Introduction to R

Víctor Peña

Contents

What do you need to download?

You'll need:

- `R`, which you can download by clicking here (or going to `https://www.r-project.org/`)
- `RStudio`, which you can download clicking here (or going to `https://www.rstudio.com/`).

The official `Rstudio` cheat sheet (download here) might be useful, especially if you want to learn some shortcuts that will help you write code faster. You should install `R` first, and then install `RStudio`.

Creating variables

We can create variables by assigning them values:

```
firstvariable = 0
secondvariable = 10
thirdvariable = TRUE
fourthvariable = "hello"
```

If we want to print a variable, we can type its name. For example:

```
fourthvariable
```

```
## [1] "hello"
```

The names of the variables are case sensitive. For example, if we try to print `Fourthvariable`, we'll get an error because `R` doesn't recognize it (try it yourself).

With `R`, we can also assign values with `<-` instead of `=`. For example, the following code is equivalent to the first chunk of code:

```
firstvariable <- 0
secondvariable <- 10
thirdvariable <- TRUE
fourthvariable <- "hello"
```

There isn't any practical difference between using one or the other. I tend to use `=` because it's less work.

We can comment code writing `#` followed by some text. For example,

```
# The code below creates a variable named greetings
greetings = "hello"
```

As we learn more `R`, we'll write somewhat complicated code. It's good practice to add comments before steps that aren't obvious (so that you don't forget what the code does next time you look at it).

We can rewrite variables. For example, if we type

```
firstvariable = 3
```

the previous value of `firstvariable` has been overwritten. Now it's equal to 3:

```
firstvariable
```

```
## [1] 3
```

We can see the "type" of the variables using the function `class`:

```
class(firstvariable)
```

```
## [1] "numeric"
```

```
class(thirdvariable)
```

```
## [1] "logical"
```

```
class(fourthvariable)
```

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

The names of the types of these variables are intuitive, but here's an explanation:

- `numeric` variables can take on numerical values.
- `logical` variables can take on the values `TRUE` and `FALSE`. The values `TRUE` and `FALSE` can be abbreviated to `T` and `F`.
- `character` variables are… characters. The characters can be numbers. If we want to define variables of type `character`, we need to type their values between quotation marks. For example, `var = "3"` creates a variable `var` whose class is `character`. An equivalent alternative is using single quotes. For example, we could have also written `var = '3'`. There is no difference. I tend to use "double" quotes, but sometimes I might use single quotes instead.

There are other types of variables and we'll see them as we learn more `R`.

Exercise What are the types of the following variables?

- `var1 = 1000`
- `var2 = 1e3`
- `var3 = "1000"`
- `var4 = "1e3"`
- `var5 = "T"`
- `var6 = FALSE`

Operations with variables

We can add, subtract, multiply, divide, and exponentiate `numeric` variables:

```
firstvariable+secondvariable
```

```
## [1] 13
```

```
firstvariable-secondvariable
```

```
## [1] -7
```

```
firstvariable*secondvariable
```

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

```
firstvariable^secondvariable # exponentiation
```

```
## [1] 59049
```

We can combine operations. For example, we can compute the average of `firstvariable` and `secondvariable` as

```
(firstvariable+secondvariable)/2
```

```
## [1] 6.5
```

`R` has a built-in `mean` function, which we'll see later.

We can't add, subtract, multiply, divide or exponentiate `character` variables (try it out: it'll give you an error), but we can add, subtract, multiply, divide or exponentiate `logical` variables. If the variable is `TRUE` it'll be treated as a `1`; if it's `FALSE`, it'll be treated as a `0`:

```
logi1 = T
logi2 = F
```

```
logi1+logi2
```

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

```
logi1*logi2
```

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

```
logi1/logi2
```

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

```
logi1^logi2
```

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

We can combine `logical` and `numeric` variables in operations. Again, `TRUE` will be treated as a `1`, and `FALSE` will be treated as a `0`.

Exercise Let `var1 = 3`, `var2 = -3`, `var3 = 2`, `var4 = TRUE`, and `var5 = "0"`. Find the values of the following operations. Try to guess what the values will be before trying in `R`, just to make sure you understand the process.

- `(var1 + var3*var4)/4`
- `var3^var1+var5^2`
- `var1^0+var2^0`

We can also do operations without using any variables at all. For example,

```
6/2*(2+1)
```

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

Exercise Find the value of the following expressions:

- $10^3 - 3 \cdot 4$

- $\frac{(6+4)\cdot 3}{2}$
- $\frac{3-4}{2+3}$
- Think about what `6/2*(2+1+TRUE)` should be without using `R`. Then, try what `R` gives you and compare.

Arithmetic functions

R has built-in functions such as `sqrt`, `exp`, `log`, etc.

```
sqrt(4)
```

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

```
exp(firstvariable)
```

```
## [1] 20.08554
```

```
log(10, base=2)
```

```
## [1] 3.321928
```

If you're not sure how a function works, you can ask for help by writing `?` before the name of the function.

The number π is "pre-defined" as a variable. That is,

```
pi
```

```
## [1] 3.141593
```

Note that we never defined a variable called `pi`, but the code above still worked.

Exercise Let `var1 = TRUE`, `var2 = 1e3`, `var3 = pi`, `var4 = -3`. Find the values of the operations below.

