light.r: Explanation

Goals of code:

- Do analysis by subgroups
- Fit models to both subgroups (ANCOVA, heterogeneous slopes regr.)

This lab uses the light.txt data set. This is case study 9.1 in the book. The study examined two groups of plants (E: early and L: late) at a range of light intensities. The response is the # of flowers produced. We want to fit regression lines that include both group and intensity.

Find unique values of a variable: `unique()`
Many times it is helpful to ask R to give you the unique values of a variable. That's what `unique()` does. I use unique() a lot when I am checking for mistakes in a data set. In the light data set, time should be E or L. If `unique()` returns e, E, l, and L, there are one or more data entry errors. An analysis on the `time=='E'` subset will ignore the observations with 'e'.

Do an analysis by subgroups: `subset()`
meat.r in Lab 6 introduced `subset()` . The meat.r explanation document described many of the logical operators that you can use with subset() to extract a subset of observations. We remind you about this here. The two arguments to subset() are the name of a data frame followed by a logical expression. You do not need to repeat the name of the data frame. `==` is the logical equals. So `time=='E'` is true for all observations in the Early group (with a time value of "E"). subset() extracts and returns all rows where the logical expression is true. When the result of subset() is saved in a new data frame, that new data frame has only one group.

Analyze a subgroup "on the fly": `subset=`
Most of the R model fitting functions, e.g., `lm()`, accept a `subset=` argument that defines the subset of observations to be included in the analysis. The right-hand-side of the `=` is (usually) the result of a logical expression. My practice is to put the logical expression inside `()`. This is usually not necessary but critical when it is.

Analyze all subgroups: `by()`
We have seen the dplyr functions `group_by()` and `summarize()`. This information

is in patty.r (Lab 6 for F 2018). These set up, compute, and store summary statistics for subgroups of observations. However summarize is limited to a small set of functions that return descriptive statistics. The `lm()` function returns a lot of pieces of information, so you can't use summarize() with it. The by() function is a more general way to analyze subgroups.

The base R `by()` function provides a way to run a specified function on all subsets of the data. The three arguments are:

a data frame
a vector defining the subgroups, this must be written out (unlike subset)
the function to be run on each subgroup.

A simple version, repeating what summarize will do without some of the benefits of summarize is `by(light$flowers, light$time, mean)`. Because mean expects a vector, the data frame going into by() is reduced to just the column with the number of flowers. The subgroup vector can not be "shortcut", i.e. even though time is (or was) in light, you must write `light$time`.

We want to get summaries of the regressions for each group. That means we want to run `summary(lm(flowers~intensity))` on the "E" subgroup, then on the "L" subgroup. The simplest way to do this is to write a function that accepts the data frame containing a subset then does what we want. This function can be one line or many lines.

Defining a function: `function(x) { }`
Functions work by accepting one (or more) arguments, doing something using those arguments, and returning a result. So, `myanalysis <- function(x)` defines a function named myanalysis which accepts one argument. The function is used by `myanalysis(lightE)`, which calls the myanalysis function with the lightE dataframe. Inside the function, there is no reference to lightE. Inside the function, the argument is called `x` because of the `x` in `function(x)`. Because of this "argument passing", a function can be used with many different sets of data. You just call the function with the name of the data to be used. You don't have to `x` as the internal name. You can choose whatever you want; I recommend something that helps you understand what your function is doing.

Note: if you wanted to pass two different arguments to the function `name`, the definition would be `function(x1, x2)` and called by `name(argument1, argument2)`.

The R code inside `{}` specifies what the function is to do with its argument(s). This

goes inside the squiggly braces `{}`. To get the summary of an lm() fit to the data frame named x, we would write `summary(lm(flowers ~ intensity, data=x))`. Or, you could write the function as two lines:

```
fit <- lm(flowers ~ intensity, data=x)
summary(fit)
```

The result of the function is the last unsaved result. In the two line version of the function, the result of lm() is saved in fit, so it can be used in summary(). The result of summary() is not saved. That output is what the function returns. Any variables defined inside a function disappear when the function finishes.

Putting the pieces together:
Define the desired function:

```
  myanalysis <- function(x) {summary(lm(flowers~intensity, data=x))}
```

Run the function on one data set to test it:

```
  myanalysis(lightE)
```

Use by() to run the function on all subsets:

```
  by(light, light$time, myanalysis)
```

Creating indicator variables automatically: `factor(time)`
When a variable is defined as a factor, R creates indicator variables automatically. The R default is that the FIRST level gets the 0 value. (Note: this can be changed. Ask me if you want to see how). You can create the factor version of the variable in the data frame: `light$time.f <- factor(light$time)` then include `time.f` in the `lm()` model, or you can create the factor "on the fly": `light.lm1b <- lm(flowers ~ factor(time) + intensity, data=light)`. I prefer to create a new variable because it simplifies predicting new observations.

If you look at the output from `summary()`, you see a line for `time.fL`. This is the name of the factor concatenated with the level of the indicator that had the value of 1.

Confidence intervals for model parameters: `confint(light.lm1)`
The confint() function provides confidence intervals for all the model parameters. These are T-statistic based intervals, because errors are assumed to be normal.

Viewing the values for an R created indicator: `model.matrix()`
After fitting a model, you can view the matrix of all X variables used for that fit by `model.matrix()`. The argument is the name of an `lm()` fit. This is especially helpful when there are factors in the model, because `model.matrix()` will

return columns for each indicator variable. If there are no factors, the result from `model.matrix()` is just the model variables from the original data frame.

Fitting ANCOVA models:
`light.lm1 <- lm(flowers ~ time.f + intensity, data=light)`
An ANCOVA model has groups with different intercepts but the same slope. We will let R automatically create indicator variables for the two time groups, then use those to define different intercepts.

fitting ANCOVA models with more interpretable intecept values:
`light.lm1 <- lm(flowers ~ -1 + time.f + intensity, data=light)`
The intercepts estimated using the previous model are combinations of the overall intercept (β_0) and the difference between the group-specific intercepts. This is a consequence of including the overall intercept in the model. Remember our discussion of overparameterized models. Same consequences here. If you omit the overall intercept, the estimates are the group-specific intercepts.

You tell R to omit the intercept by adding -1 to the model.

The intercepts reported by summary() are:
Early group: time.fE: 83.46
Late group: time.fL: 71.30

Fitting a heterogeneous slope regression: `time.f:intensity`
This model has groups with different intercepts and different slopes. The model statement includes time.f to generate different intercepts for each group. We also need the interaction (time by intensity) term to generate different slopes for each group. R can create that interaction variable automatically. The syntax is the two variable names separated by a : without any spaces.

Because models with many variables may have many interactions, R provides a short-hand for "variables and their interactions". That is `time.f*intensity`, which R expands into `time.f intensity time.f:intensity`, i.e., the two variables and their interaction. The result in `light.lm2b` (using `time.f*intensity`) is identical to that in `light.lm2` (using `time.f + intensity + time.f:intensity`)

heterogeneous slope regression, with more interpretable coefficients
`light.lm3 <- lm(flowers ~ -1 + time.f + time.f:intensity, data=light)`
We get group-specific intercepts and slopes by suppressing both the overall inter-

cept and the overall slope. We suppress the overall intercept by adding -1 to the model equation (just as done above). We suppress the overall slope by writing a model **without** intensity.

The results from summary() give us the following equations (rounding coefficients a bit) for the fitted lines:
Early group: $\hat{Y}_i = 83.15 - 0.0399\text{intensity}$
Late group: $\hat{Y}_i = 71.62 - 0.0411\text{intensity}$