The purpose of this section is to introduce you to writing your own packages in `R`, and to get you started on calculating R^2 values.

Last Section

I'm looking forward to seeing your problem sets nicely typed up using `knitr`. While it isn't explicitly required, I strongly urge you to do so.

I usually write `R` code using RStudio but edit `.Rnw` files using TexStudio. Instructions for using TexStudio to compile from `.Rnw` to `.pdf` can be found at:

`http://yihui.name/knitr/demo/editors/`

My build command (which is discussed in the link) is, for example:

```
"C:/Program Files/R/R-3.1.2/bin/x64/Rscript.exe" -e "library(knitr); knit('
%.Rnw')"|"C:/Program Files (x86)/MiKTeX 2.9/miktex/bin/pdflatex.exe" -inter
action=nonstopmode %.tex|texs:///view-pdf-internal
```

The user defined build command can be accessed using `Alt + Shift + F1`. What the above commands do is 1) open an `R` instance on the fly and run the `knit()` function on the open `.Rnw` file, creating a `.tex` file in the same folder. Then 2) run `pdflatex` on the created `.tex` file, creating/updating the `.pdf` file, then 3) opening the internal pdf viewer to see the new/updated `.pdf`.

As with `Git`, a section hour is not enough time to really know how to use `knitr`, so you should read up/Google a bit more on what it can do.

Packages

It's likely that most of you will not develop packages in `R` for the purposes of sharing code with others. However, you will almost certainly end up writing utility functions along the way that you want to keep for yourself, and packages are a great way to nicely collate and store this code.

For a good exposition on the basics of writing packages, see:

`http://r-pkgs.had.co.nz/`

I was extremely surprised at how simple package development is using RStudio. Let's see how it works!

Getting Started

First things first, install the `devtools` and `roxygen2` packages:

```
install.packages("devtools")
install.packages("roxygen2")
```

Next, in RStudio, go to `File` → `New Project` and choose "New Directory", then "R Package". Name your package, uncheck "Create a git repository" (you can do this yourself later) and click "Create Project".

My version of RStudio creates a folder named whatever-I-chose-for-the-package-name that contains 1) `DESCRIPTION`, `NAMESPACE`, and `.Rbuildignore` files, and 2) `man`, and `R` folders. The absolute essentials are a folder named `R` that contains some `R` code, a `DESCRIPTION` file, and a `NAMESPACE` file.[1]

The DESCRIPTION file

Open your `DESCRIPTION` file in RStudio by clicking on it in the Files view in the bottom right corner. Fill in `Title` and `Description` with something appropriate.

> The DESCRIPTION file contains important metadata about your package. Who wrote it, what it's for, who can use it, etc.

Authors@R

I like to remove the `Author` and `Maintainer` fields and replace them with the `Authors@R` field which handles both. You can do something like:

```
Authors@R:person("Kendon", "Bell", email = "kmb56@berkeley.edu",
  role = c("aut", "cre"))
```

This says I am both the author (`aut`) and the maintainer (`cre`) of the package.

License

If you ever want to share your code, you should read up on the different licenses at:

`http://r-pkgs.had.co.nz/description.html#license`

Otherwise it doesn't matter.

Important fields that are not included by default in the RStudio default are `Depends`, and `Imports`.

Depends

You should only use `Depends` to specify a version of `R` that is required for your package to work.[2] For example:

```
Depends: R (>= 3.1.2)
```

Imports

Imports is used to load packages that functions in your package will use. For example:

[1] See `http://r-pkgs.had.co.nz/description.html#description` for a detailed discussion of the DESCRIPTION file, and `http://r-pkgs.had.co.nz/namespace.html` for a detailed description of the NAMESPACE file.

[2] See Wickham's website for the few exceptions to this rule.

```
Imports:
  dplyr (>= 0.3.0.1),
  ggplot2 (>= 1.0.0)
```

We won't worry about the `NAMESPACE` file for now.

R Code

Now, let's add an R-function to the package. Create a `.R` file containing just the OLS function that you created last week and save it in the `/R` folder in your package directory.

Now, in RStudio, hit the `Build & Reload` button in the `Build` tab in the top right corner. Ask me questions if you get a error - especially sensitive is the `DESCRIPTION` file. The syntax needs to be perfect. Make sure that, if you have fields that go for more than one line, you indent them with 2 spaces.

