Week 4 workbook: Data exploration and uncertainty

Site:	Moodle VLE	Printed by:	Marta Shocket
Course:	24/25: LEC.243: Experimental Design and Analysis [1]	Date:	Monday, 14 October 2024, 22:03
Book:	Week 4 workbook: Data exploration and uncertainty		

Table of contents

5. Descriptive statistics with R

Last week we learned the basics of how to use RStudio, including:

- The basic parts that make up code
- How to run code from a script file
- How to load data from a .csv file
- How to perform basic calculations

This week we're going to build on that foundation and learn some new skills, including:

- How to install and load packages
- How to use a couple different packages for data exploration
- How to generate histograms and box-and-whisker plots to visualise distributions of our data
- How to summarise and plot data using tidyverse methods

Specifically, we're going to show you how to create the column graph of carbon store estimations that you made last week in Excel. You can use either version of the figure in your Data Presentation coursework.

To help yourself learn how code works, before you run it, stop to ask yourself:

- What functions are being used?
- What arguments are going into the function?
- Are any new objects being created?

5.1. Installing packages

A package is a set of extra functions that are not part of the basic R installation but can be added on later. You can think of them like apps that enhance R.

Today we will be using two packages:

1. `psych` - a package for exploring data and generating a specific set of descriptive statistics
2. `tidyverse` - a collection of related packages for 'wrangling' (i.e., manipulating) data and plotting figures. Within the tidyverse, the main packages we will be using are `dplyr` (data manipulation) and `ggplot2` (plotting).

For tidyverse packages, you can load the specific packages that you want individually, or all together as the `tidyverse` package. We'll do the latter since it will require fewer lines of code.

> You only need to *install* a package once on a given computer (or in this case, in the Apporto virtual computer). However, you need to *load* it every new R session.

Directions to install packages

1. Click on the Packages tab in the bottom right quadrant, and then click on the Install button at the top left of the panel.

OR

1. Click on 'Tools' in the top menu, and then click on the first option 'Install Packages...'

Both options will bring up the same window.

2. Type the name of the first package that we want (`psych`) into the 'Packages' blank. A menu with autocomplete suggestions will populate as you type, and you can either click or press the arrow keys and hit 'Enter' to select the one you want.

3. You can select multiple packages at a time, separated by commas. Add the `tidyverse` package.

4. Hit 'Install'. In general, this takes anywhere from a few seconds to a few minutes, depending on the sizes of the package(s) you select.

5. As the packages install, you will see text printing in the Console window. There will often be some warning messages about certain functions being masked. Don't worry about that, you can ignore them.

5.2. Setting up our workspace

Creating a new R script

Create a new R script and save it in your `LEC243/scripts` folder as "Week4Practical". Add a header like the one we created at end of Practical 3.

```
##### Marta Shocket
##### LEC 243 - Week 4 Practical
##### 28 Oct 2024
```

Loading the packages

Type the following code into your R script and run it to load the packages that you just installed. Luckily the autocomplete feature works for packages too!

```
# Load libraries
library(psych)
library(tidyverse)
```

Uploading the data to Apporto

Upload the new file `LEC243_2024_results.csv` that you just made to your Apporto desktop, and then move it into your `LEC243/data` folder.

Loading the data

Last week you loaded the area data set using the following code:

```
# Load data
data_area <- read.csv("LEC-243/data/LEC243_2024_area.csv")
```

Type this code into your new script, because we are still using the original area data.

Now copy and paste it once below the original line.

Modify this copied code so that it will load the new results data.

Some hints for how to do that if you are having trouble:

- You will need to change the names of the object being created and stored (I suggest `data_results` to match the rest of the example code in this workbook).
- You will also need to change the file name.

Run the code to load both data sets.

Make sure you can see them in your Environment tab before moving on.

5.3. Numerical data exploration

We'll start by doing some numerical data exploration on the original area data with two different functions.

summary()

Base R has a function called `summary()` that you can use on a lot of different types of objects.

