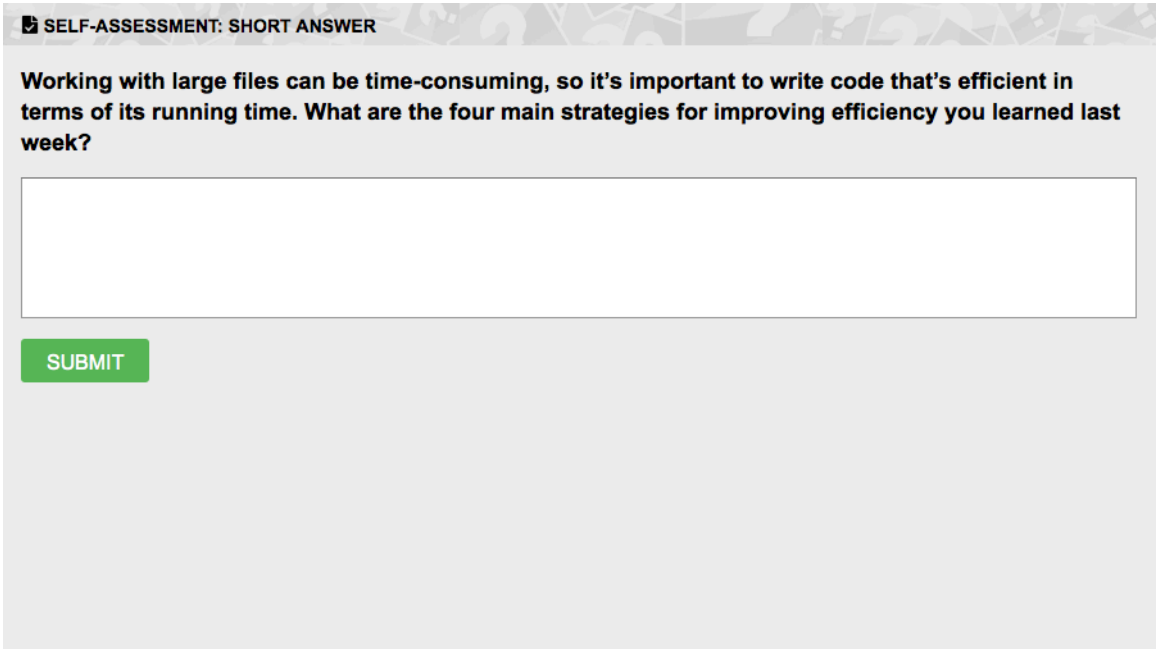
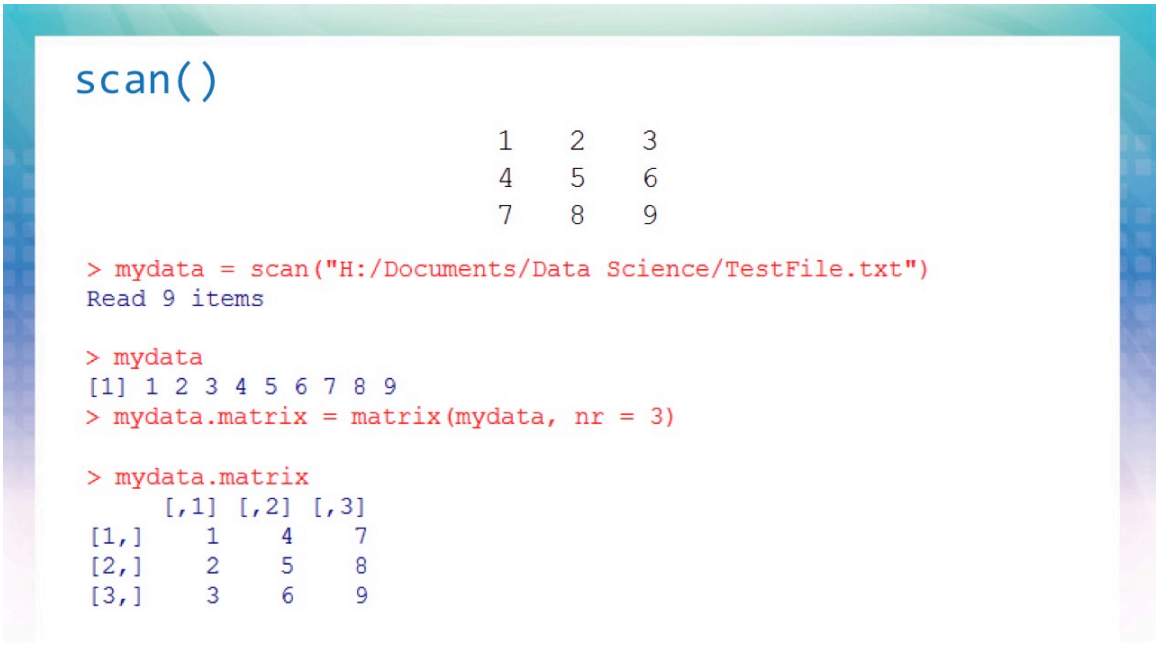

DS 710
Programming for Data Science
Working with large files in R



Question 1

See correct answer at the end of the transcripts.



Both read.csv and read.table can be very slow when reading large data files. So the best way to read a large data file into R is using the Scan function. Let's look at an example using the data in testfile.txt. This is a very small file. It just has three columns and three rows of data. So we wouldn't need to use Scan here. We could use read.table, because the numbers in each column are separated by spaces rather than commas.

