

Data wrangling

Víctor Peña

Contents

Importing and exporting data	1
Data subsetting	2
Missing data	7
Sorting data	9
Type conversions	12
Reformatting datasets	14
References	25

Importing and exporting data

R can read data in many different formats and it has several functions that can help us with that. The functions themselves have numerous parameters and options that can be used to read messy data correctly. I am not going to cover the ins and outs of that (it's rather tedious). If you're interested in learning more, I recommend the following article: <https://www.datacamp.com/community/tutorials/r-data-import-tutorial>.

My personal workflow to importing data is (1) clean the data using some spreadsheet software (Excel, Numbers, Google Sheets) and then (2) read the spreadsheet using the “Import Dataset” option in RStudio (top-right corner).

However, if the data are nicely formatted in *.csv or plain text format, using the functions `read.csv` and `read.table` is relatively painless.

For example, you can read `depression.csv` (hosted on my website) with the instruction

```
depression = read.csv("http://vicpena.github.io/sta9750/fall18/depression.csv")
```

If the dataset doesn't have column names, you only need to add `header = FALSE`. For example, suppose that we want to read in the following dataset: <http://users.stat.ufl.edu/~winner/data/femrole.dat>. It doesn't have variable names. We can read it in with

```
femrole = read.table("http://users.stat.ufl.edu/~winner/data/femrole.dat", header=F)
```

Exporting data with R is easy. If we want to export an existing `data.frame` to a *.csv file (which can be opened with Excel, Numbers, or any statistical package), we can use the function `write.csv`. For example, if we want to export the `iris` dataset into a file named `iris.csv` in the working directory:

```
data(iris)
write.csv(iris, file = "./iris.csv")
```

If you want the file to be saved somewhere else, you can change `./` by any path you want.

Another option is saving the workspace. That is, creating a file that has all the objects that we are currently working with (variables, `data.frames`, etc.). The function that allows us to do that is `save`. If we want

to save all the variables and objects, we can simply type `save(file='<path>/<filename>.RData')`, where `<path>` is the path where the file will be saved and `<filename>` is the filename. We can also save only a subset of the variables. For example, suppose we want to save 2 objects named `var1` and `df`. The command `save(var1, df, file = '<path>/<filename>.RData')` will do that for us.

Data subsetting

In this section, we'll cover how to subset variables and rows of datasets (mainly `data.frames`). We'll cover 2 different ways of filtering. We'll use the "traditional" way to do that (which doesn't require any extra libraries) and we'll use functions in `library(dplyr)` (which are faster in big datasets, cleaner, and more "intuitive").

Subsetting variables

We saw some of that in the previous chapter. Let's load the `iris` dataset.

```
data("iris")
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 ...
```

If we want to create a subset that contains, say, the first, the second, and the fifth columns, it's as easy as typing

```
sub1 = iris[,c(1,2,5)]
str(sub1)
```

```
## 'data.frame':   150 obs. of  3 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 ...
##  $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

We can also create subsets by specifying which columns we want to remove. For example,

```
sub2 = iris[,-c(1,2,5)]
str(sub2)
```

```
## 'data.frame':   150 obs. of  2 variables:
##  $ 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 ...
```

contains all the columns in *except* the first, the second, and the fifth.

If we want to access specific columns of `iris`, we can use `$` followed by the name of the variable. For example, if we want to take a look at `Species`:

```
iris$Species
```

```
## [1] setosa setosa setosa setosa setosa setosa
## [7] setosa setosa setosa setosa setosa setosa
## [13] setosa setosa setosa setosa setosa setosa
## [19] setosa setosa setosa setosa setosa setosa
## [25] setosa setosa setosa setosa setosa setosa
## [31] setosa setosa setosa setosa setosa setosa
## [37] setosa setosa setosa setosa setosa setosa
## [43] setosa setosa setosa setosa setosa setosa
## [49] setosa setosa versicolor versicolor versicolor versicolor
## [55] versicolor versicolor versicolor versicolor versicolor versicolor
## [61] versicolor versicolor versicolor versicolor versicolor versicolor
## [67] versicolor versicolor versicolor versicolor versicolor versicolor
## [73] versicolor versicolor versicolor versicolor versicolor versicolor
## [79] versicolor versicolor versicolor versicolor versicolor versicolor
## [85] versicolor versicolor versicolor versicolor versicolor versicolor
## [91] versicolor versicolor versicolor versicolor versicolor versicolor
## [97] versicolor versicolor versicolor versicolor virginica virginica
## [103] virginica virginica virginica virginica virginica virginica
## [109] virginica virginica virginica virginica virginica virginica
## [115] virginica virginica virginica virginica virginica virginica
## [121] virginica virginica virginica virginica virginica virginica
## [127] virginica virginica virginica virginica virginica virginica
## [133] virginica virginica virginica virginica virginica virginica
## [139] virginica virginica virginica virginica virginica virginica
## [145] virginica virginica virginica virginica virginica virginica
## Levels: setosa versicolor virginica
```

What we just covered is the traditional way of subsetting variables with R. With `library(dplyr)`, we can use the command `select`. First, let's load the library (if you don't have it, you can install it with the command `install.packages("dplyr")`).

```
library(dplyr)
```

The following command creates a subset that contains the first, the second, and the fifth variables,

```
sub3 = iris %>% select(1,2,5)
str(sub3)
```

```
## 'data.frame': 150 obs. of 3 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 ...
## $ Species : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

Note that there's a `%>%` separating `iris` and `select`. The operator `%>%` is what we call a “pipe”. The commands in `library(dplyr)` (and `library(tidyr)`, which we'll also cover) can be concatenated using the pipe operator. It ends up being pretty convenient. It looks odd at first, but you'll get used to it.

