Assignment 1

J.M.T. Roos

Last updated: 2017-01-21 11:38:04

Due: Friday, February 3, 2017 at 9.00

In this assignment, you will download, install, and use R, an open-source statistical programming and graphics program. We will be using R throughout the course, and the more comfortable you are with this software, the easier this course will be for you.

The remainder of this assignment requires you to read this document, typing commands into R and verifying the results match what you see on the printed page as you go. Why am I asking you to do type as you read? Because typing commands into R and observing the output makes it much easier to learn how R works, compared to only reading this document. Hence, typing commands into R is a requirement of this assignment.

Note. I will say this many times, so get used to hearing it: Typing is not the same as copying and pasting! You will learn to program in R considerably faster if you type!

Install and Setup R and RStudio

There are four concepts to keep straight when reading what follows.

R Language R is the name of a programming language.

R Interpreter R is also the name of a program that runs on Mac, Windows, and Linux, that provides the "engine" for understanding and then executing instructions written in the R Language.

R Console The R Console is the name of the interactive, command-line, and somewhat human-friendly interface provided by R (i.e., issuing instructions written in the R Language and observing the resulting output).

RStudio RStudio is a complete development environment for writing and executing R code (plus more).

To get R running on your computer, there is one required and another optional but strongly recommended step:

1. **Install R** (required). You can download and install R from this location:

   `https://cran.r-project.org`

I suggest using the appropriate "installer" for your computer type. The installer will set up the core R interpreter (i.e., console), as well as a very primitive graphical user interface (GUI). This GUI application includes a window with an R Console and a simple text editor. It is primitive, and rather frustrating to use. For this reason, I strongly recommend you follow the next step and install RStudio.

2. **Install RStudio** (recommended). You can download and install RStudio from this location:

   `https://www.rstudio.com/products/rstudio/download/#download`

   RStudio provides an "integrated development environment" (IDE) for R. It provides a small window panel with a running R Console, another panel for editing documents (e.g. code scripts), another for viewing graphical output (e.g. plots). **Highly recommended.**

If you cannot install R or RStudio on your computer, you can use the lab computers instead.

Running R or RStudio for the first time

After you have installed R (and optionally RStudio), open either application. You will see a running instance of the R Console containing startup text similar to the following:

```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin14.5.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

The "greater than sign" (>) is a "prompt" for the interactive R Console. The prompt indicates that the R interpreter is ready for you to tell it what to do. We will return to this momentarily—first, we need to install additional libraries which provide all the extra functionality that has made R such a popular tool.

Installing Packages

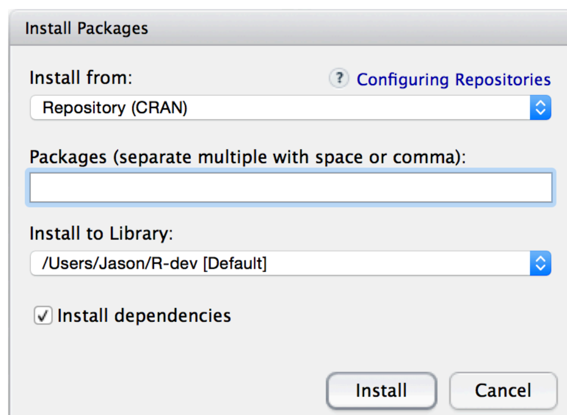
R has been designed so that users can extend its capabilities via "packages". Packages are bundles of functions and sometimes data as well which facilitate certain types of analysis. Some packages

(for example the "stat" package, providing many statistical functions) are included in the "core" version of R and are installed by default. Others (for example the "ggplot2" package, providing a rich interface for creating data visualizations) must be downloaded and installed by the user. In what follows, you will install the `tidyverse`, a collection of related packages used throughout this course.

It is recommended that you install packages using RStudio, but instructions for the basic GUI's for the OS X and Windows platforms are included here as well.

RStudio

Using the mouse, click on the menu named `Tools`, then select the item called `Install Packages...`.



In RStudio, you will see a window prompting you to select a "mirror" (a download location), a list of packages to install, and a location to install them. Use the default settings for the mirror and install location. In the text box, type the following package name: `tidyverse`. Be sure the "Install dependencies" check box is selected, then click the Install button.

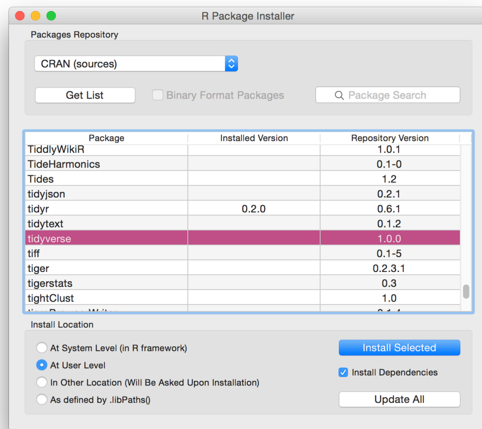
R on Windows

Using the mouse, click on the menu named `Packages` then select the item called `Install Package(s)...`. A window should open showing of repositories (locations on the web where you can download R packages from). Choose the default, which might be labeled `0-Cloud`, and click OK.

Next, you will see a list of packages. Scroll to the item called `tidyverse`, highlight it, and click OK. At some point, you will be asked whether you want to use or create a "personal library". I recommend clicking yes. If you are asked whether to install dependencies as well, you should do so.

R on OS X

Using the mouse, click on the menu named `Packages & Data`, then select the item called `Package Installer`.



After selecting `Package Installer` you will see a window with a list of repositories (use the default) and a button labeled `Get List`. Click on this button, then scroll until you see the item called `tidyverse`. Highlight this item. Make sure the `Install dependencies` box is checked. You should also probably click the radio button next to `At User Level` as well. Finally, click `Install Selected`.

Using the R console

A quicker method for installing packages (if you know the exact spelling for the package you want) is to enter the following into the R Console and hit Enter:

```
> install.packages('tidyverse')
```

Note that the > character above shows where the prompt is—you should not type > here. That is, you should type `install.packages('tidyverse')` and not `> install.packages('tidyverse')`.

Important note for users of the lab computers

When installing packages, always create a "personal library." On the lab computers, you might need to create this folder manually (once) before installing new packages.

To do so, navigate in Windows to `Library > Documents` and create a folder under `Documents` called `R`. Open the new folder called `R`. Inside that folder, create another folder called `win-library`. Open the newly created `win-library` folder, and inside that one, create a new folder called `3.3` (or perhaps `3.1` or `3.2` depending on the version of R that is currently installed in the lab). If you don't know the version of R that is running, you can find it by entering the following into the R Console and hit Enter.