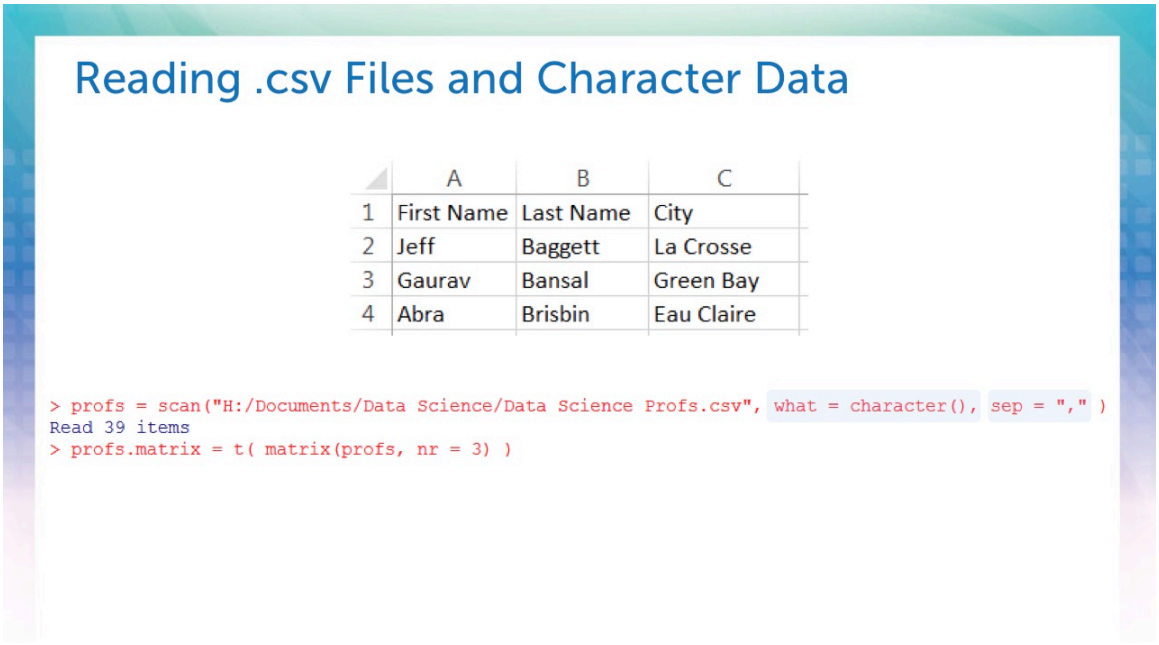
But let's try using Scan here as an example. The syntax will look like this. When you do that, the mydata variable now contains the numbers that were in the file. But they're just listed as a vector rather than a matrix or a data frame. To convert it into a matrix, we can use the Matrix function and specify how many rows we want our matrix to have.

But when you look at this matrix, you see that there's a problem. Because R fills matrices with the first column, then the second column, then the third column, it fills the numbers in the order in which it read them in from the file. So our matrix ends up being flipped diagonally compared to the way it was in the file.

Transposing a Matrix

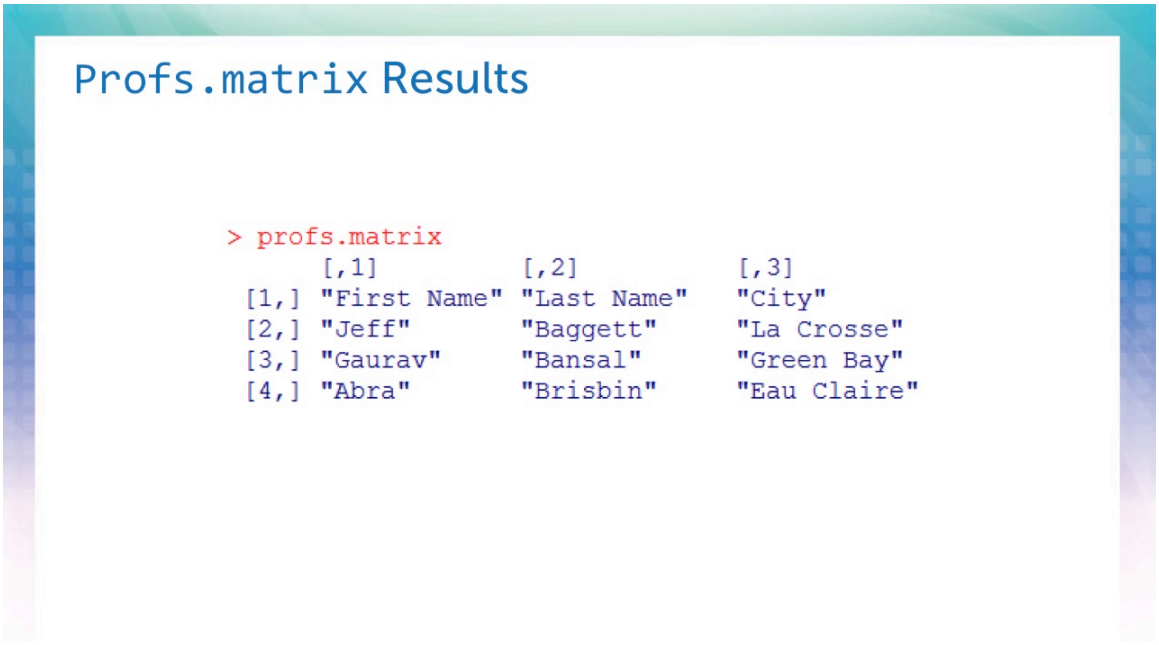
```
> mydata.matrix = t(mydata.matrix)
> mydata.matrix
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

The way to fix this is to transpose the matrix using the function t.



