Accessing data files in R

The here package has a function called here() that uses a set of sensible rules to work out what the working directory is. If you use an RStudio project (like I’ve told you to) here() can easily locate the working directory because it’s the project directory. Let’s assume that the location of your project is C:/Users/andyfield/Documents/my\_new\_project/, executing the function will return that location:

> here::here()

[1]"C:/Users/andyfield/Documents/my\_new\_project"

If you type the location of a file within the project folder into the function, it appends this text to the location of the project directory, to give you the absolute path. For example, if we want to access the file called ‘my\_funky\_data.csv’ which is within a folder called ‘data’ within our project folder, we could get the filepath to this file by executing:

> here::here("data/my\_funky\_data.csv")

[1] C:/Users/andyfield/Documents/my\_new\_project/data/my\_funky\_data.csv"

The here() function generates the filepath for the project directory each time it is called, so if you move your project folder to another machine or a different location on the same machine your code will still work. For example, previously we looked at loading some data using a relative path:

funky\_tib <- readr::read\_csv("../data/my\_funky\_data.csv")

We can avoid the relative path by using:

funky\_tib <- readr::read\_csv(here::here("data/my\_funky\_data.csv"))

To sum up, we use here() to create a filepath to the file we want to read into R, and use readr::read\_csv() to open that file.

Creating variables from variables

We can add variables (columns) to a tibble using the mutate() function from the dplyr package (which is part of tidyverse). Often when doing this we want to use operators to manipulate variables: for example we might want to use + to create a new variable by adding two existing variables together. Table 1 shows a list of commonly-used operators.

| Table 1: Summary of the main operators in R | |
| --- | --- |
| **Operator** | **What\_it\_does** |
| + | Adds things together |
| - | Subtracts things |
| \* | Multiplies things |
| / | Divides things |
| ^ or \*\* | Exponentiation (i.e., x^2 or x\*\*2 is x squared |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| == | Equals (note that ‘==’ is used rather than the conventional ‘=’) |
| != | Not equal to |
| !x | Not x |
| x | y | x OR y (e.g., name == “Zach”|“Jessika” means ‘the variable name is equal to either Zach or Jessika’) |
| x & y | x AND y (e.g., age == 27 & name == “Zach” means ‘the variable age is equal to 27 and the variable name is equal to James) |
| isTRUE(x) | Test if x is TRUE |
| sqrt(x) | Take the square root of x (e.g., sqrt(9) returns 3) |
| log(x) | Take the natural log of x |
| mean(x) | Comopute the mean of x |

Using the pipe we could create a new variable that contains the average scores across the variables **humour** and **kind** by executing:

ha\_tib <- ha\_dat %>%

dplyr::mutate(mean\_hum\_kind = (humour + kind)/2)

let’s break this command down:

* ha\_tib <- tells **R** to re-create the tibble *ha\_tib* from whatever comes next
* ha\_tib %>% starts to recreate the tibble from the original *ha\_tib* tibble. The pipe operator tells R that there’s some processing on that tibble to follow
* dplyr::mutate(mean\_hum\_kind = (humour + kind)/2) uses the mutate() function to add a variable called **mean\_hum\_kind** which contains the scores of **humour** and **kind** added together and divided by two. (Inside mutate is one of the rare occasions you will use a single = sign in **R**).

Having done this we convert the variable to a factor using the built-in factor() function, which takes the general form:

factor(variable\_name, levels = c(x,y, … z), labels = c("label\_1", "label\_2", … label\_n"))

This looks a bit scary, but let’s break it down:

* factor(variable\_name) is all you really need to create the factor — in our case factor(sex) would do the trick.
* levels = c(1, 2, 3, 4, …): we need to tell R which values we want to use to denote different groups and we do this with the levels = argument. We can use the c() function to list the values we have used (c(0, 1)) or if we have used a regular series such as 0, 1, 2, 3, 4 we can use the shorthand x:y in which the colon means ‘all the values between’, so this would mean ‘all of the values between *x* and *y*’. For example, 1:4 is the same as c(1, 2, 3, 4) and 0:6 is the same as c(0, 1, 2, 3, 4, 5, 6). In our case, we used 0 and 1 to denote the two categories, so we could specify this as levels = 0:1 or levels = c(0, 1).
* labels = c("label\_1", "label\_2", … label\_n"): The final step is to assign labels to these levels using labels = c(“label”, …). Again, we use c() to collect together the labels that we wish to assign. The labels must be ordered consistent with the numeric levels, and you must provide a label for every level. In our case, 0 corresponds to male and 1 to female, so we would want to specify labels as labels = c("Male", "Female").
* Method two is to create a character variable and then use the as\_factor() function to convert it. For example, we could execute:
* sex <- c("Male", "Male", "Male", "Female")
* sex <- forcats::as\_factor(sex)
* The first command creates a character variable called sex containing the words male and female, the second command recreates the object sex by converting the original variable to a factor. Notice that with this method we don’t specify the levels of labels of the factor. This is done automatically, ordering the levels by the order they appear in the data (in this casel level 1 will be Male and level 2 Female). You can see this if you inspect the variable by executing its name. Try out this code:
* Exercise Run Code Start Over Solution
* 
* 1
* 2
* 3
* sex <- c("Male", "Male", "Male", "Female")
* sex <- forcats::as\_factor(sex)
* sex
* [1] Male Male Male Female
* Levels: Male Female
* If we don’t like the order of factor levels we can use the fct\_relevel() function (also from the forcats package) to change the order. For example, to reverse the order of the levels of sex we would execute:
* sex <- forcats::fct\_relevel(sex, "Female", "Male")
* The code box below repeats the code from the previous box, try executing it and then adding the code to relevel the factor so that Females are the first level.