A nice feature of `select` is that we can easily select variables using their names:

```
sub4 = iris %>% select(Sepal.Length, Sepal.Width, Species)
head(sub4)
```

```
##   Sepal.Length Sepal.Width Species
## 1          5.1          3.5  setosa
## 2          4.9          3.0  setosa
## 3          4.7          3.2  setosa
## 4          4.6          3.1  setosa
## 5          5.0          3.6  setosa
## 6          5.4          3.9  setosa
```

As you can imagine, we can also create subsets by specifying which variables we want to exclude:

```
sub5 = iris %>% select(-c(1,2,5))
str(sub5)
```

```
## 'data.frame':   150 obs. of  2 variables:
## $ 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 ...
```

```
sub6 = iris %>% select(-c(Sepal.Length, Sepal.Width, Species))
str(sub6)
```

```
## 'data.frame':   150 obs. of  2 variables:
## $ 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 ...
```

Subsetting rows

We can subset rows by indicating which row numbers we want to keep (or exclude). For example, we can create a subset with the first, the thirtieth, and the fiftieth observations in the `iris` dataset as follows

```
sub1 = iris[c(1, 30, 50),]
str(sub1)
```

```
## 'data.frame':   3 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.7 5
## $ Sepal.Width : num  3.5 3.2 3.3
## $ Petal.Length: num  1.4 1.6 1.4
## $ Petal.Width : num  0.2 0.2 0.2
## $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1
```

And if we want to create a subset that includes all but the first, the thirtieth and the fiftieth observations:

```
sub2 = iris[-c(1, 30, 50),]
str(sub2)
```

```
## 'data.frame':   147 obs. of  5 variables:
## $ Sepal.Length: num  4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 5.4 ...
## $ Sepal.Width : num  3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 ...
## $ Petal.Length: num  1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 ...
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 0.2 ...
## $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

This is alright, but not very useful in practice. We're usually interested in subsets of rows that satisfy a certain condition. For example, we might be interested in creating a subset that only contains flowers of the `setosa` species. The following commands will do that for us

```
cond1 = (iris$Species == 'setosa')
str(cond1)
```

```
## logi [1:150] TRUE TRUE TRUE TRUE TRUE TRUE ...
```

```
sub3 = iris[cond1,]
str(sub3)
```

```
## 'data.frame': 50 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 ...
```

In the first command, we create a logical type variable that takes on the value `TRUE` if `Species` is equal to `setosa` and `FALSE` otherwise (note that there are 2 equal signs between `iris$Species` and `setosa`). In the second command, we use the logical variable to filter the `iris` dataset. We can use a similar strategy to create all sorts of subsets according to logical conditions. The operators are

- `==`: equal to
- `!=`: not equal to
- `>`: greater than
- `<`: less than
- `>=`: greater or equal to
- `<=`: less than or equal to

For example, we can create a subset that contains only observations whose `Sepal.Length` is greater than 5

```
cond2 = (iris$Sepal.Length > 5)
sub4 = iris[cond2,]
```

And we can create a subset that contain all the observations whose `Species` isn't equal to `setosa` with

```
cond3 = iris$Species != 'setosa'
sub6 = iris[cond3,]
str(sub6)
```

```
## 'data.frame': 100 obs. of 5 variables:
## $ Sepal.Length: num 7 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 ...
## $ Sepal.Width : num 3.2 3.2 3.1 2.3 2.8 2.8 3.3 2.4 2.9 2.7 ...
## $ Petal.Length: num 4.7 4.5 4.9 4 4.6 4.5 4.7 3.3 4.6 3.9 ...
## $ Petal.Width : num 1.4 1.5 1.5 1.3 1.5 1.3 1.6 1 1.3 1.4 ...
## $ Species : Factor w/ 3 levels "setosa","versicolor",...: 2 2 2 2 2 2 2 2 2 2 ...
```

Logical conditions can be combined with “and”, “or”, and “not” operators, which in R are:

- `&`: and
- `|`: or
- `!`: not

For example, we can create a subset that contains `setosas` whose `Sepal.Length` is greater than 5 with

```
cond4 = (iris$Species == 'setosa') & (iris$Sepal.Length > 5)
sub7 = iris[cond4,]
str(sub7)
```

```
## 'data.frame':  22 obs. of  5 variables:
## $ Sepal.Length: num  5.1 5.4 5.4 5.8 5.7 5.4 5.1 5.7 5.1 5.4 ...
## $ Sepal.Width : num  3.5 3.9 3.7 4 4.4 3.9 3.5 3.8 3.8 3.4 ...
## $ Petal.Length: num  1.4 1.7 1.5 1.2 1.5 1.3 1.4 1.7 1.5 1.7 ...
## $ Petal.Width : num  0.2 0.4 0.2 0.2 0.4 0.4 0.3 0.3 0.3 0.2 ...
## $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

We can create a subset that contains observations that are not `setosas` or whose `Sepal.Width` is less than or equal to 4 with

```
cond5 = !(iris$Species == 'setosa') | (iris$Sepal.Width <= 4)
sub8 = iris[cond5,]
str(sub8)
```

```
## 'data.frame':  147 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 could’ve also written `iris$Species != 'setosa'`.

What we just covered is the traditional way of subsetting rows with R. `library(dplyr)` has the function `filter`, which does the same thing with a cleaner syntax.

For example, we can create a subset that contains `setosas` whose `Sepal.Length` is greater than 5 as follows

```
sub1 = iris %>% filter(Species == 'setosa' & Sepal.Length > 5)
```

And we can create a subset that contains flowers whose `Species` isn’t `setosa` or whose `Sepal.Width` is less than or equal to 4

```
sub2 = iris %>% filter(Species != 'setosa' | Sepal.Width <= 4)
```

As you can see, with `filter` we don’t have to type in `iris$` whenever we want to specify a condition for variables in `iris`.

We can combine `select` and `filter` statements. For example, we can create a subset that excludes `Species` and only contains `setosas` as follows

```
sub3 = iris %>% filter(Species == 'setosa') %>% select(-Species)
str(sub3)
```

```
## 'data.frame': 50 obs. of 4 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 ...
```

The order in which `filter` and `select` appear matters. If we typed the `select` statement first, we would get an error because when R tries to apply the `filter`, `Species` has already been excluded.

Tophat exercises

Read in the `hsb2` dataset.

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

Answer the following questions:

1. What is the average 'math' score in the dataset?
2. What is the average 'math' score for those who scored 50 or greater in 'read'? Is it greater or smaller than the overall mean? Think about the result.
3. What is the average 'read' score in the dataset?
4. What is the average 'read' score for those who scored 50 or greater in 'math'? Is it greater or smaller than the overall mean? Compare your result to your answer in part 2.
5. What is the average difference in 'math' scores between individuals whose race is 'white' and those whose race is not 'white'?
6. Now, consider only those students whose 'ses' is 'high'. What is the average difference in 'math' scores between individuals whose race is 'white' and those whose race is not 'white'?
7. What is the percentage of individuals in the sample whose 'race' is 'white'?
8. What is the percentage of individuals of high 'ses' that are 'white'?
9. What percentage of students of low 'ses' went to 'public' schools?
10. What is the percentage of students with a 'math' score greater than 50 who went to 'public' schools?

Missing data

Sometimes, our datasets have missing values. In R, missing values are marked as `NA`.

For example, we can a vector with a missing value as follows

```
x = c(1:5, NA)
x
```

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

When we have missing values, we have to be careful. For example, if we try to take the average of `x` with `mean`:

```
mean(x)
```

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

In general, arithmetic operations with NAs return NAs:

```
0+NA
```

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

```
3*NA
```

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

```
5/NA
```

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

The function `is.na` can be used to filter missing values. For example,

```
cond = is.na(x)
cond
```

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

```
x = x[!cond]
x
```

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

For `data.frames`, the functions `complete.cases` and `na.omit` are useful.

Let's load the `airquality` dataset, which is built-in in 'R'.

