RStudio Tutorial

Samuel DeWitt, Ph.D.

RStudio - What is it?

RStudio is a UI overlay for the base R package, which is an open-source statistical program.

R is quickly becoming a *very* popular programming language due to its:

1) Versatility (lots **can** be done in R - does *not* necessarily mean you should, though)
2) It's free!
3) That's basically it - #2 is pretty big for government agencies (stats programs like SPSS, SAS, or Stata are hard on budgets)

RStudio - What is it?

In turn, RStudio is a popular overlay because:

1) It makes R look more like other popular stats programs
2) It allows for many functions to be completed through point-and-click (I highly advise against this!)

How to Install R and & RStudio on your Home PC

First step - install R

1) Navigate to https://archive.linux.duke.edu/cran/
2) Select the R download for your OS (Linux, Mac, or Windows)
3) Download and run the installer (default install options are appropriate for most users)

Second step - install RStudio

1) Navigate to https://www.rstudio.com/products/rstudio/download/#download
2) Select the RStudio download for your OS (Linux, Mac, or Windows)
3) Download and run the installer (default install options are appropriate for most users)

Third step - there isn't one. Just open the program (RStudio) now.

A Quick Reminder on File Paths in R

Remember that the backslash (“\”) is an *escape* character in R - you’ll need to escape each backslash in the file path if you are using Windows.

So, the following file path: “C:\Users\yourid\etc. . . ” will **not** work and you’ll get an error.

Instead, use the following path: “C:\\Users\\yourid\\etc. . . ” and you will avoid that error.

Alternatively, you could also replace all backslashes (\) in the file path with forward slashes (/) - up to you.

Exploring Some Common Functions

We will begin by exploring a few common functions you will need to use throughout the semester.

1) The help functions: ? and ??
2) Installing new packages: install.packages()
3) Loading packages: library()
4) Setting a working directory: setwd()
5) Summarizing an object: summary()
6) The assignment operator: <-
7) Creating a new data frame: data.frame()
8) An advanced plot command suite: ggplot2
9) The dplyr package: Variable transformation

The Help Functions - ? and ??

In addition to Googling issues (trust me, you will), R has built in help features.

A single ? attached to a function name will bring up the PDF file for that function in the Help page (bottom right window in R Studio).

This is interchangeable with the help() function, also.

The Help Functions - ? and ??

So, to get help with the install.packages() function, you could type either of the following:

1) ?install.packages
2) help(install.packages)

The Help Functions - ? and ??

But, what if you don't know the function names?

In this case, you must search the help files for what you **think** the function name might be.

The Help Functions - ? and ??

Using the same function as an example, I need to know how to install a package, but can't remember the full name of the function:

1) ??install
2) help.search("install")

Note the quotation marks around the term - this means that R will search its help files for that specific string of characters.

Note: I have had a *lot* of difficulty getting this function to work properly and I'm not sure why. If you don't know the name of the function, just use Google.

Installing New Packages

The base R package includes all the functions you will need to conduct most analyses.

However, there are user-built functions which expand upon base R.

Many make coding simpler, others allow for statistical analysis not (yet) available in base R.

One of the most popular packages is ggplot2 - an alternative to the visual plotting available in base R.

Installing New Packages

Installing a new package is remarkably simple.

And, once you've installed it, you don't have to again.

You will need to keep it updated, though - as with any computer program, R packages are not without bugs the developers fix later on.

Installing New Packages

I'll demonstrate how to install a new package with the ggplot2 addon we will be going over in more detail later.

```
#install.packages("ggplot2",repos='https://cloud.r-project.org')
```

Installing New Packages

And that’s it!

You will typically not have to specify the ‘repos’ option, I only had to do so because I was installing through Markdown.

Always be sure to put single or double quotation marks around package names - R will not read them if you don’t.

Loading Packages

When you open R only the base package will be *loaded*.