You can also read CSV files using Scan by using the sep, or separator argument, to tell R what character separates the values in different columns. In this case, a comma. This is the Data Science Profs.csv file which contains data on professors who are teaching data science courses. This is character data. So we need to tell R not to expect numeric data. We do that using the what argument.

Down below, you'll notice that I'm converting the data into a matrix and transposing that matrix using one line of code. You'll also notice, in the nr = 3 argument, that I'm setting the initial number of rows of the initial matrix based on the number of columns of our original data set.



This produces these results. And you'll notice a slight problem. The header row was read in as if it was another row of data.

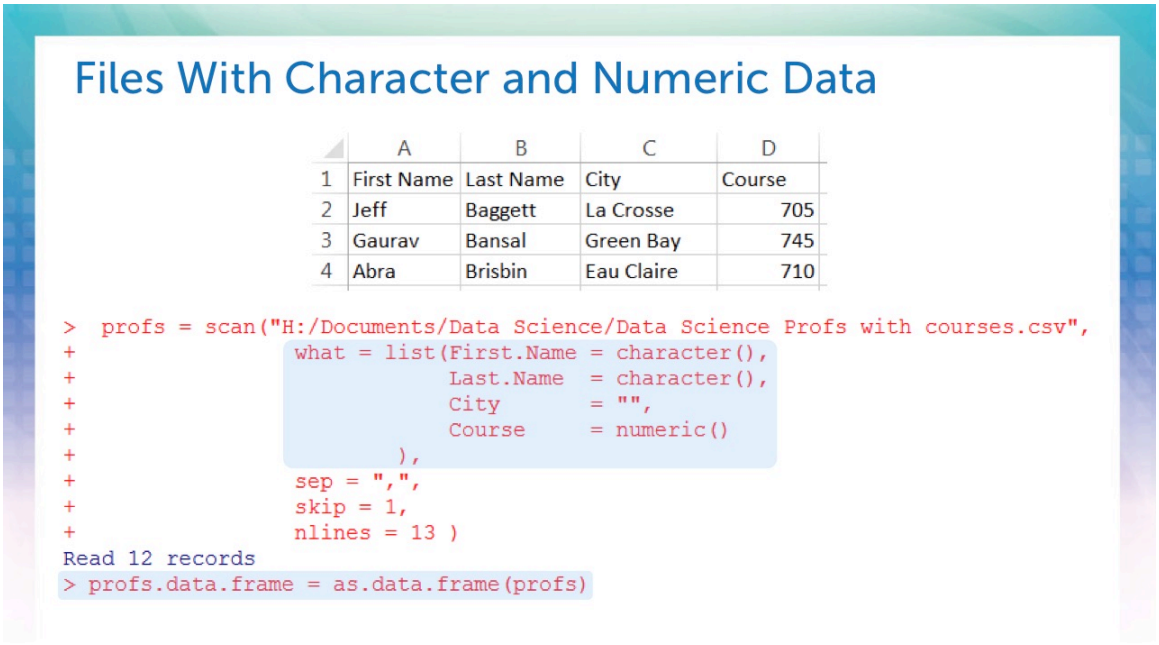
Dealing with a header row

```
> profs = scan("H:/Documents/Data Science/Data Science Profs.csv",
+              what = character(),
+              sep = ",",
+              skip = 1,
+              nlines = 13 )
Read 36 items
>  profs.matrix = t( matrix(profs, nr = 3) )


> profs.header = scan("H:/Documents/Data Science/Data Science Profs.csv",
+                     what = character(),
+                     sep = ",",
+                     nlines = 1 )
Read 3 items
>  colnames(profs.matrix) = profs.header
```

We can fix this by using the argument skip to skip one or more lines of data at the start of the file. In this case, we want to skip one line. That's the header row. I've also added the argument nlines = 13 to tell R to read in the first 13 lines after the one that it skips. This allows R to pre-allocate the memory for this data set, which, as we've seen before, is more efficient than having to increase the amount of memory. The number that you put after nlines can either be the exact number of rows that you want to read, or it can be a slight overestimate.

Then we can use another Scan function to read in the header row by using the argument nlines = 1. And then we can use the colnames function to set the column names of that matrix profs.matrix.