- $\frac{(\exp(\text{var1})+\sqrt{\text{var3}})^2}{\text{var4}+\text{var3}}$
- $3 - \frac{\text{var2}-\text{var4}}{3+\text{TRUE}}$
- $\frac{\log_{10}(\text{var2})}{\text{var1}}$
- $\exp(\sqrt{-1}\pi) + 1$

Vectors

We can define vectors as follows:

```
x1 = c(1, 2, 3, 4, 5, 6)
y1 = c("a","b","c","d","efg")
z1 = c("a", 2, 3, "e")
```

We can print them by writing their name. For example,

```
z1
```

```
## [1] "a" "2" "3" "e"
```

We can find the types of their entries with `class`, just as we did with variables that only had one entry:

```
class(x1)
```

```
## [1] "numeric"
```

```
class(y1)
```

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

```
class(z1)
```

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

Note that if a vector has mixed `numeric` and `character` entries, it gets saved as `character` (see `z1`).

We can create ranges of values with `:`

```
1:10
```

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

```
10:6
```

```
## [1] 10  9  8  7  6
```

We can also find the length of a vector:

```
length(z1)
```

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

Operations with `numeric` vectors

We can add, multiply, divide, etc. all the components of a `numeric` vector by the same number:

```
x1+5
```

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

```
x1*5
```

```
## [1]  5 10 15 20 25 30
```

```
x1/5
```

```
## [1] 0.2 0.4 0.6 0.8 1.0 1.2
```

We can add and subtract vectors of the same length:

```
x2 = c(7, 8, 9, 10, 11, 12)
x1+x2
```

```
## [1]  8 10 12 14 16 18
```

```
x1-x2
```

```
## [1] -6 -6 -6 -6 -6 -6
```

Similarly, we can do **componentwise** multiplication and division:

```
x1*x2
```

```
## [1]  7 16 27 40 55 72
```

```
x1/x2
```

```
## [1] 0.1428571 0.2500000 0.3333333 0.4000000 0.4545455 0.5000000
```

If two vectors are of different lengths, we have to be careful! `R` won't give us a warning message:

```
x1
```

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

```r
x3 = c(2,3)
x1+x3
```

```
## [1] 3 5 5 7 7 9
```

```r
x1*x3
```

```
## [1]  2  6  6 12 10 18
```

We can compute means, standard deviations, variances, etc:

```r
mean(x1)
```

```
## [1] 3.5
```

```r
sd(x1)
```

```
## [1] 1.870829
```

```r
var(x1)
```

```
## [1] 3.5
```

We can also take the sum or the product of the elements of a `numeric` vector:

```r
sum(x1)
```

```
## [1] 21
```

```r
prod(x1)
```

```
## [1] 720
```

Exercises

- What is the value of the sum $1 + 2 + 3 + \cdots + 2019$?
- What about the product $1 \cdot 2 \cdot 3 \cdot \cdots \cdot 10$? [It's not part of the (graded) question, but try computing $1 \cdot 2 \cdot 3 \cdot \cdots \cdot 2019$. What do you get?]
- The final grade of a course is equal to the average of 3 assignments. A student got a 100% in 2 of the tests and didn't show up for the last one. What is her final grade? (in grade %).
- Consider the same setup that we had in the previous question, but now assume that the instructor computes the final grade using the geometric mean (`https://en.wikipedia.org/wiki/Geometric_mean`) instead. What is her final grade?
- Find the usual (arithmetic) average, the geometric mean, and the standard deviation of the grades of the following 3 students in the class. Alice got a 75 in the first, second, and third exam. Bob got a 100 in the first one, another perfect 100 in the second exam, but he tanked the third and got only a 25. Finally, Carol started strong with a 93, but then lost steam. In the second exam, she got a 70, and in the last exam, she got a 63. Comment on the results. [To be discussed in class.]
- Let `x = 1:10` and `y = 11:20`. What is the value that you get after applying the `R` equivalent of the function `SUMPRODUCT` in Excel?

Concatenating

We can add new entries in an existing vector as follows:

```r
x1
```

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

```r
c(x1,10) # add at the end
```

```
## [1]  1  2  3  4  5  6 10
```

```r
c(10, x1) # add at the beginning
```

```
## [1] 10  1  2  3  4  5  6
```

We can concatenate vectors, too:

```r
c(x1, x2)
```

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

Indexing

We can look at particular entries of a vector using brackets.

```r
x1
```

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

```r
x1[1] # first entry
```

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

```r
x1[4] # 4th entry
```

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

```r
x1[length(x1)] # last entry
```

```
## [1] 6
```

In `R`, indices start at `1` (in some other programming languages, indices start at `0`).

We can access subsets of vectors using vectors. For example, if we want to print the third and fifth entries of `x1`:

```r
x1[c(3,5)]
```

```
## [1] 3 5
```

We can subset using ranges of values with `:`. For instance, if we want to select the second, third, fourth, and fifth entries of `x1`:

```r
x1[2:5]
```

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

We can also index by excluding certain observations. For example, if we want a vector that contains all the components in `x1` except the first one, we can write

```r
x1[-1]
```

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

This trick can also be used with vectors and ranges:

```r
x1[-c(3,5)]
```

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

```r
x1[-(2:5)]
```

```
## [1] 1 6
```

Exercise This exercise is not on *TopHat*, but please do it. Otherwise, you might get stuck further on.

- Create a vector that contains the second, fourth, and sixth entries of `x1 = c(1,2,3,1,2,5,2,2,2)`.
- Create a vector that contains all but the second, fourth, and sixth entries of `x1 = c(1,2,3,1,2,5,2,2,2)`.
- Let `x = 1:50` and `y = 51:100`. Create a new vector `z` that concatenates the entries of `x` and `y`. Then, create a vector that contains the even entries of `z` and another one that contains the odd entries of `z`.