This means that any addon packages need to be loaded manually, otherwise its functions will **not** work.

Loading packages is also very simple.

Loading Packages

Here's how you would load the *ggplot2* package to be able to use it:

```
library(ggplot2)
```

Loading Packages

Notice how you do **not** have to include quotation marks around the package name anymore.

This is because R now knows the name of the package since you have already installed it.

Setting a Working Directory

Some advise against this, and to instead use the project feature of R to just set a project directory where all files are located.

I don't do this as a personal preference.

Mostly because I primarily use Stata for data analysis and cleaning and I never learned how to use the project feature in that, either.

Setting a Working Directory

Setting a working directory is important because it determines part of the file path R uses to find files used in your program.

For example, suppose I want to load a data set into R, I could:

A) Specify the entire file path or,
B) Specify a partial path or just the name of the file

You can only go with Option B if you set a working directory.

Setting a Working Directory

Luckily, this is also simple, with one important caveat.

For Windows users file paths contain a character that R uses to *escape* functions.

You will need to use the methods I detail above to avoid this issue.

Setting a Working Directory

Here's how to use the 'setwd()' function:

```
##setwd('C:\\Users\\sd662\\Desktop')
```

Setting a Working Directory

You'll notice that nothing will appear in the console window to confirm the directory has changes.

That's where the getwd() function comes in.

This function allows you to print out the current working directory to the console window.

Setting a Working Directory

```
#getwd()
```

Setting a Working Directory

Now you will see in the console window the directory you are currently in.

Note: In practice, I strongly advise against setting your Desktop as the working directory.

Instead, create a folder specifically for this class.

Setting a Working Directory - Redux

Okay, so Macs have weird issues with setting a working directory that I'm not capable of debugging (never used a Mac).

Here's an easy solution - use the dropdown menu (yeah, just this one time...)

Session -> Set a Working Directory -> Choose Directory

Then navigate to the directory where your data is located.

The actual command to do this should then be in your console pane - copy and paste that into your script file.

Loading Data Into R - Opening R Data

We're actually going to combine some lessons here. Loading data into R is simple, but referencing file paths can be a chore.

So, we are going to use the setwd() function and the load() package to accomplish this task (but I will show you an alternative just using the latter).

Loading Data into R - Opening R Data

One way to make this process easier on yourself is to put the R data in a particular folder, then set your working directory to that folder.

This way, you only have to type the name of the data set, not the entire file path leading to it.

Loading Data Into R - Opening R Data

Example #1 - using setwd() before load()

```
##setwd('C:\\Users\\sd662\\Desktop')
##load('test.Rdata')
```

Loading Data Into R - Opening R Data

Example #2 - without using setwd() before load()

```
##load('C:\\Users\\sd662\\Desktop\\test.Rdata')
```

Loading Data Into R - Opening R Data

Either option will work, but I suggest using the first for simplicity.

You could always also use the drop-down menus, but doing so means you have to load data each time you open R.

Since that is pretty tedious, I suggest loading data sets using the coding language instead of the point-and-click interface.

The Assignment Operator: <-

The assignment operator is a special character set in R.

To use the assignment operator, you specify a name for an object on its left, and the function creating the object on its right.

Think of this as you telling R to store the result of the function as something you can use later on.

The Assignment Operator: <-

Here's an example - I can create a random dummy variable using the rbinom() function. I will show you what happens when I do or do not use the assignment operator.

```
rbinom(10,1,0.50)
```

```
##  [1] 1 0 1 0 1 0 0 0 1 0
```

The Assignment Operator: <-

If you copy and paste that function you will see similar output, but just in the console window.

Further, nothing will show up in the global environment (top right window in RStudio).

That means you will not be able to reference that exact result again later on in your syntax.

The Assignment Operator: <-

Let’s now use the assignment operator and see what happens.

```
dummy<-rbinom(10,1,0.50)
dummy
```

```
##  [1] 0 1 0 0 1 1 0 0 1 0
```

The Assignment Operator: <-

Notice that I needed two lines of code to display the results?