```
> version$version.string
```

```
[1] "R version 3.3.2 (2016-10-31)"
```

(Again, only type what follows the > prompt.) The folder gets its name from the first two numbers of the version string. In the example above, I would create a folder named 3.3.

Did it work?

If there are any errors, try your luck googling them. This is an important skill to develop when learning to program in a new language. If you are still stuck, ask a classmate for help. If you are *still* stuck, email me (roos@rsm.nl)!

If everything worked correctly, you should see a bunch of text in the R Console that ends with something like this:

```
.
.
.
* DONE (tidyverse)

The downloaded source packages are in
    '/private/var/folders/d8.../downloaded_packages'
```

You are now ready to use R!

Your First Commands

As noted above, the R Console provides an interface for interacting with the R Interpreter. The > with the blinking vertical bar next to it indicates R is ready for you to type something. Click to the right of the >, type `3 + 2`, and then hit Enter:

```
> 3 + 2
```

```
[1] 5
```

The above text should look like what you see in the R Console. First there is the prompt (>), then the text you typed (`3 + 2`). Then, on the next line, you see `[1] 5`. The `[1]` is meaningless in this context and will be explained later; ignore it for now. Finally, 5 is the sum of 3 and 2. By typing `3 + 2` and hitting Enter, you have issued the following commands to R: "add 3 to 2, then show me the result."

Completing this assignment

To complete this assignment, you must read it in its entirety. And as you read, you should *type* (do not copy and paste!) all of the R code into the R Console and hit Enter. I'm repeating myself here, but this is important.

As you go, first confirm that the results produced by your code are the same as those shown in this document. If they aren't, try to figure out why not. Perhaps you typed `'` when you should have typed `"`? Or maybe you forgot `"` at the end? Or maybe you don't have the same number of `(`'s and `)`'s?

Note that if you hit Enter in the middle of a command, R will let you continue typing on the next line, and will replace the normal > prompt with a `+`. For example, typing `3 +`, hitting Enter, then typing 2 and hitting enter once again, should look like this:

```
> 3 +
+ 2
```

```
[1] 5
```

The + on the second line of code is a prompt, not a plus sign. If you see a + at the start of a line when you didn't expect one, you've made a typing mistake and need to correct it. Try typing `)` or `'` or `"` to get R to issue an error and return to the normal > prompt. Then check what you originally typed, correct the mistake, and try again.

After you have confirmed your code does what it was supposed to do, you should experiment and find out what happens when you change things around a little bit. For example, does R return 6 when you type `3 + 3` and then hit Enter? What about `8 + 3`? Make a prediction, then see if you were correct—this is the best way to learn a new programming language!

How R Code is Formatted in this Document

In the examples above, I formatted the code and R's responses so that they would look as much like what is happening in your R Console as possible. But from this point forward, the format will change so that 1) it is easier to copy and paste from this document into the R Console in those *rare* cases when such a thing is needed, and 2) you get used to seeing how R code is formatted in the many examples available on the Internet.

Hence, instead of seeing

```
> 3 + 2
```

```
[1] 5
```

you will now see

```
3 + 2
```

```
## [1] 5
```

Don't worry about the ## in the output for now—like the `[1]` it is not meaningful here and will be explained later.

Basic Commands in the R Language

In this section, you will learn a few basic ideas about how the R Language works.

Arithmetic

As we have just seen, R is able to perform basic arithmetic operations. Type the following three expressions into your R Console, then verify you get the same results as those displayed below each line of input.

```
4.1 - 1.5
```

```
## [1] 2.6
```

```
9 * 10
```

```
## [1] 90
```

```
12 / 2.1
```

```
## [1] 5.714286
```

The `+` and `-` signs should be familiar to you—users of Excel will recognize the asterisk `*`, which represents multiplication, and the slash `/` which represents division. The carat ˆ sign represents raising a number to a particular power—hence `3 ˆ 2` is equivalent to 3^2.

```
3 ^ 2
```

```
## [1] 9
```

To make a number negative, place a minus sign in front of it.

```
3 ^ -2
```

```
## [1] 0.1111111
```

The answer is equal to $3^{-2} = 1/\left(3^2\right) = 1/9 = 0.111\ldots$.

R also has a number of built-in arithmetic functions. In R, functions are designated by a word followed by parentheses—the parenthesis can contain expressions, called "arguments," which tell the function what to do. For example, R has a function that takes the square root of a number:

```
sqrt(2)
```

```
## [1] 1.414214
```

In the example above, the name of the function above is sqrt, and the argument is 2. Hence typing the expression `sqrt(2)` and hitting Enter tells R to execute the square root function using the number 2 as input (and then print out the result).

There are many other useful mathematical functions available in R. Two that will be particularly useful in this course are the `log` function, which evaluates the natural logarithm of a number,

```
log(3)
```

```
## [1] 1.098612
```

and the `exp` function, which raises the number e to a particular power. To evaluate e^4, simply type `exp(4)` and hit Enter.

```r
exp(4)
```

```
## [1] 54.59815
```

It is possible to nest function calls: the inner-most nested function (i.e. the function surrounded by the most parentheses) is evaluated first, and its value is passed to the next outer-most function, etc.:

```r
log(exp(15.2))
```

```
## [1] 15.2
```

Of course, the result is 15.2 because the log and exp functions are inverses (i.e., they "undo" each other). The value of the expression `exp(15.2)` is $e^{15.2}$. R calculates this value, passes it to the `log` function, and the `log` function returns the final result, 15.2.

A person who has had a little math training can recognize that log and exp are inverses, and that the expression $\log\big(\exp(15.2)\big)$ equals 15.2 without needing to calculate the intermediate value $\exp(15.2)$. A small number of programming languages are clever enough to recognize this "short cut," but not R. Rather, R first calculates $\exp(15.2) = 3992786.8352$ and then calculates $\log(3992786.8352) = 15.2$.]

Note. At this point, R might appear to be nothing more than a fancy replacement for a pocket calculator: we give R a mathematical expression to evaluate, and R prints out the answer. Much of R's power comes from being able to execute long sequences of such expressions—computer programs, really—but for now we will just focus on these simple one-line expressions.

Precedence and Grouping

When confronted with an expression such as `3 + 2 * 6`, R needs to decide which part of the expression to evaluate first. For example, R might first add 3 to 2 and then multiply by 6, or it could multiply 2 by 6 and then add 3. Because each of these different procedures returns a different value, R (and all programming languages) have strict rules determining the order in which expressions are evaluated. These rules are called "operator precedence." The symbols `+` and `*`, etc. are called "operators" in the context of programming languages. Here is a listing of the operators we have seen so far, ordered from first to last in execution:

- Exponentiation via the `^` operator (i.e. exponents are evaluated first)
- Negation via the `-` operator (i.e. making four into negative four by placing a minus in front of it)
- Multiplication and division via the `*` and `/` operators
- Addition and subtraction via the `+` and `-` operators (i.e. addition and subtraction come last)

Thus an expression such as `3 + 2 * -6` has a clear meaning to R: it is the sum of 3 and -12. Expressions can be grouped together using parentheses. Hence the product of -6 and the sum of 3 and 2 can be evaluated by typing `(3 + 2) * -6`. Verify that the following expressions produce different results.

```
3 + 2 * -6
```

```
## [1] -9
```

```
(3 + 2) * -6
```

```
## [1] -30
```

Logical Comparisons

We can ask R to confirm whether the preceding two expressions are equivalent by using the `==` operator (both equals signs together form a single operator that tests for equivalence).

```
3 + 2 * -6 == (3 + 2) * -6
```

```
## [1] FALSE
```

The result of this expression is the special value `FALSE`. `FALSE`, and its counterpart `TRUE`, are special types of numbers, known as Booleans. Whenever appropriate, R will quietly convert these into `0` and `1` respectively. Thus the expression `TRUE + TRUE` is equal to `2`, as you can verify:

```
TRUE + TRUE
```

```
## [1] 2
```

To determine whether two expressions are not equivalent, we use the `!=` operator, which means "not equal to."