If you use it on a data frame, it will print the following values for each numeric variable:

- Minimum
- 1st quartile (25% percentile value)
- Median (50% percentile value)
- Mean
- 3rd quartile (75% percentile value)
- Maximum
- Number of NAs (if applicable)

Type the following code into your R script and run it.

```
# Basic descriptive statistics
summary(data_area)
```

> **Q:** What information does it provide for non-numeric variables like surname and forename?

describe()

The `psych` package has a function called `describe()` that provides a larger and slightly different set of statistics. Most of these should already be familiar to you:

- count of observations (n)
- mean
- standard deviation (sd)
- median
- minimum (min)
- maximum (max)
- range
- standard error (se)

However, some may be less familiar (see below).

Type the code below into your R script and run it.

Here, we are providing the required data frame and two optional keyword arguments to change the settings. Optional arguments can be written in any order (after the required arguments) and use the keyword followed by `=` to tell the function which specific setting the input belongs to.

(We could also write out "`x = data_area`" if we wanted, but instead we are taking advantage of R's defaults for handling required arguments.)

- The `omit` argument tells it to ignore non-numeric data - otherwise it will convert them to numbers and provide nonsense output. (You can try it and give it a look if you want.)
- The `IQR` argument tells it to provide the interquartile range, which otherwise it would exclude.

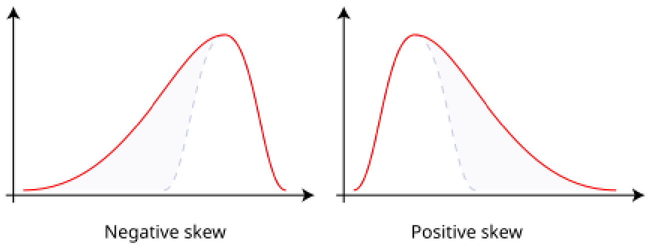
```
describe(data_area, omit = TRUE, IQR = TRUE)
```

The **interquartile range** is calculated by subtracting the 1st quartile value from the 3rd quartile value - this number tells you the spread of the middle 50% of your data.

The output from `describe()` includes two very important statistical values, which are both used when we are evaluating the normality of a distribution:

- **Skew** - a measure of how asymmetrical the data distribution is, and in which direction (see picture from Wikipedia below).
- **Kurtosis** - a measure of "tailedness" or how much of a distribution is found in its tails.

Negative skewness values imply that the data are negatively skewed (tail to the left), and positive skewness indicates positive skew (tail to the right). The further away from zero the skewness is, the more skewed the distribution.



Positive kurtosis means fatter tails, while negative kurtosis means thinner tails.

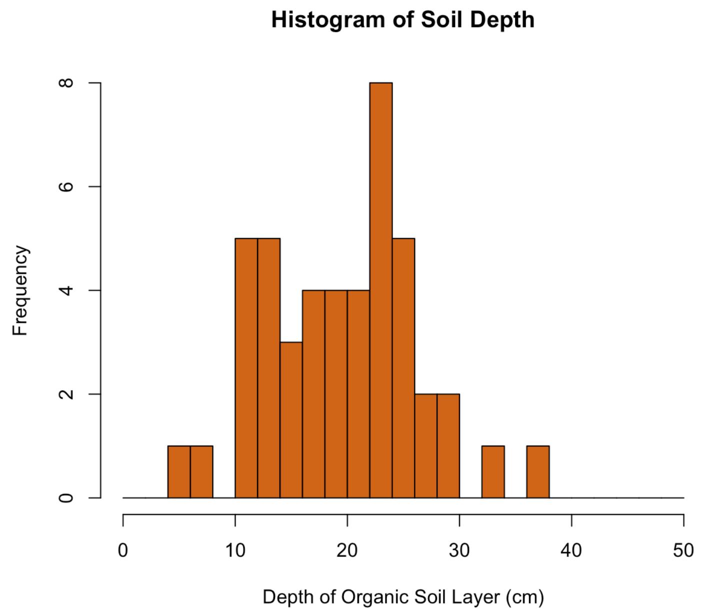
The `describe()` output also includes some unusual values that are not commonly used in environmental science or ecology, so we'll ignore them. I'm including definitions here only for the curious / completionist among us, but feel free to ignore and/or immediately forget:

- **Trimmed mean** - the mean calculated with the top 10% and bottom 10% of values removed.
- **MAD (median absolute deviation)** - another measure of variability. It's calculated by taking the absolute value of the difference between every observation and the median, and then taking the median of that list of numbers.