The first line **assigns** the values from the function to an **object** I decided to call ‘dummy’

Notice also that if you run that first line of code by itself all that happens is an object named ‘dummy’ now appears in the global environment.

This means that you can now use this result in later functions, since it is now *stored* as an **object**.

The Assignment Operator: <-

You can use the assignment operator to store literally anything as an **object** in R.

This includes:

1) Results from a multivariate regression (or any other model results, for that matter)
2) Individual column vectors
3) Lists of names or numbers
4) Data visualizations - plots can be stored and amended at will.

This is why R is called an **object-oriented** programming language!

Summarizing a Vector or Data Frame - The summary() Function

The summary() function provides basic descriptive statistics about a vector or data frame.

It's often helpful, especially after creating a variable, to verify that you created it correctly.

For the purposes of this example, we will use the 'mtcars' data set - data from a 1974 Motor Trend magazine concerning the design and performance of 32 popular cars (at that time, at least).

There's no need to download these data - they come pre-packaged with base R.

Summarizing a Vector or Data Frame - The summary() Function

```
data(mtcars)
summary(mtcars)
```

```
##       mpg             cyl             disp             hp
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
##       drat             wt             qsec             vs
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
##        am              gear            carb
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
##  Median :0.0000   Median :4.000   Median :2.000
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Summarizing a Vector or Data Frame - The summary() Function

That's a lot of output!

Probably too much to look at all at once.

Perhaps you just want to look at the summary statistics for one column in the *mtcars* data.

To do that, you need to reference that specific column somehow in the summary() function.

Summarizing a Vector or Data Frame - The summary() Function

Here's how to do just that:

```
summary(mtcars$mpg)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   10.40   15.43   19.20   20.09   22.80   33.90 
```

That's a lot less output to sort through.

Can anyone guess at what each of these statistics means?

Summarizing a Vector or Data Frame - The summary() Function

The $ sign after the data set name tells R to look within that data set for a column named whatever you type after the $. In this case, we summarized the ‘mpg’ column.

Keep this structure in mind - you’ll be using it frequently.

Summarizing a Vector or Data Frame - The summary() Function

You can also get some additional information about this data set by typing in ?mtcars into the console window.

This will bring up a description file in the Help window.

Creating a Data Frame: the data.frame() function

Data frames are a specific type of data storage in R and are unique for their ability to accommodate different types of variables.

As an example, a matrix (another storage format) can only accept *one* type of data. All the columns (variables) must be the same type of data (numeric, string, etc. . .).

Therefore, if you create a matrix using a numeric variable, all other columns must also be numeric.

Creating a Data Frame: the data.frame() function

A data frame object does not have this strict restriction.

Although much of the data we will use this semester is numeric, using data frames allows us to avoid any issues with variables (columns) being measured in different types of storage units.

Creating a Data Frame: the data.frame() function

I am going to create a few different variables and store them as column vectors.

A column vector is simply a single column containing one type of data.

I will then collect these separate columns into a single data frame using the data.frame() function.

Here goes.

Creating a Data Frame: the data.frame() function

```r
d1<-rbinom(100,1,.25) ## this creates a dummy (0/1) variable)
a1<-rnorm(100,mean=16,sd=1) ## this creates a continuous variable
c1<-rpois(100,1) ## this creates a discrete Poisson variable
t1<-sample(c("Freshman","Sophomore","Junior","Senior"),100,replace=TRUE)
## This creates a character variable
example_data<-data.frame(d1,a1,c1,t1)
```

Creating a Data Frame: the data.frame() function

This will create a data frame object (called ‘example_data’) stored in your global environment.

You can then describe this data frame by using the summary() function as so:

```
summary(example_data)
```

```
##        d1             a1              c1            t1
##  Min.   :0.00   Min.   :14.12   Min.   :0.00   Length:100
##  1st Qu.:0.00   1st Qu.:15.42   1st Qu.:0.00   Class :character
##  Median :0.00   Median :16.02   Median :1.00   Mode  :character
##  Mean   :0.29   Mean   :16.04   Mean   :0.92
##  3rd Qu.:1.00   3rd Qu.:16.64   3rd Qu.:2.00
##  Max.   :1.00   Max.   :18.59   Max.   :3.00
```

Creating a Data Frame: the data.frame() function

You can also click on the ‘example_data’ data object in the global environment.

This will bring up a new window in the script area which lists all row and column values in the data frame.

An Advanced Plotting Function: ggplot2

ggplot2 is a suite of data visualization commands otherwise referred to as the ‘Grammar of Graphics’

It offers a variety of plotting commands for different types of variable combinations.

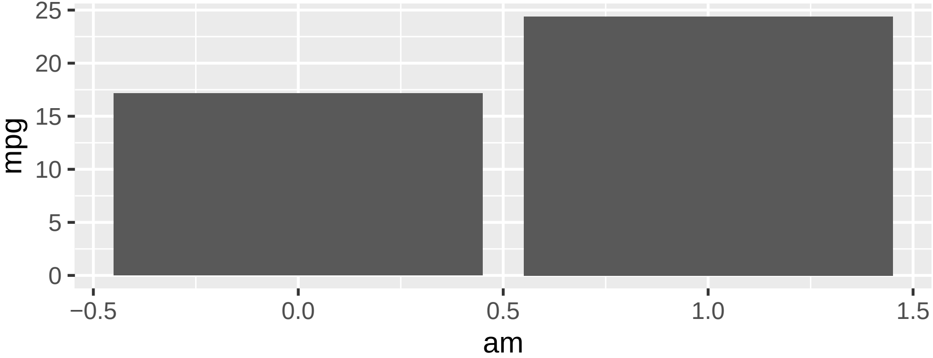
Some of these commands will automatically select the right plot for you (e.g., qplot()), while others will require that you specify the proper visualization technique for a specific plot of two or more variables (e.g., geom_bar()).

An Advanced Plotting Function: ggplot2

What follows is a few examples of plotting using ggplot2 and the mtcars data set. I'll begin with a simple bar plot then move forward to more complicated examples.

An Advanced Plotting Function: ggplot2

```
ggplot(mtcars, aes(x=am, y=mpg)) +
  geom_bar(position='dodge', stat='summary', fun.y='mean')
```



An Advanced Plotting Function: ggplot2

The last plot was a simple bar graph plotting miles per gallon (mpg) by transmission type (am).

Automatic transmissions are represented by a ‘0’ in the data and manual transmissions by a ‘1’

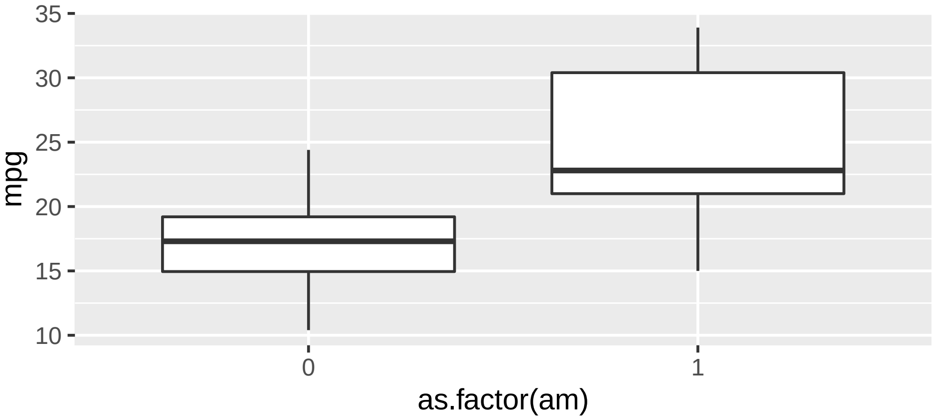
From the last figure, which type of transmission appears to have better fuel efficiency, on average?

An Advanced Plotting Function: ggplot2

Let's get a little more complicated - a boxplot. A boxplot tells you a lot more about the distribution of a continuous variable across different categories. Let's redo the last plot but using the geom_boxplot() command.

An Advanced Plotting Function: ggplot2

