Week 3 R Tutorial

Statistics and Statistical Programming
Northwestern University
MTS 525

Aaron Shaw

September 28, 2020

Contents

- More Rstudio, RMarkdown, and R background + tips . . . 1
  - Working in "Base R" . . . 1
  - Notebooks vs. scripts vs. other ways of developing R code . . . 2
  - Working directories . . . 2
  - Adding comments to your R code chunks . . . 3
  - Importing datasets from libraries, the web, and locally (on your computer) . . . 3
- More (complicated) variable types . . . 3
  - Factors . . . 3
  - Lists . . . 4
  - Matrices . . . 5
  - Data frames . . . 6
- Vectorized operations: "Apply" functions and beyond . . . 10
- Some basic graphs with ggplot2 . . . 12

More Rstudio, RMarkdown, and R background + tips

Last week, I encouraged you to learn some basic things about RStudio, RMarkdown, and R in order to get started. All three have a vast (overwhelming?) number of available options, opportunities for customization, and documentation. While I think you'll be able to do most of what you need for the course with what I'm providing in these tutorials and/or links to other resources, you should absolutely explore further (I'll keep adding materials to the list of resources on the course wiki page. In the meantime, here are a few more things that I think are likely to be useful in the coming weeks of our course. Partly, I want to keep introducing other "features" of these tools that may not have been obvious. I also want to continue to explain some of the choices I've made in how I introduce you to these resources and encourage you to explore more broadly, cultivate your own preferences, and find ways to work in R that suit your goals, needs, and skill level.

Working in "Base R"

You may encounter a distinction between what people often call *Base R* and other tools or software packages. *Base R* usually refers to the syntax and functions that R employs without the addition of extra libraries or packages (or with libraries and packages that merely extend that syntax rather than develop an entirely different syntax). The most common syntax people currently use that is not Base R comes from a large family of packages called the Tidyverse. The Tidyverse is maintained and distributed by the same organization that maintains and distributes RStudio. The Tidyverse packages provide a suite of tools developed to facilitate data science and statistical analysis with a programming syntax (or "grammar" as the creators might prefer) designed to overcome some of the initial obstacles and idiosyncracies of Base R.

For the purposes of our course, you might want to learn more about Tidyverse code and packages. You will almost certainly encounter example code snippets and suggestions around the web that assume you are familiar with the Tidyverse syntax and software. Indeed, those of you who are accustomed to working in Python or other "modern" programming languages may find it easier to work with the Tidyverse than with Base R. Books like those by Healy and Wickham and Grolemund linked from the course wiki page forego almost any introduction to Base R in favor of working almost exclusively within the Tidyverse.

All of this is a bit far out in the weeds, so let me skip to the key takeaway: I start off teaching you Base R and will not require you to learn the Tidyverse. That said, I strongly encourage you to embrace the Tidyverse, especially as you become familiar with some of the basic tasks and operations you can perform with Base R. I especially encourage you to become familiar with the `ggplot2` package as it produces visualizations superior to those you can make in Base R in almost every way. My rationale is that even the Tidyverse packages and syntax operate *within* the R software environment and rely on the underlying functionalities of Base R. In that respect, starting with Base R provides you with a foundation that should allow you greater flexibility as you deepen your R skills. You will not be constrained to any particular software package within R.

Notebooks vs. scripts vs. other ways of developing R code

My first tutorial made at least one other leap/assumption worth noting. If you've taken other introductory courses or tutorials in data science or programming with R or Python, you may have interacted with other languages/tools via software "Notebooks." Notebooks are a type of software development environment that make it possible to work with text and execute code alongside each other. Sound familiar? It should! The R Markdown scripts I've introduced and suggested here are similar in many ways. The big difference is that in a notebook you can iteratively execute code chunks and view the results of that execution as you go along.

Notebooks are fantastic software development and learning tools and R Studio/R Markdown support them as well. You can learn more via the R studio documentation, the (much more exhaustive) R Markdown "cookbook", and try one out by selecting the 'File → New File → R Notebook' from the R Studio dropdown menus.

My attitude here is analogous to my thoughts on the Tidyverse. Even if you wind up working primarily/exclusively in Notebooks, you should have some grounding in working with R via the console, scripts, and "regular" R Markdown files. This is because R Markdown Notebooks rely on all of these tools as well and by knowing even a little bit about what's going on in the background you'll be better able to deepen your knowledge and extend your work beyond the limitations of Notebooks.

tl;dr: If you (want to) love Notebooks, you can/should use them. I wanted to make sure you knew how to work with some more foundational stuff too.

Working directories

The concept of "working directories" refers to the location on your computer where R is running, looking for files, and/or storing output files. Sounds simple enough, but managing working directories seems to often induce confusion as you start working with external datasets from the web or stored locally elsewhere on your computer.

For your purposes here, the simplest way to manage/avoid working directory issues may be to select an option from the "Session → Set Working Directory" dropdown menu in RStudio. My best guess is that while you're working on a given .Rmd file you'll be happy most of the time if you choose the "To source file location" option. That said, you might have other preferences and I don't mean to suggest this as a rule/requirement. Whatever the case, when you choose that menu option, you'll see RStudio generate something at the console that might look a bit like this:

```
> setwd("~/Documents/Courses/2020/stats/r_tutorials")
```

This `setwd()` ("set working directory") command is what R calls under the hood to..set the working directory. You can also ask R to tell you where your current working directory is with the related command, `getwd()`. Try running it just like that with nothing in the parentheses to see what it returns.

```r
getwd()
```

```
## [1] "/home/ads/Documents/Teaching/2020/stats/r_tutorials"
```

Whatever this says is where R thinks it's "doing stuff" on your machine, so, in the scenario where you are asking R to load a dataset from a file stored somewhere else on your computer, R might struggle to find any files located elsewhere unless you point to exactly where it lives. More on this in one of the examples below.