Files in the working directory/project folder

Most likely you will find it easier to input data into a spreadsheet software (e.g., *Microsoft Excel*, *OpenOffice Calc*, *Numbers*, *NeoOffice*, *Google Sheets* etc.). I don’t recommend using *SPSS Statistics* for data entry (it is quite slow compared to, say, *Excel*). Although you can read data directly from an *Excel* or *SPSS* file it is generally more straightforward to export data from these packages into the comma-separated values, CSV, format. There are native functions for reading csv files (e.g., read.csv()) but the tidyverse way is to use the functions from the readr package that loads with tidyverse:

* read\_delim() and write\_delim(): read and write (respectively) a text file specifying how values are delimited with the argument delim = "". For example, you could read a csv file using read\_delim(file\_name, delim = ",").
* read\_csv() and write\_csv(): special cases of the above that assume comma separated values.
* read\_csv2(): special cases of read\_delim() that assume semi-colon separated values.
* read\_sav() and write\_sav(): read and write SPSS Statistics files. (You need to execute library(haven) to use these functions.)
* read\_excel(): read Microsoft Excel files (both .xls and .xlsx format). (You need to execute library(readxl) to use this function.)

## **ggplot2**

The best package for producing graphs in **R** is [ggplot2](http://ggplot2.tidyverse.org/) which automatically installs as part of the tidyverse package. ggplot2 is great because it is so versatile, but the price for its versatility is that it is extremely complicated. In my other book (Field, Miles, and Field 2012) I dedicate an entire chapter to it and still only scratch the surface. Through these tutorials we will learn by doing rather than me trying to explain every aspects of ggplot.

Figure 1 shows how ggplot2 works. You begin with some data and you initialize a plot with the ggplot() function within which you name the tibble or data frame that you want to use, then you set a bunch of aesthetics using the aes() function. Primarily, you name the variable you want plotted on the x-axis, the variable for the y-axis and any aesthetics that you want to set for the plot using a variable (for example, you might want to vary the colour of bars by levels of a variable.). You then add layers to the plot that control what the plot shows and the visual properties. For example, you might add bars to show group means, then layer on top error bars. There are various key concepts that relate to controlling aspects of the layers of the plot:

* **Aesthetics**: objects in a plot have visual aesthetics that control how they look. Examples of aesthetics are the fill colour of points and bars, line colours (of lines, error bars, lines around bars), the shape of data points, the size of data points, the type of line (full, dashed, dotted etc.). These can be set directly for an object (e.g., making all data points red) or can be set using a variable (e.g., colouring data points based on whether it came from an experimental or control group). When setting an aesthetic based on a variable we use the aes() function.
* **Geometric objects**: these are objects that represent data. Some examples are bars to represent means, dots to represent raw data (or a summary such as a mean), lines connecting data points or summarizing data (e.g., a line of best fit, lines connecting group means), error bars, and so on. There are different functions relating to different geoms, within which there will be different options that you can set, for example:
  + geom\_point() plots data points (by default dots)
  + geom\_bar() plots bars
  + geom\_boxplot() plots boxplots
  + geom\_histogram() plots histograms
  + geom\_errorbar() plots error bars
  + geom\_smooth() plots summary lines (e.g., linear models and splines)
* **Statistics**: there are situations where rather than using a geom function to display the data it is easier to map a summary of the data directly to the plot with various stat functions (usually stat\_summary()). It’s a little complex to explain when you use stats instead of geoms, so we’ll just learn by doing!
* **Scales**: These control the details of how the data are mapped to their visual objects. For example, you can control what appears on the x and y axis (i.e. intervals between values) using scale\_x\_continuous() and scale\_y\_continuous(), axis labels are controlled with labs(), and you can over-ride the default colour schemes for fills and colours by specifying your own colours within scale\_fill\_manual() and scale\_colour\_manual().
* **Coordinate system**: by default ggplot2 uses a Cartesian system, which will be fine for our purposes. However, we will use coord\_cartesian() to set the limits of the x and y axis. You may find coord\_flip() useful if you ever want to transpose the x and y axes.
* **Position adjustments**: sometimes elements of a plot overlap (e.g., lots of data points in the same place). There are various position adjustments that can be useful such as position\_dodge() which forces objects not to overlap side by side (handy for complex bar charts) and position\_jitter() which adds a small random adjustment to data points.
* **Facets**: facets can be used to plot different parts of the data in different panels. For example, if you wanted a plot of data from men and a separate plot of the same data for women and you wanted these plots side by side, you could do this with facet\_wrap().
* **Themes**: Themes are a set of instructions about how a plot should look (what font is used, font sizes, whether to show grid lines, colours of all visual elements, etc.). There are a number of built in themes that you can apply to your plots. At an advanced level you can even write your own themes and apply them. For example, when writing textbooks, I write a theme that controls fonts (family and size), fill colours, background colours, grid lines etc.. Whenever I create a plot I apply that theme and know that all of the plots within my book will look consistent. For our purposes we will just use built in themes, but occasionally over-ride defaults with the theme() function.

