraries; they are of excellent quality, and are very well documented. However, they are updated frequently, so make sure that you are reading the current documentation. For ggplot2, this can be found at...

`https://ggplot2.tidyverse.org/reference/`

In this workshop, we will also be using his **tidyr** and **dplyr** packages.

ggplot2 relies entirely on data frames for input.

1. Let's make our first ggplot with the `ablation` data that we imported earlier.

   ```
   ggplot(ablation, aes(x = Time, y = Score)) + geom_point()
   ```

   At a minimum, the two things that you need to give ggplot are:

   a. The dataset (which must be a data frame or an object that can be interpreted as one), and the variable(s) you want to plot

   b. The type of plot you want to make.

2. ggplot gives you exquisite control over plotting parameters. Here, we'll change the color and size of the points.

   ```
   ggplot(ablation, aes(x = Time, y = Score)) + geom_point(color = "red", size = 4)
   ```

   Aesthetics are used to bind plotting parameters to your data.

   ```
   ggplot(ablation, aes(x = Time, y = Score)) +
     geom_point(aes(color = Experiment), size = 4)
   ggplot(ablation, aes(x = Time, y = Score)) +
     geom_point(aes(color = Experiment, shape = CellType), size = 4)
   ```

3. When using ggplot, layers are added to a ggplot object, in order. You can add multiple layers.

   ```
   ggplot(ablation, aes(x = Time, y = Score)) +
     geom_point(aes(color = Experiment), size = 4) +
     geom_text(aes(label = CellType), hjust = 0, size = 3)
   ```

   It is sometimes useful to save off the base ggplot object and add layers in separate commands. The plot is only rendered when R "prints" the object. This is useful for several reasons:

   a. We don't need to create one big huge command to create a plot, we can create it piecemeal.

   b. The plot will not get rendered until it has received all of its information, and therefore allows ggplot2 to be more intelligent than R's built-in plotting commands when deciding how large a plot should be, what the best scale is, etc.

   ```
   p <- ggplot(ablation, aes(x = Time, y = Score))
   p <- p + geom_point(aes(color = Experiment, shape = Measurement), size = 4)
   p <- p + geom_line(aes(group = interaction(Experiment, Measurement, CellType),
                          color = Experiment,
                          linetype = CellType))
   print(p) # plot gets rendered now
   ```


Sourcing a file will not automatically generate output, so here we have to explicitly ask for the plot to be printed.

Here we've added a layer that plots lines. We want a separate line for each unique combination of Experiment, Measurement, and CellType. The `interaction()` function takes a set of factors, and computes a composite factor. Try running...

```
interaction(ablation$Experiment, ablation$Measurement, ablation$CellType)
```

...to see what this does. This composite factor is passed to the group aesthetic of `geom_line()` to inform ggplot which data values go together.

We have also added a new binding to `geom_point()`. The shape of each point is determined by the corresponding Measurement. Note that ggplot prefers six or fewer distinct shapes (i.e., there are no more than six levels in the corresponding factor). You can, however, use more using a command like...

```
scale_shape_manual(values = 1:11)
```

Here we specify the shapes we want to use, as well as jittering the points slightly so they no longer sit directly on top of one another.

```
p <- ggplot(ablation, aes(x = Time, y = Score))
p + geom_point(aes(color = Experiment, shape = Measurement),
                   size = 4, position = position_dodge(0.5)) +
  scale_shape_manual(values = c(1,16))
```

We'll show you how to draw a plot listing all the possible shapes at the end of this section.

4. In the above example, we specified that the color of both points and lines should be determined by the Experiment. It is therefore tidier to specify this binding once in the ggplot aesthetic. Any bindings defined there are inherited by all layers (but can be overridden by any individual layer's aesthetic).

```
p <- ggplot(ablation, aes(x = Time, y = Score, color = Experiment))
p <- p + geom_point(aes(shape = Measurement), size = 4)
p <- p + geom_line(aes(group = interaction(Experiment, Measurement, CellType),
                       linetype = CellType))
print(p)
```

5. Some layers don't plot data, but affect the plot in other ways. For example, there are layers that control plot labels and plot theme (there are eight themes built-in to `ggplot2` and many more available at `https://github.com/jrnold/ggthemes`). The `labs()` function also modifies legend labels.

```
( p <- p + labs(title = "Ablation", x = "Time (minutes)", y = "% Saturation") )
( p <- p + theme_bw() + theme(plot.title = element_text(h = 0.5)))
```

6. ggplot gives you control over the scales of your plot. There is one scale for each binding. In the plot we just made, there are five scales that we can manipulate: the x and y axes and the three legends.

   Let's change our x-axis to include the 5 minute timepoint. This is achieved with yet another layer.

```
p + scale_x_continuous(breaks = c(0, 5, 10, 20, 30))
p + scale_x_continuous(breaks = unique(ablation$Time))
```

   Tip: In the second example above, we have computed the breaks from the data, rather than listing them individually. This makes the code we are writing usable even when the data changes. This is an essential strategy for reproducibly analyzing data at scale.

7. We can also manipulate legends with scale layers.


```
p <- p + scale_shape_manual(values = c(1,16), labels = c("LDLR", "TfR")) +
         scale_linetype_discrete(name = "Cell type") +
         scale_color_manual(values = c("blue", "red", "green"))
```

Here we provide the labels for the Measurement scale (remember that we used an aesthetic to bind shape to Measurement). Note that ggplot will always order the labels according to the levels of the underlying factor, so the labels should be provided in that order. If you want to change the order in which the legend elements are displayed, change the underlying factor.