Adding comments to your R code chunks

The concept of "comments" in code is pretty intuitive. A comment is just some text that the programming language interpreter ignores and repeats. Comments are generally inserted in code to make it easier for people to read in various ways. R interprets the `#` character and anything that comes after it as a comment. R will not try to interpret whatever comes next as a command:

```r
2+2
```

```
## [1] 4
```

```r
# This is a comment. The next line is too:
# 2+2
```

Comments are often less common in the context of R Markdown scripts and notebooks. That said, you may encounter them in examples, R documentation, and elsewhere. You may also want to use them to leave notes for yourself or the teaching team in your R scripts. Whatever the case, it's good to know how to comment.

Importing datasets from libraries, the web, and locally (on your computer)

. You will want to import datasets.

More (complicated) variable types

In the previous tutorial we introduced some basic variable types (numeric, character, and logical). Here are some other common Base R variable types that you will encounter and need to learn to recognize/manage in your work:

Factors

Factors usually work like character variables that take a finite and pre-specified set of values. They are useful for encoding categorical variables. You can create them with the `factor()` command or by running `as.factor()` on a character vector.

```r
## Create a vector of cities as a factor:
cities <- factor(c("Chicago", "Detroit", "Milwaukee"))
summary(cities)
```

```
##   Chicago   Detroit Milwaukee 
##         1         1         1
```

```r
class(cities)
```

```
## [1] "factor"
```

```r
## Create another vector as a character first...
more.cities <- c("Oakland", "Seattle", "San Diego")
summary(more.cities)
```

```
##    Length     Class      Mode 
##         3 character character
```

```r
class(more.cities)
```

```
## [1] "character"
```

```r
## ...and coerce it to a factor:
more.cities <- as.factor(more.cities)
summary(more.cities)
```

```
##   Oakland San Diego   Seattle
##         1         1         1
```

```r
class(more.cities)
```

```
## [1] "factor"
```

You can usually run `as.factor()` to coerce other kinds of variables into a factor; however, be warned that doing so has some risks as R may not share your intuitions about what ought to happen. Whenever you coerce or convert or reshape data you should immediately run something like `summary()` and `class()` to inspect the results and confirm whether the results seem to align with your aspirations.

Lists

Lists are a bit like vectors, but can contain many other kinds of object (e.g., other variables). In this sense they are more of a data structure than a type, but for the purposes of R, the distinctions between data structures and types are a bit mushy...

```r
cities.list <- list(cities, more.cities)
class(cities.list)
```

```
## [1] "list"
```

```r
cities.list
```

```
## [[1]]
## [1] Chicago   Detroit   Milwaukee
## Levels: Chicago Detroit Milwaukee
##
## [[2]]
## [1] Oakland   Seattle   San Diego
## Levels: Oakland San Diego Seattle
```

We can name the items in the list just like we did for a vector:

```r
names(cities.list) <- c("midwest", "west")

# This works too:
cities.list <- list("midwest" = cities, "west" = more.cities)
```

You can index into the list just like you can with a vector except that instead of one set of square brackets you have to use two:

```r
cities.list[["midwest"]]
```

```
## [1] Chicago   Detroit   Milwaukee
## Levels: Chicago Detroit Milwaukee
```

```r
cities.list[[2]]
```

```
## [1] Oakland   Seattle   San Diego
## Levels: Oakland San Diego Seattle
```

With a list you can also index recursively (down into the individual objects contained in the list). For example:

```
cities.list[["west"]][2]
```

```
## [1] Seattle
## Levels: Oakland San Diego Seattle
```

Some functions that operate on vectors or other data structures "just work" on lists. Others don't or produce weird output that you probably weren't expecting (if you're coming to R from Python this may induce tears and gnashing of teeth). As a practical matter, you should not assume R will be perfectly consistent with how functions treat different variable types and inspect the output of commands to see what happens:

```
# summary works as you might hope:
summary(cities.list)
```

```
##         Length Class  Mode
## midwest 3      factor numeric
## west    3      factor numeric
```

```
# table produces something very weird:
table(cities.list)
```

```
##            west
## midwest     Oakland San Diego Seattle
##   Chicago         1         0       0
##   Detroit         0         0       1
##   Milwaukee       0         1       0
```

Matrices

Matrices are a little less common for everyday use, but it's good to know they're there and that R can help you do matrix arithmetic to your heart's content. In my day-to-day work, I most frequently encounter matrices in R as an intermediate format used or generated by functions to complete specific tasks (e.g., running regression models). This means that I sometimes want/need to interact with matrix objects and, by extension, you might too.

An example is below. Check out the help documentation for the `matrix()` function for more.

```
m1 <- matrix(c(1:12), nrow=4, byrow=FALSE)
m1
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
m2 <- matrix(seq(2,24,2), nrow=4, byrow=FALSE)
m2
```

```
##      [,1] [,2] [,3]
## [1,]    2   10   18
## [2,]    4   12   20
## [3,]    6   14   22
## [4,]    8   16   24
```

```
m1*m2
```

```
##      [,1] [,2] [,3]
## [1,]    2   50  162
## [2,]    8   72  200
## [3,]   18   98  242
```

```
## [4,]   32  128  288
```

```r
t(m2) # transposition
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    4    6    8
## [2,]   10   12   14   16
## [3,]   18   20   22   24
```

Data frames

A data frame is a structured format for storing tabular data. More formally in R, a data frame consists of a list of vectors of equal length.