```
data(airquality)
```

The dataset has some air quality measurements that were taken in NYC from May to September in 1973 (see `?airquality` for more details). The dataset has some missing values

```
summary(airquality)
```

```
##      Ozone      Solar.R      Wind      Temp
## Min.   : 1.00   Min.     : 7.0   Min.    : 1.700   Min.     :56.00
## 1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00
## Median : 31.50   Median :205.0   Median : 9.700   Median :79.00
## Mean   : 42.13   Mean    :185.9   Mean    : 9.958   Mean    :77.88
## 3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00
## Max.   :168.00   Max.     :334.0   Max.    :20.700   Max.     :97.00
## NA's    :37      NA's     :7
```



```
##      Month      Day
## Min.   :5.000   Min.    : 1.0
## 1st Qu.:6.000   1st Qu.: 8.0
## Median :7.000   Median :16.0
## Mean   :6.993   Mean    :15.8
## 3rd Qu.:8.000   3rd Qu.:23.0
## Max.   :9.000   Max.    :31.0
##
```

There are 37 missing `Ozone` readings and 7 missing values in `Solar.R`. The function `complete.cases`, when applied to `airquality`, will create a logical vector whose values will be `TRUE` if the observation is “complete” (i.e., doesn’t have any missing values) and `FALSE` if there is at least one variable with a missing value.

We can create a new dataset called `aircomp` that only contains complete observations as follows

```
aircomp = airquality[complete.cases(airquality),]
```

The command above is equivalent to

```
aircomp = na.omit(aircomp)
```

We are covering `complete.cases` because having a logical vector can help us identify the observations that have missing values. Indeed, we can filter the observations that are NOT complete cases, that is:

```
miss = airquality[!complete.cases(airquality),]
head(miss)
```

```
##      Ozone Solar.R Wind Temp Month Day
## 5      NA      NA 14.3   56     5   5
## 6      28      NA 14.9   66     5   6
## 10     NA     194   8.6   69     5  10
## 11      7      NA   6.9   74     5  11
## 25     NA      66 16.6   57     5  25
## 26     NA     266 14.9   58     5  26
```

Sorting data

We can sort variables with the `sort` function. The default ordering is increasing. For example,

```
sort(iris$Sepal.Length)
```

```
##      [1] 4.3 4.4 4.4 4.4 4.5 4.6 4.6 4.6 4.6 4.7 4.7 4.8 4.8 4.8 4.8 4.8 4.9
##     [18] 4.9 4.9 4.9 4.9 4.9 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.1 5.1
##     [35] 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.2 5.2 5.2 5.2 5.3 5.4 5.4 5.4 5.4 5.4
##     [52] 5.4 5.5 5.5 5.5 5.5 5.5 5.5 5.5 5.5 5.6 5.6 5.6 5.6 5.6 5.6 5.7 5.7
##     [69] 5.7 5.7 5.7 5.7 5.7 5.8 5.8 5.8 5.8 5.8 5.8 5.8 5.9 5.9 5.9 6.0 6.0
##     [86] 6.0 6.0 6.0 6.0 6.1 6.1 6.1 6.1 6.1 6.1 6.2 6.2 6.2 6.2 6.3 6.3 6.3
##    [103] 6.3 6.3 6.3 6.3 6.3 6.3 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.5 6.5 6.5 6.5
##    [120] 6.5 6.6 6.6 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.8 6.8 6.8 6.9 6.9 6.9
##    [137] 6.9 7.0 7.1 7.2 7.2 7.2 7.3 7.4 7.6 7.7 7.7 7.7 7.7 7.7 7.9
```

```
sort(iris$Species)
```

```
## [1] setosa setosa setosa setosa setosa setosa
## [7] setosa setosa setosa setosa setosa setosa
## [13] setosa setosa setosa setosa setosa setosa
## [19] setosa setosa setosa setosa setosa setosa
## [25] setosa setosa setosa setosa setosa setosa
## [31] setosa setosa setosa setosa setosa setosa
## [37] setosa setosa setosa setosa setosa setosa
## [43] setosa setosa setosa setosa setosa setosa
## [49] setosa setosa versicolor versicolor versicolor versicolor
## [55] versicolor versicolor versicolor versicolor versicolor versicolor
## [61] versicolor versicolor versicolor versicolor versicolor versicolor
## [67] versicolor versicolor versicolor versicolor versicolor versicolor
## [73] versicolor versicolor versicolor versicolor versicolor versicolor
## [79] versicolor versicolor versicolor versicolor versicolor versicolor
## [85] versicolor versicolor versicolor versicolor versicolor versicolor
## [91] versicolor versicolor versicolor versicolor versicolor versicolor
## [97] versicolor versicolor versicolor versicolor virginica virginica
## [103] virginica virginica virginica virginica virginica virginica
## [109] virginica virginica virginica virginica virginica virginica
## [115] virginica virginica virginica virginica virginica virginica
## [121] virginica virginica virginica virginica virginica virginica
## [127] virginica virginica virginica virginica virginica virginica
## [133] virginica virginica virginica virginica virginica virginica
## [139] virginica virginica virginica virginica virginica virginica
## [145] virginica virginica virginica virginica virginica virginica
## Levels: setosa versicolor virginica
```

If we want descending order, we can add the option `decreasing = TRUE`:

```
sort(iris$Sepal.Length, decreasing = TRUE)
```

```
## [1] 7.9 7.7 7.7 7.7 7.7 7.6 7.4 7.3 7.2 7.2 7.2 7.1 7.0 6.9 6.9 6.9 6.9
## [18] 6.8 6.8 6.8 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.6 6.6 6.5 6.5 6.5 6.5
## [35] 6.5 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.3 6.3 6.3 6.3 6.3 6.3 6.3 6.3 6.3
## [52] 6.2 6.2 6.2 6.2 6.1 6.1 6.1 6.1 6.1 6.1 6.0 6.0 6.0 6.0 6.0 6.0 5.9
## [69] 5.9 5.9 5.8 5.8 5.8 5.8 5.8 5.8 5.8 5.7 5.7 5.7 5.7 5.7 5.7 5.7 5.7
## [86] 5.6 5.6 5.6 5.6 5.6 5.6 5.5 5.5 5.5 5.5 5.5 5.5 5.5 5.4 5.4 5.4 5.4
## [103] 5.4 5.4 5.3 5.2 5.2 5.2 5.2 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.0
## [120] 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 4.9 4.9 4.9 4.9 4.9 4.9 4.8 4.8
## [137] 4.8 4.8 4.8 4.7 4.7 4.6 4.6 4.6 4.6 4.5 4.4 4.4 4.4 4.4 4.3
```

This only works with variables. What if we want to order a `data.frame` according to the values of one of the variables? For that task, we can use `order`.

For example, if we want to order `iris` in ascending order by `Sepal.Length`:

```
head(iris[order(iris$Sepal.Length),])
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 14          4.3         3.0          1.1          0.1 setosa
```

```
## 9      4.4      2.9      1.4      0.2 setosa
## 39     4.4      3.0      1.3      0.2 setosa
## 43     4.4      3.2      1.3      0.2 setosa
## 42     4.5      2.3      1.3      0.3 setosa
## 4      4.6      3.1      1.5      0.2 setosa
```

I'm adding `head()` so that R doesn't print the full dataset.

If we want descending order instead:

```
head(iris[order(-iris$Sepal.Length),])
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
## 132           7.9         3.8         6.4         2.0 virginica
## 118           7.7         3.8         6.7         2.2 virginica
## 119           7.7         2.6         6.9         2.3 virginica
## 123           7.7         2.8         6.7         2.0 virginica
## 136           7.7         3.0         6.1         2.3 virginica
## 106           7.6         3.0         6.6         2.1 virginica
```

When there are “ties”, we can also sort the data by a second variable. For example, if we sort the data in descending order by `Species`, there will be a lot of observations that will share the same value of `Species`. If, given the species, we want to sort in ascending order by `Petal.Width`, this will do that for us

```
head(iris[order(-iris$Species, iris$Petal.Width),])
```

```
## Warning in Ops.factor(iris$Species): '-' not meaningful for factors
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 10           4.9         3.1         1.5         0.1 setosa
## 13           4.8         3.0         1.4         0.1 setosa
## 14           4.3         3.0         1.1         0.1 setosa
## 33           5.2         4.1         1.5         0.1 setosa
## 38           4.9         3.6         1.4         0.1 setosa
## 1           5.1         3.5         1.4         0.2 setosa
```

`library(dplyr)` has the function `arrange`, which is the analogue of `order`.

The following piece of code sorts the dataset in ascending order by `Sepal.Length`

```
head(iris %>% arrange(Sepal.Length))
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           4.3         3.0         1.1         0.1 setosa
## 2           4.4         2.9         1.4         0.2 setosa
## 3           4.4         3.0         1.3         0.2 setosa
## 4           4.4         3.2         1.3         0.2 setosa
## 5           4.5         2.3         1.3         0.3 setosa
## 6           4.6         3.1         1.5         0.2 setosa
```

If we want descending order

```
head(iris %>% arrange(desc(Sepal.Length)))
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
## 1         7.9         3.8         6.4         2.0 virginica
## 2         7.7         3.8         6.7         2.2 virginica
## 3         7.7         2.6         6.9         2.3 virginica
## 4         7.7         2.8         6.7         2.0 virginica
## 5         7.7         3.0         6.1         2.3 virginica
## 6         7.6         3.0         6.6         2.1 virginica
```

And the following sorts in descending order by `Species`, and then in ascending order by `Petal.Width`.

```
head(iris %>% arrange(desc(Species), Petal.Width))
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
## 1         6.1         2.6         5.6         1.4 virginica
## 2         6.0         2.2         5.0         1.5 virginica
## 3         6.3         2.8         5.1         1.5 virginica
## 4         7.2         3.0         5.8         1.6 virginica
## 5         4.9         2.5         4.5         1.7 virginica
## 6         6.3         2.9         5.6         1.8 virginica
```

An advantage of using `arrange` is that we don't have to type `iris$<variable name>` all the time.

Type conversions

Oftentimes, categorical variables are coded as numerical. For example, let's look at the dataset `femrole.dat`, which is uploaded on Professor Winner's website. A description of the dataset can be found [here](#) and the data can be accessed [here](#). As you can see, there are 4 categorical variables that are coded as numerical. How do we convert these variables to `factors`?

The following instruction reads in the data

```
femrole = read.table("http://users.stat.ufl.edu/~winner/data/femrole.dat", header = FALSE)
```

Now, we can print it

```
femrole
```

```
##      V1 V2 V3 V4 V5
## 1     1  1  1  1 11
## 2     1  2  1  1 12
## 3     2  1  1  1 10
## 4     2  2  1  1 12
## 5     1  1  1  2 13
## 6     1  2  1  2 12
## 7     2  1  1  2  8
## 8     2  2  1  2 29
## 9     1  1  2  1 11
## 10    1  2  2  1  6
## 11    2  1  2  1  4
```

```
## 12  2  2  2  1 13
## 13  1  1  2  2 17
## 14  1  2  2  2  8
## 15  2  1  2  2  9
## 16  2  2  2  2 33
```

First of all, the columns don't have interpretable names. We can change the names as follows:

```
colnames(femrole) = c("personality", "role", "friends", "dates", "count")
```

The variables `personality`, `role`, `friends`, and `dates` are categorical, but in `femrole` they are coded as numerical. To see this, we can run

```
str(femrole)
```

```
## 'data.frame':  16 obs. of  5 variables:
## $ personality: int  1 1 2 2 1 1 2 2 1 1 ...
## $ role       : int  1 2 1 2 1 2 1 2 1 2 ...
## $ friends    : int  1 1 1 1 1 1 1 1 2 2 ...
## $ dates      : int  1 1 1 1 2 2 2 2 1 1 ...
## $ count      : int 11 12 10 12 13 12 8 29 11 6 ...
```

The output tells us that `personality`, `role`, `friends`, `dates`, and `count` are of type `int`, which means that they're coded as integers.

R functions can treat variables differently depending on whether they are numerical or categorical. If we don't convert the variables, we can get meaningless output.

We can convert the variables to factors using `as.factor`:

```
femrole$personality = as.factor(femrole$personality)
femrole$role = as.factor(femrole$role)
femrole$friends = as.factor(femrole$friends)
femrole$dates = as.factor(femrole$dates)
```

Let's run `str` again:

```
str(femrole)
```

```
## 'data.frame':  16 obs. of  5 variables:
## $ personality: Factor w/ 2 levels "1","2": 1 1 2 2 1 1 2 2 1 1 ...
## $ role       : Factor w/ 2 levels "1","2": 1 2 1 2 1 2 1 2 1 2 ...
## $ friends    : Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 2 2 ...
## $ dates      : Factor w/ 2 levels "1","2": 1 1 1 1 2 2 2 2 1 1 ...
## $ count      : int 11 12 10 12 13 12 8 29 11 6 ...
```

We have successfully changed their type from `integer` to `factor`. However, the levels of the factors are noninformative. We can change them using `levels`:

```
levels(femrole$personality) = c("Modern", "Traditional")
levels(femrole$role) = c("Modern", "Traditional")
levels(femrole$friends) = c("Low", "High")
levels(femrole$dates) = c("Low", "High")
```

TopHat Exercises. Read in the dataset `interfaith.dat`, which is available on Professor Winner's website or by clicking [here](#) (the description is available [here](#)). Change the variable names to something informative, convert the appropriate variables into **factors**, and rename the levels of the factors using meaningful labels.

- What percentage of catholics are of low socioeconomic status?
- What percentage of protestants are of low socioeconomic status?
- What percentage of catholics are in an interfaith relationship?
- What percentage of protestants are in an interfaith relationship?

As you can imagine, other type conversions are possible. For instance, we can convert from `matrix` to `data.frame` with `as.data.frame`:

```
mat = matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 3)
df = as.data.frame(mat)
```

Now, `df` is of class `data.frame`:

```
class(df)
```

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

We can also convert `data.frames` that contain numeric variables to `matrix` using `as.matrix`.

```
df = data.frame(var1 = 1:3, var2 = 4:6)
mat = as.matrix(df)
```

And, unsurprisingly,

```
class(mat)
```

```
## [1] "matrix"
```

Reformatting datasets

Aggregated data

Let's take a closer look at the `femrole` dataset, which we formatted in a previous section.

```
femrole
```

```
##      personality      role friends dates count
## 1      Modern      Modern      Low   Low    11
## 2      Modern Traditional      Low   Low    12
## 3 Traditional      Modern      Low   Low    10
## 4 Traditional Traditional      Low   Low    12
## 5      Modern      Modern      Low  High    13
## 6      Modern Traditional      Low  High    12
## 7 Traditional      Modern      Low  High     8
```

```
## 8 Traditional Traditional Low High 29
## 9 Modern Modern High Low 11
## 10 Modern Traditional High Low 6
## 11 Traditional Modern High Low 4
## 12 Traditional Traditional High Low 13
## 13 Modern Modern High High 17
## 14 Modern Traditional High High 8
## 15 Traditional Modern High High 9
## 16 Traditional Traditional High High 33
```

The data are aggregated: each row corresponds to a certain social *profile*. The last column counts how many observations there are for each profile. Unfortunately, R isn't very good at working with data in this format. For example, suppose that we're interested in knowing how many women in the dataset have a **High** number of dates. If we type in

```
table(femrole$dates)
```

```
##
## Low High
## 8 8
```

we get the wrong answer.

It's not only tables: plots and statistical methods in R are coded in a way that makes working with aggregated data difficult.

The most convenient format is a dataset where the rows correspond to different individuals (in this case, each row should correspond to a different woman). Thankfully, the function `uncount` in `library(tidyr)` makes the conversion easy.

```
unaggregated = femrole %>% uncount(count)
```

The argument of `uncount` is the variable that contains the counts (which, in this case, is conveniently named `count`). If you want to convert individual data to an aggregated format, you can use the function `count`.

We can convert `unaggregated` into an aggregated format as follows:

```
unaggregated %>% count(personality, role, friends, dates)
```

```
## # A tibble: 16 x 5
##   personality role      friends dates    n
##   <fct>      <fct>      <fct>  <fct> <int>
## 1 Modern    Modern    Low    Low    11
## 2 Modern    Modern    Low    High   13
## 3 Modern    Modern    High   Low    11
## 4 Modern    Modern    High   High   17
## 5 Modern    Traditional Low    Low    12
## 6 Modern    Traditional Low    High   12
## 7 Modern    Traditional High   Low    6
## 8 Modern    Traditional High   High    8
## 9 Traditional Modern    Low    Low   10
## 10 Traditional Modern    Low    High    8
## 11 Traditional Modern    High   Low    4
```

```
## 12 Traditional Modern      High    High      9
## 13 Traditional Traditional Low     Low      12
## 14 Traditional Traditional Low     High     29
## 15 Traditional Traditional High    Low      13
## 16 Traditional Traditional High    High     33
```

The arguments in `count` are the variables which we use for counting. For instance, compare the result above to

```
unaggregated %>% count(personality, dates)
```

```
## # A tibble: 4 x 3
##   personality dates      n
##   <fct>        <fct> <int>
## 1 Modern      Low      40
## 2 Modern      High     50
## 3 Traditional Low      39
## 4 Traditional High     79
```

TopHat Exercises. Answer the following questions using the `interfaith.dat` dataset

- What is the percentage of low socioeconomic status individuals in an interfaith relationship?
- What is the percentage of high socioeconomic status individuals in an interfaith relationship?
- What is the value of (% interfaith relationship among men) - (% interfaith relationship among women)?
- Let's consider protestants only. What is the value of (% interfaith relationship among men) - (% interfaith relationship among women)?
- Let's consider catholics only. What is the value of (% interfaith relationship among men) - (% interfaith relationship among women)?

gather: from wide format to long format

Suppose you want to compare outcomes with 3 treatments, and your data look like this

```
wide
```

```
## # A tibble: 5 x 3
##   Treat1 Treat2 Treat3
##   <dbl> <dbl> <dbl>
## 1 -1.60  0.983 -0.2
## 2  0.409 -0.671 -0.022
## 3 -0.019 -0.313 -1.74
## 4 -0.251  3.25  1.88
## 5  0.306  2.06 -0.083
```

Some people would say that the data is in “wide format.”

Data in wide format aren't convenient for running our analyses: if you want to run statistical methods or create plots, most R functions expect to have all the outcomes in one column, and the categories (treatments) in another column. This alternative formatting is called “long format”. You can go from wide to long format using `gather` in `library(tidyr)`.