```
-9 != -30
```

```
## [1] TRUE
```

This statement is true because indeed $-9 \neq -30$.

R can also compare values using notions of greater-than and less-than:

```
3 > 2
```

```
## [1] TRUE
```

```
2 > 2
```

```
## [1] FALSE
```

The operator >= means greater-than-or-equal-to and is equivalent to the algebraic symbol $\geq$.

```r
2 >= 2
```

```
## [1] TRUE
```

Operators for less-than relations are available as well.

```r
1 <= 2
```

```
## [1] TRUE
```

```r
1 <= 1
```

```
## [1] TRUE
```

Vectors and Sequences

When working with R, we are typically analyzing large arrays or matrices of data. Such structures are analogous to tables or columns (or even rows) in Excel spreadsheets. The simplest of these data structures is the "vector," which represents an array of values that all have the same type of data (e.g. decimal numbers, Booleans [`TRUE` and `FALSE`], etc.). We can define a simple vector using the concatenation function, `c`:

```r
c(1, 3, 5.2, 8, 13, 21)
```

```
## [1]  1.0  3.0  5.2  8.0 13.0 21.0
```

R responds to our command by creating and then displaying a vector containing the numbers we typed in the order we typed them. More specifically, when we hit Enter, we are telling R to first create a vector by concatenating (joining together) a set of numbers, and then to print out this new vector. The numbers we wish to join together are passed as arguments to the function `c` (which stands for "concatenate"). Until now, we have only worked with functions that are only given a single argument (e.g. `sqrt(2)`, `exp(3)`, etc.)—the function `c`, however, can take many arguments, separated by commas.

Another way to define a vector is to create a sequence:

```r
1:5
```

```
## [1] 1 2 3 4 5
```

```r
15:100
```

```
##  [1]  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
## [18]  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48
## [35]  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65
## [52]  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82
## [69]  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
## [86] 100
```

The special operator, denoted by the colon `:` defines a sequence of numbers. The sequence begins with the first number (to the left of the colon), adding 1 to each preceding value, until the second number (to the right of the colon) is reached (but never exceeded).

Note. Now it should be clear what the `[1]` has meant in all of our earlier examples. To make it easier for humans to read its output, R labels each row with the *index* of the first element in the row.

R will create a sequence before performing arithmetic operations against it (i.e., the `:` operator has higher precedence than the mathematical operators). Hence `1:4 + 3` first creates the sequence `1 2 3 4`, and then adds `3` to each element.

```
1:4 + 3
```

```
## [1] 4 5 6 7
```

Of course, we can override this default behaviore by placing parentheses around expressions we want evaluated first (because parentheses have higher precendence). Thus, the following expression evaluates the sum $4 + 3 = 7$ first, and then creates the sequence $1, \ldots, 7$.

```
1:(4+3)
```

```
## [1] 1 2 3 4 5 6 7
```

R can create sequences starting with numbers that are not integers:

```
4.5:15.5
```

```
##  [1]  4.5  5.5  6.5  7.5  8.5  9.5 10.5 11.5 12.5 13.5 14.5 15.5
```

To define a sequence that does not increase by 1 with each element, we can use the special function `seq`.

```
seq(5, 100, by = 5)
```

```
##  [1]   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85
## [18]  90  95 100
```

The items inside parentheses are, once again, the "arguments" to the function `seq`. The first argument represents the start of the sequence, the second argument represents the supremum of the sequence (i.e. a number not to be exceeded); the final argument, `by = 5`, tells the `seq` function to count by fives. Hence `seq(5, 100, by = 5)` represents a sequence starting at 5, increasing by 5, and not exceeding 100.

The sequence defined above can also be created using the `:` operator:

```r
1:20 * 5
```

```
##  [1]   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85
## [18]  90  95 100
```

Here we have created the sequence $1, \dots, 20$, then multiplied each element of that sequence by 5. Our expression works as written (without parentheses) because the sequence operator `:` has higher precedence than the multiplication operator `*`.

How did I know about the `seq()` function? If I know there is a function called `seq` but don't know what arguments to pass it, I can type:

```r
?seq
```

This command displays the "help text" for the function called `seq` (if it exists and R can find it).

seq {base} R Documentation

Sequence Generation

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

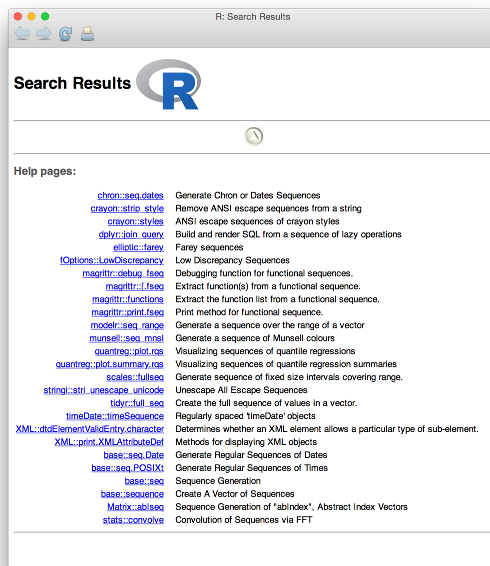
```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

Arguments

What if I don't know there's a function called `seq`? I might try searching through the documentation for commands whose help text contains the word "sequence":

```r
??sequence
```

This opens a window listing commands whose documentation contains the term "sequence."



Storing Values in Variables

When evaluating mathematical expressions, it is often desirable to save an intermediate value from the computation so that we can re-use it in subsequent computations (e.g., repeatedly evaluating the same expression might be slow). We store such values in "variables" using the assignment operator, `<-`.

In R, a variable is a symbol that we define and assign a particular value. In the example below, we define a variable called `x` and assign it the value of the expression `2 + 3`.

```
x <- 2 + 3
```

Note that R first evaluates `2 + 3` and then assigns the result to `x`. If we want to know the current value of `x`, we can type `x`.

```
x
```

```
## [1] 5
```

There are many ways to do the same thing in R, and another way to see what's inside a variable is to use the `print` command:

```
print(x)
```

```
## [1] 5
```

A word of caution: When you type the `<-` operator, there should not be any space between the `<` and `-`. That is, the following two lines of code do not do the same things:

```
x <- 2 + 3
x < -2 + 3
```

```
## [1] FALSE
```

The first line creates a variable `x` holding the value `5`; the second line asks whether `5` is less than `-2 + 3` and evaluates to `FALSE`.

Note. R's `<-` operator is unusual for programming languages, and has led to a lot of confusion and frustration among its user base. Nobody's perfect.