For our purposes, data frames are the most important data structure (or type) in R. We will use them constantly. They have rows (usually units or observations) and columns (usually variables). There are also many functions designed to work especially (even exclusively) with data frames. Let's take a look at another built-in example dataset, `faithful` (note: you can read the help documentation on `faithful` to learn about the dataset):

```r
## This first command calls a dataset into my working "Global" environment and makes it available for s
data("faithful")

dim(faithful) # Returns the number of rows and columns. Often the first thing I do with any data frame
```

```
## [1] 272   2
```

```r
nrow(faithful)
```

```
## [1] 272
```

```r
ncol(faithful)
```

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

```r
names(faithful)  ## try colnames(faithful) too
```

```
## [1] "eruptions" "waiting"
```

```r
head(faithful) ## look at the first few rows of data
```

```
##   eruptions waiting
## 1     3.600      79
## 2     1.800      54
## 3     3.333      74
## 4     2.283      62
## 5     4.533      85
## 6     2.883      55
```

```r
summary(faithful)
```

```
##    eruptions        waiting
##  Min.   :1.600   Min.   :43.0
##  1st Qu.:2.163   1st Qu.:58.0
##  Median :4.000   Median :76.0
##  Mean   :3.488   Mean   :70.9
##  3rd Qu.:4.454   3rd Qu.:82.0
##  Max.   :5.100   Max.   :96.0
```

Some datasets are built-in to packages. Once you've installed the package (see the R Tutorial from Week 1 of the help documentation for `install.packages()`), you can call a dataset from that package by first

invoking the package

You can index into a data frame using numeric values or variable names. The notation uses square brackets again and requires you to remember the convention of `[<rows>, <columns>]`:

```
faithful[1,1] # The item in the first row of the first column
```

```
## [1] 3.6
```

```
faithful[,2] # all of the items in the second column
```

```
##   [1] 79 54 74 62 85 55 88 85 51 85 54 84 78 47 83 52 62 84 52 79 51 47 78 69 74
##  [26] 83 55 76 78 79 73 77 66 80 74 52 48 80 59 90 80 58 84 58 73 83 64 53 82 59
##  [51] 75 90 54 80 54 83 71 64 77 81 59 84 48 82 60 92 78 78 65 73 82 56 79 71 62
##  [76] 76 60 78 76 83 75 82 70 65 73 88 76 80 48 86 60 90 50 78 63 72 84 75 51 82
## [101] 62 88 49 83 81 47 84 52 86 81 75 59 89 79 59 81 50 85 59 87 53 69 77 56 88
## [126] 81 45 82 55 90 45 83 56 89 46 82 51 86 53 79 81 60 82 77 76 59 80 49 96 53
## [151] 77 77 65 81 71 70 81 93 53 89 45 86 58 78 66 76 63 88 52 93 49 57 77 68 81
## [176] 81 73 50 85 74 55 77 83 83 51 78 84 46 83 55 81 57 76 84 77 81 87 77 51 78
## [201] 60 82 91 53 78 46 77 84 49 83 71 80 49 75 64 76 53 94 55 76 50 82 54 75 78
## [226] 79 78 78 70 79 70 54 86 50 90 54 54 77 79 64 75 47 86 63 85 82 57 82 67 74
## [251] 54 83 73 73 88 80 71 83 56 79 78 84 58 83 43 60 75 81 46 90 46 74
```

```
faithful[10:20, 2] # ranges work too. This returns the 10-20th values of the second column.
```

```
##  [1] 85 54 84 78 47 83 52 62 84 52 79
```

```
faithful[37, "eruptions"] # The 37th value of the column called "eruptions."
```

```
## [1] 1.867
```

It is very useful to work with column (variable) names in a data frame using the `$` symbol:

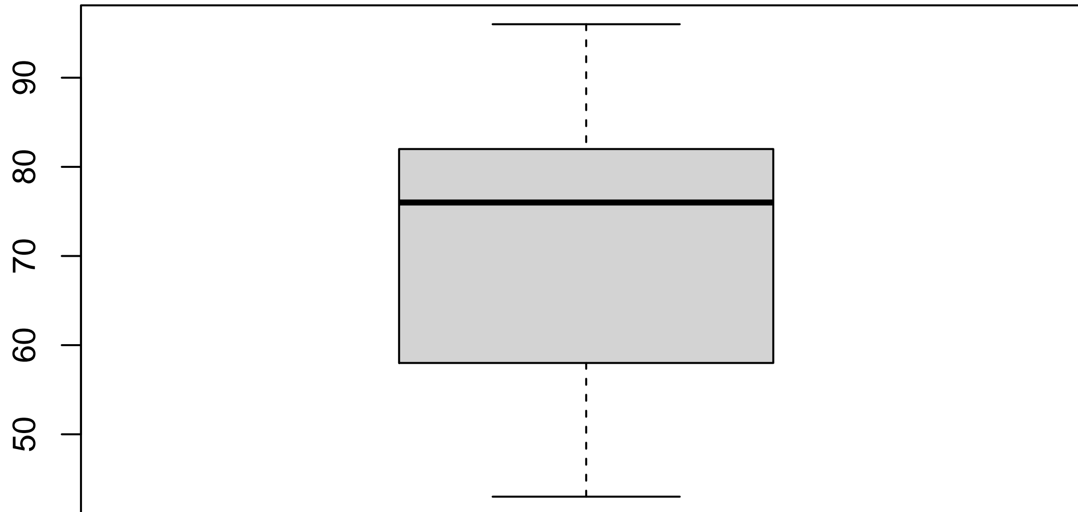
```
faithful$eruptions
```

```
##   [1] 3.600 1.800 3.333 2.283 4.533 2.883 4.700 3.600 1.950 4.350 1.833 3.917
##  [13] 4.200 1.750 4.700 2.167 1.750 4.800 1.600 4.250 1.800 1.750 3.450 3.067
##  [25] 4.533 3.600 1.967 4.083 3.850 4.433 4.300 4.467 3.367 4.033 3.833 2.017
##  [37] 1.867 4.833 1.833 4.783 4.350 1.883 4.567 1.750 4.533 3.317 3.833 2.100
##  [49] 4.633 2.000 4.800 4.716 1.833 4.833 1.733 4.883 3.717 1.667 4.567 4.317
##  [61] 2.233 4.500 1.750 4.800 1.817 4.400 4.167 4.700 2.067 4.700 4.033 1.967
##  [73] 4.500 4.000 1.983 5.067 2.017 4.567 3.883 3.600 4.133 4.333 4.100 2.633
##  [85] 4.067 4.933 3.950 4.517 2.167 4.000 2.200 4.333 1.867 4.817 1.833 4.300
##  [97] 4.667 3.750 1.867 4.900 2.483 4.367 2.100 4.500 4.050 1.867 4.700 1.783
## [109] 4.850 3.683 4.733 2.300 4.900 4.417 1.700 4.633 2.317 4.600 1.817 4.417
## [121] 2.617 4.067 4.250 1.967 4.600 3.767 1.917 4.500 2.267 4.650 1.867 4.167
## [133] 2.800 4.333 1.833 4.383 1.883 4.933 2.033 3.733 4.233 2.233 4.533 4.817
## [145] 4.333 1.983 4.633 2.017 5.100 1.800 5.033 4.000 2.400 4.600 3.567 4.000
## [157] 4.500 4.083 1.800 3.967 2.200 4.150 2.000 3.833 3.500 4.583 2.367 5.000
## [169] 1.933 4.617 1.917 2.083 4.583 3.333 4.167 4.333 4.500 2.417 4.000 4.167
## [181] 1.883 4.583 4.250 3.767 2.033 4.433 4.083 1.833 4.417 2.183 4.800 1.833
## [193] 4.800 4.100 3.966 4.233 3.500 4.366 2.250 4.667 2.100 4.350 4.133 1.867
## [205] 4.600 1.783 4.367 3.850 1.933 4.500 2.383 4.700 1.867 3.833 3.417 4.233
## [217] 2.400 4.800 2.000 4.150 1.867 4.267 1.750 4.483 4.000 4.117 4.083 4.267
## [229] 3.917 4.550 4.083 2.417 4.183 2.217 4.450 1.883 1.850 4.283 3.950 2.333
## [241] 4.150 2.350 4.933 2.900 4.583 3.833 2.083 4.367 2.133 4.350 2.200 4.450
## [253] 3.567 4.500 4.150 3.817 3.917 4.450 2.000 4.283 4.767 4.533 1.850 4.250
## [265] 1.983 2.250 4.750 4.117 2.150 4.417 1.817 4.467
```

```r
mean(faithful$waiting)
```