```
data %>% gather(key=treatment, value=outcome, Treat1, Treat2, Treat3)
```

```
## # A tibble: 15 x 2
##   treatment outcome
##   <chr>      <dbl>
## 1 Treat1    -1.60
## 2 Treat1     0.409
## 3 Treat1    -0.019
## 4 Treat1    -0.251
## 5 Treat1     0.306
## 6 Treat2     0.983
## 7 Treat2    -0.671
## 8 Treat2    -0.313
## 9 Treat2     3.25
##10 Treat2     2.06
##11 Treat3    -0.2
##12 Treat3    -0.022
##13 Treat3    -1.74
##14 Treat3     1.88
##15 Treat3    -0.083
```

The first argument in `gather` is for naming the new column that contains the categories (the `key`), the second one is for naming the column where the new outcomes will be stored (the `value`), and then you write the names of the columns that contain the outcomes you want to `gather`. An equivalent way of writing the same thing is:

```
data %>% gather(key=treatment, value=outcome, Treat1:Treat3)
```

```
## # A tibble: 15 x 2
##   treatment outcome
##   <chr>      <dbl>
## 1 Treat1    -1.60
## 2 Treat1     0.409
## 3 Treat1    -0.019
## 4 Treat1    -0.251
## 5 Treat1     0.306
## 6 Treat2     0.983
## 7 Treat2    -0.671
## 8 Treat2    -0.313
## 9 Treat2     3.25
##10 Treat2     2.06
##11 Treat3    -0.2
##12 Treat3    -0.022
##13 Treat3    -1.74
##14 Treat3     1.88
##15 Treat3    -0.083
```

In `Treat1:Treat3` we gave R a range of columns which we want to `gather` (first to last). This is useful if you have many variables.

What if your data is in wide format, but you have an uneven number of observations? That is, your data looks something like this

```
uneven
```

```
## # A tibble: 5 x 3
##   Treat1 Treat2 Treat3
##   <dbl> <dbl> <dbl>
## 1 -1.60  0.983 -0.2
## 2  0.409 -0.671 -0.022
## 3 -0.019 -0.313 -1.74
## 4 -0.251  3.25  NA
## 5  NA      2.06  NA
```

Let's try to gather:

```
uneven %>% gather(key = treatment, value = outcome, Treat1:Treat3)
```

```
## # A tibble: 15 x 2
##   treatment outcome
##   <chr>      <dbl>
## 1 Treat1    -1.60
## 2 Treat1     0.409
## 3 Treat1    -0.019
## 4 Treat1    -0.251
## 5 Treat1     NA
## 6 Treat2     0.983
## 7 Treat2    -0.671
## 8 Treat2    -0.313
## 9 Treat2     3.25
## 10 Treat2    2.06
## 11 Treat3    -0.2
## 12 Treat3   -0.022
## 13 Treat3   -1.74
## 14 Treat3    NA
## 15 Treat3    NA
```

Unfortunately, we get some NAs. We can get rid of them with `na.omit`:

```
uneven %>% gather(key = treatment, value = outcome, Treat1:Treat3) %>%
  na.omit
```

```
## # A tibble: 12 x 2
##   treatment outcome
##   <chr>      <dbl>
## 1 Treat1    -1.60
## 2 Treat1     0.409
## 3 Treat1    -0.019
## 4 Treat1    -0.251
## 5 Treat2     0.983
## 6 Treat2    -0.671
## 7 Treat2    -0.313
## 8 Treat2     3.25
## 9 Treat2     2.06
## 10 Treat3    -0.2
## 11 Treat3   -0.022
## 12 Treat3   -1.74
```

spread: from long to wide format

If you want to go from long to wide format, you can use `spread`.

For example, if your data are

```
data2
```

```
## # A tibble: 15 x 3
## # Groups:   treatment [3]
##   treatment outcome   ind
##   <chr>      <dbl> <int>
## 1 Treat1     -1.60     1
## 2 Treat1      0.409     2
## 3 Treat1     -0.019     3
## 4 Treat1     -0.251     4
## 5 Treat1      0.306     5
## 6 Treat2      0.983     1
## 7 Treat2     -0.671     2
## 8 Treat2     -0.313     3
## 9 Treat2      3.25     4
##10 Treat2      2.06     5
##11 Treat3     -0.2      1
##12 Treat3     -0.022     2
##13 Treat3     -1.74     3
##14 Treat3      1.88     4
##15 Treat3     -0.083     5
```

You can convert it to wide format as follows

```
data2 %>% spread(treatment, outcome)
```

```
## # A tibble: 5 x 4
##   ind Treat1 Treat2 Treat3
##   <int> <dbl> <dbl> <dbl>
## 1     1 -1.60  0.983 -0.2
## 2     2  0.409 -0.671 -0.022
## 3     3 -0.019 -0.313 -1.74
## 4     4 -0.251  3.25  1.88
## 5     5  0.306  2.06 -0.083
```

Note that `data2` isn't just our dataset that came out of `gathering`. In fact, if we start with

```
gath = data %>% gather(key=treatment, value=outcome, Treat1:Treat3)
```

and we try to `spread`, we'll get an error. R complains because the rows of `gath` aren't uniquely identifiable. A way to get around that is creating index variables within the treatments

```
gath = gath %>% group_by(treatment) %>% mutate(id=row_number())
gath
```

```
## # A tibble: 15 x 3
## # Groups:   treatment [3]
##   treatment outcome    id
##   <chr>      <dbl> <int>
## 1 Treat1     -1.60     1
## 2 Treat1      0.409     2
## 3 Treat1     -0.019     3
## 4 Treat1     -0.251     4
## 5 Treat1      0.306     5
## 6 Treat2      0.983     1
## 7 Treat2     -0.671     2
## 8 Treat2     -0.313     3
## 9 Treat2      3.25     4
## 10 Treat2     2.06     5
## 11 Treat3     -0.2      1
## 12 Treat3     -0.022     2
## 13 Treat3     -1.74     3
## 14 Treat3      1.88     4
## 15 Treat3     -0.083     5
```

and then, we can spread (and get rid of id):

```
gath %>% spread(treatment, outcome) %>% select(-id)
```

```
## # A tibble: 5 x 3
##   Treat1 Treat2 Treat3
##   <dbl> <dbl> <dbl>
## 1 -1.60  0.983 -0.2
## 2  0.409 -0.671 -0.022
## 3 -0.019 -0.313 -1.74
## 4 -0.251  3.25  1.88
## 5  0.306  2.06 -0.083
```

Creating, modifying, and renaming variables

We can use `mutate` if we want to transform/create new variables. Let's load in the `hsb2` dataset and use it in our examples.