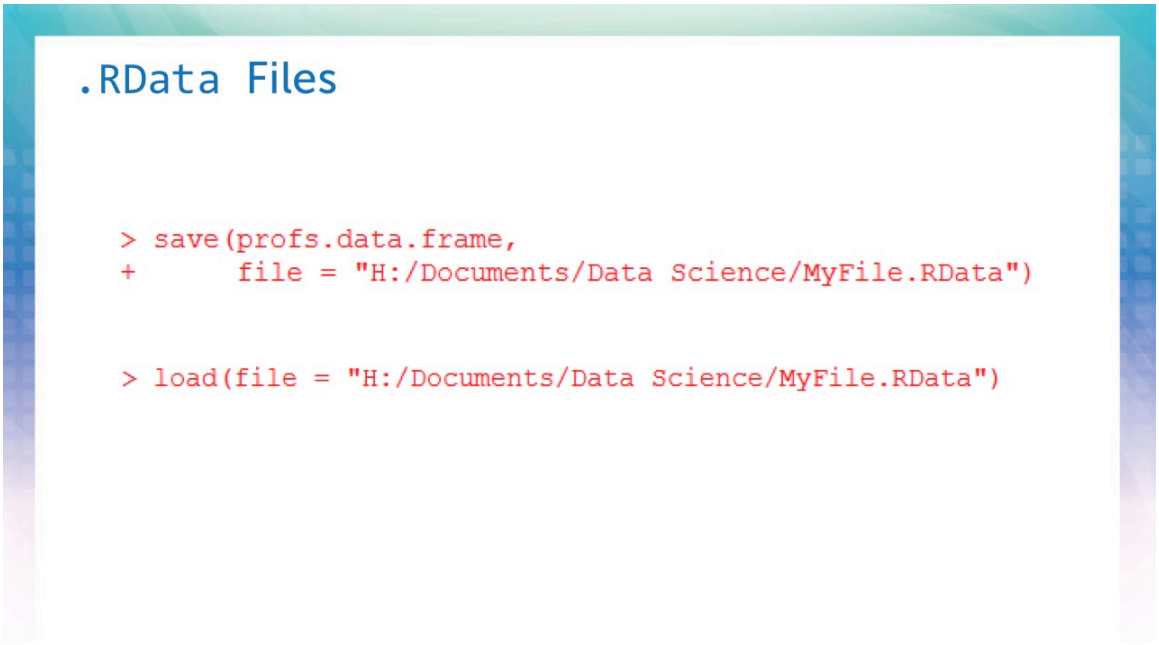


The file Data Science Profs with courses contains three columns of character data and one column of numeric data. We can still read this into R using Scan using a command that looks like this. This is the same as what we've seen before, just with a different segment for the what argument.

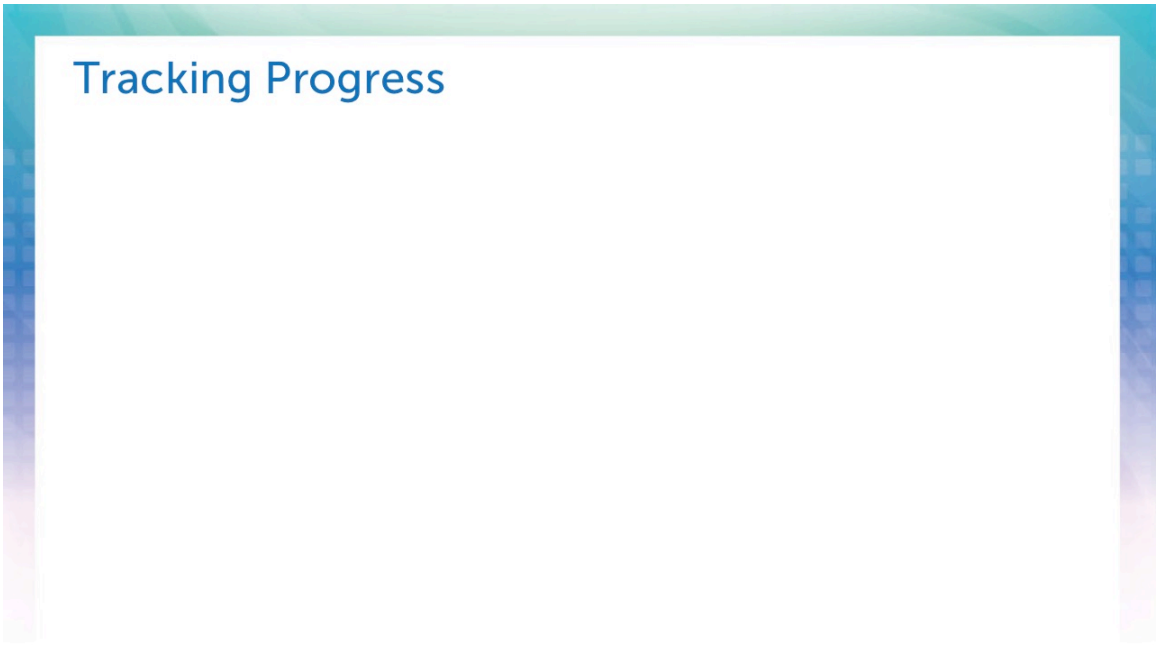
It's a little more complicated, but we can handle it. The what argument is now a list that contains the names of each of the columns set equal to the type of data that we want to include in them, either character with parentheses or numeric with parentheses. Or you could use quotation marks with nothing in between them to represent character data.

We can then create a data frame to hold the data that we've read in using the as.data.frame function. However, remember that for large data sets, data frames can be slightly less efficient than matrices, because for a data frame, you also need to store information about the type of data in each column.