un the code below to view the histogram so far.

At the moment the *x*-axis is scaled from 5 to 10. Let’s show the full range of the scale. To do this we need to set the limits of the *x*-axis using the coord\_cartesian() function:

coord\_cartesian(xlim = c(begin, end), ylim = c(begin, end))

You set the limits of the *x*-axis using xlim and the limits of the *y*-axis with ylim. After each you specify numbers representing the start and end values for the axis. You need to collect these values into a single object by enclosing them in c(). So, to set the *x*-axis to begin at 1 and end at 10 (the lowest and highest points of the scale) add a + after the geom\_histogram() function and on the next line type:

coord\_cartesian(xlim = c(1, 10))

Add this line to the code box below and run the code to see how the limits of the *x*-axis change.

Tip: You may notice that you can also set limits using scale\_x\_continuous(). However, if you do this then data outside of the limits you set are discarded. Therefore, most of the time we would use coord\_cartesian() because it leaves the underlying data alone.

Now let’s add a layer that changes what breaks are displayed on the *x*-axis. At the moment the *x*-axis displays 2.5, 5, 7.5, and 10 Let’s change this to display the numbers 1 to 10 In **R** we can generate this sequence of numbers by using :. For example, 1:5 will return 1, 2, 3, 4, 5. The *x*-axis displays a continuous variable (**humour**) so we can use the function scale\_x\_continuous() to change aspects of this axis. In particular, the breaks = option will over-ride the default breaks along the axis.

In the code box below, type + after the coord\_cartesian() function and on the next line type:

scale\_x\_continuous(breaks = 1:10)

Milton also calculates the variance, standard deviation and inter-quartile range (IQR) of Alice’s ratings (without the outlier). Using **R** we can get these values using the functions var(), sd() and IQR() respectively. These functions behave exactly like mean() in that we input the variable for which we want the variance, standard deviation or IQR and specify how we treat missing values (by default they are not removed):

* var(variable\_name, na.rm = FALSE)
* sd(variable\_name, na.rm = FALSE)
* IQR(variable\_name, na.rm = FALSE, type = 7)

The IQR() function has an additional option of type = which allows you to specify one of 8 different ways to calculate the IQR. The default is 7. If you want to mimic what IBM SPSS Statistics does then you’d need to over-ride this default with type = 6, and there is an argument for using type = 8, which uses a method recommended by (Hyndman and Fan 1996). In the code box below, use these functions to get the variance, standard deviation and IQR for Alice’s ratings (with the outlier excluded).

Summmarize ()

Creating summaries by group

In several graphs in Nutcot’s report the data are summarized by group (for example JIG:SAW employees vs. non-employees). We can summarize data by applying the group\_by() and summarize() functions from the tidyverse package (specifically the dplyr package) to our tibble. For example, lets saw we wanted to get the mean visual acuity separately for the JIG:SAW employees and non-employees and store these values in a tibble called vis\_mean. We could achieve this using the following pipe command:

vis\_mean <- jig\_tib %>%

dplyr::group\_by(employee) %>%

dplyr::summarize(

mean\_vis = mean(vision)

)

Let’s break this command down:

* vis\_mean <- jig\_tib %>% translates as create an object called vis\_mean and the starting point is to copy the tibble called jig\_tib. The pipe operator (%>%) indicates that more processing will be done on the jig\_tib tibble before vis\_mean is created.
* dplyr::group\_by(employee) %>% tells **R** to group by the variable **employeee** and because this command comes after the pipe operator we know that the thing that we’re going to ‘group by’ employee is the object before the pipe operator (i.e. jig\_tib). If group\_by() is used outside of a pipe command then you need to specify a data frame or tibble within it otherwise it won’t know what object to group. the pipe operator again tells **R** that there is more processing to come.
* dplyr::summarize(mean = mean(vision)) creates a new variable called mean\_vis within the tibble and then uses the function mean(), which we learnt about in a previous tutorial, to place values into that variable.