We can use indexing to modify certain entries of a vector. For example, consider the vector

```
x = c("a", "b", "c", "e")
```

If we want to modify the fourth entry so that it's `d` instead of `e`:

```
x[4] = "d"
```

Now, let's print `x`:

```
x
```

```
## [1] "a" "b" "c" "d"
```

Matrices

Matrices are "boxes" that can contain `numeric`, `logical`, or `character` entries.

We can create matrices as follows:

```
A1 = matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=TRUE) # read by row
A2 = matrix(c(1,3,2,4), nrow=2, ncol=2, byrow=FALSE) # read by column
```

And we can print them by typing in their names (as usual):

```
A1
```

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

```
A2
```

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

If `byrow` isn't specified, the default is `byrow = FALSE`:

```
A3 = matrix(c(TRUE, FALSE, TRUE, FALSE), nrow = 2, ncol = 2)
A3
```

```
##       [,1]  [,2]
## [1,]  TRUE  TRUE
## [2,] FALSE FALSE
```

Doing operations with matrices is straightforward:

```
A1*A2 # componentwise product
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    9   16
```

```r
A1+A2  # componentwise addition
```

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

```r
log(A1) # taking the log of the components
```

```
##          [,1]      [,2]
## [1,] 0.000000 0.6931472
## [2,] 1.098612 1.3862944
```

```r
7*A1 # multiply all of A1 by 7
```

```
##      [,1] [,2]
## [1,]    7   14
## [2,]   21   28
```

Indexing matrices is similar to indexing vectors. For example, if we want to access the element in the first row and second column of `A1`:

```r
A1[1,2] # accessing entries: rows first, then columns
```

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

We can also index by full rows and / or columns of matrices. For example, if we want to access the first row of `A1`:

```r
A1[1,]
```

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

If we want to access the second column:

```r
A1[,2]
```

```
## [1] 2 4
```

We can also access subsets of rows and columns matrices. For example, let

```r
B = matrix(c(1:9), nrow = 3, ncol=3)
```

We can access the first two rows and columns as follows

```r
B[1:2,1:2]
```

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

We can name the rows and columns of matrices. For example if we want the rows of `B` to be called `R1`, `R2`, and `R3` and the columns of `B` to be named `C1`, `C2`, `C3`, we can type

```r
rownames(B) = c("R1", "R2", "R3")
colnames(B) = c("C1", "C2", "C3")
```

And then print

```r
B
```

```
##    C1 C2 C3
## R1  1  4  7
## R2  2  5  8
## R3  3  6  9
```

The functions `colSums` and `rowSums` give us column sums and row sums of matrices, which is often useful. For example:

```r
rowSums(B)
```

```
## R1 R2 R3
## 12 15 18
```

```r
colSums(B)
```

```
## C1 C2 C3
##  6 15 24
```

We can also apply the function `sum` if we want to add up all the numbers in the matrix:

```r
sum(B)
```

```
## [1] 45
```

Exercise

Bob is on a health kick and is keeping track of the macronutrients and calories in what he eats. Yesterday, he ate

- Breakfast: 50g of carbs, 8g of fat, and 20g of protein
- Lunch: 60g of carbs, 30g of fat, and 40g of protein
- Dinner: 40g of carbs, 30g of fat, 40g of protein

Create a matrix that combines the information above. Each meal should be in a different row and the columns should contain the grams of carbs, fat, and protein in the meals. The `rownames` of the matrix should be `breakfast`, `lunch`, and `dinner`. The `colnames` should be `carbs`, `fat`, and `protein`. Once you've done that, use `R` to answer the following questions:

- How many grams of carbs, fat, and protein did Bob eat yesterday? (give them separately; your answer should look something like 50g of carbs, 30g of fat, 20g of protein, but with different numbers)
- Assume that each gram of carbs yields 4 calories, each gram of protein yields 4 calories, and each gram of fat yields 9 calories. How many calories did Bob eat for breakfast, lunch, and dinner yesterday? (give them separately; your answer should look something like 200 calories for breakfast, 500 calories for lunch, and 300 for dinner, but with different numbers) How many calories did he eat in total? Did he stay under his goal of 1800 calories per day?
- What percentage of the calories he ate yesterday come from carbs, protein, and fat, respectively? (give them separately; your answer should look something like 100 cal from carbs, 300 cal from fat, 300 cal from protein, but potentially with different numbers). He is trying to follow the so-called 40/30/30 diet, where 40 percent of the calories eaten should come from carbs, 30 percent from protein, and 30 percent from fat. Is he close to his goal? If not, suggest how he could get closer

We can add rows and columns to a `matrix` using `rbind` and `cbind`, respectively.

Let's define 2 matrices, `A` and `B`:

```r
A = matrix(c(1,2,3,4), nrow=2, ncol=2)
B = matrix(c(5,6,7,8), nrow=2, ncol=2)
```

Let's use `rbind`:

```r
rbind(A,B)
```

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

```
## [3,]    5    7
## [4,]    6    8
```

And `cbind`:

```
cbind(A,B)
```

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

As you can imagine, order matters. Try out `rbind(B,A)` and check that you get the output that you'd expect.

Installing libraries

Statisticians use `R` because there are many libraries that contain useful functions. We can install libraries with `install.packages`. For example, if we want to install `ggplot2`, which is a useful library for plotting:

```
install.packages("ggplot2")
```

Once the library is installed, we can load it using `library()`. If we want to load `ggplot2`, we need to type:

```
library(ggplot2)
```

Introduction to `data.frames`

We'll use the dataset `iris`, which is in the built-in library `datasets`. First, we load it:

```
data(iris)
```

You can get information about the dataset by typing `?iris`.