5.4. Histograms

Histograms are the most basic way to plot a data distribution to see what it looks like.

A histogram has the observation value of a variable on the x-axis (e.g. the sample soil depth) and the number of observations on the y-axis. Observation values for the variable are usually grouped into bins (e.g. all soil depths between 10 cm and 12 cm).



The function to plot a histogram in base R is `hist()`. Add the following code into your R script and run it.

```
# Histograms in base R
hist(data_area$WoodlandArea_m2)
```

> **Q:** Where does the plot appear? Does the data distribution look like what you expected? Why or why not?

> **Q:** What else do you notice about the plot? What might you want to change if you were sharing it with someone else, or putting it in a report?

Now let's add historgrams of the other two areas. Add the following code to your script and run it.

- Copy the `hist()` line and paste it below the original twice.
- On the new lines, double click the variable name after the `$` and it will highlight it.
- Then you can easily delete it and replace it with the new variable name.
- Add another line of code at the top (the `par` command shown below) to tell Base R to plot multiple panels in a single figure

```
# Histograms in base R
par(mfrow = c(1,3) # This tells R to plot multiple panels in 1 row, 3 columns
hist(data_area$WoodlandArea_m2)
hist(data_area$GrasslandArea_m2)
hist(data_area$TotalCampusArea_m2)
```

You can change the size of your bottom-right quadrant (where the plots tab is) to change the size of the plot. The plot will re-render each time you drag the panel borders.

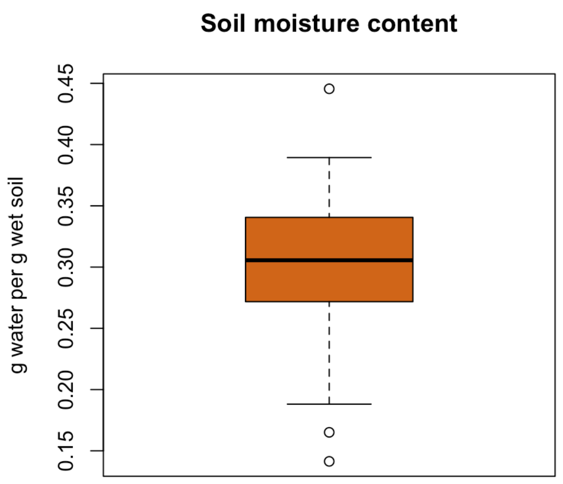
Q: What do you notice about the three distributions? How are they similar or different? Do they look like stereotypical normal distributions? Why or why not?

5.5. Box and whiskers plots

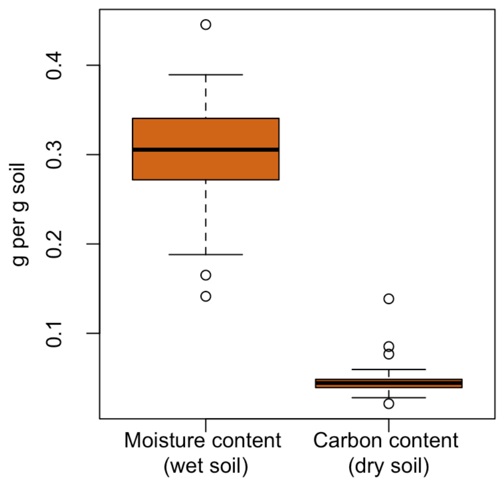
Box and whiskers plots, sometimes shortened to boxplots, are another common way to visualise data during the initial exploration phase.