```
hsb2 = read.csv("http://vicpena.github.io/sta9750/spring19/hsb2.csv") %>% select(-X, -id)
```

For example, if we want to create a new variable called `avg` which contains the average score in `read`, `write`, `science`, and `socst`:

```
hsb2 = hsb2 %>% mutate(avg=(read+write+science+socst)/4)
```

One would expect `hsb2 %>% mutate(avg=mean(read,write,math,science,socst))` to work, but it doesn't. The problem is that `mutate` operates by columns. We can force R to operate by row using `rowwise`:

```
hsb2 %>% rowwise() %>% mutate(avg=mean(c(read,write,science,socst)))
```

```
## Source: local data frame [200 x 11]
## Groups: <by row>
##
## # A tibble: 200 x 11
##   gender race    ses schtyp prog   read write  math science socst  avg
##   <fct> <fct> <fct> <fct> <fct> <int> <int> <int> <int> <int> <dbl>
## 1 male   white   low   public gene~    57    52    41     47    57  53.2
## 2 female white  midd~ public voca~    68    59    53     63    61  62.8
## 3 male   white  high   public gene~    44    33    54     58    31  41.5
## 4 male   white  high   public voca~    63    44    47     53    56  54
## 5 male   white  midd~ public acad~    47    52    57     53    61  53.2
## 6 male   white  midd~ public acad~    44    52    51     63    61  55
## 7 male   africa~ midd~ public gene~    50    59    42     53    61  55.8
## 8 male   hispan~ midd~ public acad~    34    46    45     39    36  38.8
## 9 male   white  midd~ public gene~    63    57    54     58    51  57.2
## 10 male  africa~ midd~ public acad~    57    55    52     50    51  53.2
## # ... with 190 more rows
```

We can use `transmute` to create new variables and keep only the new variables that we create. For example, if we want to compute the average of the scores and a new variable called `white` that takes on the values `white` if the student is white and `nonwhite` otherwise:

```
hsb2new = hsb2 %>% transmute(white = ifelse(race == "white", "white", "nonwhite"), avg = (read+write+sc
```

The function `ifelse` expects 3 arguments. The first one is a logical condition. The second argument is the value that should be assigned if the condition is `TRUE`. The third argument is the value that should be assigned if the condition is `FALSE`.

If we want to apply the same transformation to more than one variable, we can use the `mutate_at`. For example, if we want to convert the test scores (in grade %) to z-scores:

```
hsb2_zscores = hsb2 %>% mutate_at(c("read", "write", "math", "science", "socst"), scale)
head(hsb2_zscores)
```

```
##   gender race    ses schtyp    prog   read   write    math
## 1  male white   low  public  general 0.4652326 -0.08176325 -1.24300207
## 2 female white middle public vocational 1.5380959 0.65674353 0.03789315
## 3  male white   high  public  general -0.8026968 -2.08628164 0.14463442
## 4  male white   high  public vocational 1.0504307 -0.92577099 -0.60255446
## 5  male white middle public  academic -0.5100977 -0.08176325 0.46485822
## 6  male white middle public  academic -0.8026968 -0.08176325 -0.17558939
##   science    socst  avg
## 1 -0.4898549 0.4280075 53.25
## 2 1.1261613 0.8005929 62.75
## 3 0.6211562 -1.9937977 41.50
## 4 0.1161512 0.3348611 54.00
## 5 0.1161512 0.8005929 53.25
## 6 1.1261613 0.8005929 55.00
```

We can use `mutate_at` for type conversions. Let's read in the `femrole` dataset again

```
femrole = read.table("http://users.stat.ufl.edu/~winner/data/femrole.dat", header=F)
summary(femrole)
```

```
##           V1           V2           V3           V4           V5
## Min.      :1.0    Min.      :1.0    Min.      :1.0    Min.      :1.0    Min.      : 4.00
## 1st Qu.:1.0    1st Qu.:1.0    1st Qu.:1.0    1st Qu.:1.0    1st Qu.: 8.75
## Median :1.5    Median :1.5    Median :1.5    Median :1.5    Median :11.50
## Mean      :1.5    Mean      :1.5    Mean      :1.5    Mean      :1.5    Mean      :13.00
## 3rd Qu.:2.0    3rd Qu.:2.0    3rd Qu.:2.0    3rd Qu.:2.0    3rd Qu.:13.00
## Max.      :2.0    Max.      :2.0    Max.      :2.0    Max.      :2.0    Max.      :33.00
```

Variables V1, V2, V3, and V4 are actually categorical and we want to convert them to **factors**. In a previous section, we did the type conversion one variable at a time. A shorter way of converting the variables to factors is

```
femrole2 = femrole %>% mutate_at(c("V1", "V2", "V3", "V4"), as.factor)
str(femrole2)
```

```
## 'data.frame':    16 obs. of  5 variables:
## $ V1: Factor w/ 2 levels "1","2": 1 1 2 2 1 1 2 2 1 1 ...
## $ V2: Factor w/ 2 levels "1","2": 1 2 1 2 1 2 1 2 1 2 ...
## $ V3: Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 2 2 ...
## $ V4: Factor w/ 2 levels "1","2": 1 1 1 1 2 2 2 2 1 1 ...
## $ V5: int   11 12 10 12 13 12 8 29 11 6 ...
```

If we want to use `mutate_at` by specifying the columns on which the transformation won't be applied, we have to be a little careful: we have to add `vars()` to our command.

```
femrole2 = femrole %>% mutate_at(vars(-V5), as.factor)
str(femrole2)
```

```
## 'data.frame':    16 obs. of  5 variables:
## $ V1: Factor w/ 2 levels "1","2": 1 1 2 2 1 1 2 2 1 1 ...
## $ V2: Factor w/ 2 levels "1","2": 1 2 1 2 1 2 1 2 1 2 ...
## $ V3: Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 2 2 ...
## $ V4: Factor w/ 2 levels "1","2": 1 1 1 1 2 2 2 2 1 1 ...
## $ V5: int   11 12 10 12 13 12 8 29 11 6 ...
```

If we want to transform variables that satisfy a certain logical condition, we can use `mutate_if`. For example, let's take a look at a `summary(femrole)`.