The easiest way to get a feel for what’s happening is to execute the code and see what happens. The code is already types into the code box to show you a good way to format this kind of piped command. Note that the top line sets up the object to be created, then the commands to control *how* the object is created are indented on the lines below. You start a new line after each pipe operator (so that it’s easy to see the components of the pipe), and within the summarize() command we indent any commands. It may not be obvious why we do this, but it will when we move onto more complex examples.

mean(variable, trim = 0, na.rm = FALSE)

Which just says that you need to include a reference to the data that you want the mean for, and that you can set two options:

* **trim**: allows you to trim the value before calculating the mean by specifying a value between 0 and 0.5. The default is 0 (no trim), but if you wanted to trim 10% of of each end of the data you could set *trim = 0.1*, similarly for a 20% trim from each end set *trim = 0.2*.
* **na.rm**: you’ll see this in many functions and it stands for ‘NA remove’. In **R** missing values are denoted as ‘NA’ (not available), so by setting *na.rm = TRUE* (or *na.rm = T* for short) you ask **R** to remove missing values before computing the mean. The default is *na.rm = FALSE* so if the function throws an error this might be because you have missing values and you need to set na.rm to true.

The function for the median has a similar format except that there isn’t an option to trim the data because that wouldn’t make sense (the median is effectively the data with a 50% trim):

## **Plotting grouped data**

### Boxplots

Figure 5.6 in An adventure in statistics shows a boxplot of **footspeed** split by **employee**. This is easy to create in ggplot using the geom\_boxplot() function. We could set up the plot with this command:

speed\_box <- ggplot2::ggplot(jig\_tib, aes(employee, footspeed))

speed\_box +

geom\_boxplot() +

labs(x = "Employee status", y = "Footspeed (mph)") +

theme\_bw()

Let’s break down this command (refer back to the tutorial on ggplot2 if you need to):

* speed\_box <- ggplot2::ggplot(jig\_tib, aes(employee, footspeed)) creates an object called speed\_box that contains the plot. The ggplot() function is then used to specify that the plot uses the jig\_tib tibble and plots the variable **employee** on the x-axis and the variable **footspeed** on the y-axis.
* speed\_box + geom\_boxplot() + takes the object speed\_box and adds a boxplot geom to it.
* labs(x = "Employee status", y = "Footspeed (mph)") + applies descriptive labels to the x and y axes.
* theme\_bw() applies a black and white theme to the plot

Copy this code into the box and run it to see the boxplot.

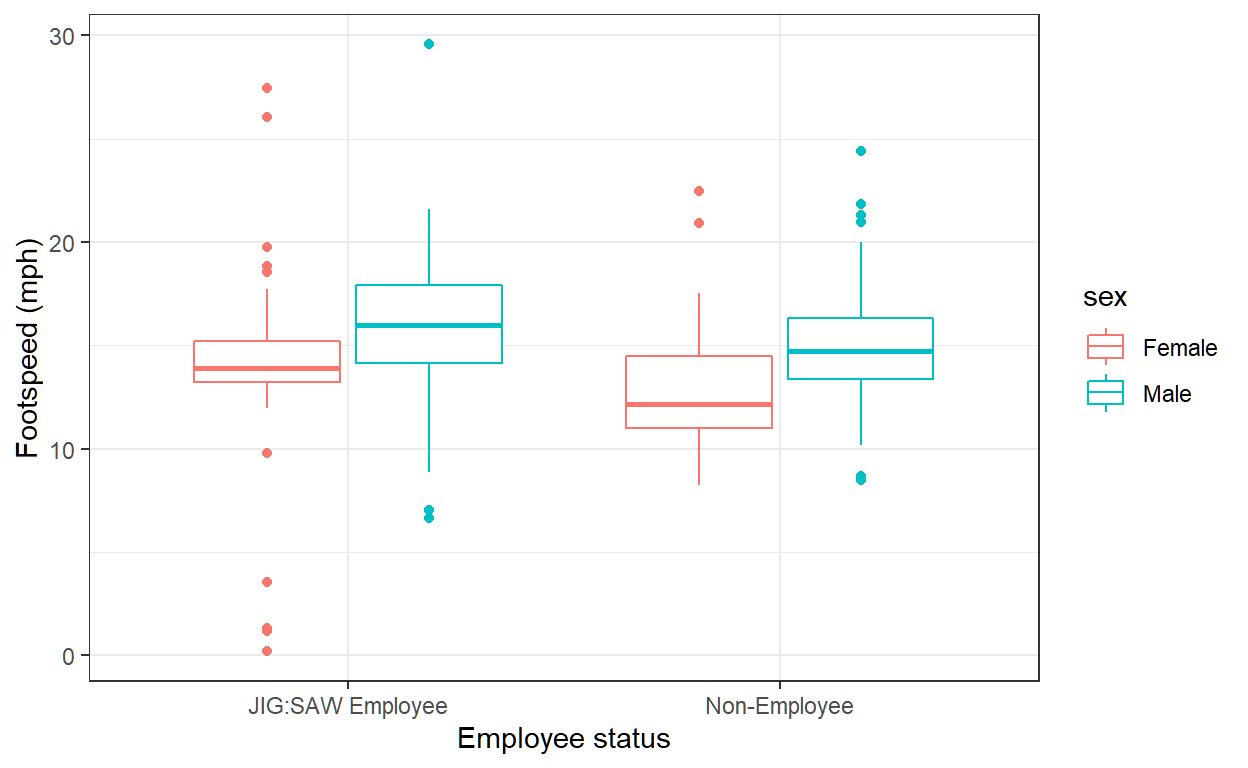
speed\_box <- ggplot2::ggplot(jig\_tib, aes(employee, footspeed, colour=sex))

speed\_box +

geom\_boxplot() +

labs(x = "Employee status", y = "Footspeed (mph)") +

theme\_bw()