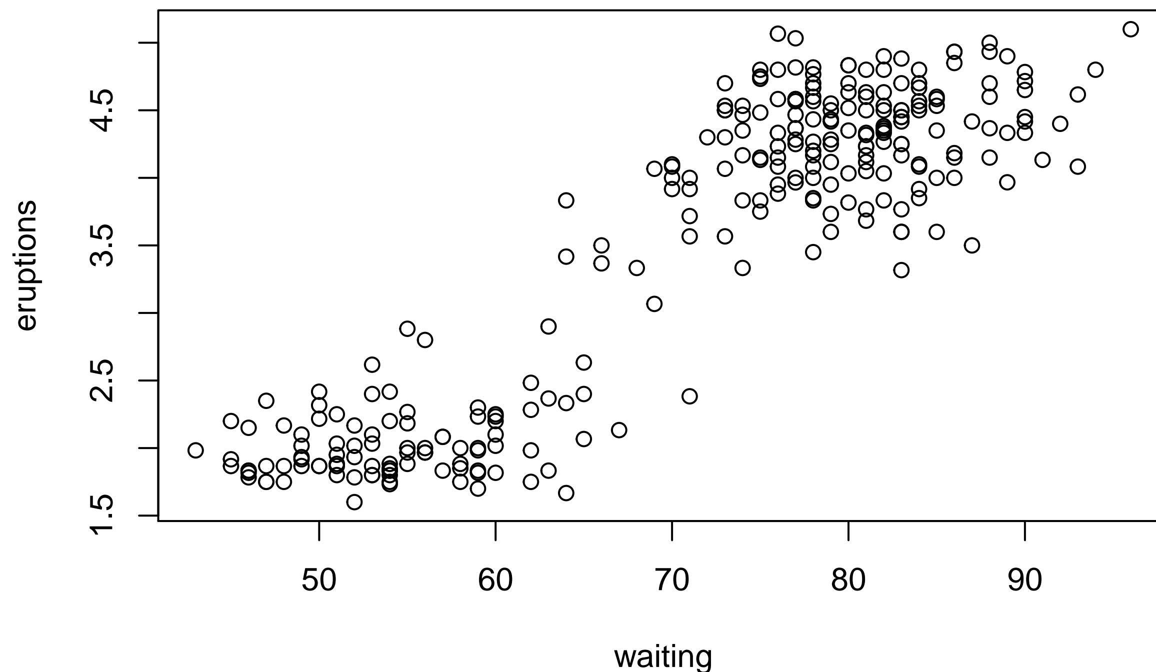
```
## [1] 70.89706
```

```r
boxplot(faithful$waiting)
```



Data frames are very useful for bivariate analyses (e.g., plots and tables). The base R notation for a bivariate presentation usually uses the ~ character. If both of the variables in your bivariate comparison are within the same data frame you can use the `data=` argument. For example, here is a scatterplot of eruption time (Y axis) over waiting time (X axis):

```r
plot(eruptions ~ waiting, data=faithful)
```



Data frames can have an arbitrary number of columns (variables). Another built in dataset used frequently in R documentation and examples is `mtcars` (read the help documentation! it contains a codebook that tells you about each variable). Let's look at that one next:

```r
data("mtcars")

dim(mtcars)
```

```
## [1] 32 11
```

```r
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

There are many ways to create and modify data frames. Here are some examples using the `mtcars` data. I create new vectors from the variables (columns) in the dataset, then I use the `data.frame` command to build a new data frame from the three vectors and do some data cleanup/recoding:

```r
my.mpg <- mtcars$mpg
my.cyl <- mtcars$cyl
my.disp <- mtcars$disp

df.small <- data.frame(my.mpg, my.cyl, my.disp)
class(df.small)
```

```
## [1] "data.frame"
```

```r
head(df.small)
```

```
##   my.mpg my.cyl my.disp
## 1   21.0      6     160
## 2   21.0      6     160
## 3   22.8      4     108
## 4   21.4      6     258
## 5   18.7      8     360
## 6   18.1      6     225
```