```
summary(femrole)
```

```
##           V1           V2           V3           V4           V5
## Min.      :1.0    Min.      :1.0    Min.      :1.0    Min.      :1.0    Min.      : 4.00
## 1st Qu.:1.0    1st Qu.:1.0    1st Qu.:1.0    1st Qu.:1.0    1st Qu.: 8.75
## Median :1.5    Median :1.5    Median :1.5    Median :1.5    Median :11.50
## Mean      :1.5    Mean      :1.5    Mean      :1.5    Mean      :1.5    Mean      :13.00
## 3rd Qu.:2.0    3rd Qu.:2.0    3rd Qu.:2.0    3rd Qu.:2.0    3rd Qu.:13.00
## Max.      :2.0    Max.      :2.0    Max.      :2.0    Max.      :2.0    Max.      :33.00
```

The maximum value that variables V1 through V4 can take on is 2. Therefore, we can create a filter that checks if the maximum of a variable is 2 or not, and apply `as.factor` as needed. Unfortunately, `mutate_if` expects an argument that is a function that will be applied to each of the columns. The following command works

```
femrole2 = femrole %>% mutate_if( ~ max(.) == 2, as.factor)
str(femrole2)
```

```
## 'data.frame':    16 obs. of  5 variables:
## $ V1: Factor w/ 2 levels "1","2": 1 1 2 2 1 1 2 2 1 1 ...
## $ V2: Factor w/ 2 levels "1","2": 1 2 1 2 1 2 1 2 1 2 ...
## $ V3: Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 2 2 ...
## $ V4: Factor w/ 2 levels "1","2": 1 1 1 1 2 2 2 2 1 1 ...
## $ V5: int   11 12 10 12 13 12 8 29 11 6 ...
```

The tilde `~` indicates that what comes after will be a function. Within the function, the argument is denoted with `..`. An equivalent (and perhaps easier to understand) way to do this is the following. First, define a function that checks whether the maximum of a variable `x` is 2 or not:

```
max2 = function(x) { max(x) == 2 }
```

Then, you can use `max2` in `mutate_if`:

```
femrole2 = femrole %>% mutate_if(max2, as.factor)
str(femrole2)
```

```
## 'data.frame':    16 obs. of  5 variables:
## $ V1: Factor w/ 2 levels "1","2": 1 1 2 2 1 1 2 2 1 1 ...
## $ V2: Factor w/ 2 levels "1","2": 1 2 1 2 1 2 1 2 1 2 ...
## $ V3: Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 2 2 ...
## $ V4: Factor w/ 2 levels "1","2": 1 1 1 1 2 2 2 2 1 1 ...
## $ V5: int   11 12 10 12 13 12 8 29 11 6 ...
```

Same thing.

There is a `select_if` function that works the same way as `mutate_if`. For example, if, after doing the type conversion, we want to create a subset that only contains the factors:

```
femrole3 = femrole2 %>% select_if(is.factor)
str(femrole3)
```

```
## 'data.frame':    16 obs. of  4 variables:
## $ V1: Factor w/ 2 levels "1","2": 1 1 2 2 1 1 2 2 1 1 ...
## $ V2: Factor w/ 2 levels "1","2": 1 2 1 2 1 2 1 2 1 2 ...
## $ V3: Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 2 2 ...
## $ V4: Factor w/ 2 levels "1","2": 1 1 1 1 2 2 2 2 1 1 ...
```

We can also rename variables. If we want to change `ses` to `status`:

```
hsb2 = hsb2 %>% rename(status = ses)
```

In “old R”, we can rename columns by indexing `colnames()`.

Some utility functions for identifying columns

Sometimes we want to select or transform columns that satisfy some condition. The functions `starts_with()`, `ends_with()`, `contains()`, and `num_range()` can help us get the subsets we want.

For example, let's go back to the `iris` dataset. If we want to select the variables that have to do with the sepal of the flower, we can use

```
sepal = iris %>% select(starts_with("Sepal"))
```

If we want the variables that have to do with widths:

```
width = iris %>% select(ends_with("Width"))
```

In fact, we didn't need to use `start_with` or `ends_with`. We could've used `contains`, which checks if a column contains the string or not.

`num_range()` is useful in datasets where there are variables whose names are something like a prefix, followed by a number. That is, something like `V1`, `V2`, etc. For example, in the unformatted `femrole` dataset, we can select the columns `V1` through `V4` as follows

```
fem14 = femrole %>% select(num_range("V", 1:4))
```

This example is a little silly, because we could've just written

```
fem14 = femrole %>% select(1:4)
```

Or even

```
fem14 = femrole %>% select(V1:V4)
```

The advantage of `num_range()` is that it works even if the columns are all scrambled. For example, try applying the code above to the dataset

```
fem_scramble = femrole %>% select(2,3,1,5,4)
head(fem_scramble)
```

```
##   V2 V3 V1 V5 V4
## 1  1  1  1 11  1
## 2  2  1  1 12  1
## 3  1  1  2 10  1
## 4  2  1  2 12  1
## 5  1  1  1 13  2
## 6  2  1  1 12  2
```

In this case, only `num_range()` will get it right (and it will rearrange the order of the columns).

Obtaining summaries by categories of variables

We can create objects which contain summaries for different groups by combining `group_by` and `summarize`:


```
hsb2 %>% group_by(race) %>% summarize(medMath = median(math), sdMath = sd(math))
```

```
## # A tibble: 4 x 3
##   race          medMath sdMath
##   <fct>        <dbl>  <dbl>
## 1 african american    45    6.49
## 2 asian              61   10.1
## 3 hispanic           47    6.98
## 4 white              54    9.38
```

And we can combine these function with the other functions we learned today. For example:

```
hsb2 %>% group_by(race) %>% filter(math > 70) %>% summarize(n=n())
```

```
## # A tibble: 2 x 2
##   race      n
##   <fct> <int>
## 1 asian     1
## 2 white     9
```

Tells us that there are 10 people who got a `math` score greater than 70, and that 1 of them is `asian` and 9 of them are `white`.

References

- [dplyr cheat sheet](#)
- [tidyr website](#)
- [Tutorial by Bradley Boehmke](#)
- [Tutorial by Olivia L. Holmes](#)
- Chapter 12 of *R Programming for Data Science*, by Roger D. Peng