hamburger.r: Explanation of code

Goals of code:

- Installing add-on libraries (also called packages)
- Read an excel sheet (.xls or .xlsx file)
- Wilcoxon rank sum test
- Diagnostics
  - Using base plot graphics
  - Using ggplot graphics
- Transformations

Reading an excel sheet:
When you have a .xls or .xlsx file, there are two ways you could read this into R. One is to store a specific spreadsheet as a comma delimited file (.csv format). The other is to read the Excel workbook (.xlsx format file). The ability to read an Excel workbook is not part of base R. It is added to R by functions in an add-on package (also called library). There are many (nearly 10,000) add-on packages that do all sorts of different analyses. There are multiple packages that read .xls or .xlsx files. I demonstrate the `read_excel` function in the readxl package. This is the one I find most useful. You are welcome to use a different one.

The R archive site, called CRAN: comprehensive R archive network, is so active, it is distributed over 10's of mirror sites. I suggest you use the mirror site closest to you. For us, that is the one in IA.

Installing a new package: `install.packages('readxl')`
Before you can use any add-on library, you need to download it from CRAN, the Comprehensive R Archive Network. Since there are now so many packages, I find it easiest to download it by name. `install.packages()` installs the package you name. You will have to specify a mirror site (I recommend IA). The package is now downloaded and unzipped.

Note: the package name is in quotes.

If you want to look for a package in a menu, from the main menu: Packages / Install Package. Choose a mirror site (I recommend the IA one), then choose the readxl library.

Downloading the library (either by name or by menu) only needs to be done once, until you update your version of R. When you install a new version of R, you will need to download new versions of libraries.

Activating a library: `library(readxl); read_excel()`
Before you can use a package, you have to enable it. That's what `library(readxl)` does.

Note: Here the filename is not in quotes.

library() has to be done once each session before you can use the read.excel() function. If you get the error message: Error: could not find function "read.excel", you forgot to enable the library. The required argument is the file name, or replace the filename with choose.files() to use the mouse to select the file. The default is to read Sheet1, but you can specify the name of another sheet. `read_excel()` also allows you to designate a specific range of cells to read. Default is to read all rows and all columns in the specified worksheet.

Reading an excel file: `read_excel()`
`read_excel()` will read both xlsx and xls format files. Just specify the appropriate filename.

The result from `read_excel()` is a tibble. This is an extension of the data frame that we use to store data. You can treat the tibble just like a data frame. The biggest difference is that a large tibble is printed more nicely than a large data frame.

Wilcoxon rank sum test: `wilcox.test(cfu~treatment, data=hb)`
The syntax is exactly like t.test: a formula with the response variable (here, cfu) $\sim$ the treatment variable (here, treatment), and the name of the data frame.

A couple of details: I prefer to not use a continuity correction, so I add correct=F. Lecture will briefly discuss this. If the data set is small, I prefer the permutation test p-value, which you get by adding exact=T, as in the second line.

One slightly irritating limitation: the algorithm R uses to get the exact p-value does not work when there are tied values (two observations with the same response value). R will print a warning when it can't compute the exact p value. The exact p-value is well defined, even when there are ties. It's just that R won't calculate it.

Numerical diagnostics: equal variance:
`with(hb, tapply(cfu, treatment, sd))`: You want to compare the sd's for each group. Remember `tapply()` applies a function to subsets of the data. The three arguments are the variable to use, the grouping variable, and the function. `with()` is a shortcut to avoid typing the data frame name for each variable.