```
ggplot(mtcars, aes(y=mpg, x=as.factor(am))) + geom_boxplot()
```



An Advanced Plotting Function: ggplot2

It's still clear that manual transmissions have better fuel economy, but it does look like some cars with automatic transmissions can have similar fuel efficiency.

In either of the above cases, we chose plotting techniques suited to the combination of variables - Y was continuous (fuel efficiency) and X was a categorical (nominal) variable. This would also work well with an ordinal variable on the X-axis (but definitely not the Y-axis!).

An Advanced Plotting Function: ggplot2

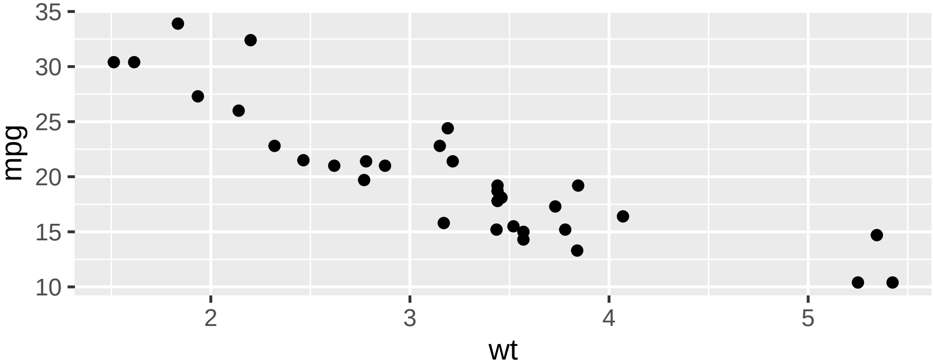
Let's try some more advanced plotting techniques using two continuous variables. We will keep fuel efficiency in each, but substitute transmission type for other continuous variables in the data.

When you have two continuous variables, the most appropriate plot type is a scatterplot.

This type of plot merely places points at the places in which the data on the X- and Y-axis meet. You can then add 'best fit' lines to examine the plot for possible trends.

An Advanced Plotting Function: ggplot2

```
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point()
```



An Advanced Plotting Function: ggplot2

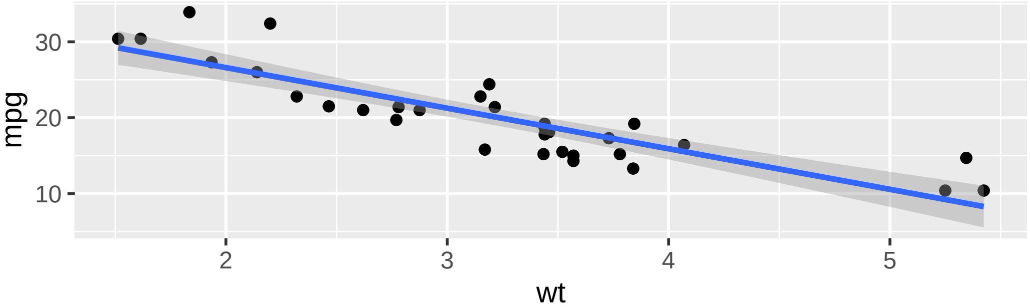
Without fitting a trend line, what do you think the relationship is between the weight of a vehicle and its miles per gallon?

Positive, negative, or no relationship?

An Advanced Plotting Function: ggplot2

Now let's fit a trend line to the original scatterplot.