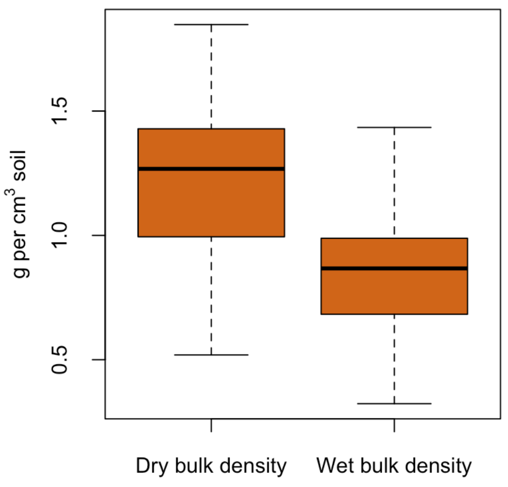
Boxplots show:

- A box encompassing the **interquartile range (IQR)**, the values between the 1st and 3rd quartiles, i.e. the 25th and 75th percentiles, q_{25} and q_{75} respectively.
- A line across the centre of the box at the median
- "Whiskers" on each end representing the 5th and 95th percentiles.
- **Outliers** (extreme values) plotted as individual points. Specifically, outliers are identified as any points outside the whiskers with values:
    - $< q_{25} - 1.5 \times \mathrm{IQR}$ or
    - $> q_{75} + 1.5 \times \mathrm{IQR}$



Unlike for histograms, you can plot make boxplots with more than one variable at a time.





Q: Which of the two figures above is more useful? What is the problem with one of them?

The function to plot a histogram in base R is `boxplot()`. Add the following code into your R script and run it.

You will want to reset the graphical parameters to the default settings - one row and one column.

- Copy the same text you used for plotting the histograms (the one with the `par()` function) on the line where it says "*Insert code...*" below.
- Modify it to set it to one row and one column.

```
# Boxplot
*Insert code to reset the graphical parameters here*
boxplot(data_area)
```

Q: What happened? Can you figure out what the message in the console means? Which variable(s) in `data_area` do you think are causing the problem?

Remember, you can view the data frame by clicking on it in the Environment tab, or running the commands `View(data_area)` or `head(data_area)` from the console or your script.

"Non-numeric argument" means that we tried to give `boxplot()` non-numerical data to plot, and it really didn't like that and just gave up on the whole task.

So now we have two options: give it the data one variable a time (like we did with `hist()`), or trim our data to exclude the forename and surname columns.

Let's trim the data to exclude the name columns. And given what we know about the values of our different variables - that total area is much larger than the woodland and grassland areas - let's trim out the total area as well.

Indexing in R

Most coding languages have an **indexing** system that you can use to refer to specific items in a data object. We can use this to select specific rows and/or columns within our data frames.

In R, indexing is done with with square brackets `[]`. Within the brackets:

- Rows are specified first, then columns - separated by a comma
- If you leave the row or column setting blank, this means "all of them"
- Remember from last week that:
  - you can use a colon operator to list out all integer numbers between A and B (written as `A:B`)
  - you can also use the letter `c` to make a list of numbers separated by commas.

In R, the first item, row, or column in a data object is denoted by with the number 1, the second by the number 2, etc.

This may seem obvious, but actually many languages are "zero-indexed" (instead of "one-indexed" like R), so the first item is denoted by the number zero.

Edit your code so it looks exactly like this and run it. It tells R to:

- make a boxplot,
- using the area data frame,
- with data from all rows and the columns 4 through 5

```
boxplot(data_area[, 4:5])
boxplot(data_area[, c(4,5)])
```

> **Q:** What do you notice about outliers in our area data?

5.6. Summarising data with dplyr

Now we're going to learn how to summarise data (i.e. calculate means, standard deviations, etc. from individual observations) using tidyverse methods.

Tidyverse code frequently uses the **pipe** operator: `|>`. This operator passes data to the next function in the code to use as its first argument.

> NOTE: Until 2023 the pipe operator was `%>%`, so you will probably see that older notation if you look up code examples online. Both pipes will do the same thing.

In the tidyverse, you summarise data with the `summarise()` (or `summarize()`) function.

Type the code below into your R script and run it. Here is a place where the autofill will really increase your efficiency!

An explanation of the code structure as you go down the lines:

- *Line 1*: Store the new object (`area_data_summary`) that will be created by the rest of the code. That process starts with our `data_area` data frame that gets passed down (via the pipe) to the next function.
- *Line 2*: Use the `summarise()` function (from the `dplyr` package) to create a new data frame with summary statistics calculated from the old data frame that we passed down.
- *Lines 2-8*: Specifically, we want the means and standard deviations for our three different types of area estimations. Each summary statistic that we want is separated by a comma. This part would work as one really long line of code, but it's much easier to read formatted like this.