Obtaining residuals and predicted values: lm(), predict() and resid()
Various graphical diagnostics require residuals; some also require predicted values. For example, the graphical diagnosis for equal variance is to plot residuals (on the Y axis) against the predicted values (on the X axis). This requires the residuals and the predicted values from a model. `t.test()` does not provide either. Instead, we will use `lm()`. This function fits a linear model to the data. Linear models include t.tests, regressions, and all sorts of analysis of variance. `lm()` will be our workhorse for most of the semester.
`cfu ~ treatment`. Specifies the model you want to use, as a formula. The response variable is on the left-hand side of the $\sim$; the model is on the right-hand side. For a two-sample t-test, there is

one model variable that indicates the group.
`diff ~ 1`. If there is only one sample of observations, e.g., the differences in paired data analysis, the model is just an intercept (i.e, the mean). That is represented by the number 1 as the model formula.
`data=hb`. Specifies the data frame that contains the model variables. Note: it is not necessary to put the variables you want to use in a data frame, but it is highly recommended.
Later, we will print the output or print a more detailed summary of the output from `lm()`. For now, we only want the predicted values and the residuals. So we save the `lm()` results in a new variable. `hb.lm` is a list with many different pieces of information. The help file for `lm()` tells you about each one. I will introduce the most useful ones. Although we can look directly at the contents of `hb.lm`, it is preferable to use "extractor" functions to extract desired pieces.
`predict()`: Extract the predicted values for each observation. The result is a vector with one value for each observation in the initial data set.
`resid()`: Extract the residuals for each observation. The result is a vector with one value for each observation in the initial data set.
Residuals and predicted values can be stored in new variables, or can be computed "on demand" within other functions.

Graphical diagnostics using base graphics:
Plot of residuals against predicted values: `plot(predict(hb.lm), resid(hb.lm))` See above section to fit a model and extract the residuals and predicted values. This code line doesn't save those values. It just passes the output from predict() and resid() directly to the `plot()` function.
Adding a dotted line at Y=0: `abline(h=0, lty=3)` `abline()` adds a line to a pre-existing plot. h=0 is a horizontal line at the specified Y value. The default is a solid line. The type of line can modified by the `lty=` argument. 1 is a solid line, 2 is a dashed line, 3 is a dotted line, and there are a few other combinations.

Modifying plot characteristics: The default plot symbol is a black open circle. All the base R graphics functions can be modified in a huge number of ways. My preferred plot is blue solid circles. You may prefer something else.
Changing the plot symbol `pch=`: The pch argument changes the plot character. You can put a single character in quotes or use a number to indicate the desired symbol. 19 is a solid circle.
Changing the symbol color `col=`: The col argument changes the symbol color. You can specify the color as a number (for 8 major colors) or a color name in quotes. If you want to see the names of the 657 named colors, type `colors()`.
Showing all symbols and 8 numbered colors `plot(1:25, ...); points()`: These two lines draw 25 points, one in each plot symbol, and add 8 points, one in each color. `points()` adds points to a pre-existing plot.
Seeing all possible plot modifications: Type `?par` to see the very long list of all the ways you can change figures and plots in base R graphics.

Diagnostics, normality `qqnorm(); qqline()`: `qqnorm()` draws a normal quantile-quantile (QQ) plot. The Y axis is the sorted values, the X axis is the expected values if the population is normal. We use `resids()` to extract the residuals from the linear model because we want to draw the QQ plot using the residuals. `qqline()` draws a line to aid your interpretation of the plot. That line is

fitted in a way that is resistant to outliers (details not important, but available in the qqline() help file).

Graphical diagnostics using ggplot graphics:
See the code and explanation for creativityV2 in week 1. This describes basic use of ggplot graphics and is essential to understand the code provided this week.

See the Obtaining residuals and predicted values section (found above) to compute the residuals and predicted values. For clarity, the code here saves the predicted values and residuals in vectors, not in a data frame.

Draw a residual vs predicted value plot
`ggplot(data=hb, aes(x = predvalues, y = residuals)) + geom_point()`:
ggplot() really wants a data frame as the first argument, so we give it one, even though it isn't needed (the residuals and predicted values are stored in vectors outside the data frame). The aes() and `geom_point()` are explained in the week 1 material. A very short summary is that the `aes()` component specifies the X and Y variables and `geom_point()` plots a dot at each (x,y) pair.