Go to the console, type `OLS` (without the '`()`'), and hit `ENTER`. You should now see the function definition as it was written last week. What `R` is doing when it 'builds' your function is simply sourcing (or running) all the `.R` files in the `/R` folder.

Now, open the `NAMESPACE` file. It should read something like `exportPattern("^[[:alpha:]]+")` by default. Remove that line, save the `NAMESPACE` file, and `Build & Reload` again.

Now, you shouldn't see the `OLS` function. What happened??

The export line in the `NAMESPACE` file is telling `R` which functions to load, and you're no longer telling it to load everything using `exportPattern("^[[:alpha:]]+")`.[3] So, it's not really running *all* the `R` code, just what is specified in the `NAMESPACE` file.

Help File Documentation

Package documentation is very simple using the package `roxygen2`. `roxygen2` allows you to ignore most of the dirty details of how documentation works, which is great.

> *Note that "documentation" in coding can refer to both the comments a coder makes throughout her functions, and the help files that are produced for users. We're talking about the latter here.*

[3] We will cover regular expressions near the end probably.

Now, lets document our OLS function:

```
#' @title OLS Coefficients
#' @description Calculates OLS coefficients using matrix methods.
#' @param y \code{numeric} or \code{matrix}. The left hand
#'             side variable of the model to be estimated.
#' @param X \code{matrix}. Contains the right hand side variables of the
#'           model to be estimated. Must include a column of 1's if you
#'           wish to include an intercept.
#' @return A vector of linear regression coefficients.
OLS <- function(y, X) {
        b <- solve(t(X) %*% X) %*% t(X) %*% y
        b
}
```

Above, we can see the basic structure of `roxygen2` documentation where each line begins with '`#'`', and fields are all prefixed by an '`@`' symbol. The only essential field in the above is `@title`. There are also several other fields you may use which you can explore in the `roxygen2` documentation online.

`@title, @description,` and `@details` (omitted here) are differing levels of description of what the function does. These are simply used by typing the field name followed by a space and the content (all 3 can go onto multiple lines). `@details` is usually only used for more complicated functions and will typically contain more than one paragraph.

`@param` documents the function arguments. This is used by typing the field name, followed by a space and the argument name, followed by a space and a description of the argument.

`@return` documents the return value.

It is good practice for both `@param` and `@return` to include the expected classes of the variables involved.

There is a bunch of syntax you can use to style the documentation, such as `\code` above. A full list is available at:

`http://cran.r-project.org/doc/manuals/r-release/R-exts.html#Marking-text`

Let's now get `roxygen2` doing its thing! Make sure the package is installed and call:

```
roxygen2::roxygenize()
```

> *The syntax:* `packageName::functionName()` *allows you to call a function from a package without explicitly loading the function using* `library()`*. Using* `:::` *allows you to access functions that are only available internally to the package.*

Now, build/reload the package again using `Ctrl/Cmd + Shift + B` or clicking the button.

You should be able to see your documentation now by typing:

```
?OLS
```

in the console. You've just made your first `R` package! You can now read in your package in any `R` session using `library(yourPackage)`.

Calculating R^2

First, we'll go over how to calculate the R^2 values. We're going to write the code so that we can retrieve the uncentered R^2_{uc}, the centered R^2, and the adjusted $\bar{R}^2$ with *or* without an intercept in our model. However, computing R^2 values can be trickier than it seems, particularly since many statistical packages (including `R`) are not fully transparent about which R^2 they are displaying. To ground ourselves before we start coding, we'll first go through the intuition behind the R^2 measures, then the math, and finally the code.

Intuition

Some intuition may be helpful here. The uncentered R^2_{uc} is a measure of how much of the variance in the data our model explains relative to a hyperplane[4] where $y = 0$. The centered R^2 is a measure of how much of the variance in our model we can explain relative to a hyperplane where $y = \bar{y}$. The adjusted $\bar{R}^2$ is the R^2 with a penalty applied for adding more covariates.