Back to variables: We are not limited to single letters in our variable names:

```
this_is_a_long_variable_name <- 14
this_is_a_long_variable_name
```

```
## [1] 14
```

We can assign vectors to variables as well:

```
x <- c(1, 3, 6, 2)
x <- 1:8
```

Each time we assign a value to `x`, we are replacing whatever value that variable previously held with a new value. Again, to learn the current value of `x`, we can type `x`.

```
x
```

```
## [1] 1 2 3 4 5 6 7 8
```

After saving an expression into a variable, we can use it to create new expressions:

```
3 + x
```

```
## [1]  4  5  6  7  8  9 10 11
```

```
y <- 4
y + x
```

```
## [1]  5  6  7  8  9 10 11 12
```

It should be clear that using a variable as part of an expression does not change its value—in the example above, the expression `3 + x` creates a new vector (and prints it), leaving the original value of `x` intact. We can verify this by typing `x`:

```
x
```

```
## [1] 1 2 3 4 5 6 7 8
```

It should also be clear that when we define a new expression using variables we have already assigned values to, the expression is immediately evaluated using the *current* values of those variables. This can be illustrated with an example:

```
x <- 1:4
y <- 2
z <- x * y
z
```

```
## [1] 2 4 6 8
```

```
y <- 4
z
```

```
## [1] 2 4 6 8
```

As we can see, the variable `z` is equal to `1:4` times `2`, not `x` times `y`. Assigning a new value to `y` has no impact on `z`.

Data

Next, we will learn how structured data is represented in R. There are many small data sets that are included in the basic R distribution. One of these is called `pressure`. To see the data, simply type `pressure` and hit Enter.

```
pressure
```

```
##    temperature pressure
## 1            0   0.0002
## 2           20   0.0012
## 3           40   0.0060
## 4           60   0.0300
## 5           80   0.0900
## 6          100   0.2700
## 7          120   0.7500
## 8          140   1.8500
## 9          160   4.2000
## 10         180   8.8000
## 11         200  17.3000
## 12         220  32.1000
## 13         240  57.0000
## 14         260  96.0000
```

```
## 15          280 157.0000
## 16          300 247.0000
## 17          320 376.0000
## 18          340 558.0000
## 19          360 806.0000
```

The data set is stored in a special data structure called a `data.frame`.

Probably the simplest analogy to explain what a data frame is, is to say that it is very much like a spreadsheet or database table. That is, there are many rows and many columns. All of the items in each column represent different instances of the same variable, whereas all of the items in each row collectively describe a single entity. The `pressure` data frame has two columns, named `temperature` and `pressure`, and 19 rows. The values in each row "go together" in the sense that we would not want to sort one of the columns without also sorting all other columns in the same way.

We will return to the topic of data frames momentarily after a brief interlude.

Using Packages

In the earlier instructions for setting up R and RStudio, you should have installed a set of packages called `tidyverse`. If you skipped over those instructions or ran them on another computer, go back and install `tidyverse` now.

Installing `tidyverse` triggers the installation of about 20 packages. A few that we will use in this course are `dplyr`, `ggplot2`, and `readr`.

Before we can access the functionality provided by a package, we must instruct R to load it from computer disk and make its features available to your current R session. We accomplish this task using the `library` function. Do this now the `ggplot2` package:

```
library(ggplot2)
```

Alternatively, if you know you will use many of the packages in the `tidyverse` bundle, you can load them all at once, which you should do now:

```
library(tidyverse)
```

Note. The `ggplot2` package was already loaded, hence there is no mention of it when loading `tidyverse` (i.e., there is no `Loading tidyverse: ggplot2` message).

The `Conflicts with tidy packages` message is a warning that both the `stats` package (which is loaded automatically at the start of your R session) and the `dplyr` package provide functions named `filter` and `lag`. Because `dplyr` was loaded after `stats`, the `dplyr` versions of those functions are what get called unless you specify the package name using the `::` operator.

```
lag(1:3)
```

```
## [1] NA  1  2
```

```
stats::lag(1:3)
```

```
## [1] 1 2 3
## attr(,"tsp")
## [1] 0 2 1
```

```
dplyr::lag(1:3)
```

```
## [1] NA  1  2
```

Data Frames and Tibbles

Earlier, you looked at the `pressure` data frame. Next we will look at another data set that is included in (and loaded with) the `ggplot2` library. This data set is called `mpg` and contains information about automobile fuel efficiency ("mpg" means "miles per gallon"). To display this data set, you can type its name and hit Enter:

```
mpg
```

```
## # A tibble: 234 × 11
##    manufacturer      model displ  year   cyl      trans  drv   cty   hwy
##           <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int>
## 1          audi         a4   1.8  1999     4   auto(l5)     f    18    29
## 2          audi         a4   1.8  1999     4 manual(m5)     f    21    29
## 3          audi         a4   2.0  2008     4 manual(m6)     f    20    31
## 4          audi         a4   2.0  2008     4   auto(av)     f    21    30
## 5          audi         a4   2.8  1999     6   auto(l5)     f    16    26
## 6          audi         a4   2.8  1999     6 manual(m5)     f    18    26
## 7          audi         a4   3.1  2008     6   auto(av)     f    18    27
## 8          audi a4 quattro   1.8  1999     4 manual(m5)     4    18    26
## 9          audi a4 quattro   1.8  1999     4   auto(l5)     4    16    25
## 10         audi a4 quattro   2.0  2008     4 manual(m6)     4    20    28
## # ... with 224 more rows, and 2 more variables: fl <chr>, class <chr>
```

The `mpg` data set is a special kind of `data.frame` called a `tibble`. Tibbles are not very different from data frames. The main differences have to do with certain types of default behaviors that aren't very important for this tutorial. Probably the most visible difference is that tibbles are displayed much more sensibly than data frames.

Returning to the `mpg` data set, notice the first line of output indicates this is a tibble with 234 rows and 11 columns. Moreover, there are more rows and columns than are displayed (a nice feature of tibbles). Information about the data that isn't displayed is in the last row of output.

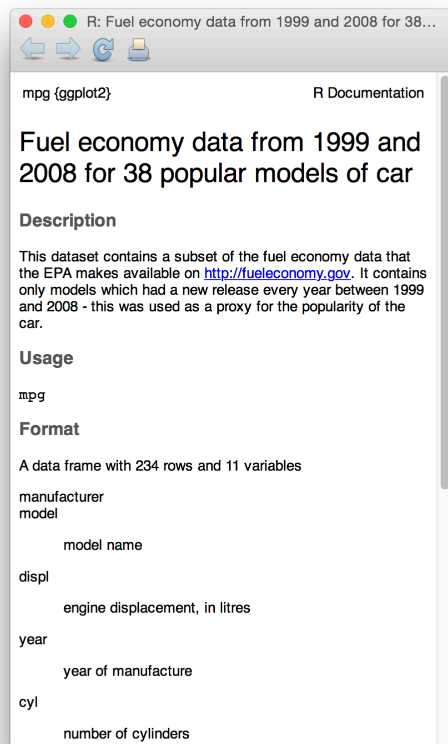
In between, you can see the structure of the `mpg` tibble and some of its data. For example, the `model` column (second from the left) is of type `chr` (which means "character" or "text"). It contains values such as `a4` and `a4 quattro`. The column to its right, `displ`, by contrast, contains numeric data (`dbl`

or `double`, which is computer for "numeric with decimal points"). The `year` column contains `int` (meaning "integer") data.

Looking at the first row, we see it describes a model of car: the 1999 Audi A4 with a 4-cylinder 1.8l engine and 5-speed automatic transmission. This model of car was rated as getting 18 miles to the gallon in the city, and 29 on the highway.