We have also changed the title of the CellType legend (the linetype binding) to be two words and used a different color palette (for the binding to Experiment).

8. You can use built-in color palettes from ColorBrewer (`https://colorbrewer2.org/`). To see all available palettes:

   ```
   library(RColorBrewer)
   display.brewer.all()
   ```

   and to use a ColorBrewer palette in your plot:

   ```
   scale_color_brewer(palette = "Set1")
   ```

9. Note the general form of the scale layer functions:

   ```
   scale_aestype_colortype
   ```

   where the aestype is the bound aesthetic, and the colortype is the type of color associated with that binding.

   Common values for the aestype and colortype include:

Aestype	
colour	Color of lines and points
fill	Color of area fills (e.g. bar graph)
linetype	Solid/dashed/dotted lines
shape	Shape of points
size	Size of points
alpha	Opacity
x,y	x and y axes

Colortype	
hue	Equally-spaced colors from the color wheel
manual	Manually-specified values (e.g., colors, point shapes, line types)
gradient	Color gradients
grey	Shades of grey
discrete	Discrete values (e.g., colors, point shapes, line types, point sizes)
continuous	Continuous values (e.g., alpha, colors, point sizes)

Table 1: Scale layer function components.

10. This plot is probably showing too much data at once. One approach to resolve this would be to make separate plots for the LDLR and TfR measurements. You can make multiple plots at once using facets. Here are a few options.

    ```
    p + facet_grid(Measurement ~ .)
    p + facet_grid(. ~ Measurement)
    ```



```
p + facet_grid(Experiment ~ Measurement)
p + facet_grid(Measurement ~ Experiment)
```

In these plots, you can remove the color and shape legends entirely (an option that can be specified in each of the respective legend layers)...

```
p + facet_grid(Measurement ~ Experiment) +
  scale_color_discrete(guide = "none") + scale_shape_discrete(guide = "none")
```

...or you may no longer want to bind the Measurement and Experiment variables to shape and color at all.

Tip: The `facet_wrap()` function in ggplot can be used to wrap a 1D ribbon of plots into a 2D layout. You can also use the **gridExtra** package to place independently generated plots on the same page.

11. When plotting many points, controlling opacity can be useful. Let's model an ant-infested park. (To create a reproducible data set, you can set the seed for the random number generator with the `set.seed()` function).

```
trees <- data.frame(x = rnorm(100), y = rnorm(100),
  size = rnorm(100, mean = 5, sd = 2))
ants <- data.frame(a = rnorm(10000, sd = 0.4, mean = 1),
  b = rnorm(10000, sd = 0.2, mean = -1))
p1 <- ggplot() +
  geom_point(data = ants, aes(x = a, y = b),
    color = "brown", size = 2, alpha = 0.01) +
  geom_point(data = trees, aes(x = x, y = y, size = size),
    color = "green", shape = 8)
print(p1)
```

Note that here we are plotting points from two different data frames, so we don't provide a default dataset or default bindings to x or y in the `ggplot()` function. These can always be set (or overridden) in individual layers.

Exercise:

a. Compare the result without specifying alpha.

12. ggplot can do statistical analyses on the fly. We won't cover the details here, but here's an example to whet your appetite:

```
p2 <- ggplot() +
  stat_density2d(data = ants, aes(x = a, y = b, fill= ..level.. ),
                 alpha = 0.5, geom = "polygon") +
  geom_density2d(data = trees, aes(x = x, y = y)) +
  geom_point(data = trees, aes(x = x, y = y, size = size),
             color = "green", shape = 8) +
  scale_fill_gradient(low = "white", high = "brown") +
  scale_size_continuous(guide = "none")
   # or scale_size_continuous(name = "Tree size")
print(p2)
```

Note the use of the binding to the `..level..` variable. This binds to a statistic (in this case, the 2D density of points) computed by the `stat_density2d()` layer (computed variables are identified by the appended and prepended "..").

13. You can ask R to produce your plots as PDF files rather than display them on the screen.

```
pdf(file = "figures.pdf", paper = "letter")
print(p)
print(p1)
print(p2)
dev.off()
```

    Whenever a PDF device is open, all plotting (with ggplot or otherwise) will be to the PDF file, not to the RStudio plot tab. So don't forget that `dev.off()` command.

    Note that you can also use RStudio to export individual plots as PDFs or PNGs from the Plot tab.

14. As promised above, here we show some code that prints a reference of all the 26 shapes that are available. Note that several shapes can also have a fill aesthetic.

```
i <- 0:25; x <- i %/% 6; y <- i %% 6
df <- data.frame(name = i, row = x, column = y)
ggplot(df, aes(x = row, y = column)) +
  geom_point(shape = 0:25, fill = "yellow", size = 4) +
  geom_text(label = i, vjust = 2) +
  scale_y_reverse() +
  theme(axis.title.x=element_blank(), axis.text.x=element_blank(),
        axis.ticks.x = element_blank(),
        axis.title.y=element_blank(), axis.text.y=element_blank())
```

15. A great resource to check out is `https://r-graph-gallery.com/`, which showcases many different types of graphs that can be made with various external packages. Note the ggplot2 section down at the bottom. Also note that it gives you the code for all the example plots!

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 what exactly 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_sim_weights, # 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))
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 = 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 = 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?

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.