Math

Now the math. First, the uncentered R^2_{uc}:

$$R^2_{uc} \equiv \frac{\hat{\boldsymbol{y}}'\hat{\boldsymbol{y}}}{\boldsymbol{y}'\boldsymbol{y}} = 1 - \frac{\boldsymbol{e}'\boldsymbol{e}}{\boldsymbol{y}'\boldsymbol{y}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - 0)^2} = 1 - \frac{\text{SSR}}{\text{SST}_0} \tag{1}$$

The above is perfectly general. The first equality comes from $\boldsymbol{y}'\boldsymbol{y} = \hat{\boldsymbol{y}}'\hat{\boldsymbol{y}} + \boldsymbol{e}'\boldsymbol{e}$, the second is matrix decomposition, and the third is just my favorite way to think about this measure. SSR is the sum of squared residuals, and SST_0 is the total sum of squares *relative to zero*. The latter is new notation to you, so be careful: it is NOT the SST that Max refers to in his notes — we'll use that soon, when we define the centered R^2. Which we'll do now.

The centered R^2 looks very similar to the uncentered R^2_{uc}. The only difference is the denominator:

$$R^2 \equiv 1 - \frac{\boldsymbol{e}'\boldsymbol{e}}{\boldsymbol{y}^{*\prime}\boldsymbol{y}^*} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} = 1 - \frac{\text{SSR}}{\text{SST}} \tag{2}$$

Remember that $\boldsymbol{y}^* = \boldsymbol{A}\boldsymbol{y}$, the demeaned version of $\boldsymbol{y}$. I diverge a bit from the lecture notes here. My definition is a bit more general, since it will allow us to calculate the R^2 for models without an intercept[5]. Here, the SST is as it was in the lecture notes — it is sum of the squared differences between all of the values of y and the mean of y.

The adjusted $\bar{R}^2$ is just the R^2 with a penalty applied for additional covariates:

$$\bar{R}^2 \equiv 1 - \frac{n-1}{n-k}(1 - R^2) \tag{3}$$

What happens when $k = 1$?

[4] When you see "hyperplane," think "line."

[5] If you're interested in exploring why the definition in the lecture notes won't work for models without an intercept, see Greene 3.5.2.

```
demeanMat <- function(n) {
        ones <- rep(1, n)
        diag(n) - (1/n) * ones %*% t(ones)
}
```

Now, the main event! We'll define a function that runs OLS, computes all of our R^2 values, and returns them in a matrix. There is a lot of code here to process, but if you refer to the math above you'll see that it all matches up.

```
        R.squared <- function(y,X) {
                n <- nrow(X)
                k <- ncol(X)
                b <- OLS(y,X)
                yh <- X %*% b
                e <- y - yh # yh is y hat, the predicted value for y

                SSR <- t(e) %*% e
                SST.0 <- t(y) %*% y # == sum(y^2)
                R2.unc <- 1 - SSR / SST.0

                A <- demeanMat(n)
                ys <- A %*% y # this is ystar
                SST.yb <- t(ys) %*% ys # ==  sum((y - mean(y)^2))
                R2.cen <- 1 - SSR / SST.yb

                R2.adj <- 1 - ((n-1)/(n-k))*(1-R2.cen)

                return(cbind(R2.unc,R2.cen,R2.adj))
        }
```

We'll test our function by calling it with `X`, our data matrix that contains an intercept and `mpg`. We'll also confirm our results with `lm()`.

```
(Rsq.X2 <- R.squared(y,X))

## Error in nrow(X): object 'X' not found

summary(lm(y ~ X))$r.squared

## Error in eval(expr, envir, enclos):  object 'y' not found

summary(lm(y ~ X))$adj.r.squared

## Error in eval(expr, envir, enclos):  object 'y' not found
```

`lm()` does not, as far as I know, have a method for returning the uncentered R^2 for this model. Now let's try with `X2`, the data matrix with `mpg` but *no intercept*.

```
(Rsq.X2 <- R.squared(y,X2))

## Error in nrow(X): object 'X2' not found

summary(lm(y ~ 0 + X2))$r.squared
```

```
## Error in eval(expr, envir, enclos):  object 'y' not found
summary(lm(y ~ 0 + X2))$adj.r.squared
## Error in eval(expr, envir, enclos):  object 'y' not found
```

Our results for R^2 and $\bar{R}^2$ are totally insane, but for some reason our R^2_{uc} matches the output from `lm()`. Does this make any sense? Actually, yes! Remember that the centered R^2 compares our model, $\boldsymbol{y} = \boldsymbol{X}\beta + \varepsilon$ (with no intercept) to $\boldsymbol{y} = \bar{\mathbf{y}}$. Since $\boldsymbol{y} = \bar{\boldsymbol{y}}$ is actually a much better predictor of $\boldsymbol{y}$ than $\boldsymbol{Xb}$, we get that SSR > SST. Using that $R^2 = 1 - \frac{\text{SST}}{\text{SST}}$, we can see why we get a negative R^2 and $\bar{R}^2$.