Job done. If we wanted to split the data by a second grouping variable (for example **sex**) we can do this by specifying that ggplot varies the fill of the boxes or the colour of the lines around the boxes by this second variable. To do this we add this specification to the aes() function in the original command to set up the graph. For example, we’d change the first line to be:

speed\_box <- ggplot2::ggplot(jig\_tib, aes(employee, footspeed, fill = sex))

Note that all I have done is to add fill = sex to the initial aesthetic. The rest of the command can stay the same. Adapt the code above to include fill = sex and run it again. Note that the graph now splits the data by **employee** along the x-axis and within that splits it by **sex** using different colours to fill the boxes. Now change the word ‘fill’ to ‘colour’ and run the code again. You should see that now the data and the box outlines and whiskers are coloured differently for men and women.

Having added another variable to the plot, we might want to exit the label displayed on the graph. Currently we have specified labels for the x- and y-axis by including:

labs(x = "Employee status", y = "Footspeed (mph)")

If we want to edit the label for the variable that is used to determine the fill or colour of the plot, we can add this to the labs() function. If we used **sex** to determine the fill of the plot then we’d add fill = "label", where label is the text we want to use:

labs(x = "Employee status", y = "Footspeed (mph)", fill = "Biological sex")

If we had used **sex** to determine the colour of the plot then we’d add colour = "label" to the function:

labs(x = "Employee status", y = "Footspeed (mph)", colour = "Biological sex")

What if you want to change the colours that are used to fill the boxes or the box outlines? That’s easy. You can add the scale\_fill\_manual() or scale\_colour\_manual() functions to the command. Both functions have a similar syntax:

scale\_fill\_manual(values = c("colour\_1", "colour\_2", "colour\_3" ...))

In which you change “colour\_1” etc. to be the name of a colour of a hex code. In this example, we’re using two colours. Let’s change them to be a colour-blind friendly blue (hex code #56B4E9) and a colour-blind friendly orange(hex code #E69F00). The function would be:

scale\_fill\_manual(values = c("#56B4E9", "#E69F00"))

Our completed code would be:

speed\_box <- ggplot2::ggplot(jig\_tib, aes(employee, footspeed, fill = sex))

speed\_box +

geom\_boxplot() +

labs(x = "Employee status", y = "Footspeed (mph)", fill = "Biological sex") +

scale\_fill\_manual(values = c("#56B4E9", "#E69F00")) +

theme\_bw()

Copy this code into the box below and run it to see how the fill of the boxes has changed to the colours we specified. Then edit the code to change the colour rather than the fill to be these new colours (i.e., change ‘fill’ to ‘colour’ in the first line and the labs() function and change scale\_fill\_manual to be scale\_colour\_manual). Run the code to see the effect.

Exercise Run Code Start Over Solution



1

2

3

4

5

6

7

speed\_box <- ggplot2::ggplot(jig\_tib, aes(employee, footspeed, fill = sex))