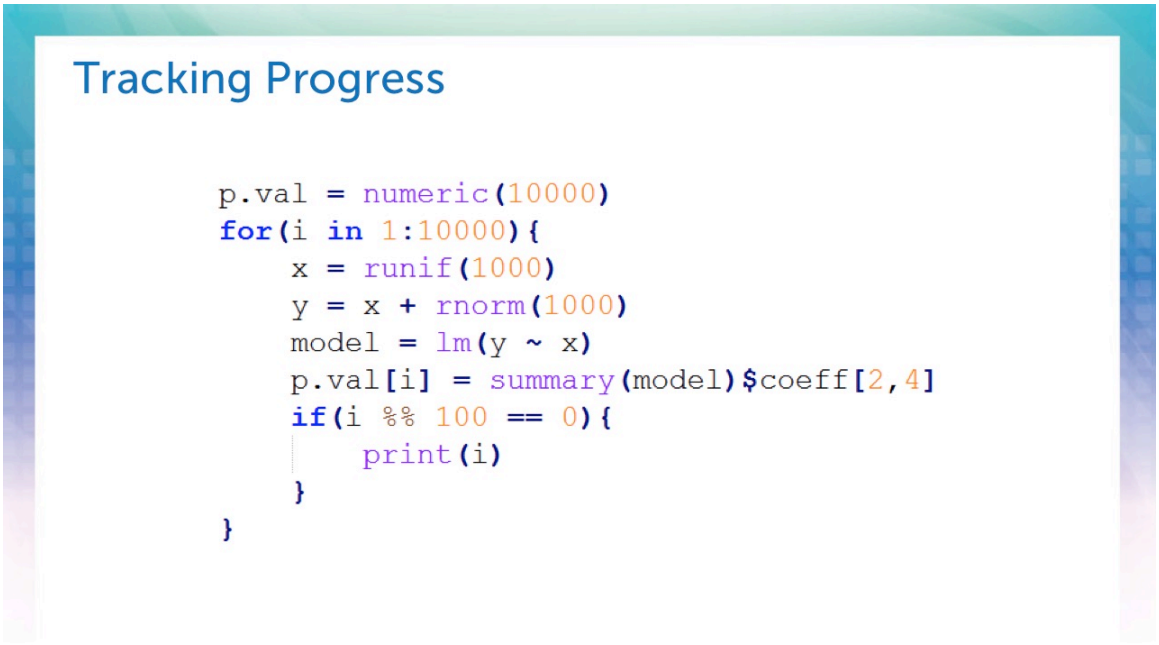
So for a very large data set, you might prefer to separate the columns containing character data from the columns containing numeric data, and store each of those separately in a matrix.



If you're planning on reusing the same data set in another session of R later on, you can save some time in the future by saving the data set as a .RData file. This is a compressed file format that's specifically designed for R. After you've read the data in using Scan, you can save it using the Save function. And the arguments here are the name of the variable you want to store. Or you could put more than one variable. And the last argument is file = and the file where you want to save that data set. To load the data set in later, you can use the Load function.



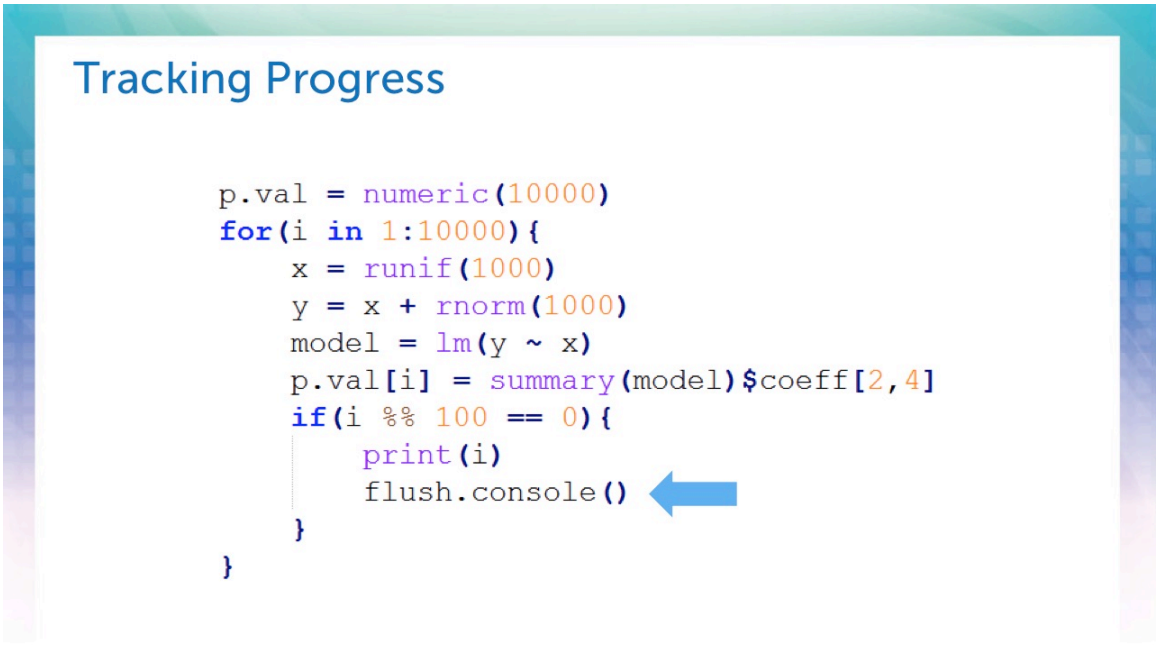
When you're analyzing large data sets, it's frequently helpful to have a way of tracking your progress, so that you can step away from your computer for a few minutes, get a cup of coffee. And when you come back, you can know whether your analysis is 10% done or 90% done.