```r
# recode a value as missing
df.small[5,1] <- NA

# removing a column
df.small[,3] <- NULL
dim(df.small)
```

```
## [1] 32  2
```

```r
head(df.small)
```

```
##   my.mpg my.cyl
## 1   21.0      6
## 2   21.0      6
## 3   22.8      4
## 4   21.4      6
## 5     NA      8
## 6   18.1      6
```

Creating new variables, recoding, and transformations look very similar to working with vectors. Notice the

na.rm=TRUE argument I am passing to the mean function in the first line here:

```
df.small$mpg.big <- df.small$my.mpg > mean(df.small$my.mpg, na.rm=TRUE)

table(df.small$mpg.big)
```

```
##
## FALSE  TRUE
##    17    14
```

```
df.small$mpg.l <- log1p(df.small$my.mpg) # notice: log1p()
head(df.small$mpg.l)
```

```
## [1] 3.091042 3.091042 3.169686 3.109061       NA 2.949688
```

```
## convert a number into a factor:

df.small$my.cyl.factor <- factor(df.small$my.cyl)
summary(df.small$my.cyl.factor)
```

```
##  4  6  8
## 11  7 14
```

Some special functions are particularly useful for working with data frames:

```
is.na(df.small$my.mpg)
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
sum(is.na(df.small$my.mpg)) # sum() works in mysterious ways sometimes...
```

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

```
complete.cases(df.small)
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
sum(complete.cases(df.small))
```

```
## [1] 31
```

Vectorized operations: "Apply" functions and beyond

R has a lot of built-in functionality to support working on objects very quickly/efficiently in a "vectorized" way. The specifics of vectorization are beyond the scope of our course, but the significance is that while you may have learned to do things with "loops" in other programming languages (and we'll learn how to use them in R) vectorized functions in R almost always provide a faster/better way to achieve the same goals. As a result, I like to introduce vectorized functions first.

The reasons you might use a vectorized function are kind of simple: imagine you have an object (vector, dataframe, list, or whatever) and you want to perform the same operation over all of the rows or columns. Loops are good if you want to do this in a strict order (e.g., first row #1, then row #2, etc.), but most of the time the order doesn't matter. Vectorized functions allow R to apply operations over data objects in an arbitrary order, resulting in massively faster and less-verbose code.

Most of the base R versions of these functions have "apply" in the name. There are also Tidyverse alternatives. I will stick to the base R versions here. Please feel free to read about and use the alternatives! I'll try to

introduce and document them a little later on in our course.

Let's start with an example using the `mtcars` dataset again. The `sapply()` and `lapply()` functions both "apply" the second argument (a function) iteratively to the items (variables) in the first argument. The differences emerge in the structure of the output: `sapply()` returns a (usually more user-friendly) vector or matrix by default whereas `lapply()` returns a list:

```
sapply(mtcars, quantile)
```

```
##         mpg cyl    disp    hp  drat      wt    qsec vs am gear carb
## 0%   10.400   4  71.100  52.0 2.760 1.51300 14.5000  0  0    3    1
## 25%  15.425   4 120.825  96.5 3.080 2.58125 16.8925  0  0    3    2
## 50%  19.200   6 196.300 123.0 3.695 3.32500 17.7100  0  0    4    2
## 75%  22.800   8 326.000 180.0 3.920 3.61000 18.9000  1  1    4    4
## 100% 33.900   8 472.000 335.0 4.930 5.42400 22.9000  1  1    5    8
```

```
lapply(mtcars, quantile) # Same output, different format/class
```

```
## $mpg
##     0%    25%    50%    75%   100%
## 10.400 15.425 19.200 22.800 33.900
##
## $cyl
##   0%  25%  50%  75% 100%
##    4    4    6    8    8
##
## $disp
##      0%     25%     50%     75%    100%
##  71.100 120.825 196.300 326.000 472.000
##
## $hp
##    0%   25%   50%   75%  100%
##  52.0  96.5 123.0 180.0 335.0
##
## $drat
##    0%   25%   50%   75%  100%
## 2.760 3.080 3.695 3.920 4.930
##
## $wt
##      0%     25%     50%     75%    100%
## 1.51300 2.58125 3.32500 3.61000 5.42400
##
## $qsec
##      0%     25%     50%     75%    100%
## 14.5000 16.8925 17.7100 18.9000 22.9000
##
## $vs
##   0%  25%  50%  75% 100%
##    0    0    0    1    1
##
## $am
##   0%  25%  50%  75% 100%
##    0    0    0    1    1
##
## $gear
##   0%  25%  50%  75% 100%
```

```
##    3    3    4    4    5
##
## $carb
##   0%  25%  50%  75% 100%
##    1    2    2    4    8
```

Experiment with that idea on your own a little bit before moving on. For example, can you find the mean of each variable in the `mtcars` data using either `sapply` or `lapply`?

The `tapply` function allows you to apply functions conditionally. For example, the chunk below finds the mean gas mileage by number of cylinders. The second argument (`mtcars$cyl`) provides an index into the first (`mtcars$mpg`) before the third argument (`mean`) is applied to each of the conditional subsets:

```
tapply(mtcars$mpg, mtcars$cyl, mean)
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

Try some other calculations using `tapply()`. Can you calculate the average engine discplacement conditional on number of cylinders? What about the average miles per gallon conditional on whether the car has an automatic transmission?

Note that `apply()` works pretty smoothly with matrices, but it can be a bit complicated/surprising otherwise.