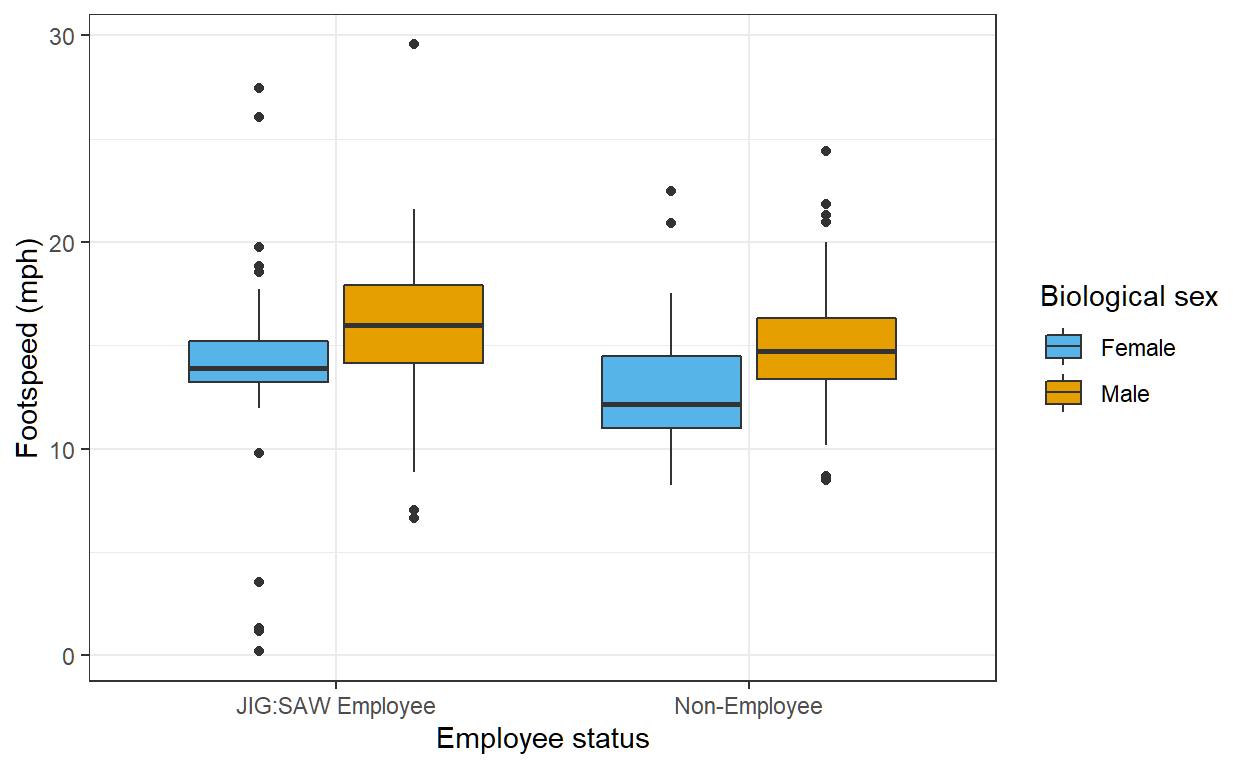
speed\_box +

geom\_boxplot() +

labs(x = "Employee status", y = "Footspeed (mph)", fill = "Biological sex") +

scale\_fill\_manual(values = c("#56B4E9", "#E69F00")) +

theme\_bw()



### Plotting means

Plotting means is slightly more tricky. If you want to plot from the raw data (rather than a tibble containing the summary information) then your best bet is to use the stat\_summary() function and then specify the geom to use within it. Let’s begin by plotting the mean **strength** split by **employee**. We can do this with the following command:

strength\_plot <- ggplot2::ggplot(jig\_tib, aes(employee, strength))

strength\_plot +

stat\_summary(fun = "mean", geom = "point", size = 4)

All we’re doing here is setting up the plot as normal and then adding stat\_summary() to it and within that function asking it to calculate the means and their confidence intervals (fun = "mean") and displaying them using geom\_point (geom = "point"). size = 4 simply makes the points that display the means bigger than the default (you can omit this option if you like). Copy the code into the code box and run it to see the resulting plot. Bar charts are generally a massive waste of ink, but try replacing geom = "point" with geom = "bar" and re-running the code. Congratulations, you have just displayed the same information as the graph with points but using a tonne more ink!