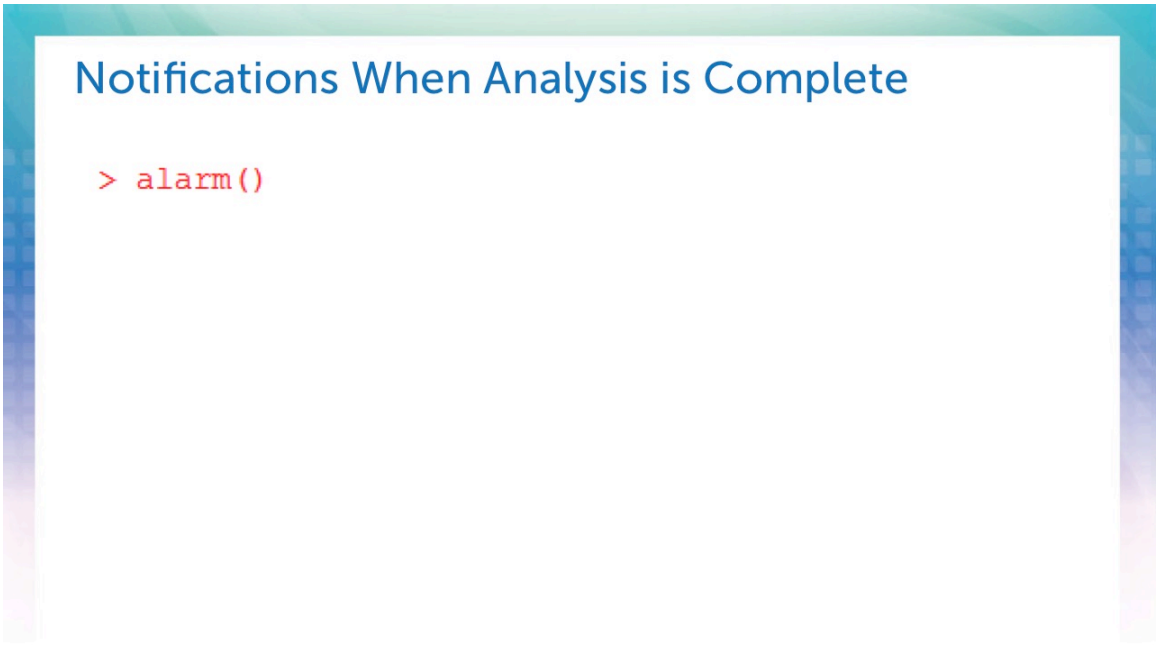
For example, here's a section of code that simulates values for x and y, 1,000 pairs, and then does linear regression, and checks the p value for an association between these two variables. And it does this 10,000 times. This section of code doesn't take too long, about 20 seconds on my computer. But it could easily take several minutes or even several hours if we increased the size of the vectors x and y or the number of iterations in the For loop.

So to track our progress, we can insert a statement like this. We use an If statement to test whether i is divisible by 100. And if it is, then print i. So we're printing the number of the iteration we're currently on.

One problem with this, however, is that sometimes there's a lag between when you tell R to print something and when it actually shows up on the screen. The print statement gets stored in the buffer. And R will sometimes wait through several more iterations before it actually prints the number of the iteration you were on.

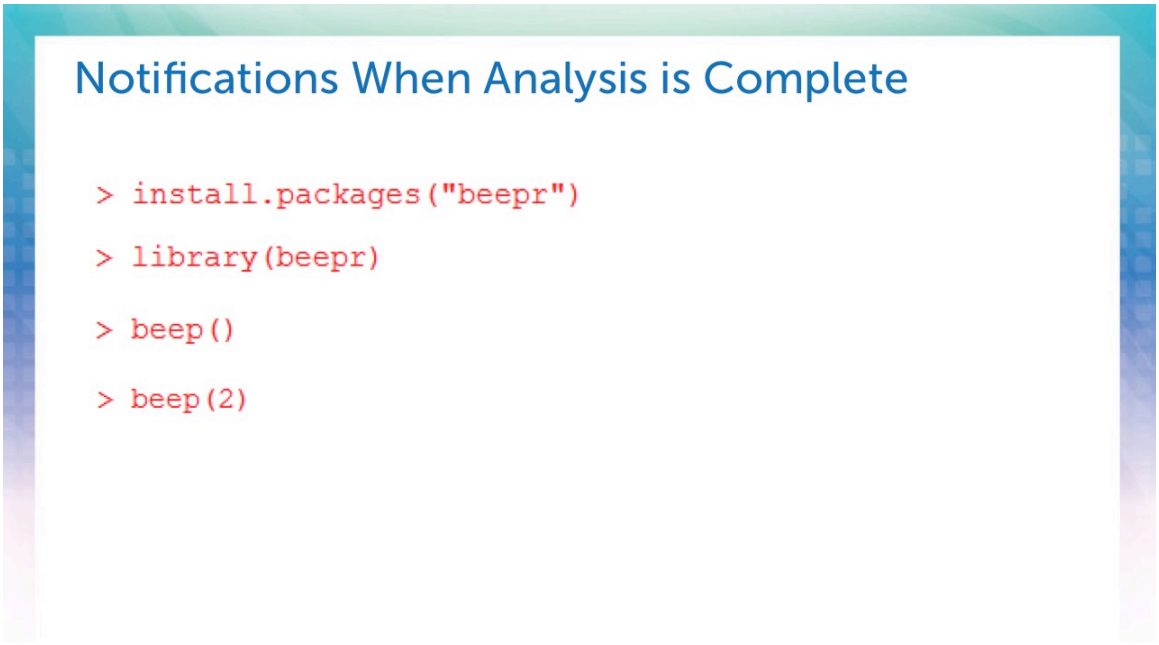


This can be fixed by inserting a line to tell R explicitly to flush the buffer and print everything to the console. This is just the command flush.console.