Speaking of formatting, notice that RStudio will automatically indent some of your lines to give it structure that makes it easier to read and interpret.

```
area_data_summary <- data_area |>
    summarise(MeanTotalArea = mean(TotalCampusArea_m2),
              MeanWoodlandArea = mean(WoodlandArea_m2),
              MeanGrasslandArea = mean(GrasslandArea_m2),
              SDTotalArea = sd(TotalCampusArea_m2),
              SDWoodlandArea = sd(WoodlandArea_m2),
              SDGrasslandArea = sd(GrasslandArea_m2))
```

> **Q:** What does the output - `area_data_summary` - look like?

Whoops! All the functions we've been using so far today - `summary()`, `describe()`, `hist()`, and `boxplot()` - have been excluding the NAs in our data, so we didn't have to remove them.

But now it looks like `summarise()` needs us to do that.

Open your R script from last week's practical and find the code you used to remove the NAs. Re-type the comment + command into this week's script.

- You could put it near the top, right after we load the data.
- Or you could put it just above where we are currently coding with `summarise()`, so the `summary()` function we used earlier will still report the number of NAs.

Re-run the code now that you've removed the NAs.

> **Q:** Now what does the output - `area_data_summary` - look like?

I hope that you can start to see how, once you know what you're doing, coding in R can make summarising data much faster and easier than doing it in Excel.

Additionally, unlike for Excel, this code can also be easily used on a similar data set from a different location or year, even if the number of observations or the order of columns is different (as long as the variable names are the same).

5.7. Plotting with ggplot

The tidyverse function for basic plotting is called `ggplot()`.

Today we'll learn the basic methods for how to use it by making the same column plot that you made last week in Excel (the one that you'll need for coursework assignment 1).

We'll start with a very simple version of the figure, and slowly build up complexity to get something of the quality that you could submit for your coursework.

This approach (start with a very simple plot and gradually make it prettier and more complete) is actually the same process that I use to make figures for my papers and presentations.

In general, `ggplot()` works by:

- First creating a plotting "aesthetic" (`aes()`) that specifies which variables you want to plot.
- Then it layers the actual points, lines, labels, etc. on top of the aesthetic. Each layer is separated by a \`+\` sign that tells `ggplot()` to keep going.

Type the code below into your R script and run it.

An explanation of the code structure as you go down the lines:

- *Line 1*: Store the object (`figure_cw1`) created by `ggplot()` using our data frame (`data_results`). We want `ggplot()` to use an aesthetic where our x-axis is the carbon store type (soil_grasslands, etc.) and the y-axis is the mean value of the store.
- *Line 2*: Add a column plot layer with `geom_col()`.
- *Line 3*: Print the plot we just made.

> The `ggplot()` function can automatically generate a figure in the Plots panels, just like `hist()` and `boxplot()`. However, the standard practice for tidyverse code is to store it as an object and then print it.

```
figure_cw1 <- ggplot(data = data_results, aes(x = type, y = mean)) +
  geom_col()
print(figure_cw1)
```

Hmmm. It definitely made a column plot, but the columns aren't in the right order. It's sorting them alphabetically, when we want them in numerical order!

We can change that by using the function `reorder()` to modify the x-axis variable inside the aesthetic. You need to give it two arguments:

1. the name of the variable being sorted (`type`)
2. the name of the variable you want it sorted by (`mean`), with either a `+` for ascending order or a `-` for descending order.

Now that that line of code is longer, I'm also going to put in a line break so you don't have the scroll side-to-side to read it in this workbook, but you can leave it on one line in your (likely much wider) R script.

For each of the code chunks below, edit your previous plotting code to match and then re-run it to see how the figure changed.

```
figure_cw1 <- ggplot(data = data_results,
                     aes(x = reorder(type, +mean), y = mean)) +
    geom_col()
print(figure_cw1)
```

Much better!

The next step is to add an error bar layer showing our standard error calculation.

Since the error bars use different data (arithmatic combinations of the `se` and `mean` variables instead of just the `mean` variable), we need to create a new aesthetic just for this layer.