But, our results don't match the results from `lm()`. This is because `lm()` is actually pulling a bait-and-switch here: since we aren't passing it an intercept, it computes the uncentered R^2_{uc} instead of the centered R^2.[6] So rather than comparing our model $\boldsymbol{y} = \boldsymbol{Xb} + \varepsilon$ to $\boldsymbol{y} = \bar{\mathbf{y}} + \varepsilon$, it compares it to $\boldsymbol{y} = \mathbf{0} + \varepsilon$. If this doesn't make any sense to you, don't worry about it too much. Just remember to include an intercept in your model and everything will be fine.

Linear algebra puzzles

1. Define vectors $\boldsymbol{x} = [1 \;\; 2 \;\; 3]'$, $\boldsymbol{y} = [2 \;\; 3 \;\; 4]'$, and $\boldsymbol{z} = [3 \;\; 5 \;\; 7]$. Define $\boldsymbol{W} = [\boldsymbol{x} \;\; \boldsymbol{y} \;\; \boldsymbol{z}]$. Calculate $\boldsymbol{W}^{-1}$. If you cannot take the inverse, explain why not and adjust $\boldsymbol{W}$ so that you /can/ take the inverse. *Hint*: the `solve()` function will return the inverse of the supplied matrices.
2. Show, somehow, that $(\boldsymbol{X}p)^{-1} = (\boldsymbol{X}^{-1})'$.
3. Generate a 3×3 matrix $\boldsymbol{X}$, where each element is drawn from a standard normal distribution. Let $\boldsymbol{A} = \boldsymbol{I}_3 - \frac{1}{3}\boldsymbol{B}$ be a demeaning matrix, with $\boldsymbol{i}$ a 3×3 matrix of ones. First show that $\boldsymbol{A}$ is idempotent and symmetric. Next show that each row of the matrix $\boldsymbol{XA}$ is the deviation of each row in $\boldsymbol{X}$ from its mean. Finally, show that $(\boldsymbol{XA})(\boldsymbol{XA})' = \boldsymbol{XAX}p$, first through algebra and then `R` code.
4. Demonstrate from random matrices that $(\boldsymbol{XYZ})^{-1} = \boldsymbol{Z}^{-1}\boldsymbol{Y}^{-1}\boldsymbol{X}^{-1}$.
5. Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be square 20×20 matrices. Show that $tr(\boldsymbol{X} + \boldsymbol{Y}) = tr(\boldsymbol{X}) + tr(\boldsymbol{Y})$.
6. Generate a diagonal matrix $\boldsymbol{X}$, where each element on the diagnonal is drawn from $U[10, 20]$. Now generate a matrix $\boldsymbol{B}$ s.t. $\boldsymbol{X} = \boldsymbol{BB}p$. *Hint*: There is a method in `R` that makes this easy. Does the fact that you can generate $\boldsymbol{B}$ tell you anything about $\boldsymbol{X}$?
7. Demonstrate that for any scalar c and any square matrix $\boldsymbol{X}$ of dimension n that $\det(c\boldsymbol{X}) = c^n \det(\boldsymbol{X})$.
8. Demonstrate that for an $m \times m$ matrix $\boldsymbol{A}$ and a $p \times p$ matrix $\boldsymbol{B}$ that $\det(\boldsymbol{A} \otimes \boldsymbol{B}) = \det(\boldsymbol{A})^p \det(\boldsymbol{B})^m$. /Hint/: Note that $\otimes$ indicates the Kronecker product[7]. Google the appropriate =R= function.

[6] For more information, see here: http://bit.ly/1c0nA6N.

[7] The Kronecker product is a useful mathemagical tool for econometricians, allowing us to more easily describe block-diagonal matricees for use in panel data settings. I wouldn't lose sleep over it, though.

Here's a nice problem set hint for those of you who actually read these! Check out the `WDI` package in `R`; it should make downloading the World Bank data much simpler. This is a canned package which is definitely OK to use for the problem set.