The objective of this section is to review the syllabus and to introduce the `R` environment. If there is remaining time, I'll work through some basic code puzzles that will require you to work in `R`. The first two or three sections will feel slow for those of you with substantial experience in `R`[1], but I promise we'll speed up soon.

Installing `R`

Download `R`: The download of `R` will vary by operating system, but it will begin here in any event:

`cran.r-project.org`

The online documentation and installer routines are comprehensive. If you are new to `R`, then it might make sense to use the Mac or Windows distribution, along with the built-in editor to write and evaluate code. `Rstudio` is a popular IDE that provides a somewhat more user-friendly interface than the base `R` installation. For the tech-oriented, the Linux distribution is very flexible; I use Emacs with the ESS package for editing. If you are interested in using the Linux distribution and are having trouble with the setup, please see me.

Learning `R`

I have included links in the syllabus to a few of the many resources on the web that provide gentle introductions to the `R` language. Those of you who have no experience with `R` or with programming in general will find it well worth your time to spend a few hours browsing these, in particular the starter resources for `R` from the UCLA Statistics department. In section I will focus on presenting examples of code piece-by-piece in order to illustrate certain concepts. My intention is to expose you to a small part of the `R` language in section so that you'll feel more comfortable exploring the rest.

Once you feel sufficiently competent in the language you will find that the optimal strategy for learning how to put together a particular piece of code is usually to search the web for "R [whatever you want to do]". RSeek is also an immensely useful resource. If you want to learn about a specific function, simply type `?func` into your `R` console, where `func` is the name of the function you want to look up.

Creating and testing matricies

The lingua franca of this course is matrix algebra, so we will start by introducing some of the more common commands for working in matrix-world[2].

There are a variety of data objects in `R`, including numbers, vectors, matrices, strings, and dataframes. We will mainly be working with vectors and matrices, which are quick to create and manipulate in `R`. The `matrix` function will create a matrix, according to the supplied arguments.

[1] If you're bored, skip ahead to the puzzles. I won't tell.

[2] Unfortunately not quite as cool as The Matrix, but probably cooler than The Matrix: Reloaded and undoubtedly cooler than The Matrix: Revisited.

```
A <- matrix(1:6, ncol=2)
B <- matrix(1:6, ncol=3, byrow=TRUE)
```

For convenience, we use `->`[3] to assign the matrices to the variables `A` and `B` for use in subsequent manipulations. The `ncol` option specifies the number of columns for the output matrix; and the default behavior of `matrix` is to cycle through by column. To cycle through by rows we set the optional argument `byrow=TRUE`.

Suppose we wanted to check to see if the first matrix was equal to the transpose of the second. This is clearly the case — we can see that it is. But when we're working with larger matrices it will be convenient to have a way to do this programmatically. The `==` comparison operator will yield `TRUE` or `FALSE`:

```
A == t(B)
```

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

Note that `t()` will return the transpose of the supplied matrix. Each element is checked individually, and each is identical in matrix $\mathbf{A}$ and $\mathbf{B}'$. To check the truthiness of the statement that all elements are identical, we need only to employ the `all` function:

```
all(A == t(B))
```

```
[1] TRUE
```

Keeping track of your matrix dimensions is a Good Idea™. That's where the `dim()` command comes in handy:

```
dim(A)
dim(B)
```

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

With the dimensions of our matrices in mind, we'll move on to matrix operations.

Matrix operations

Matrix muliplication in `R` is bound to `%*%`, whereas scalar multiplication is bound to `*`. Consider the product $\mathbf{BA}$:

```
B %*% A
```

```
     [,1] [,2]
[1,]   14   32
[2,]   32   77
```

[3]The === operator also assigns values, with a slightly different behavior. It is also common practice to use the === assignment for function arguments. See the [[http://goo.gl/hgOJ][Google style sheet]] for a description of other standard practices in =R=.

