patty.r: Explanation of code

Goals of code:

- Averaging observations to one value per experimental unit
- Manipulating data using Piping and dplyr functions

Averaging observations: description of the data in patty.txt
The hamburger study used last week was run a second time with a slight modification. In the second experiment, hamburger patties were randomly assigned to the treatment (active or control) but the bacterial concentration in each patty was measured three times per patty. So there are 12 experimental units (pattys) and 36 observations. The experimental unit is the patty; the observational unit (in the original data) is plate count. This is a violation of independence. A more appropriate analysis is based on the average count per patty. After averaging, the observational unit is the patty (because one row of data per patty), which is the same as the experimental unit (good).

One appropriate analysis is to calculate the averages for each patty and save them in a new data set. Then, we will analyze that new data set. More advanced courses discuss other approaches.

Read the patty.txt file. You should have a data table with three columns: trt, rep and cfu. There are three cfu values for each rep. We want to average each set of three to get a single mean for each combination of trt and rep.

The dplyr library
We have seen how to use tapply() to compute summaries (e.g., averages or sd's) for subsets of observations. When we want to average ou's to get a single value for each eu, we want to retain the treatment information. It might also be useful to retain the replicate number. While this can be done using some careful use of tapply(), it isn't easy.

The dplyr library provides functions that easily manipulate data. The data wrangling cheat sheet produced by RStudio,
`https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf`,
describes how to do all sorts of manipulations using some base R functions, dplyr functions and functions in the tidyr library. tidyr is a second collection of data management functions.

On the cheat sheet, functions are named by their "long" name: `library::function`. So anything on the cheat sheet that starts with dplyr:: requires the dplyr library. Both dplyr and tidyr are automatically installed by RStudio. To use either, you just need to activate the library with `library(dplyr)`. In R, you will need to install the package first. Remember, this is `install.packages('dplyr')`. Then you can activate the library.

Averaging observations for subsets of observations: `group_by()` and `summarize()`
Two steps are necessary to average the three observations for each experimental unit:

- Define groups of observations to average together: `group_by()`
- Define what function(s) to apply to each group: `summarize()`

I show you two ways to combine these operations. The second version (using pipes) is the one I use regularly.

Version 1: two separate lines, saving the intermediate result
`patty.g <- group_by(patty, trt, rep)`: adds grouping information to a data frame
The first argument is the name of the data frame (or the tibble variation on a data frame). All other arguments are names of variables that define groups. Here, each combination of the trt variable and the rep variable.

`patty.ave1 <- summarize(patty.g, meancfu=mean(cfu))`: calculate summaries
The first argument is the name of a data frame or tibble, almost always with grouping information. There are easier ways to compute summaries of all observations. The rest of the arguments (1 or multiple) have the form `name=function(variable)`. The right-hand side , `function(variable)`, specifies the variable in the starting data frame and the function to apply to it. `mean(cfu)` will compute the average of values in the cfu variable. Because the data set has been grouped (by `group_by`), this will be done for each group of observations. The left-hand side is the variable name to store the result.

`summarize(patty.g, n=n(), meancfu=mean(cfu), sdcdf=sd(cfu))`
You can request multiple summaries by adding keywords, n() gives the number of observations, sd() gives you the standard deviations. When there are multiple response variables, you can request summaries of them specifically, e.g. meanA=mean(A), meanB=mean(B), or use the `summarize_all()` function.

Version 2: Using piping :
`patty.ave3 <- patty %>% group_by(trt,rep) %>% summarize(meancfu=mean(cfu))`
If you know about pipe operations in a UNIX command shell, piping data manipulations is exactly the same idea. Version 2 does exactly the same thing as version 1 but has a more readable flow of information. This is especially clear when you spread a command out of multiple lines and use indenting to indicate groups of operations.

The pipe operator is %>%. `patty %>% group_by(trt,rep)` is exactly the same as `group_by(patty, trt, rep)`. Data frame %>% function means to use the specified data frame as the first argument of the function. With only one function, this isn't a huge benefit. Where there is a chain of functions, as here, piping makes the flow of information extremely clear. `patty %>% group_by(trt, rep)` runs the `group_by` function on the patty data frame. The result from that operation is then passed (because of the second pipe operator) into the summarize() function. Information flows from top to bottom and left to right.

Unfortunately, piping only works for a limited set of functions. All the dplyr functions can be piped. Functions requiring data= do not accept piped input. Eventually they might, but not right now.

Code using piping is especially clear to read if you spread the commands over multiple lines and use indenting.

Warning: Each line in a multiple line command must let R know there is more coming. This code will fail:

```
patty.ave2 <- patty
  %>% group_by(treat, rep)
```

because R thinks it's done at the end of the first line, because the first line is a complete command. You will get an error, but not until the second line. You need to put the pipe at the end of the line so that it is not a complete command (needing something more after the pipe).

Demonstrating identical results: `all(patty.ave1 == patty.ave2)`
The `==` is the logical equals operation. TRUE when two values are identical; FALSE when they are not. Applied to data frames, this compares each row and column and returns a data frame of TRUEs and FALSEs. We want to see whether every element is identical. This is what `any()` does. Returns TRUE when all inputted values are TRUE; if any is FALSE, it returns FALSE. In this case, these two lines demonstrate that the results from all three approaches are identical.