The function `geom_errorbar()` has a different structure for its aesthetic: instead of taking the arguments `x` and `y`, it takes `ymin` and `ymax`. These describe the bottom and top values of the error bars, and can be described by adding and subtracting the standard error around our mean values.

```
figure_cw1 <- ggplot(data = data_results,
                     aes(x = reorder(type, +mean), y = mean)) +
    geom_col() +
    geom_errorbar(aes(ymin = mean - se, ymax = mean + se))
print(figure_cw1)
```

We successfully added error bars, but they look ugly. Let's add an argument - `width` - to change the width to something more reasonable.

```
figure_cw1 <- ggplot(data = data_results,
                     aes(x = reorder(type, +mean), y = mean)) +
    geom_col() +
    geom_errorbar(aes(ymin = mean - se, ymax = mean + se, width = 0.2))
print(figure_cw1)
```

Now we'll make a couple changes to make our figure prettier.

First, add a theme layer - `theme_classic()` - to get rid of the grey background.

Second, change the colour of the columns by adding arguments to `geom_col()`:

- `colour` (or `color`) changes the outline around the bars
- `fill` changes the actual bars

There are several ways to specify a colour in R:

- R has a library of pre-set colours that be specified using text inside quotation marks (e.g. `"limegreen"`).
- You can specify any colour using hexidecimal colour codes inside quotation marks (e.g. `"#009982"`).
- There are packages of themed colour palettes, like the `wesanderson` package based on scenes from Wes Anderson movies.

You can google "R color chart" or "hexadecimal color code chart" to help you find the colour that you want for this figure.

```
figure_cw1 <- ggplot(data = data_results,
                     aes(x = reorder(type, +mean), y = mean)) +
    geom_col(colour = "limegreen", fill = "limegreen") +
    geom_errorbar(aes(ymin = mean - se, ymax = mean + se, width = 0.2)) +
    theme_classic()
print(figure_cw1)
```

Looking much better, but none of our labels make sense for a human- they're just the commands we gave R to tell it what to plot.

First, we'll add a separate line of code (above the `ggplot()` command) to replace the values of the `type` variable with what we want printed. This code:

- Creates a vector with our new labels and uses it to overwrite the old labels in the data frame.
- Needs quotation marks around each label to keep the spaces from breaking our code.

> Because `read.csv()` replaces any spaces with periods (to prevent it from breaking the code), we would have to replace the labels even if we written them out exactly how we wanted them to appear in the .csv file.

Second, we'll use `labs()` to add another layer with labels for the axes and a title. We can leave the x-axis label blank since each column is already labled with its `type` variable value.

Obviously you'll need to come up with your own labels that actually describe the data instead of the placeholders I put in the example code.

```
# Replace the names with better formatting
data_results$type <- c("Grassland soil", "Woodland soil", "Tree biomass", "Total")

figure_cw1 <- ggplot(data = data_results,
                     aes(x = reorder(type, +mean), y = mean)) +
    geom_col(colour = "limegreen", fill = "limegreen") +
    geom_errorbar(aes(ymin = mean - se, ymax = mean + se, width = 0.2)) +
    theme_classic() +
    labs(title = "A graph title", x = "", y = "y-axis label (units)")
print(figure_cw1)
```

Almost there!

A few final touches:

- Add a `size` argument to the `geom_errorbar()` to increase the line thickness.
- Add a new theme layer with three `element_text()` functions and `size` arguments to change the sizes of the plot title, the axis title, and the axis text.
- Use the `hjust` argument (i.e. horizontal justification) to center justify the title.

Remember that you can change the aspect ratio of your figure by changing the size of the Plots panel.

```
figure_cw1 <- ggplot(data = data_results,
                     aes(x = reorder(type, +mean), y = mean)) +
    geom_col(colour = "limegreen", fill = "limegreen") +
    geom_errorbar(aes(ymin = mean - se, ymax = mean + se, width = 0.2), size = 0.95) +
    theme_classic() +
    labs(title = "A graph title", x = "", y = "y-axis label (units)")
    theme(plot.title = element_text(hjust = 0.5, size = 20),
          axis.title = element_text(size = 14),
          axis.text = element_text(size = 11))
print(figure_cw1)
```

Yay! A high-quality figure that's suitable for publication. Now we just need to save it.