The class of the dataset is `data.frame` (and others), which are matrices that have columns that can have different types.

The function `str` gives us some information about the variables in the dataset:

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

We have 150 observtions and 5 variables. The first 4 variables are `numerical` and `Species` is a `factor`. We'll cover `factors` in the next section.

You can get a quick summary of the data with

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300  
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
```

```
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

We'll come back to this later.

We can print the first and last 5 observations in the dataset using `head` and `tail`:

```r
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
tail(iris)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

We can index the rows and columns of `mpg` using the same syntax we used for indexing matrices:

```r
iris[3:7,c(1,4:5)]
```

```
##   Sepal.Length Petal.Width Species
## 3          4.7         0.2  setosa
## 4          4.6         0.2  setosa
## 5          5.0         0.2  setosa
## 6          5.4         0.4  setosa
## 7          4.6         0.3  setosa
```

With `data.frame`s, we can extract variables using `$` followed by their name. For example, if we want to create a variable named `spec` that contains the column of `iris` which has named `Species`:

```r
spec = iris$Species
```

We can also index by logical conditions. For instance, if we want to work with the subset of the data where `Species` is equal to `setosa`, we can save it in a subset with the following command:

```r
setosa = iris[iris$Species == "setosa",]
```

Notice that we used `==` instead of a single `=`. We'll learn more about data-subsetting in the next chapter.

We can print the first few observations with `head`:

```r
head(setosa)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

```
## 1          5.1         3.5         1.4         0.2  setosa
## 2          4.9         3.0         1.4         0.2  setosa
## 3          4.7         3.2         1.3         0.2  setosa
## 4          4.6         3.1         1.5         0.2  setosa
## 5          5.0         3.6         1.4         0.2  setosa
## 6          5.4         3.9         1.7         0.4  setosa
```

Exercise Not on *Top Hat*, but I strongly recommend doing it.

- Create a new `data.frame` named `color` that contains all the rows in `iris` so that `Species` is equal to `versicolor`.
- Create a new `data.frame` named `first100` that contains the first 100 rows of `iris`.
- Create a new `data.frame` named `Petal` that only contains the variables `Petal.Length` and `Petal.Width`.

Creating `data.frames` from scratch is straightforward. For example,

```
df = data.frame(var1 = c(1, 2, 3), var2 = c("A","B","C"))
```

creates a `data.frame` with two columns named `var1` and `var2` which contain the `vectors` `c(1,2,3)` and `c("A","B","C")`. You can create a `data.frame` from existing vectors as well. For example, the following chunk of code is equivalent to the previous one

```
var1 = c(1, 2, 3)
var2 = c("A","B","C")
df = data.frame(var1, var2)
```

The rows and columns of `data.frames` can be named using `rownames` and `colnames` (just as we did with `matrix`-type variables).

We can easily add new variables to an existing `data.frame`. For example, we can add the variable

```
var3 = c("X", "Y", "Z")
```

By writing

```
df$var3 = var3
```

We could've also skipped a step and written

```
df$var3 = c("X", "Y", "Z")
```

Another alternative is using `cbind`, which we saw when we were working with matrices. The function `rbind` works with `data.frames` as well.

Factors

`factor` is a variable type in `R` useful for encoding categorical variables. In the `iris` dataset, `species` is a `factor`. Variables of type `character` and `factor` are (conceptually similar) but they have different properties. When we work with `data.frames`, we'll work with factors. When we work with matrices, we'll work with `character`-type variables. This can be a bit of a headache because, sometimes, we want to convert `matrix` objects to `data.frames`. We'll see some of that later in the semester.

Defining a factor from scratch is easy:

```
fac1 = factor(c("dog","cat","cat","dog"))
```

We can use `summary` to create a quick table:

```r
summary(fac1)
```

```
## cat dog
##   2   2
```

Exercise

In the `iris` dataset, how many species of type `setosa` are there?

We can see the different categories (in `R` lingo, levels) of a factor and its ordering using `levels`:

```r
levels(fac1)
```

```
## [1] "cat" "dog"
```

The default ordering of the categories in a factor is alphabetic, which isn't always the best or most intutive. Let's use a dataset named `hsb2` to illustrate this point.

```r
hsb2  = read.csv("http://vicpena.github.io/sta9750/spring19/hsb2.csv")
```

The dataset contains a variable called `ses`, which is socioeconomic status of the student. It can take on the values `low`, `middle`, and `high`. Unfortunately, the default ordering of the factor is alphabetical, that is:

```r
levels(hsb2$ses)
```

```
## [1] "high"   "low"    "middle"
```

The problem with this ordering is that if we create tables, plots, etc. `R` will use this ordering, which is counterintuitive. For instance, if we create a 2 x 2 table of `ses` and `race`, we get

```r
table(hsb2$ses, hsb2$race)
```

```
##
##          african american asian hispanic white
##   high                  3     3        4    48
##   low                  11     3        9    24
##   middle                6     5       11    73
```

This is not great.

How can we reorder the levels of a factor? The answer is

```r
hsb2$ses = factor(hsb2$ses, levels = c("low", "middle", "high"))
```

The code above rewrites the `ses` variable in `hsb2` to an ordered factor whose levels are `low`, `middle`, and `high` (in that order).

Here's the code to verify that `ses` is now ordered:

```r
levels(hsb2$ses)
```

```
## [1] "low"    "middle" "high"
```

```r
table(hsb2$ses, hsb2$race)
```

```
##
##          african american asian hispanic white
##   low                  11     3        9    24
##   middle                6     5       11    73
##   high                  3     3        4    48
```

Exercise Not on *TopHat*.