The dimensions have to line up properly for matrix multiplication to be appropriately applied, otherwise R returns an error, as is the case with the product $\mathbf{BA}'$:

```
B %*% t(A)
```

```
Error in B %*% t(A) : non-conformable arguments
```

If scalar multiplication is applied to matrices of exactly the same dimensions, then the result is element-wise multiplication. This type of operation is sometimes called the Hadamard product, denoted $\mathbf{B} \circ \mathbf{A}'$:

```
B * t(A)
```

```
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
```

This is rarely what we want to do. More common, if we want to scale all elements by a factor of two, say, we just multiply a matrix by a scalar; but note that `class(2)` must be not be `matrix` but rather `numeric` so as to avoid a non-conformable error:

```
A * 2
```

```
     [,1] [,2]
[1,]    2    8
[2,]    4   10
[3,]    6   12
```

```
A * matrix(2)
```

```
Error in A * matrix(2) : non-conformable arrays
```

Consider a more complicated operation, whereby each column of a matrix is multiplied element-wise by another, fixed column. Here, each column of a particular matrix is multiplied in-place by a fixed column of residuals. Let $\mathbf{e}$ be a vector defined as an increasing sequence of length three:

```
e <- matrix(1:3)
```

Note first that the default sequence in R is a column vector, and not a row vector. We would like to `apply` a function to each column of $\mathbf{A}$, specifically a function that multiplies each column in-place by $\mathbf{e}$. We must supply a 2 to ensure that the function is applied to the second dimension (columns) of $\mathbf{A}$:

```
apply(A, 2, function(x) {x * e})
```

```
     [,1] [,2]
[1,]    1    4
[2,]    4   10
[3,]    9   18
```

The function that is applied is anonymous, but it could also be bound to a variable – just as a matrix is bound to a variable:

```
whoop <- function(x) {x * e}
apply(A, 2, whoop)
```

```
     [,1] [,2]
[1,]    1    4
[2,]    4   10
[3,]    9   18
```

We will often need to define an identity matrix of dimension n, or $\mathbf{I}_n$. This is quick using `diag`:

```
I <- diag(5)
```

There are many ways to calculate the trace of $\mathbf{I}_5$. One method has been bundled into a function, called `tr()`, that is included in a package called `psych` which is not included in the base distribution of `R`. We will need to grab and call the library to have access to the function, installing it with the command `install.packages("psych")`. For this, you'll need an internet connection.

```
library(psych)
tr(I)
```

```
[1] 5
```

We can get a list of all the object currently available in memory with the `ls()` function, which is useful as the assignments begin to accumulate:

```
ls()
```

```
[1] "A"     "B"     "e"     "I"     "whoop"
```

Note that the objects we did not explicitly assign, such the transpose of $\mathbf{B}$, `t(B)`, or the trace of $\mathbf{I}$, `tr(I)`, are created on the fly and not stored in memory.

When paired with the `rm()` function, we can use `ls()` to delete all of the objects in memory. This is similar to the command `clear` in Stata.

```
rm(list = ls())
```

What's going on here? `list` is actually the name of an argument built in to the `rm()` command. The default behavior of `rm` is to accept character strings; we could have alternatively specified `rm("A","B","e","I","whoop")` and the outcome would have been the same. But by passing it a list of all of the objects in memory, we are telling `rm()` to clear everything, not just the variables we name.
Next week we will begin using actual data (!) to do operations in `R`.

Linear algebra puzzles

These notes will provide a code illustration of the Linear Algebra review in Chapter 1 of the lecture notes. Don't worry if you can't solve these puzzles, many of them require commands that we have not covered in section. Come back to them later, once we have gone over `R` code in more detail. There are many correct ways to solve these puzzles. If time remains, I will go over a couple of these next week.

1. Let $\mathbf{I}_5$ be a 5×5 identity matrix. Demonstrate that $\mathbf{I}_5$ is symmetric and idempotent using simple functions in `R`.

2. Generate a 2×2 idempotent matrix $\mathbf{X}$, where $\mathbf{X}$ is not the identity matrix. Demonstrate that $\mathbf{X} = \mathbf{X}\mathbf{X}$.

3. Generate two random variables, $\mathbf{x}$ and $\mathbf{e}$, of dimension $n = 100$ such that $\mathbf{x}, \mathbf{e} \sim N(0, 1)$. Generate a random variable $\mathbf{y}$ according to the data generating process $y_i = x_i + e_i$. Show that if you regress $\mathbf{y}$ on $\mathbf{x}$ using the canned linear regression routine `lm()`, then you will get an estimate of the intercept β_0 and the coefficient on $\mathbf{x}$, β_1, such that $\beta_0 = 0$ and $\beta_1 = 1$.

4. Show that if $\lambda_1, \lambda_2, \ldots, \lambda_5$ are the eigenvectors of a 5×5 matrix $\mathbf{A}$, then $\text{tr}(\mathbf{A}) = \sum_{i=1}^{5} \lambda_i$.