Data sets provided by R packages are usually documented. To see the documentation for `mpg`, type:

```
?mpg
```



The help text can be useful when it provides details about the type of data stored in the data frame/tibble.

Accessing rows and columns

Data frames (from this point forward, I will refer to both data frames and tibbles as data frames unless the distinction matters) are extremely flexible, in the sense that we can access and manipulate their data in a variety of ways. First, data frames act like matrices in the sense that we can index individual data points using row and column subscripts:

```r
pressure[3, 2]
```

```
## [1] 0.006
```

Square brackets ([and]) are used in R to access or change subsets of values from more complex data structures. In the example above, the first index refers to row 3, and the second to column 2.

When we do the same thing with the `mpg` data frame (which is also a tibble), we get a slightly different type of result:

```r
mpg[3, 2]
```

```
## # A tibble: 1 × 1
##   model
##   <chr>
## 1    a4
```

Notice that calling `pressure[3, 2]` returns a number, whereas `mpg[3, 2]` returns a tibble containing 1 row and 1 column. This is another difference between tibbles and traditional data frames. If we wanted to get `a4` as a character value rather than a tibble, then we would need to turn it back into a traditional `data.frame` first, and then access one of its elements:

```r
as.data.frame(mpg)[3, 2]
```

```
## [1] "a4"
```

If you don't 100% follow what is happening here, don't worry for now. Just focus on what the square brackets are doing.

If we wanted to access an entire row, we can leave out the second number (which identifies the column).

```r
pressure[5,]
```

```
##   temperature pressure
## 5          80     0.09
```

Or, if we wanted to access a single column:

```r
pressure[, 2]
```

```
##  [1]   0.0002   0.0012   0.0060   0.0300   0.0900   0.2700   0.7500
##  [8]   1.8500   4.2000   8.8000  17.3000  32.1000  57.0000  96.0000
## [15] 157.0000 247.0000 376.0000 558.0000 806.0000
```

Alternatively, we can use the name of a column to access its data.

```r
pressure[, 'temperature']
```

```
##  [1]   0  20  40  60  80 100 120 140 160 180 200 220 240 260 280 300 320
## [18] 340 360
```

```r
pressure$temperature
```

```
##  [1]   0  20  40  60  80 100 120 140 160 180 200 220 240 260 280 300 320
## [18] 340 360
```

In R, often there are many ways to do the same thing, as in the example above.

Next, using the `$` operator to access individual columns, let's calculate a few summary statistics:

```r
mean(pressure$temperature)
```

```
## [1] 180
```

```r
median(mpg$cty)
```

```
## [1] 17
```

R provides a convenient way to view a number of useful statistics all at once via the `summary` command:

```r
summary(mpg$cty)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    9.00   14.00   17.00   16.86   19.00   35.00
```

This command prints out, in order, the minimum, 1st quartile, median, mean, 3rd quartile, and maximum values. We can calculate other summary statistics (e.g., to learn about the dispersion of these data):

```r
var(mpg$cty)
```

```
## [1] 18.11307
```

```r
sd(mpg$cty)
```

```
## [1] 4.255946
```

```
mad(mpg$cty)
```

```
## [1] 4.4478
```

These functions return the variance, standard deviation, and median absolute deviation respectively. The variance and standard deviation are related to each other of course:

```
# variance is the square of the standard deviation
sqrt(var(mpg$cty))
```

```
## [1] 4.255946
```

```
sd(mpg$cty)
```

```
## [1] 4.255946
```

What's going on with the extra text and the hash sign (#)? The # indicates the start of a "comment"—text that appears in your code, which R knows to ignore. Comments are for humans, not computer programs. If you want to make a note in your code (always a good idea), you can type a hash sign followed by your comments.

```
# This text will not be interpreted as R code.
```

More about indexing

To recap, we can access data in a data frame by indicating which row, column, or both we want to work with or print out:

```
mpg[1,]
```

```
## # A tibble: 1 × 11
##   manufacturer model displ  year   cyl    trans   drv   cty   hwy    fl
##          <chr> <chr> <dbl> <int> <int>    <chr> <chr> <int> <int> <chr>
## 1         audi    a4   1.8  1999     4 auto(l5)     f    18    29     p
## # ... with 1 more variables: class <chr>
```

```
mpg[,2]
```

```
## # A tibble: 234 × 1
##         model
##         <chr>
## 1          a4
## 2          a4
## 3          a4
## 4          a4
```

```
## 5          a4
## 6          a4
## 7          a4
## 8  a4 quattro
## 9  a4 quattro
## 10 a4 quattro
## # ... with 224 more rows
```

```
mpg[1,2]
```

```
## # A tibble: 1 × 1
##   model
##   <chr>
## 1    a4
```

If we want to look at more than one row or column at a time, we can use a vector (created with the c function, as we did earlier in this assignment) to indicate which rows or columns:

```
mpg[c(5, 20, 30),]
```

```
## # A tibble: 3 × 11
##   manufacturer              model displ  year   cyl    trans   drv   cty
##          <chr>              <chr> <dbl> <int> <int>    <chr> <chr> <int>
## 1         audi                 a4   2.8  1999     6 auto(l5)     f    16
## 2    chevrolet c1500 suburban 2wd   5.3  2008     8 auto(l4)     r    11
## 3    chevrolet    k1500 tahoe 4wd   5.3  2008     8 auto(l4)     4    11
## # ... with 3 more variables: hwy <int>, fl <chr>, class <chr>
```

This returns three rows: the 5th, 20th, and 30th in the data frame. Sequences can be particularly useful here:

```
mpg[1:5,]
```

```
## # A tibble: 5 × 11
##   manufacturer model displ  year   cyl      trans   drv   cty   hwy    fl
##          <chr> <chr> <dbl> <int> <int>      <chr> <chr> <int> <int> <chr>
## 1         audi    a4   1.8  1999     4   auto(l5)     f    18    29     p
## 2         audi    a4   1.8  1999     4 manual(m5)     f    21    29     p
## 3         audi    a4   2.0  2008     4 manual(m6)     f    20    31     p
## 4         audi    a4   2.0  2008     4   auto(av)     f    21    30     p
## 5         audi    a4   2.8  1999     6   auto(l5)     f    16    26     p
## # ... with 1 more variables: class <chr>
```

This gave us the first five rows.