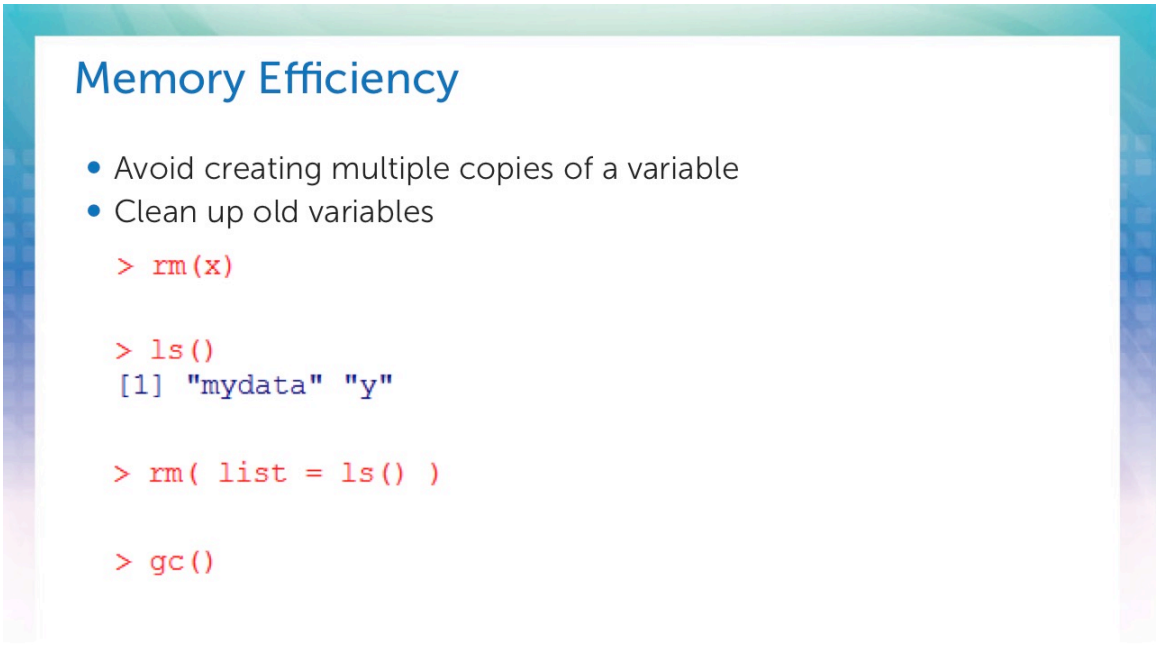
Another way you can track the progress of an analysis is to have R give you a notification when the analysis is complete. You can do this using the alarm function, which causes R to produce a beep. Just insert this function at the end of whatever segment of code you want a notification about. It doesn't require any arguments inside the parentheses.

Do make sure that the sound is turned on on your computer. And make sure to press Enter at the end of the line of code that has the alarm. And be sure to copy the blank line below it into the R console when you're copying the code in there, so that R will actually execute that last line of code.



On some computers, the alarm function doesn't work. So test it on your computer. And if it doesn't work, you can use this alternative using the beepr package. This also causes your computer to produce a beep. And it even takes different arguments to produce different kinds of sound effects. So check out the manual page, and then pick your favorite one.

These are the two simplest ways that you can get R to notify you when your analysis is complete, so that you can be working in some other program and get an audible notification when it's time to switch back to R. There are fancier ways of doing this, including a package that lets you have R send you a tweet to let you know when your analysis is finished. There's a link to that below.

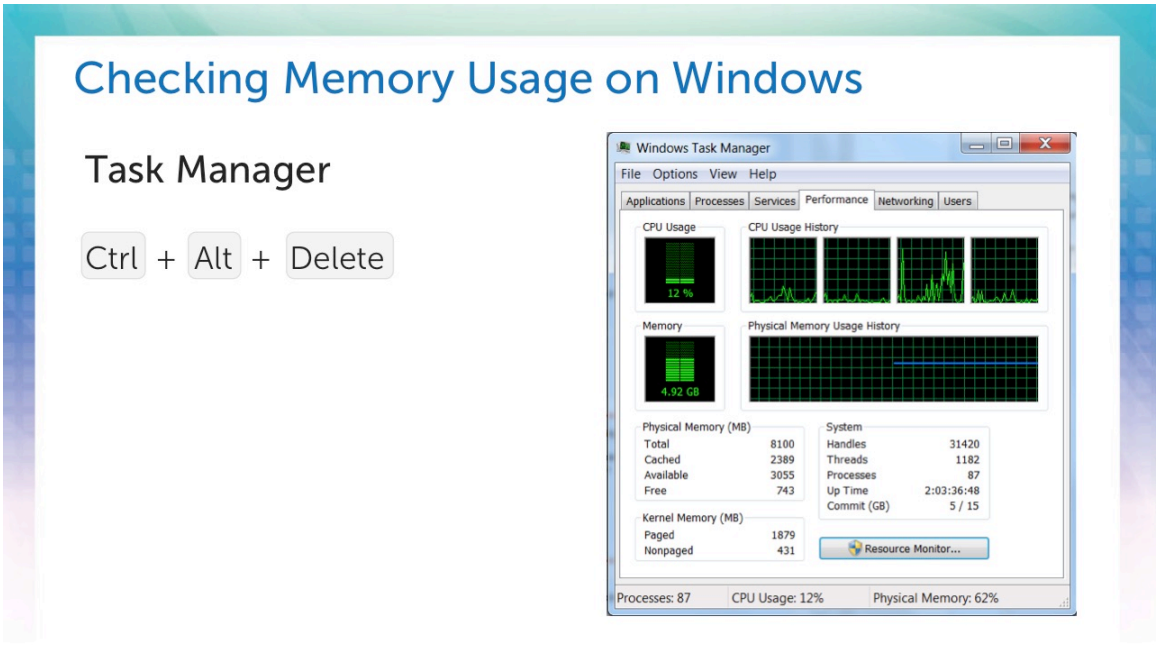


When you're working with large files, it's important to be efficient both in terms of running time and in terms of memory. To have good memory efficiency, avoid creating multiple copies of a variable or having multiple variables that basically store the same information.