Some basic graphs with ggplot2

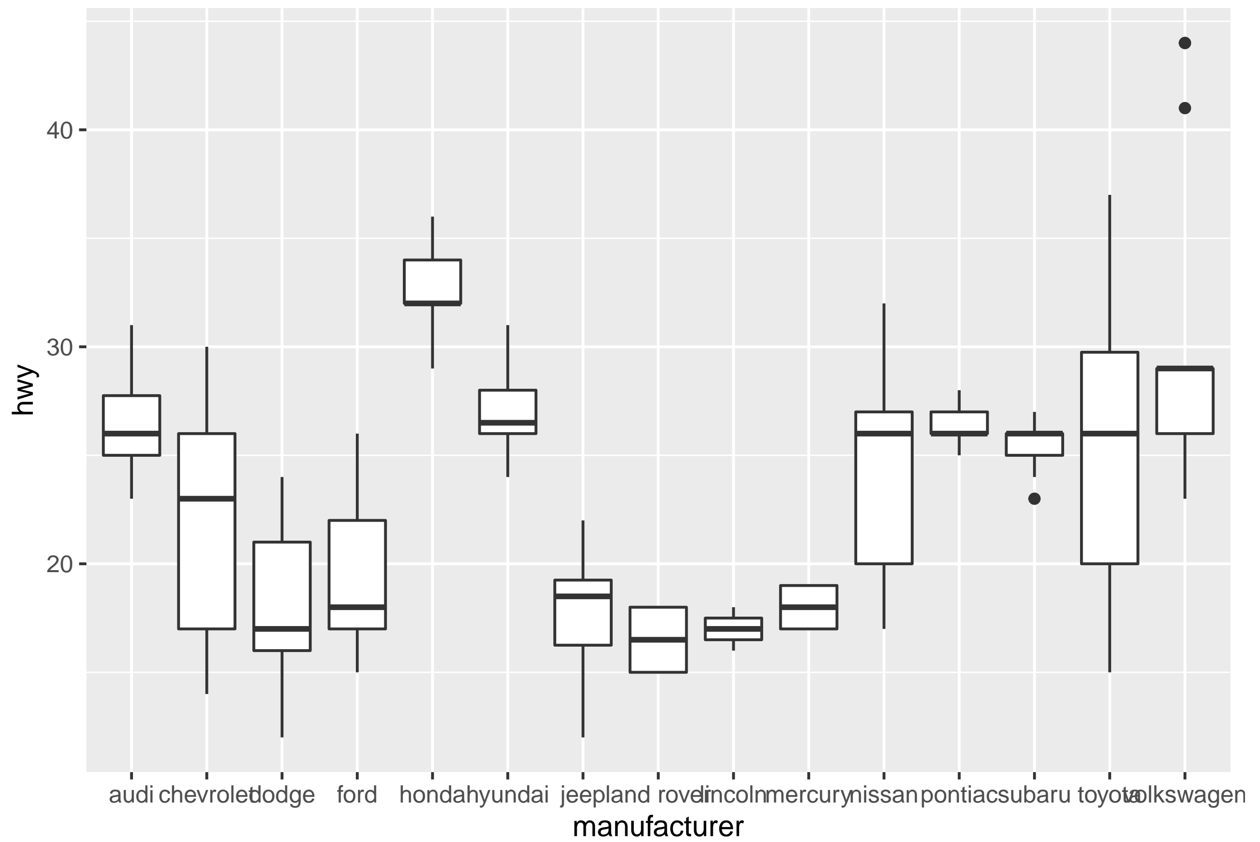
ggplot2 is what I like to use for plotting so I'll develop examples with it from here on out.

Make sure you've installed the package with `install.packages("ggplot2")` and load it with `library(ggplot2)`.

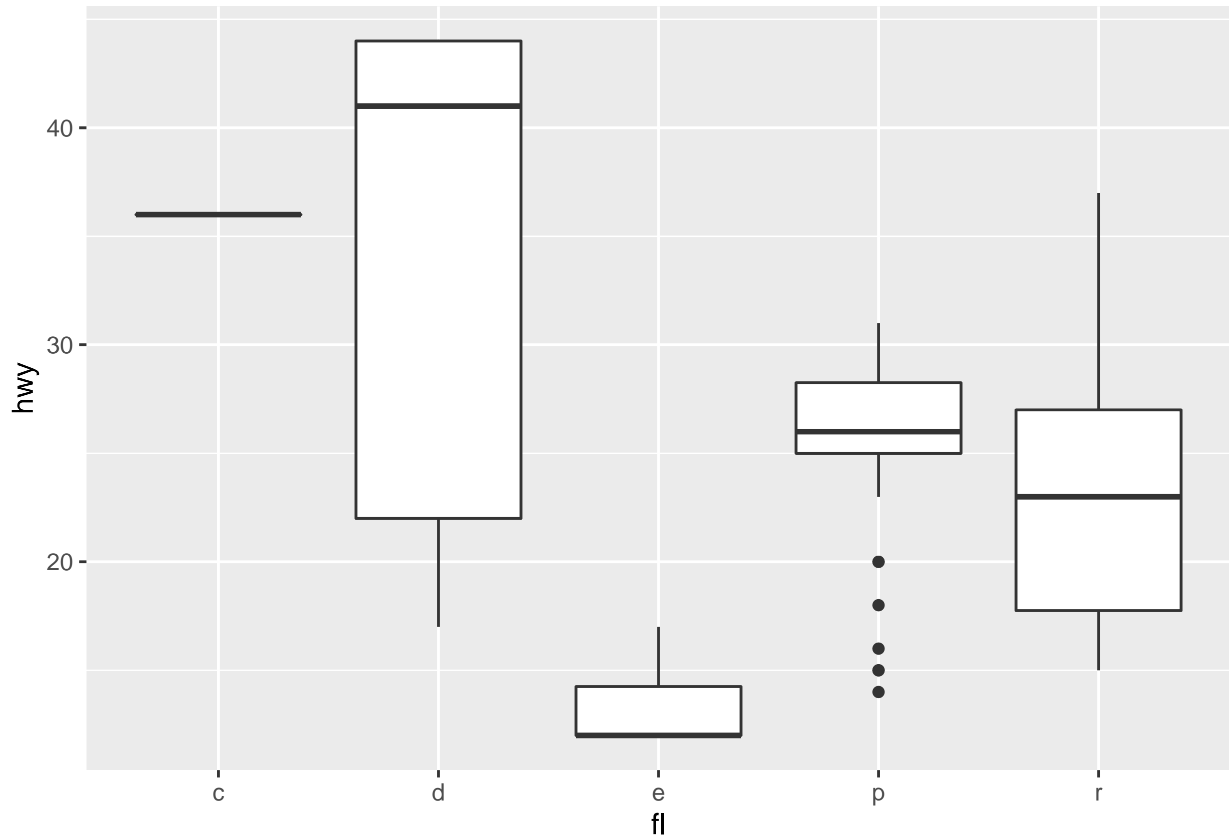
There is another built-in (automotive) dataset that comes along with the ggplot2 package called `mpg`. This dataset includes variables of several types and is used in much of the package documentation, so it's helpful to become familiar with it.