Create a factor which takes on the values `bad`, `good`, `mediocre`. Reorder it so that the levels are sorted as `bad`, `mediocre`, `good`.

Lists

We won't say much about `lists`, but they're useful if we want to keep objects of different types in a single place.

For example, suppose that we have a `vector` and a `matrix`:

```
v = 1:6
m = matrix(c(1,0,0,1),byrow=T,nrow=2)
```

Then, the following code creates a `list` whose entries are the vector `v` and the matrix `m`:

```
l = list(v,m)
```

We can access, say, the second element of the list with

```
l[[2]]
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

And we can do things such as

```
l[[2]][2,1]
```

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

```
l[[1]][4]
```

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

We can add a new element to the list indexing by a new element

```
v2 = 3:4
l[[3]] = v2
```

We probably won't see `lists` again in the course, but it's good to know that they exist.

Basic data summaries with `R`

In this section, we'll use the `mpg` dataset in `library(ggplot2)`.

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

We can get quick summaries of `numeric` variables with `summary`.

```
summary(mpg)
```

```
##  manufacturer          model               displ            year     
##  Length:234         Length:234         Min.   :1.600   Min.   :1999  
##  Class :character   Class :character   1st Qu.:2.400   1st Qu.:1999  
##  Mode  :character   Mode  :character   Median :3.300   Median :2004  
##                                        Mean   :3.472   Mean   :2004  
##                                        3rd Qu.:4.600   3rd Qu.:2008  
##                                        Max.   :7.000   Max.   :2008  
```

```
##       cyl            trans               drv                 cty
##  Min.   :4.000   Length:234         Length:234         Min.   : 9.00
##  1st Qu.:4.000   Class :character   Class :character   1st Qu.:14.00
##  Median :6.000   Mode  :character   Mode  :character   Median :17.00
##  Mean   :5.889                                         Mean   :16.86
##  3rd Qu.:8.000                                         3rd Qu.:19.00
##  Max.   :8.000                                         Max.   :35.00
##       hwy             fl               class
##  Min.   :12.00   Length:234         Length:234
##  1st Qu.:18.00   Class :character   Class :character
##  Median :24.00   Mode  :character   Mode  :character
##  Mean   :23.44
##  3rd Qu.:27.00
##  Max.   :44.00
```

We can create one-way and two-way tables with `table`

```r
table(mpg$manufacturer)
```

```
##
##       audi  chevrolet      dodge       ford      honda    hyundai
##         18         19         37         25          9         14
##       jeep land rover    lincoln    mercury     nissan    pontiac
##          8          4          3          4         13          5
##     subaru     toyota volkswagen
##         14         34         27
```

```r
table(mpg$manufacturer,mpg$year)
```

```
##
##               1999 2008
##   audi           9    9
##   chevrolet      7   12
##   dodge         16   21
##   ford          15   10
##   honda          5    4
##   hyundai        6    8
##   jeep           2    6
##   land rover     2    2
##   lincoln        2    1
##   mercury        2    2
##   nissan         6    7
##   pontiac        3    2
##   subaru         6    8
##   toyota        20   14
##   volkswagen    16   11
```

```r
table(mpg$year)
```

```
##
## 1999 2008
##  117  117
```

We can also find proportion tables with `prop.table`. If we want to use `prop.table`, we have to save a table object first, and then call `prop.table`. For example:

```r
manutable = table(mpg$manufacturer)
prop.table(manutable)
```

```
##
##       audi  chevrolet      dodge       ford      honda    hyundai
## 0.07692308 0.08119658 0.15811966 0.10683761 0.03846154 0.05982906
##       jeep land rover    lincoln    mercury     nissan    pontiac
## 0.03418803 0.01709402 0.01282051 0.01709402 0.05555556 0.02136752
##     subaru     toyota volkswagen
## 0.05982906 0.14529915 0.11538462
```

We can create two-way proportion tables using the same idea. For instance,

```r
manuyear = table(mpg$manufacturer, mpg$year)
prop.table(manuyear)
```

```
##
##                     1999        2008
##   audi       0.038461538 0.038461538
##   chevrolet  0.029914530 0.051282051
##   dodge      0.068376068 0.089743590
##   ford       0.064102564 0.042735043
##   honda      0.021367521 0.017094017
##   hyundai    0.025641026 0.034188034
##   jeep       0.008547009 0.025641026
##   land rover 0.008547009 0.008547009
##   lincoln    0.008547009 0.004273504
##   mercury    0.008547009 0.008547009
##   nissan     0.025641026 0.029914530
##   pontiac    0.012820513 0.008547009
##   subaru     0.025641026 0.034188034
##   toyota     0.085470085 0.059829060
##   volkswagen 0.068376068 0.047008547
```

The table above is a total proportions table (that is, if we add up all the numbers in the table, we get 1).

If we want row proportions,

```r
prop.table(manuyear, 1)
```

```
##
##                   1999      2008
##   audi       0.5000000 0.5000000
##   chevrolet  0.3684211 0.6315789
##   dodge      0.4324324 0.5675676
##   ford       0.6000000 0.4000000
##   honda      0.5555556 0.4444444
##   hyundai    0.4285714 0.5714286
##   jeep       0.2500000 0.7500000
##   land rover 0.5000000 0.5000000
##   lincoln    0.6666667 0.3333333
##   mercury    0.5000000 0.5000000
##   nissan     0.4615385 0.5384615
##   pontiac    0.6000000 0.4000000
##   subaru     0.4285714 0.5714286
##   toyota     0.5882353 0.4117647
##   volkswagen 0.5925926 0.4074074
```

And if we want column proportions:

```
prop.table(manuyear,2 )
```

```
##
##                   1999        2008
##   audi       0.076923077 0.076923077
##   chevrolet  0.059829060 0.102564103
##   dodge      0.136752137 0.179487179
##   ford       0.128205128 0.085470085
##   honda      0.042735043 0.034188034
##   hyundai    0.051282051 0.068376068
##   jeep       0.017094017 0.051282051
##   land rover 0.017094017 0.017094017
##   lincoln    0.017094017 0.008547009
##   mercury    0.017094017 0.017094017
##   nissan     0.051282051 0.059829060
##   pontiac    0.025641026 0.017094017
##   subaru     0.051282051 0.068376068
##   toyota     0.170940171 0.119658120
##   volkswagen 0.136752137 0.094017094
```

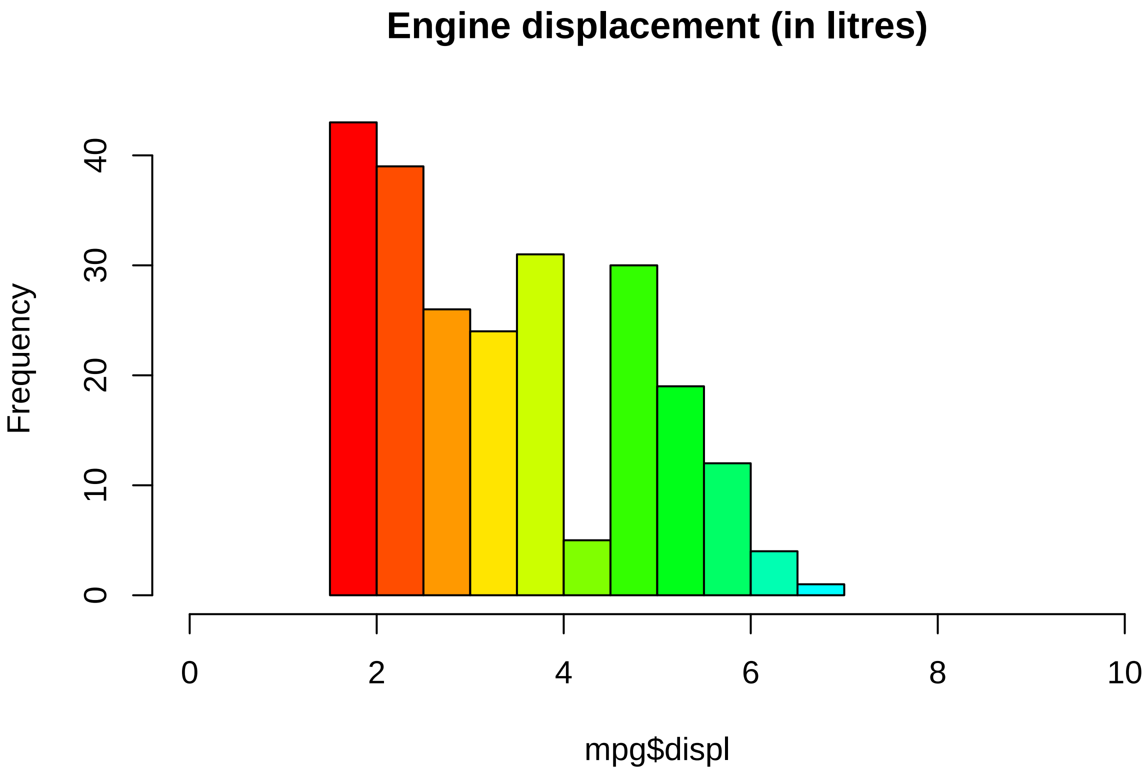
These are proportion tables. If we want percentages, we can simply multiply the `prop.table`s times 100. For example:

```
100*prop.table(manuyear)
```

```
##
##                 1999      2008
##   audi       3.8461538 3.8461538
##   chevrolet  2.9914530 5.1282051
##   dodge      6.8376068 8.9743590
##   ford       6.4102564 4.2735043
##   honda      2.1367521 1.7094017
##   hyundai    2.5641026 3.4188034
##   jeep       0.8547009 2.5641026
##   land rover 0.8547009 0.8547009
##   lincoln    0.8547009 0.4273504
##   mercury    0.8547009 0.8547009
##   nissan     2.5641026 2.9914530
##   pontiac    1.2820513 0.8547009
##   subaru     2.5641026 3.4188034
##   toyota     8.5470085 5.9829060
##   volkswagen 6.8376068 4.7008547
```

We can plot stuff, too. For example, `hist` does histograms:

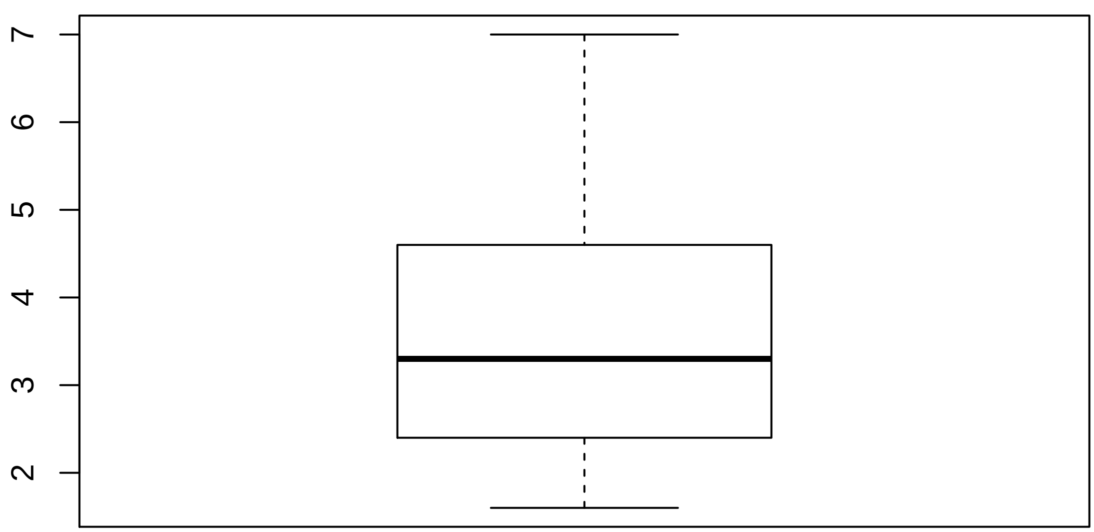
```
hist(mpg$displ, main="Engine displacement (in litres)",
     col=rainbow(20),
     xlim=c(0,10))
```



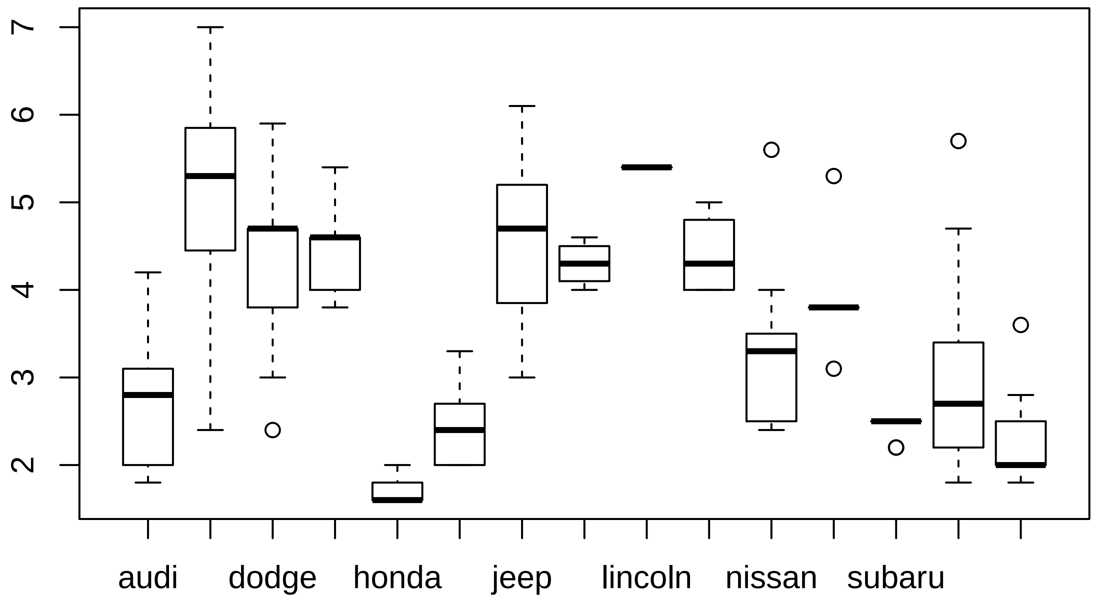
You can learn more about how to change the attributes of the plot with `?hist`.

We can create individual boxplots and boxplots grouped by values of categorical variables:

```
boxplot(mpg$displ)
```



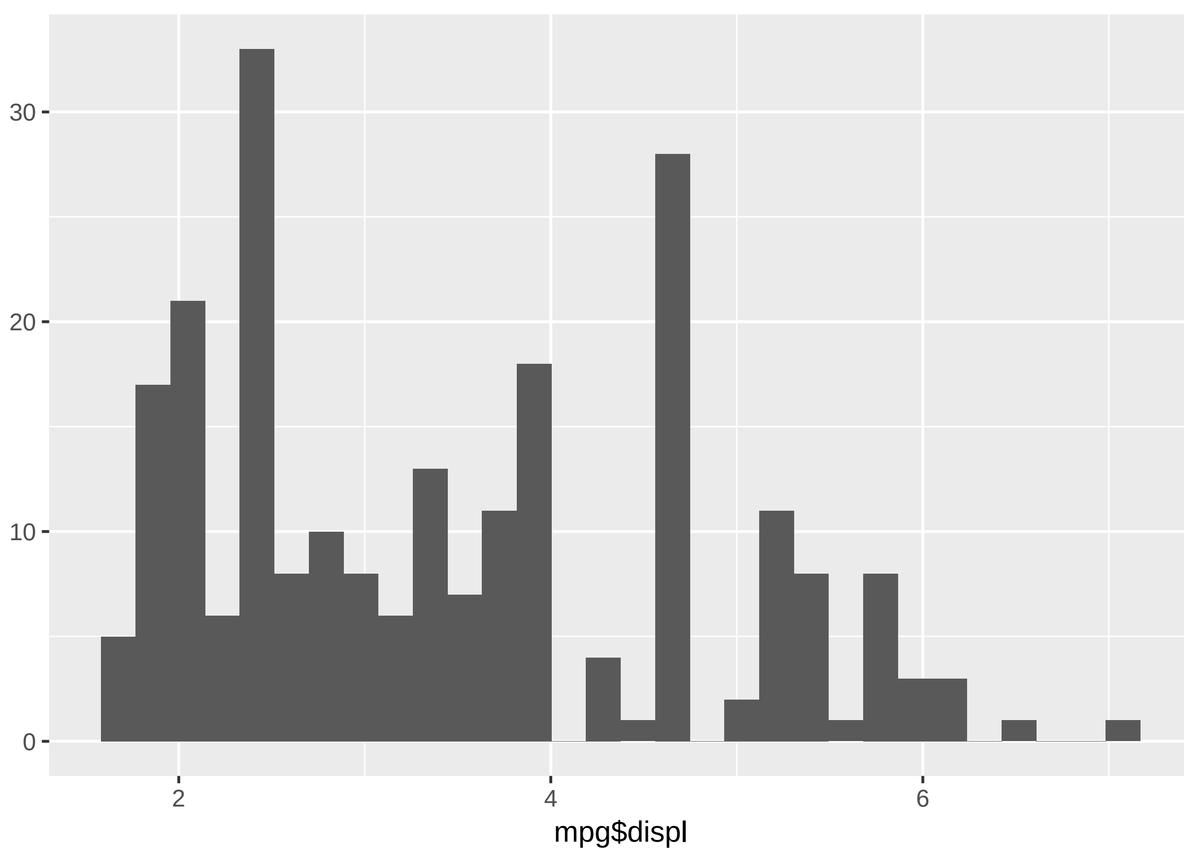

```
boxplot(mpg$displ~mpg$manufacturer)
```



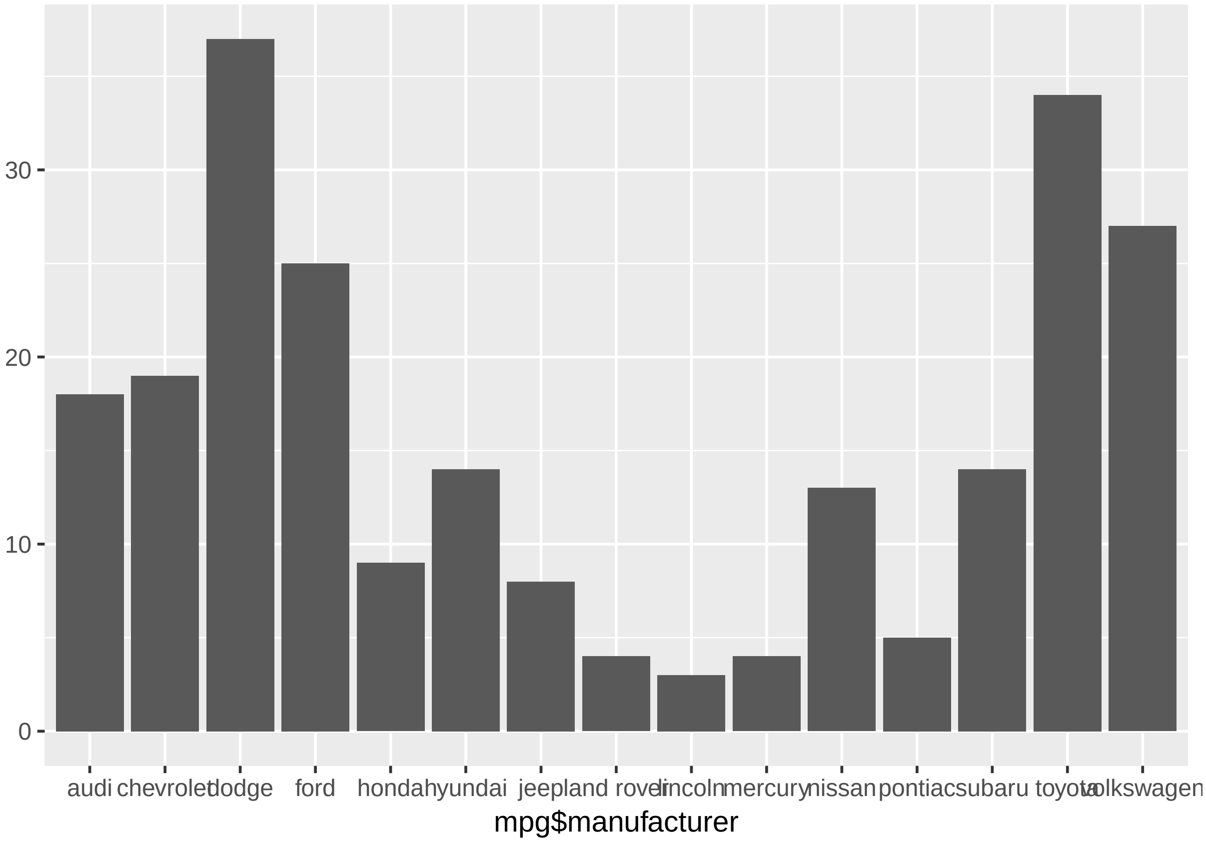
These plots are created using the `graphics` library. There are other libraries we can use to produce plots. One of them is `ggplot2`, which we installed earlier. A nice thing about `ggplot2` is that it has the function `qplot`, which produces nice-looking plots by default. For example:

```
qplot(mpg$displ)
```

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



```r
qplot(mpg$manufacturer)
```



`qplot` is smart enough to produce different plots depending on the type of the object. We'll cover `ggplot2` in more detail later in the semester.

Exercise

- What is the maximum value of highway miles per gallon in the `mpg` dataset?
- How many cars with manual transmission are there in the `mpg` dataset?
- What % of the cars in the `mpg` dataset are SUVs?
- How many SUVs in the `mpg` dataset are 4-wheel drives?
- What is the % of Toyotas in the `mpg` dataset that are SUVs?
- (Not on TopHat) Create a barplot that shows how many cars are front-wheel drives, rear-wheel drives, and 4wd

References

- Datacamp: introduction to `R`