It is often the case that we want to work with a subset of data for which some condition holds. For example, we might want the subset of cars with 6 cylinders. From the examples above, we know

that if we can create a vector that indicates which observations have 6 cylinders, then R will do the rest for us. To create that vector, we use a simple test for equality:

```
mpg$cyl == 6
```

```
##   [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
##  [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
##  [56] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
##  [89] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [111] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
## [122]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
## [155]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [177]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
## [188] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [199] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
## [210] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [221]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [232]  TRUE  TRUE  TRUE
```

In the above expression, the number 6 is compared to each value in the column `mpg$cyl`, resulting in a value of `TRUE` when equal (or `FALSE` when not). This vector tells R which observations we want:

```
mpg[mpg$cyl == 6,]
```

```
## # A tibble: 79 × 11
##    manufacturer      model displ  year   cyl      trans   drv   cty   hwy
##           <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int>
## 1          audi         a4   2.8  1999     6   auto(l5)     f    16    26
## 2          audi         a4   2.8  1999     6 manual(m5)     f    18    26
## 3          audi         a4   3.1  2008     6   auto(av)     f    18    27
## 4          audi a4 quattro   2.8  1999     6   auto(l5)     4    15    25
## 5          audi a4 quattro   2.8  1999     6 manual(m5)     4    17    25
## 6          audi a4 quattro   3.1  2008     6   auto(s6)     4    17    25
## 7          audi a4 quattro   3.1  2008     6 manual(m6)     4    15    25
## 8          audi a6 quattro   2.8  1999     6   auto(l5)     4    15    24
## 9          audi a6 quattro   3.1  2008     6   auto(s6)     4    17    25
## 10    chevrolet     malibu   3.1  1999     6   auto(l4)     f    18    26
## # ... with 69 more rows, and 2 more variables: fl <chr>, class <chr>
```

Notice that all of the observations returned have 6 cylinder engines. Recall that in the earlier example, we passed a vector of integers (i.e. counting numbers) to tell R which rows to give us (e.g., rows 5, 20, and 30). In the case above, however, we use a vector of Booleans (`TRUE`'s and `FALSE`'s). Because the vector of Booleans has the same length as our data, R correctly interprets the `TRUE`'s and `FALSE`'s as indicators of whether to include or exclude each row.

See if you can follow what is happing in this example:

```
# Create a vector indicating which rows have 6 cylinders
has_six_cylinders <- mpg$cyl == 6
print(has_six_cylinders)
```

```
##   [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
##  [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
##  [56] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
##  [89] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [111] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
## [122]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
## [155]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [177]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
## [188] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [199] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
## [210] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [221]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [232]  TRUE  TRUE  TRUE
```

```
# Create a vector with the numbers 1...234. These index the 234 rows in
# our data set
row_indexes <- 1:nrow(mpg)   # Note: nrow() returns the number of rows
print(row_indexes)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
## [103] 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
```

```
## [120] 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
## [137] 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153
## [154] 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
## [171] 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187
## [188] 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
## [205] 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221
## [222] 222 223 224 225 226 227 228 229 230 231 232 233 234
```

```r
# Create a vector containing only the indexes corresponding to cars with
# six cylinders
row_indexes_six_cylinders <- row_indexes[has_six_cylinders]
print(row_indexes_six_cylinders)
```

```
##  [1]   5   6   7  12  13  14  15  16  17  35  36  37  39  40  41  42  43
## [18]  44  45  46  47  48  49  50  51  52  58  78  79  80  81  84  85  91
## [35]  92  93  94 113 114 115 120 121 122 123 124 125 138 139 146 147 148
## [52] 149 150 151 152 153 155 156 157 158 176 177 178 184 185 186 191 192
## [69] 193 204 205 206 207 212 220 221 232 233 234
```

```r
# We can arrive at the same subset of observations in more than one way:
mpg[has_six_cylinders,]            # 234 TRUE's and FALSE's
```

```
## # A tibble: 79 × 11
##    manufacturer      model displ  year   cyl      trans   drv   cty   hwy
##           <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int>
## 1          audi         a4   2.8  1999     6   auto(l5)     f    16    26
## 2          audi         a4   2.8  1999     6 manual(m5)     f    18    26
## 3          audi         a4   3.1  2008     6   auto(av)     f    18    27
## 4          audi a4 quattro   2.8  1999     6   auto(l5)     4    15    25
## 5          audi a4 quattro   2.8  1999     6 manual(m5)     4    17    25
## 6          audi a4 quattro   3.1  2008     6   auto(s6)     4    17    25
## 7          audi a4 quattro   3.1  2008     6 manual(m6)     4    15    25
## 8          audi a6 quattro   2.8  1999     6   auto(l5)     4    15    24
## 9          audi a6 quattro   3.1  2008     6   auto(s6)     4    17    25
## 10    chevrolet     malibu   3.1  1999     6   auto(l4)     f    18    26
## # ... with 69 more rows, and 2 more variables: fl <chr>, class <chr>
```

```r
mpg[row_indexes_six_cylinders,]     # 79 integers
```

```
## # A tibble: 79 × 11
##    manufacturer      model displ  year   cyl      trans   drv   cty   hwy
##           <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int>
## 1          audi         a4   2.8  1999     6   auto(l5)     f    16    26
## 2          audi         a4   2.8  1999     6 manual(m5)     f    18    26
## 3          audi         a4   3.1  2008     6   auto(av)     f    18    27
## 4          audi a4 quattro   2.8  1999     6   auto(l5)     4    15    25
```

```
## 5          audi a4 quattro   2.8  1999      6 manual(m5)     4    17    25
## 6          audi a4 quattro   3.1  2008      6   auto(s6)     4    17    25
## 7          audi a4 quattro   3.1  2008      6 manual(m6)     4    15    25
## 8          audi a6 quattro   2.8  1999      6   auto(l5)     4    15    24
## 9          audi a6 quattro   3.1  2008      6   auto(s6)     4    17    25
## 10    chevrolet      malibu   3.1  1999      6   auto(l4)     f    18    26
## # ... with 69 more rows, and 2 more variables: fl <chr>, class <chr>
```

Be sure you understand the concepts here. In a later assignment, you will learn about the `dplyr` package, which makes tasks such as getting a subset of observations matching a particular criterion (e.g., with six cylinders) much simpler and easier to read. But the fundamental concept of using indexes to get subsets of data is central to R, so you will never entirely avoid using the `[` and `]`.

Plotting

We will now use the `ggplot2` library to create a few basic plots. If the library is not already loaded, do so now:

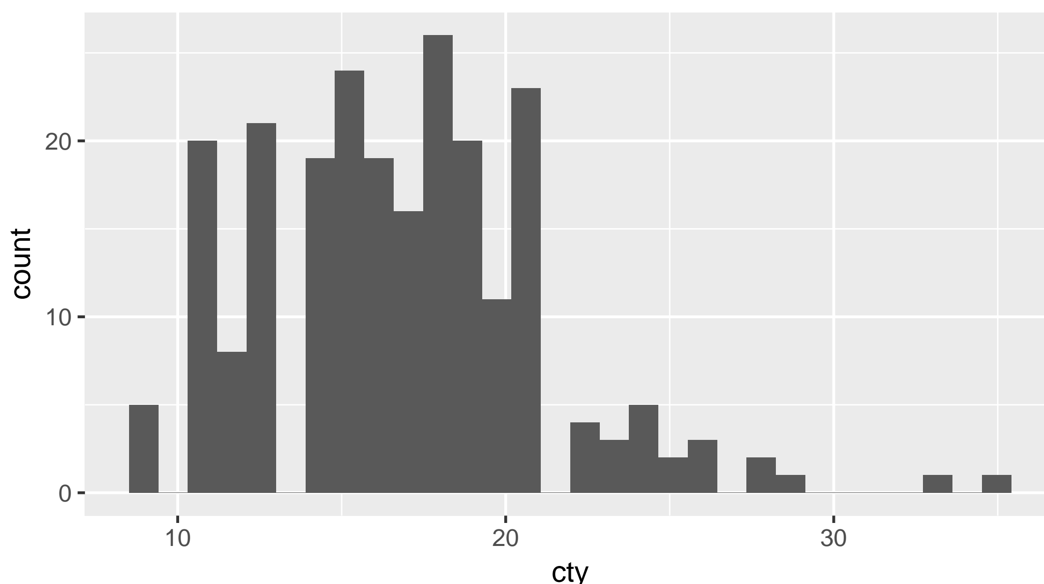
```
library(ggplot2)
```

The `ggplot2` library provides a very simple interface for building many of the most common plots you might want to produce (as well as more sophisticated interfaces for building more sophisticated plots—we will learn about these later in the course).

