Creativity2.r: Explanation

Goals of code:

- Read a space delimited file
- Compute the standard error by defining a function
- Do a randomization test

Reading a space delimited file:

```
creativity <- read.table('creativity.txt', header=T)
```

Space delimited files are read by read.table(). In fact, read.table() can be told how to read comma delimited files, but read.csv() is simpler for those.

The first argument is the name of the file. Note the name includes the file type (also called the file extension). The second argument, header=T, is optional, but you want to include that for any 587 data file (and probably for all of your data files). header=T indicates that the first line in the file is to be interpreted as a header line containing the variable names.

See creativity1.r and explanatory document for why adding as.is=T is not needed in R versions 4.X.

The output of this function is a data frame containing the contents of the data file. The `<-` assigns this to a variable, in this case creativity.

If you look at the resulting data frame and see variables called V1, V2, ..., you forgot header=T. R made up variable names (since it wasn't told them) and the contents of the variable will be a mix of a name and the real data. That's garbage. Rerun read.table() with header=T.

Computing a standard error: From lecture, the formula for the se is $se = s/sqrt(n)$ where s is the sample standard deviation and n is the number of observations. More correctly, it is the number of observations that are not missing values. My approach is to define a new function that computes the standard error given a vector (1 column) of numbers.

Optional: information about how the function works - not necessary to understand and can be skipped.

`{se <- function(x) {` starts the definition of a function that accepts one argument, *x*. Inside the function, what ever variable is "passed into" the function is called *x*. The contents of the function are stored in the object called *se*
`s <- sd(x, na.rm=T)` computes the standard deviation of the values in *x*. the `na.rm=T` bit tells R to ignore any missing values.
`n <- sum(!is.na(x))` counts the number of non-missing values. `is.na(x)` returns a TRUE value if the observation is missing and FALSE if not. `!` is the logical NOT operator. TRUE becomes FALSE, FALSE becomes TRUE. If you do arithmetic on a logical (TRUE/FALSE) value, TRUE is treated as a 1 and FALSE as a 0. So `sum(!is.na(x))` counts the number of non-missing values
`s / sqrt(n)` `sqrt()` computes the square root of a number. So this computes the se of a mean. This is not saved so it is returned when the function finishes
`}` ends the function definition

The rest of this block of code illustrates how to use the se() function once it is defined. Short version: exactly the same way as mean() or median() get used. Once defined something written by users is treated exactly like a system-defined function.

`se{creativity`score) @ calculates the se of the vector of all 47 scores in the creativity data frame

`tapply(creativity$score, creativity$treatment, se)` computes the se of each group observations, i.e., each treatment

The following two commands are the tidyverse equivalents.

Randomization p-value: The code from `response <- creativity$score` to `p.value`
Base R doesn't compute randomization test p-values. This code uses R's programming commands to do that. In this example, `creativity$score` contains the response variable; `creativity$treatment` contains the grouping variable. These are copied into two new variables, response and group, which are the variables used elsewhere in the code. To use for a new problem, change the variables on the right-hand side of the first two commands, then execute all lines down to and including the p.value line.

The two-sided p-value is printed after you execute the last line.

A short description of what the code does:
save the number of observations and the unique names of the groups
create a vector of logical values with TRUE when that observation came from the first group
`obsdiff` is the difference in means between the first group and any other observations
the core of the code is a loop, executed once for each random assignment of labels to observations
`sample(group)` permutes the group labels
we then identify which randomly permuted observations were in the first group, compute the mean difference, and save it in vector
the two-sided p-value is then the number of more extreme random differences, with +1 to include the observed sample.

I am happy to provide more explanation if you want to know the details.

Note: Those of you familiar with computer programming will realize this code could be easily converted to a user-defined function. I agree. We'll define our own functions later.