Steps for saving a figure:

- In the Plots panel, click 'Export'.
- Select 'Save As Image'
- Change the file name
- Change the save location if you want to - otherwise it will save to your working directory, in this case, your `U:/`. I would recommend creating a 'figures' folder inside your LEC243 folder.
- Change the aspect ratio if you want to - if you click "Update Preview" you can see what it would look like.
- Click 'Save'.

If there's an adjustment that you want to make to your figure that isn't shown here, you can let me know by posting about it on the Moodle forum.

5.8. Tidying up

The final tasks for today are to:

1. Tidy up your R script.
2. Update your dictionary of R functions (the markdown file you should have saved in your LEC243 folder).

Remember you'll want to:

- Add a header describing what the script is, who wrote it, and when they they wrote it.
- Add comments so that someone else (or you in a few weeks or months) can understand what you did.
- Delete spare lines of code that you don't need anymore.

I put a copy of my final script file at the end of this chapter as an example.

This week we used a lot of new functions, particularly for use inside `ggplot()`. When you update your glossary, I recommend creating a separate section for these functions to make it easier to use as a reference.

The new functions from week 4 to add to your glossary are:

General functions:

- `boxplot()`
- `describe()` (`psych` package)
- `hist()`
- `library()`
- `par()`
- `summary()`
- `summarise()` (`dplyr`/`tidyverse` package)

Functions for use inside `ggplot()` (`ggplot2`/`tidyverse` package):

- `aes()`
- `element_text()`
- `ggtitle()`
- `geom_col()`
- `geom_errorbar()`
- `labs()`
- `theme()`
- `theme_classic()`

You already saved and closed your Excel file (after making the .csv file). Now ensure that you have saved your three R-related files before closing RStudio:

- Your R script file
- Your R function glossary markdown file
- Your image file of the figure

```
##### Marta Shocket
##### LEC 243 - Week 4 Practical
##### 28 Oct 2024

############# 1. Set up

# Load package libraries
library(psych)
library(tidyverse)

# Load data
data_area <- read.csv("LEC243/data/LEC243_2023_area.csv")
data_results <- read.csv("LEC243/data/LEC243_2023_results.csv")


############# 2. Summary statistics

# Basic descriptive statistics
summary(data_area)
describe(data_area, omit = TRUE, IQR = TRUE)


############# 3. Histograms and box plots

# Histograms in base R
par(mfrow = c(1,3)) # This tells R to plot multiple panels in 1 row, 3 columns
hist(data_area$WoodlandArea_m2)
hist(data_area$GrasslandArea_m2)
hist(data_area$TotalCampusArea_m2)

# Boxplots in base R
par(mfrow = c(1,1)) # reset to 1 row, 1 column
boxplot(data_area[, 4:5])
boxplot(data_area[, c(4,5)])


############# 4. Summarising data with dplyr package

# Remove NAs from area data
data_area <- na.omit(data_area)

# Calculate means and standard deviations for all three area variables
area_data_summary <- data_area |>
    summarise(MeanTotalArea = mean(TotalCampusArea_m2),
              MeanWoodlandArea = mean(WoodlandArea_m2),
              MeanGrasslandArea = mean(GrasslandArea_m2),
              SDTotalArea = sd(TotalCampusArea_m2),
              SDWoodlandArea = sd(WoodlandArea_m2),
              SDGrasslandArea = sd(GrasslandArea_m2))


############# 5. Column graph with ggplot2 package

# Create and print column plot
figure_cw1 <- ggplot(data = data_results,
                     aes(x = reorder(type, +mean), y = mean)) +
    geom_col(colour = "limegreen", fill = "limegreen") +
    geom_errorbar(aes(ymin = mean - se, ymax = mean + se, width = 0.2),
                  size = 0.95) +
    theme_classic() +
    labs(title = "A graph title", x = "", y = "y-axis label (units)")
    theme(plot.title = element_text(hjust = 0.5, size = 20),
          axis.title = element_text(size = 14),
          axis.text = element_text(size = 11))
print(figure_cw1)
```