Exercise Run Code Start Over Solution



1

2

3

4

5

6

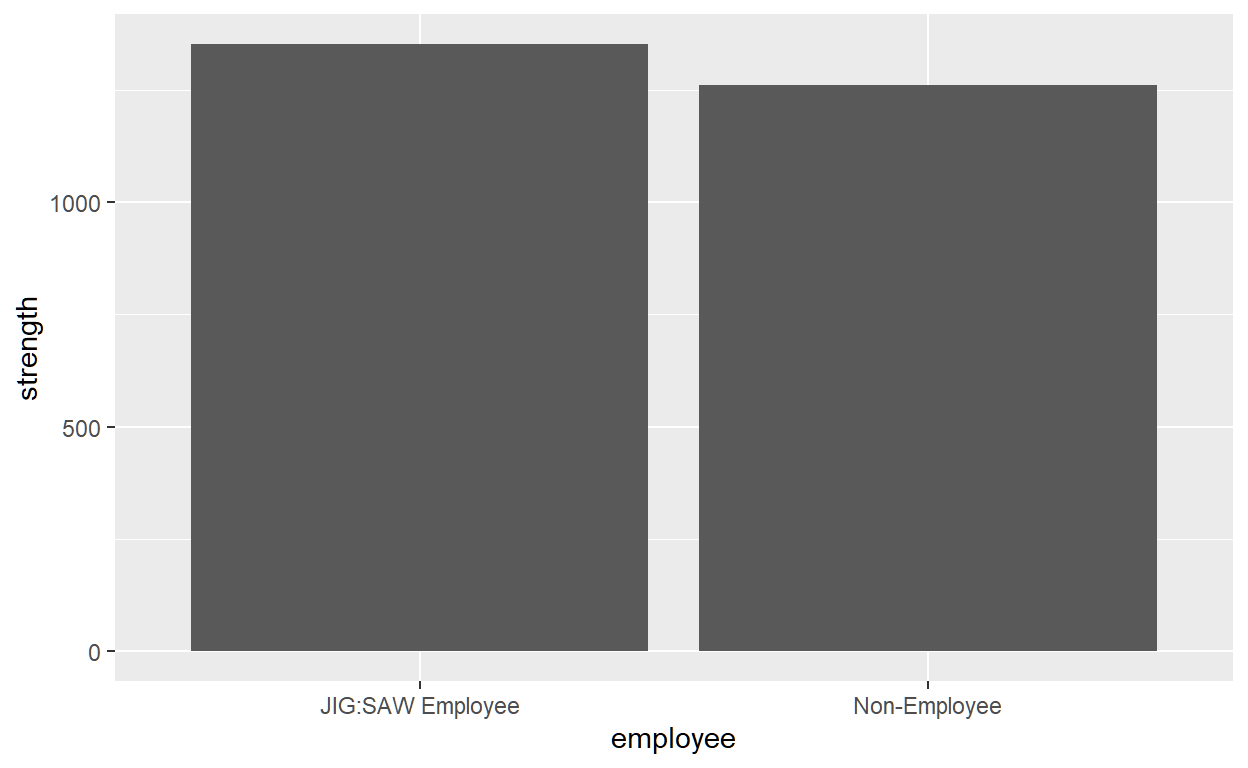
7

strength\_plot <- ggplot2::ggplot(jig\_tib, aes(employee, strength))

strength\_plot +

stat\_summary(fun = "mean", geom = "point", size = 4)+

labs(x = "Employee status", y = "Maximal push force (n)")+



The result can be tidied up by adjusting the scales, adding axis labels, and applying a nicer theme. Try adding the following commands into the code box and seeing what effect it has on the graph when you run the code (don’t forget that every line should end with a + except for the last line):

* labs(x = "Employee status", y = "Maximal push force (n)") to label the axes
* coord\_cartesian(ylim = c(1000, 1800)) to adjust the y-axis to display values from 1000 to 1800
* scale\_y\_continuous(breaks = seq(1000, 1800, 100)) to set the breaks on the y-axis. I’ve used the function seq() which takes the form seq(from, to, by) where from is the value you want to start at, to is the value you want to stop at, and by is the size of the increment you want. In this case, I have specified seq(1000, 1800, 100) which will create a sequence between 1000 and 1800 in intervals of 100.
* theme\_bw() to apply a black and white theme

### Facet\_wrap()

The fun begins if you want to divide the data by a second grouping variable (for example **sex**). One easy way is to use facet\_wrap() to create separate plots for different groups. This function takes the general form:

facet\_wrap(facet, nrow = NULL, ncol = NULL, scales = "fixed")

There are other options, but I have listed the four main ones, which are:

* facet specifies how you want to create the facet. To create separate plots for males and females our facet would be ~sex.
* nrow specifies how many rows of plots to display. There is no default, the function just tries to make sensible choices. If we wanted the male and female plots side by side we want them arranged in 1 row, so we could be explicit and include the command nrow = 1.
* ncol specifies how many columns of plots to display. There is no default, the function tries to make sensible choices. If we wanted the male and female plots on top of each other we want them arranged in 1 column, so we could be explicit and include the command ncol = 1. In reality nrow and ncol become important when you have lots of plots. For example if you were plotting data from 12 different hospitals, you might want these arranged in 2 rows and 6 columns, 4 rows and 3 columns, 6 rows and two columns and so on.
* scales by default the scales of the plots are set to be the same (“fixed”) but sometimes it’s handy to let them vary across different plots, in which case set scales = "free" or use “free\_x” or “free\_y” to allow only the x-axis or y-axis to vary across plots.

The code box below displays the code that you used above to generate the previous plot. Add facet\_wrap(~sex) to the command and execute the code to see what happens.

Exercise Run Code Start Over Solution



1

2

3

4

5

6

7

strength\_plot <- ggplot2::ggplot(jig\_tib, aes(employee, strength))

strength\_plot +

stat\_summary(fun = "mean", geom = "point", size = 4) +

labs(x = "Employee status", y = "Maximal push force (n)") +

coord\_cartesian(ylim = c(1000, 1800)) +

scale\_y\_continuous(breaks = seq(1000, 1800, 100)) +

theme\_bw()

### The more complicated way

Another approach is to plot **sex** not in different panels, but in different colours or fills. Doing so is a bit more involved. Lets start by varying males and females by colour. To set this up, we can start by adding colour = sex to the aesthetic where we set up the plot:

strength\_plot <- ggplot2::ggplot(jig\_tib, aes(employee, strength, colour = sex))

Compare this with the first line of code in the box below (which replicates the plot split solely by **employee**). Add colour = sex into the first line and run the code. What happens?

You should find that the male and female means and CIs appear in different colours, but they appear at the same horizontal location (that is, the male points are directly above the female ones). For these data that doesn’t particularly matter because the error bars don’t overlap. However, if they did overlap it would be difficult to interpret the plot.