We begin by focusing our attention on a single attribute (i.e. column) of the `mpg` data frame.

```
ggplot(data = mpg, mapping = aes(x = cty)) + geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

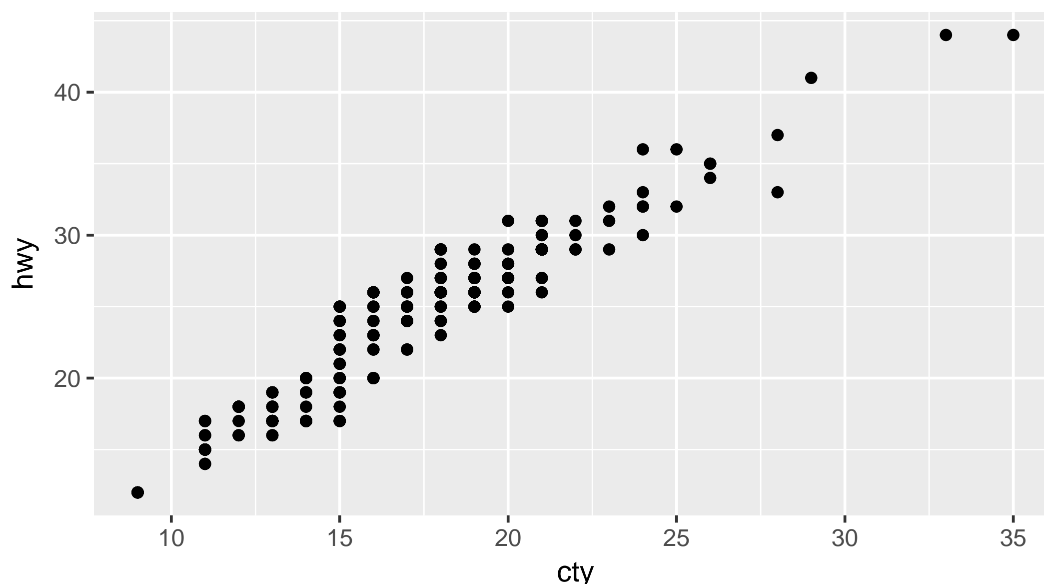


Notice that the name of the package is `ggplot2` whereas the command for creating a plot involves a function called `ggplot`. We will learn about the syntax for creating plots via `ggplot2` later in the course. For now, you should notice that we are creating a plot using the data frame called `mpg`, that we have mapped the `cty` column to the x-axis, and that we are using this mapping as the basis for creating a histogram.

A histogram is a useful way to summarize a single vector of data. The x-axis represents the values of our variable of interest, whereas the y-axis represents the count of observations observed in a particular range of values along the x-axis (these ranges are called "bins"). The ggplot2 engine automatically maps this new count variable to the y-axis, so we don't need to specify it as part of the mapping.

If we want to visualize the relationship between two variables, we need something more sophisticated. For example, to compare fuel efficiency in city versus on the highway, we can write:

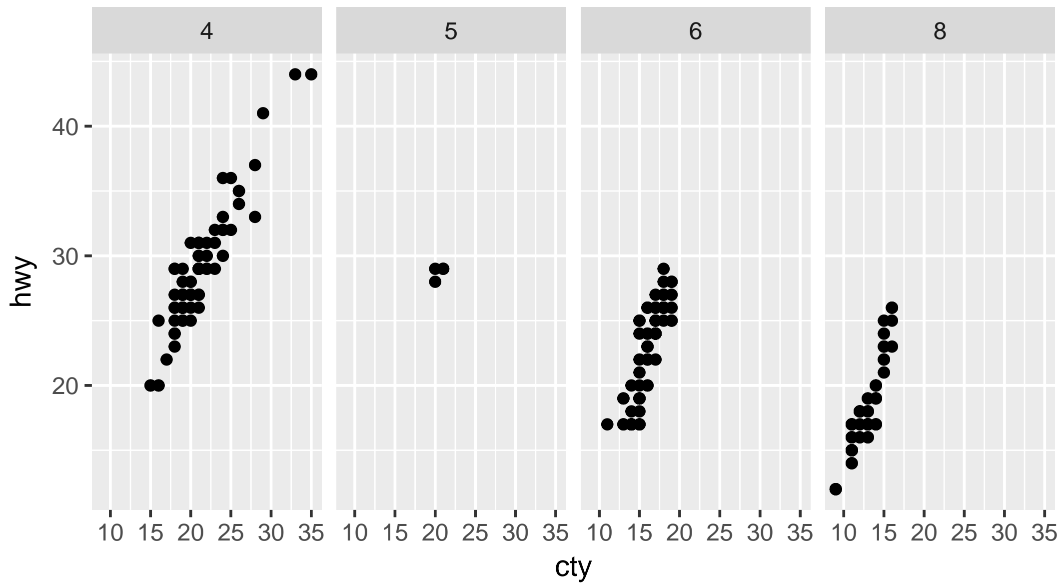
```
ggplot(mpg, aes(x = cty, y = hwy)) + geom_point()
```



We mapped `cty` to the x-axis and `hwy` to the y-axis, then instructed the `ggplot2` engine to generate a scatter plot. Each observation is represented by a single dot.

We can add a third variable to the plot. We will instruct `ggplot2` to display the third dimension of data (in this case, the variable `cyl`—the number of cylinders in the engine) using "facets." Facets are groups of plots in which each plot displays a subset of the data.

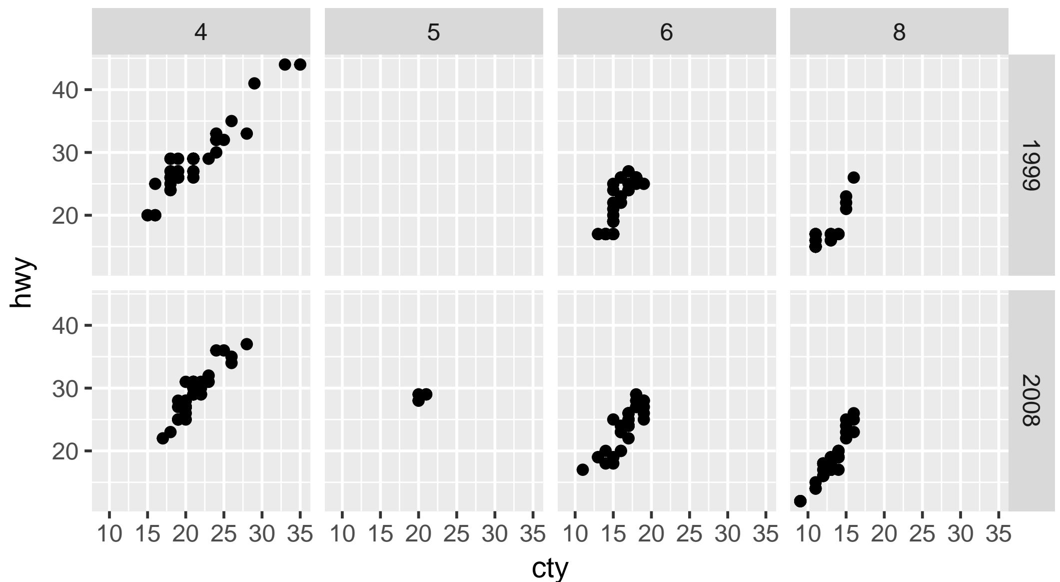
```
ggplot(mpg, aes(x = cty, y = hwy)) + geom_point() + facet_grid(~ cyl)
```



We now have 4 plots with `cty` along the x-axis and `hwy` along the y-axis. At left are cars with 4 cylinder engines, to the right those with 8 cylinder engines.

We can add a fourth variable, `year` indicating the model year, by creating more facets.

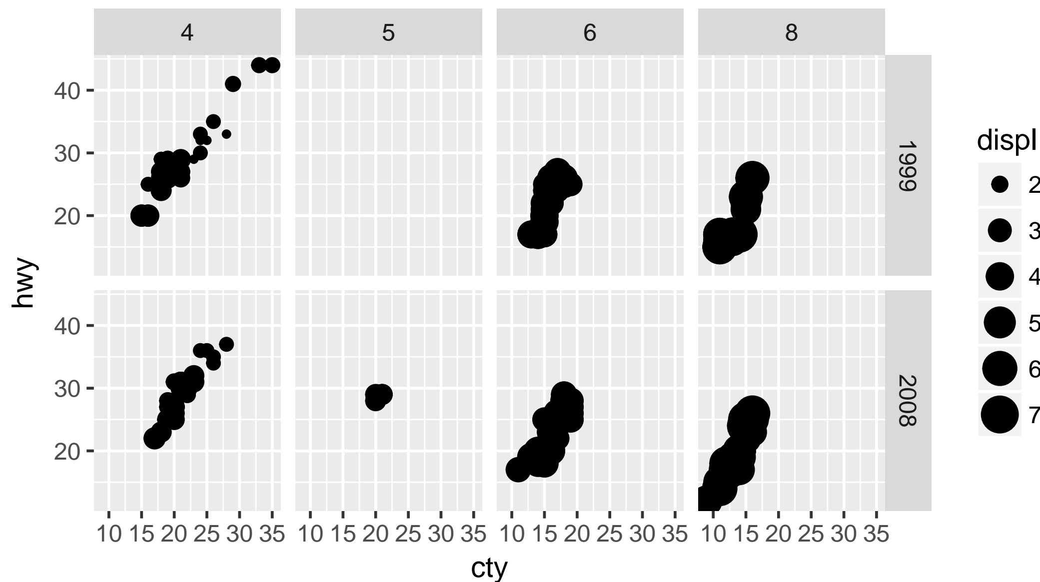
```
ggplot(mpg, aes(x = cty, y = hwy)) + geom_point() + facet_grid(year ~ cyl)
```



We now have 8 scatter plots showing with `cty` and `hwy` along the x and y axes. Each observation appears in only one of these six scatter plots. The plots on the left/right show cars with 4, 5, 6, or 8 cylinders. Those in the top row correspond with 1999 models, where those in the bottom row are from 2008.

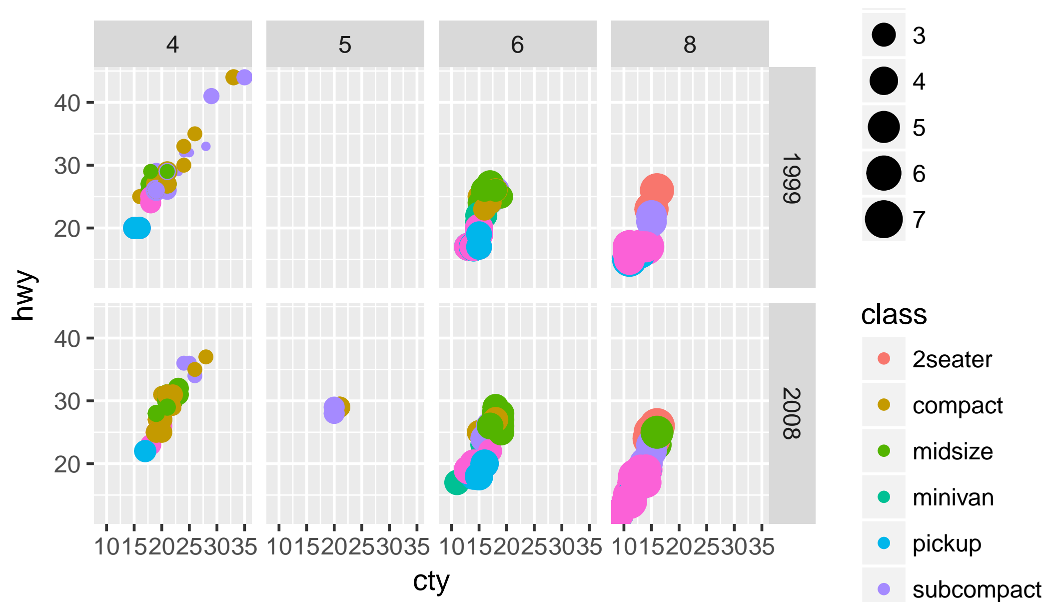
Can we add a fifth variable? Yes! We display the size of each engine, `displ`, through the size of the points, such that cars with smaller engines have smaller points. This is accomplished by mapping the values of `displ` to what `ggplot2` refers to as `size`.