```
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point() +
  stat_smooth(method=lm, se=TRUE)
```



An Advanced Plotting Function: ggplot2

Yup, definitely a negative relationship!

For clarity, I simply added an additional feature to the plot using the stat_smooth() function.

Within this function, I specified that the ‘smoothing’ technique be ‘lm’ which stands for **linear model**.

In essence, I simply added a slope to the plot - you would obtain the same plot in a bivariate regression!

Finally, the ‘se=TRUE’ statement told R to include standard error bars around the line - this just tells us where the true population slope parameter is likely to fall.

The dplyr Package: Variable Transformation

The dplyr package is a suite of functions that make variable transformation pretty painless.

The functions within the package are the combination of several base functions, but is a bit more novice-friendly.

We will review the following packages here:

1) filter() and arrange()
2) mutate() and transmute()
3) select() and rename()
4) summarise()

The dplyr Package: filter() and arrange()

The filter() and arrange() functions perform related tasks.

filter() allows you to select cases from an object based upon their values.

arrange() allows you to reorder cases, similar to sorting an Excel sheet by one or multiple columns.

Let's see both in action using the NLSY data we downloaded earlier.

The dplyr Package: filter() and arrange()

```
NLSY97 %>%
  filter(sex==1 & employed==1 & id<51) %>%
  select(id, sex, employed, peers_clubs, peers_gang, peers_college)
```

```
##   id sex employed peers_clubs peers_gang peers_college
## 1 12   1        1           4          3             3
## 2 18   1        1           5          4             1
## 3 20   1        1           4          1             3
## 4 31   1        1           3          1             4
## 5 39   1        1           5          1             5
```

A few new things here, let's review them!

The dplyr Package: filter() and arrange()

The %>% syntax is a shorthand character in R that we can use with the dplyr package (and other functions).

It basically tells R to use the result of the current function as the subject of the next function.

The dplyr Package: filter() and arrange()

So, in this example:

1) The **pipes** operator (%>%) tells R to use the result of the first line (calling the NLSY97 data frame) as the object in the following line.
2) filter(sex==1 & employed==1 & id<51) tells R to only keep observations where sex is 1, employed is 1, and id is less than 51 for the following lines of code.
3) select(id, sex, employed,. . . .) tells R to only display a summary of the variables included within the parentheses.

The dplyr Package: filter() and arrange()

```
NLSY97 %>%
  arrange(use_drugs) %>%
  select(id, sex, use_drugs) %>%
  head(5)
```

```
##   id sex use_drugs
## 1  1   2         0
## 2  2   1         0
## 3  3   2         0
## 4  4   2         0
## 5  5   1         0
```

The dplyr Package: filter() and arrange()

The example is a little different, but follows a similar path.

arrange() tells R to **sort** a variable in the NLSY97 data frame.

select() tells R to only show the *id*, *sex*, and *use_drugs* variables.

And head(5) tells R only to print out the first 5 rows of the data (otherwise it's too much to fit on a slide)

The dplyr Package: filter() and arrange()

The basic difference is that:

1) filter() **selects out** rows that do not satisfy the inclusion criteria (i.e., **sex==1**)
2) arrange() simply **sorts** the variables you tell it to

I primarily use the select() function here to limit the amount of data printed to the slide. Otherwise it would print *every* variable in the data frame and cause an error in the Markdown compiler.

The dplyr Package: mutate() and transmute()

The mutate() and transmute() functions perform similar operations with one key difference.

mutate() will allow you to create a new variable while maintaining the pre-existing ones

transmute() will allow you to create a new variable, but will erase any variables you use to create the new ones

The following are examples of both in action.

The dplyr Package: mutate() and transmute()

```
NLSY97 %>%
  select(peers_drugs, peers_gang, peers_cut) %>%
  mutate(badpeers=peers_drugs+peers_gang+peers_cut) %>%
  summary(badpeers)
```

