Goals of code:

- Creating data frames with specified X values
- Calculating confidence and prediction intervals
- Lists and matrices
- Only printing a few observations
- Overlaying predicted values
- ANOVA lack of fit test

Creating data frames with specified X values: `data.frame()`

```
c(1,1.5,2,2.5,3,3.5,4,4.5,5)
```

Last week, we used `data.frame()` to create a new data frame with a specified single value. The result is a data frame with one row. This week, we create data frames with multiple rows. The key is to create a vector (column of values) that is used to create the data frame.

The three uses of `data.frame` show three ways to do similar things. You only need one of these.

The `c()` function concatenates (hence the c) values to make a vector. `c()` can be used anywhere you need to create or specify a vector of values. The values are separated by commas.

The `seq()` function generates a sequence of values. The first two arguments are the starting and ending values. The third is the step. If the step is omitted, e.g., `seq(1,8)`, a step of 1 is assumed.

The `1:8` in the third use of `data.frame()` is a shortcut way to generate a sequence of integers from the starting to ending value.

The code then computes the logtime variable, as we’ve done before.

Lists and matrices:

R provides many ways to store information. So far, we’ve met scalars (e.g., 5) and data frames, and we’ve just met vectors (e.g., `c(1, 2, 4, 8)`). A matrix has rows and columns, just like a data frame, but the entire matrix is either numeric or character. A data frame can have some columns that contain numbers and other columns that contain character strings. If a data frame with both numbers and character strings was converted to a matrix, it would become a matrix of character strings and all the numbers would be converted to character strings. The `as.matrix()` function converts another type of object into a matrix. The code involving `test`, `test2`, and `testm` demonstrates the difference between a data frame and a matrix.

The `names()` function extracts the column names from a data frame. If you try `names()` on a matrix, the result is nothing (NULL). The `dimnames()` function extracts row and column names from a
You can access individual rows or columns by subscripting the matrix. Subscripts go in single square brackets. A matrix has two subscripts, which are separated by a comma. The first subscript indicates rows, the second indicates columns. A negative number omits the specified row(s) or column(s). If a subscript is omitted, e.g., [,1], all rows or columns are selected. So [,1] will extract the first column (and all rows). You can also subscript vectors (only one index, so no comma) and data frames (rows and columns). If you omit the comma, the matrix is subscripted as if it were one long vector composed of stacked columns. You probably want to specify a comma.

A list can be recognized by either double square brackets [[ ]] or $name followed by information. The output from dimnames() is a list with two elements. The first is the vector of row names; the second is the vector of column names. When printed as [[1]], the components of the list are unnamed. You can access each component by subscripting in double square brackets, e.g., temp[[2]].

Components of a named list, such as produced by predict() in the next section, can be accessed by number or by name. testlist is a list with two components, named a and b. Both testlist$a and testlist[[1]] refer to the first component.

If referring to list elements by name seems like referring to columns in a data frame, that’s because a data frame shares many of the characteristics of a list and also many characteristics of a matrix. A data frame just has a regular structure than a list usually doesn’t have.

**Calculating confidence intervals on model parameters:** confint()
The `confint()` function calculates confidence intervals on model parameters. It is especially useful for regression models because the model parameters are easily interpretable. The required argument is the name of the model fit. By default, intervals are calculated for all parameters and are 95%. You can specify a `parms=` argument to only report intervals for selected parameters and `level=` to change the coverage (expressed as a number between 0 and 1, not a percentage).

**Calculating confidence and prediction intervals:** predict( interval=)
The `predict()` function has optional arguments that request more information about each predicted value. Just like last week, the required argument is the name of a model fit (here, meat.lm). Predictions are make for the X values used in the original model fit unless `newdata=` is specified. The argument to `newdata=` is the name of a data frame with new values for the X variable(s). The new data can have additional variables, which are ignored. But, it must have a column (or columns) with the same names as the variable(s) used to fit the model.
The `se.fit=T` argument requests standard errors for each predicted value. The `interval='confidence'` argument requests a 95% confidence interval for the predicted values. Adding `level` with a number between 0 and 1 changes the coverage to the specified value. Changing to `interval='prediction'` gives you prediction intervals instead of confidence intervals.

The output from `predict()` is a vector when all you request are predictions; it is a list when you request standard errors. When you add `se.fit=T`, the output includes `$fit` with the vector of predicted (fitted) values and `$se.fit` with the vector of standard errors.

When you request an interval, the output is a matrix, with the 2nd and 3rd columns being the endpoints of the requested interval. If you request both the standard error and an interval, you get a list with the `$fit` component as a matrix.

**Only printing a few observations: head()**
Sometimes, I want to print a few observations in a data set to get variable names (e.g., after reading a data set) or check for gross errors. The `head()` function prints the first six rows of a data frame, matrix, or vector. `tail()` prints the last six rows.

**Overlaying observations and predicted values: plot() followed by lines()**
We have previously used `plot()` to plot data. You can follow `plot()` with commands that add stuff to the plot. The `lines()` function draws lines (as connect the dots, so it usually helps to have observations in sorted order). The arguments to `lines()` are the X and Y information. Additional arguments modify the appearance of the line. The first `lines()` command adds the fitted line, drawn as a solid black line. The second and third adds the lower and upper prediction limits as a dashed line (`lty=2`, where `lty` is “line type”). The default line type is `lty=1`.

The `legend()` function adds a legend to the plot. The first argument is the location. All the rest can be in any order and specify what to put on the plot. I prefer legends without boxes; `bty='n'` (box type) suppresses the box. I want legends for the two line types, so I specify two line types in `lty=` and my legend as a vector of character strings in `legend=`. If you used `pch=` and `legend=`, you would label points in the legend. The help file for `legend` gives you a long list of options.

**ANOVA lack of fit test: anova()**
To construct the ANOVA lack of fit test, we need to create a copy of the X variable that is a factor variable. `time.f` is that factor version of the time variable. We will use this variable to indicate groups, each with its own mean, so it doesn’t matter whether the groups are defined by time or logtime.

We can compare the fit of the regression and the fit of the ANOVA model in two different ways.

1) Fit both models, then use `anova()` to compare the two fits. That is done by providing two models to `anova()`. If you wanted to sequentially compare more than two models (e.g., a simple linear regression, a quadratic regression, then an ANOVA fit), you provide three arguments to `anova()`.

2) Fit one model with both terms, e.g., `lm(ph ~ logtime + time.f)`, with a `+` separating the two terms. Passing the output from that fit to `anova()` gives you the sequential change in fit as each
term is added to the model.

Note: If the first variable in the model is the factor variable, R refuses to fit the second term. If you specify two models to anova() and the factor variable is first, the changes in df and SS are both negative. Fix by reversing the order of the variables in the lm() formula or the order of the models in anova().