I'll develop a few simple examples below. For more, please take a look at the (extensive) ggplot2 documentation. There are **many** options and arguments for most of these functions and many more functions to help you produce publication-ready graphics. Chapter 3 of the Healy book is also an extraordinary resource for getting started creating visualizations with ggplot2.

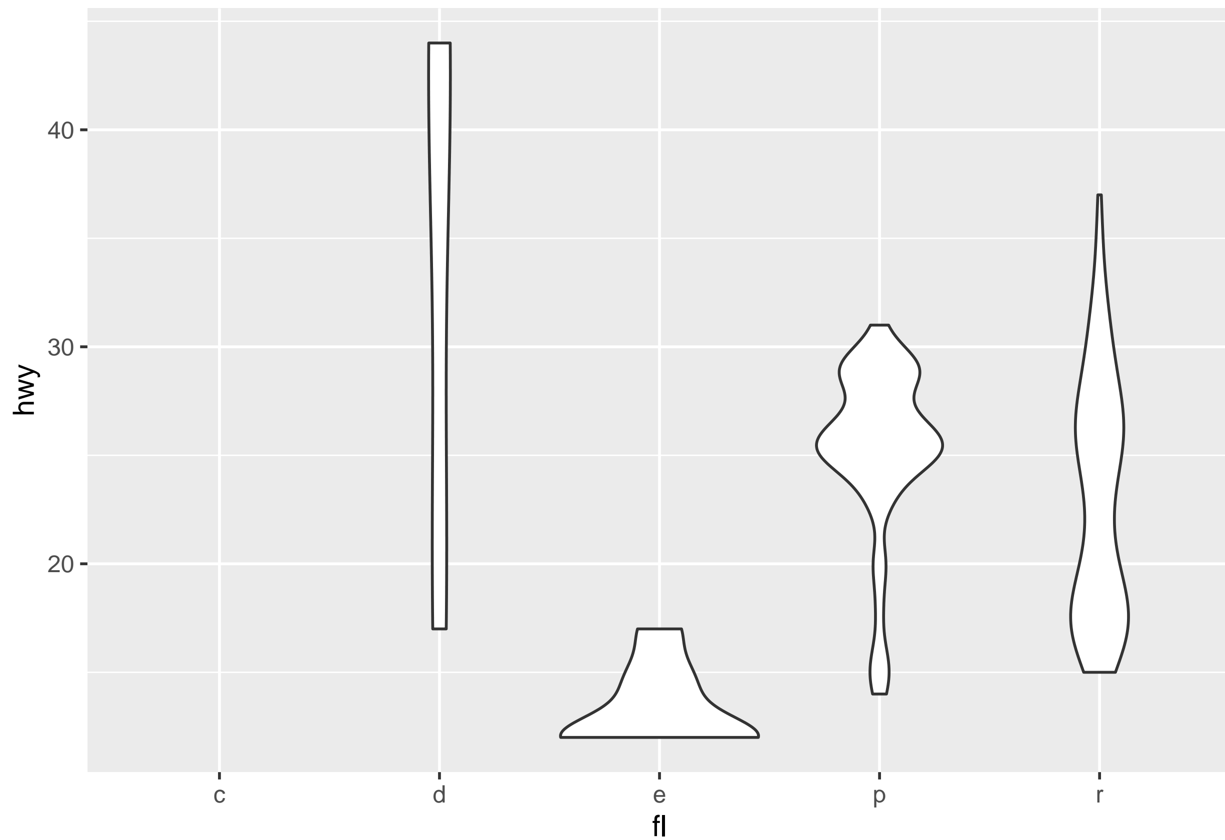
```
library(ggplot2)
data("mpg")

# First thing, call ggplot() to start building up a plot
# aes() indicates which variables to use as "aesthetic" mappings

p <- ggplot(data=mpg, aes(manufacturer, hwy))

p + geom_boxplot()
```




```
# another relationship:
p <- ggplot(data=mpg, aes(fl, hwy))
p + geom_boxplot()
```