```
##   peers_drugs      peers_gang      peers_cut        badpeers
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   : 3.000
##  1st Qu.:1.000   1st Qu.:1.000   1st Qu.:1.000   1st Qu.: 4.000
##  Median :2.000   Median :1.000   Median :2.000   Median : 6.000
##  Mean   :2.307   Mean   :1.595   Mean   :2.408   Mean   : 6.321
##  3rd Qu.:3.000   3rd Qu.:2.000   3rd Qu.:3.000   3rd Qu.: 8.000
##  Max.   :5.000   Max.   :5.000   Max.   :5.000   Max.   :15.000
##  NA’s   :226     NA’s   :172     NA’s   :64      NA’s   :315
```

The dplyr Package: mutate() and transmute()

You can see that we have a new variable - **badpeers** - which is the simple addition of three other peer variables.

The new variable and the pre-existing ones are both now in the data.

I will **NOT** demonstrate the same procedure using transmute() because the code is the same…however….

transmute() will **erase** the variables you use in the function. It would erase *peers_drugs*, *peers_gang*, and *peers_cut* and I don't want to do that (you almost never will, either).

The dplyr Package: select() and rename()

We have already seen select() in action above, now I will introduce rename()

rename() allows you to change the name of columns in the data.

This is largely helpful when you download new data and want better variable names (I have already changed the names of the NLSY97 variables for this example - they were mostly gibberish before).

The dplyr Package: select() and rename()

First, I want to summarize the existing variable.

```
NLSY97 %>%
  select(employed) %>%
  summary(employed)
```

```
##     employed
##  Min.   :0.0000
##  1st Qu.:0.0000
##  Median :0.0000
##  Mean   :0.4962
##  3rd Qu.:1.0000
##  Max.   :1.0000
##  NA's   :623
```

The dplyr Package: select() and rename()

```
NLSY97 %>%
  rename(emp_stat=employed) %>%
  select(emp_stat) %>%
  summary(emp_stat)
```

```
##     emp_stat
##  Min.   :0.0000
##  1st Qu.:0.0000
##  Median :0.0000
##  Mean   :0.4962
##  3rd Qu.:1.0000
##  Max.   :1.0000
##  NA's   :623
```

The dplyr Package: select() and rename()

If you compare the summaries you can confirm that the contents of the variable are the same - we have merely changed its name.

The dplyr Package: summarise()

The summarise command allows you to produce summaries of variables and then store them in your data or display them in the console window.

This can be helpful if we need a variable representing the mean for a certain variable to use in the calculation for a new variable.

That is, it is often helpful to *de-mean* a variable by subtracting scores from the group mean and then to use this *de-meaned* variable in statistical analysis.

A score of 0 on this new variable then means the individual is *average* for your sample.

The dplyr Package: summarise()

```
NLSY97 %>%
  mutate(age=int_year-birth_year) %>%
  summarise(avg_age=mean(age, na.rm=TRUE))
```

```
##    avg_age
## 1 15.03417
```

This tells us that the average age in this sample is about 15 years old, without accounting for birth and interview month.

The dplyr package: Some final words

The dplyr package makes R code a bit more palatable to people trained in other programming languages.

It also has function names that are more intuitively named (i.e., easier to figure out what they do from the function name).

The **pipes** operator can be *very* helpful in reducing clutter in your code. An easier way to type the operator is to hit Ctrl+Shift+M (a macro for the regular pipes operator).

Some final words on R

R is a *very* versatile coding language - this is a bit of a double-edged blade though.

Because you can do so much, in so many different ways:

1) Solutions to problems may be more varied - not immediately clear which is best!
2) You have more opportunities to screw things up (this is why it's good to keep good records of your code).
3) You also have more opportunities to get things right!
4) All sorts of new and interesting methods are being added to R regularly.

As a corollary of #4, advanced methodological (or data visualization) techniques are available in R generally much quicker than they make their way to other statistical programs.

The End

Happy programming to you!