Exercise Run Code Start Over



1

2

3

4

5

6

7

strength\_plot <- ggplot2::ggplot(jig\_tib, aes(employee, strength))

strength\_plot +

stat\_summary(fun = "mean", geom = "point", size = 4) +

labs(x = "Employee status", y = "Maximal push force (n)") +

coord\_cartesian(ylim = c(1000, 1800)) +

scale\_y\_continuous(breaks = seq(1000, 1800, 100)) +

theme\_bw()

We can avoid this potential problem by adjusting the horizontal position using position\_dodge(). This command plots geoms so that they ‘dodge’ each other on the horizontal plane. You have to include a value for the ‘dodge’, 0.9 works well for this plot, for others play around with values until it looks good. In the code box above add position = position\_dodge(width = 0.9) into the stat\_summary(). It should look like this:

stat\_summary(fun = "mean", geom = "point", size = 4, position = position\_dodge(width = 0.9))

Run the code and you should see that the male and female points are set at different locations along the horizontal axis.

### Adding raw data to a plot

This section is only really for those feeling confident! If you want to get really flash, you could add your raw data to the plot too. We want to plot it behind the geom that is displaying the mean, and we can do this by placing the function before the stat\_summary() function. Let’s just dive in. The function we will use is geom\_point() and we can set it up as follows:

geom\_point(position = position\_dodge(width = 0.9), alpha = 0.3) +

This function will add points (dots) to the plot showing the raw data. The commands inside the function are doing this:

* position = position\_dodge(width = 0.9) just like for our stat\_summary() we need the raw data for men and women to dodge each other. If you set the dodge identical to the value in stat\_summary() as we have done here then the raw data will appear in the same location as the geom displaying the mean. You can set a slightly wider dodge (e.g., width = 1) so that the data and the means are side by side. Try changing the value to see how the data points move horizontally.
* alpha = 0.3 this makes the data points semi-transparent, which is handy because it means you can see where there are lots of similar scores.

Add this function to the command in the box below (place it before the stat-summary() function). Finally, by adding the raw data we need to extend the y-axis so that we can see all of the scores. Previously we have set the limits (and breaks) to vary from 1000 to 1800, edit these values to show a range between 900 and 2500. When you have made all of those edits to the code, run it to see the plot. Try also adding a scale\_colour\_manual() function to specify the same colour-blind friendly colours that we used before.

Exercise Run Code Start Over Solution



1

2

3

4

5

6

7

strength\_plot <- ggplot2::ggplot(jig\_tib, aes(employee, strength, colour = sex))

strength\_plot +

stat\_summary(fun = "mean", geom = "point", size = 4, position = position\_dodge(width = 0.9)) +

labs(x = "Employee status", y = "Maximal push force (n)", colour = "Biological sex") +

coord\_cartesian(ylim = c(1000, 1800)) +

scale\_y\_continuous(breaks = seq(1000, 1800, 100)) +

theme\_bw()

Because we changed the size of the points displaying the mean to make them bigger, this looks OK (in that the raw scores do not obscure the mean). Howeber, we could also change the shape of the geom to distinguish the means from the raw scores. The shape = option allows us to specify a number that indicate a shape for the geom\_point() function. Figure 1 shows the numebrs representing particula shapes. If you ever forget these mappings then execute ?points; the resulting help file lists the numbers and shapes.

Figure 1: Mapping of shapes to numeric values

Figure 1: Mapping of shapes to numeric values

The code box includes the command that we have just used. First, add shape = 0 to the stat\_summary() function. In other words change:

stat\_summary(fun = "mean", geom = "point", size = 4, position = position\_dodge(width = 0.9))

to be:

stat\_summary(fun = "mean", geom = "point", size = 4, shape = 0, position = position\_dodge(width = 0.9))

Run the code, and note that the means are now displayed as hollow squares. Feel free to try other shapes!

Now change the stat\_summary() function back to the default by deleting the shape option. Now try changing the shapes that display the raw scores. Let’s change them to crosses by adding shape = 4 to geom\_point(). (We can delete alpha = 0.3 because shape 4 is not filled so setting the transparency is redundant.) So we want to change:

geom\_point(position = position\_dodge(width = 0.9), alpha = 0.3)

to be

geom\_point(position = position\_dodge(width = 0.9), shape = 4)

Run the code and you should see that the raw data points are crosses.

Exercise Run Code Start Over



1

2

3

4

5

6

7

8

9

strength\_plot <- ggplot2::ggplot(jig\_tib, aes(employee, strength, colour = sex))

strength\_plot +

geom\_point(position = position\_dodge(width = 0.9), alpha = 0.3) +

stat\_summary(fun = "mean", geom = "point", size = 4, position = position\_dodge(width = 0.9)) +

labs(x = "Employee status", y = "Maximal push force (n)", colour = "Biological sex") +

coord\_cartesian(ylim = c(900, 2500)) +

scale\_y\_continuous(breaks = seq(900, 2500, 100)) +

scale\_colour\_manual(values = c("#56B4E9", "#E69F00")) +

theme\_bw()

Previous TopicNext Topic

1. Basically you can use this tutorial for teaching and non-profit activities but do not meddle with it or claim it as your own work.[↩](http://127.0.0.1:5664/#section-fnref1)