Extensions:
`+ labs()`: adds labels to the plot. Remember that the + needs to be at the end of the previous line so R "knows" there is more to come. You can specify a title line (`title =`), X axis label (`x =`), and Y axis label (`y =`). The text for each need to be quotes, either double or single.
`+ theme_minimal()`: Changes the overall appearance to a minimal one. One of two generally preferred for scientific plots.
`+ theme_classic()`: Changes the overall appearance to a classic one. The other one generally preferred for scientific plots.
Note that a basic plot can be saved, then modified by extracting the saved plot and adding the modification, e.g. `residplot + theme_minimal()`.

Changing symbols and colors:
ggplot allows almost infinite possibilities for customizing a plot. The aes() will control the symbol and color used in a plot. The most basic application is to specify shape and color in a aes(). The code shown here uses the treatment variable (looked for in the hb data frame because treatment is not a stand-along vector) to change the shape: `shape=treatment` and the color: `color=treatment`. This can be done inside the `geom_point()`, as in this code. This only applies to drawing the points. It could also be done in the ggplot aes() call; if so, that applies to any use of the data.

If you only specify shape= and color= in the aes() call, ggplot will choose its default colors and shapes.

To specify specific colors and shapes for each treatment, add the `scale_shape_manual()` statement to specify shapes and the `scale_color_manual` statement to specify colors. The lists of desired shapes and desired colors are defined using vectors of shape names `symbols <- c("circle", "square")` and color names, `colors <- c("blue", "red")`. These vectors are then used in the `values = symbols` and `values = colors` arguments.

QQ normal plots: `ggplot(data=hb, aes(sample = residuals)) + geom_qq()`
These are drawn by the `geom_qq()` command. This requires that the aes() specifies `sample =` the vector of quantities to use in the QQ plot. The aes() call can be general, i.e., in the ggplot() as here, or specific, i.e., in the `geom_qq()`. The `labs=` is optional to specify a title and X and Y axis labels.

Transformations `log()`: `log()` computes the natural log (base e) of the specified values. Because we want to use this in a model, I generally store in as a new variable in the data frame. R provides many other mathematical functions. Some of the other transformations sometimes used are `log10()`: log base 10, `sqrt()`: square root, and `1/`: reciprocal.

Backtransformation of results: If you want to express estimates on the original data scale (e.g., of a multiplicative effect), you will need to work with results from `t.test()` (or from `lm()`). Here is how to work with `t.test()`. Later, we'll see how to work with `lm()`. By default, the results are printed. But, they can be saved in a list object (not unlike saving the `lm()` results). There are extractor functions for many results, but not for all. Here we directly access the pieces we need.
What are the names of the stored results? `names()`: To see the names of the items stored in the result list, you can look in the help file for the generating function, e.g., `?t.test`, or you can ask R to give you the names. `names()` tells you the names. The different components of the results are access by `name$component`, e.g., `hb.test$estimate`.

List or data frame? We previously used `names()` and the $ symbol to find and refer to variables in a data frame. In R, lists and data frames are very similar. Data frames are lists with a particularly clean structure: Each variable in a data frame is a vector with the same number of rows. In a list, the components can have very different structures. Some components are scalars (single value), some are vectors, some are character strings, and some are matrices or other more complicated objects.

Sample averages for each group: `hb.test$estimate`: The `estimate` component of the t.test results has the averages for each group.
Difference of sample averages `diff(hb.test$estimate)`: The `diff()` function computes the running differences along a vector. When that input vector has only 2 values, it returns the difference.
Multiplicative effect `exp(...)`: The `exp()` function exponentiates its argument. This is the inverse of `log()`. If you used log base 10, you would need `10^X` to back transform your results.

Confidence interval for the log-scale difference `hb.test$conf.int`. The `conf.int` component of the results has the 95% confidence interval (or other interval if requested by `level=` in the t.test() call).
Confidence interval for the multiplicative effect `exp(hb.test$conf.int)`. The estimated multiplicative effect is obtained by exponentiating the estimated difference. We get the confidence interval by exponentiating the confidence interval for the difference.