Also, clean up old variables once you're done using them. You can do this using the rm function, which stands for remove. So for example, rm (x) would delete the variable x. To see what variables are still in memory, you can use the ls function to list those variables.

You can also use rm list = ls to remove all of the variables currently in memory. Be careful when you're doing this. There's no way to undo it.

And finally, after you've removed a particularly large variable, it's a good idea to use gc, or garbage collection, to force R to free up the memory from the variable it just deleted.

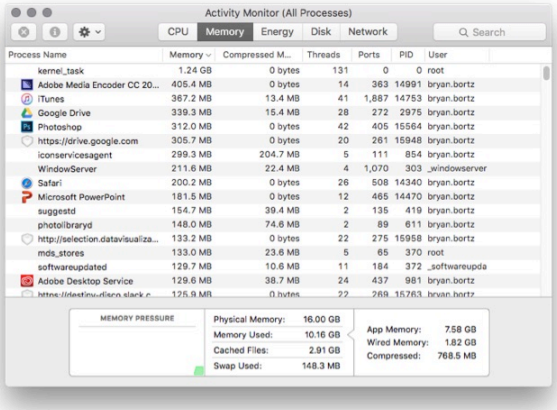


To check how much memory you're currently using, on a Windows computer, you can press Control-Alt-Delete and then choose the Task Manager. Here you can see I'm using about 62% of my computer's memory. If this number gets too high, say, around 90%, you'd probably want to close down some other programs such as your internet browser to allow R to have more memory to work with.

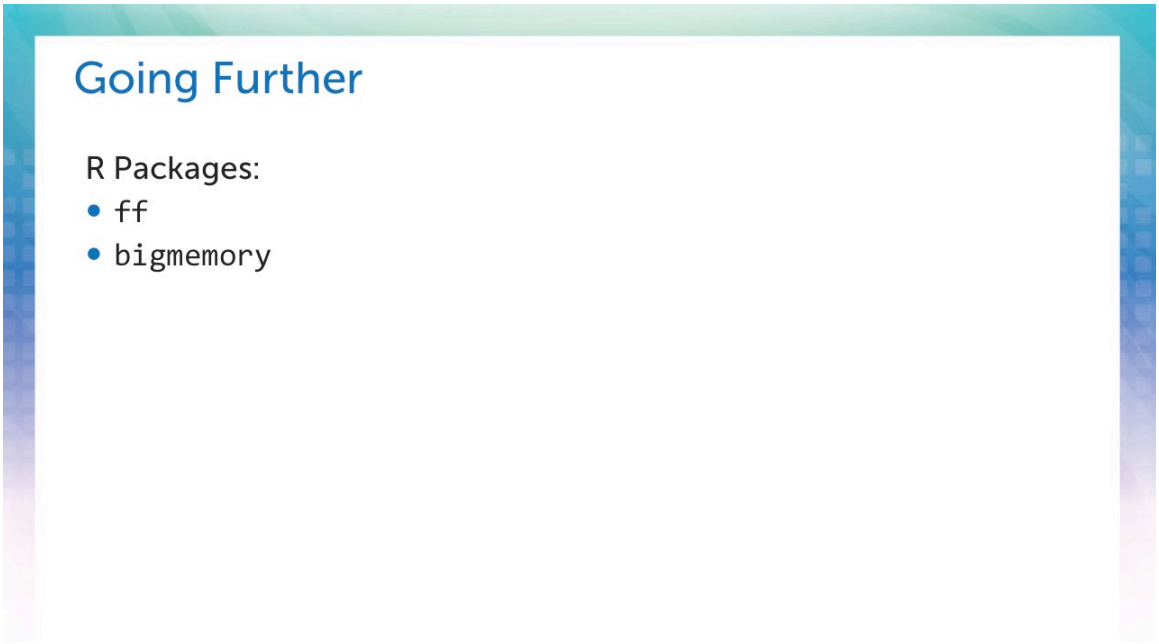
Checking Memory Usage on Macs

Activity Monitor

`/Applications/Utilities/`



To check how much memory you're currently using on a Mac, use the Finder to search for the Activity Monitor. And then choose the Memory tab. Here you can see my memory pressure is very low shown in the green bar. If this got to be higher, up into the red, then I would want to close down some of the other programs I was running such as my internet browser, in order to allow more memory for R to use.



If you need to work with very large files frequently, you might want to look into the R package's ff and bigmemory, which are designed to help deal with memory issues involved in reading and analyzing large files.

Question 1 answer:

SELF-ASSESSMENT FEEDBACK

Working with large files can be time-consuming, so it's important to write code that's efficient in terms of its running time. What are the four main strategies for improving efficiency you learned last week?

Your answer:

–

Correct answer:

1. Avoid repetition.
2. Vector calculations are better than *for* loops.
3. Use existing functions.
4. Plan ahead for memory use.