We’ll start with single-sample bootstrapping (using confidence intervals as an example) to introduce the functions you’ll need for resampling methods in general.

**Part 1: Resampling (and subsampling)**

The key function necessary for resampling is `sample()`. It can take several forms but all require a single vector (or factor, but you’ll mostly use numeric vectors) as the primary input.

The full form of the function looks like this:

```
sample(vector, n, replace=F)
```

Where `vector` is the input to be resampled (or subsampled), `n` is the number of data points to be output, and `replace=F` allows sampling without replacement or sampling with replacement (with `replace=T`).

A value only needs to be specified for `n` if you want to perform subsampling. If you want to randomize or bootstrap the entire data, returning the same number of data points as in the input, you can just write:

```
sample(vector, replace=T)  # for bootstrapping
```

or

```
sample(vector)  # for randomization (replace=F is the default)
```

**Part 2: Replicating**

The `sample()` function produces a random resampling (or subsampling) of the input data. However, because it’s random any single result may be anomalous and not representative of the underlying data. As a result, resampling methods repeat the process many times to generate a reliable sampling distribution. The recommended number of iterations has increased as computing power has increased over the years, but it is now trivial to perform 10,000 (unless the procedure is very complicated).

The function for repeating the resampling procedure is `replicate()`. It is a straightforward function to use, but you’ll want to set it up in a way to return easy-to-use output:

To repeat a function `n` times, just use:

```
replicate(n, command)  # command is the name of the function you want to repeat
```

The combination of `sample()` and `replicate()` is the fundamental basis of resampling methods.
Section 3: Working with matrices

If you consider how to set up your replication, you will be able to avoid working with matrix data. However, there are some extremely useful data manipulation functions for more sophisticated analysis.

Imagine a matrix like this:

```
[,1] [,2] [,3]
[1,]  1  1  1
[2,]  2  2  2
[3,]  3  3  3
```

For the preceding matrix, `apply(x, 1, mean)` will return the vector 1, 2, 3, the mean of each row. The function `apply(x, 2, mean)` will return the vector 2, 2, 2, the mean of each column.

You may recall the function `colMeans()` from last week, which does the same thing as `apply(x, 2, mean)`. There is also `rowMeans()`, equivalent to `apply(x, 1, mean)`.

The `apply()` function is more generic because you can apply any function to the rows or columns of a matrix. It’s also slower: `colMeans()` on the example matrix takes about 7 microseconds but `apply()` takes something like 55 microseconds (on my computer), although code optimization isn’t something you need to worry about that this stage.

Part 4: Quantiles and confidence intervals

The output from the combination of `replicate()` and `sample()` will be a sampling distribution of the parameter of interest. That means it will show the distribution of means, medians, R2 values, etc. that would be expected for many repeated samples from the population (using the sample data as the best estimate of the population).

A confidence interval is a (usually) symmetrical range around the parameter estimate itself. The 95% confidence interval will include 95% of the area under the sampling distribution around the center. Another way of saying is that that the smallest 2.5% and largest 2.5% of the repeated samples from the population will be outside of the 95% confidence interval.

The R function to identify the value that is larger than a given proportion of the data is called `quantile()`. It can give you any quantile between 0 (the smallest value) and 1 (the largest value). A quantile of 0.5 would be the median, because 50% (a proportion of 0.5) of the data is smaller and 50% is larger.

The function needs two inputs: the numeric vector of interest (called `x` here), and the desired quantile(s):

`quantile(x, quantile)`

For example, you can get the 25% quantile (also called the lower *quartile*) with:

`quantile(x, 0.25)`
You can provide multiple quantiles to the function as a vector of numbers (remember that you use \( c() \) to combine numbers into a vector):

For example; you can get the distance between the 2.5% and 97.5% quantiles:

\[
\text{quantile}(x, c(0.025, 0.975))
\]

**Bootstrapping correlation or regression:**

These tests require two continuous variables and every observation will have a value for each one (an \( x \) and a \( y \) coordinate value per point). You want to keep those associated values together during the bootstrapping, rather than mixing the \( x \) coordinate from one observation with the \( y \) coordinate from a different one. To do that, you’ll need to select entire rows of the data frame with replacement. You learned early in the quarter that rows and columns within data frames can be specified by their numeric position. For example, \( x[1,] \) is row 1 in a data frame named \( x \), and \( x[c(1, 2, 3),] \) includes rows 1, 2, and 3. Think about how you could use the \( \text{sample}() \) function to get randomly sampled rows with replacement.

The results from \( \text{cor.test} \) and \( \text{lm} \) (after running the \( \text{summary} \) function for \( \text{lm} \)) have a nice formatting for your screen, but all of those individual elements are stored as different entries in the object. You can use the \( \text{names}() \) function to see what they are called. The objects are not data frames, but you still use the $ syntax to extract individual elements from the object. For example, \( \text{summary(lmresult)}$r.squared \) to get the R2 value.