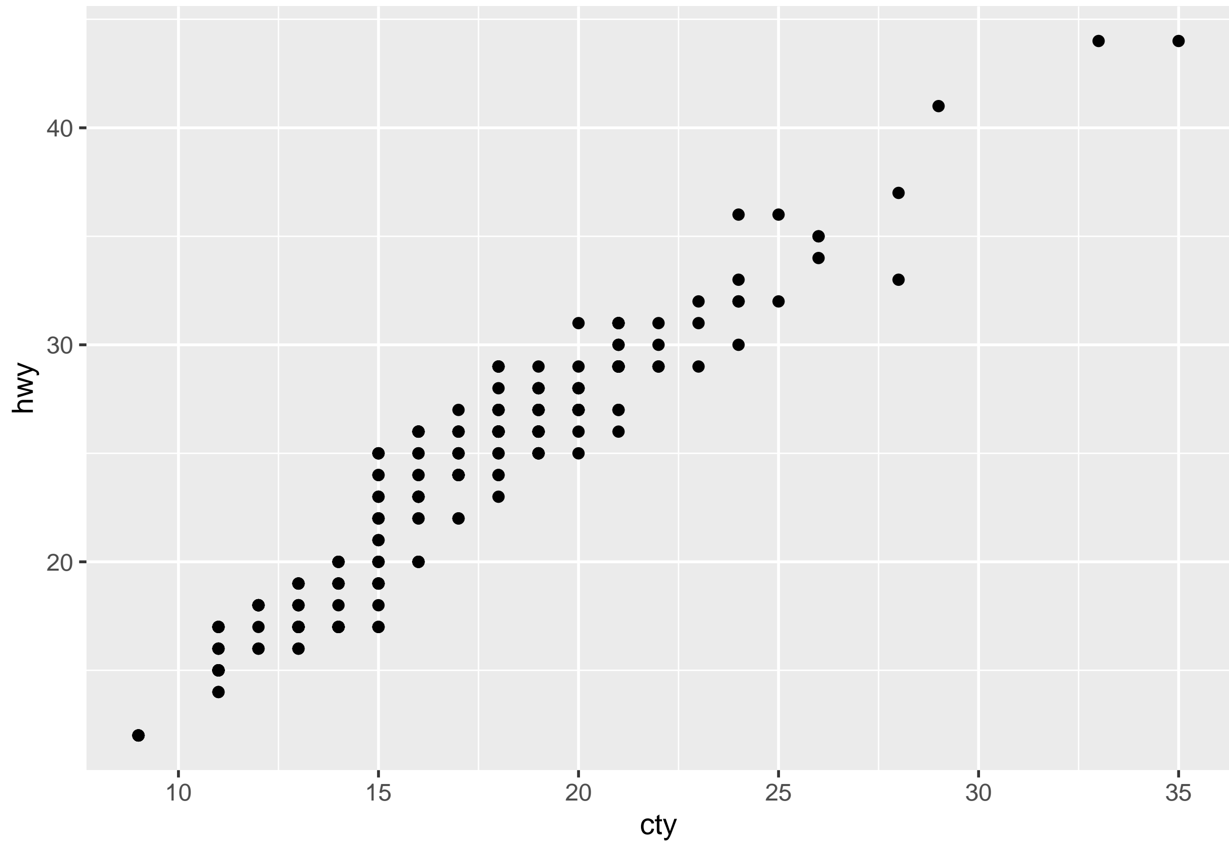



```
p + geom_violin()
```

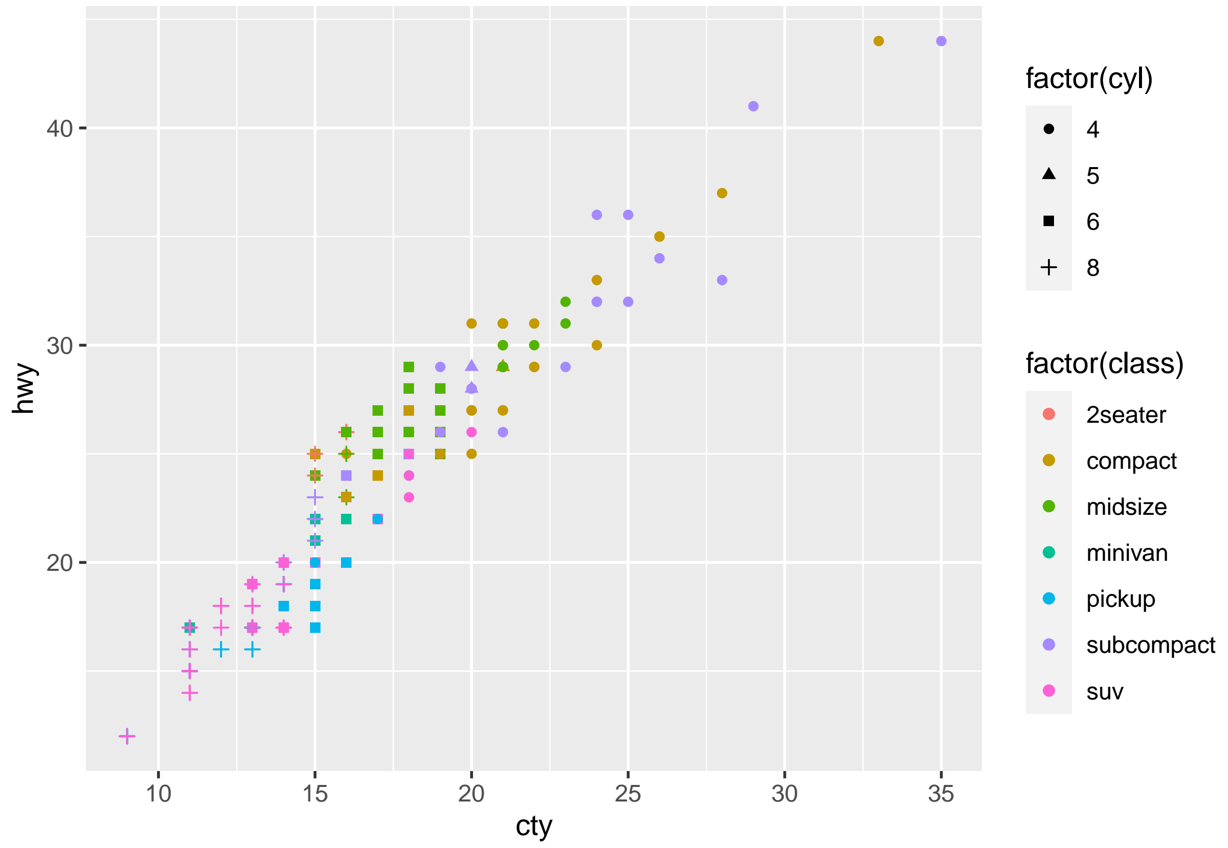


Here's another that visualizes the relationship between miles per gallon (mpg) in the city vs. mpg on the highway:

```
p <- ggplot(data=mpg, aes(cty, hwy))

p+geom_point()
```




```
# Multivariate graphical displays can get pretty wild
p + geom_point(aes(color=factor(class), shape=factor(cyl)))
```


factor(cyl)
4
5
6
8
factor(class)
2seater
compact
midsize
minivan
pickup
subcompact
suv
hwy
cty
40
30
20
10
15
20
25
30
35