```r
ggplot(mpg, aes(x = cty, y = hwy, size = displ)) +
  geom_point() + facet_grid(year ~ cyl)
```



How about a sixth variable? Why not!? We will indicate the class of vehicle by mapping `class` to the `colour` of the point.

```r
ggplot(mpg, aes(x = cty, y = hwy, size = displ, colour = class)) +
  geom_point() + facet_grid(year ~ cyl)
```



We now have a plot that is displaying 6 dimensions of data describing 234 automobile models. We managed to compress a great deal of information into a very small picture, but is this picture informative? Depending on our needs, this plot might be perfect: after all, we might need to show the complex interactions between all six variables.

But usually we can make our point *more effectively* with a simpler picture—one that contains less information, and, despite its relative simplicity, still manages to be more informative.

We will return to these ideas later in the course. For now, congratulate yourself for having completed this tutorial!

Confirmation of Completion of Assignment 1

Please go to Blackboard and indicate that you have completed this assignment. If for some reason you are unable to indicate completion of this assignment on Blackboard, you can instead print and sign this page (just this page), and bring it to class on Friday, February 3, 2017.

I, ________________________, attest that the following statements are true:

1. I read Assignment 1 in its entirety.
2. I installed R on my computer, or had access to a computer with R already installed.
3. Wherever indicated in this assignment, I typed the appropriate commands into the R console and observed the results. For example, after seeing

```
3 + 2
```

I typed `3 + 2` into R, hit Enter, and observed the result.

4. I did this for every example in this document.

Signed ________________________________

Date ________________________________