Data manipulation and I/O

December 11, 2017

Data structures (cont.)

`data.frame` is the building block for most of what we will do in data analysis. Think about them as a `matrix` that can hold columns of different types and with column names.

```
states <- data.frame("code"       = c("CA", "NY", "NE", "AZ"),
                     "population" = c(38.8, 19.7, 2.1, 6.8),
                     "region"     = c("West", "Northeast", "Midwest", "West"),
                     "landlock"   = c(FALSE, FALSE, TRUE, TRUE))
```

We can access elements via indexing the same way as we would do in a `matrix` or we can access by name:

```
states[, 3]
states[, "region"]
states$region
```

We can also add new variables:

```
states$spanish <- c(28.5, 15.7, NA, 19.5)
```

Transformation of variables is straightforward:

```
states$spanish <- states$spanish * states$population # But we could have used a new variable
```

But we will see soon that we don't need to do most of this transformations.

The same approach can be used for subsetting a dataset, i.e., selecting rows from a `data.frame` based on given values:

```
states[states$population > 10,] # Notice the comma!
subset(states, population > 10)
```

Let's take the code apart. We want to take the dataset `states` and keep only the rooms for which the statement `states$population > 10` is true. As we saw before, the comparison actually returns a logical vector that has value `TRUE` when the condition is met. We can then select rows from the data by simply passing this logical vector to the dimension that captures the rows.

We can use the same strategy to do recoding:

```
states$population[states$code == "NE"] <- 1.8
```

Carefully examine what's happening in the previous line. `states$code` gets the column `code` in our dataset. `states$code == "NE"` returns a logical vector in which only the third observation will be `TRUE`. `states$population[states$code == "NE"]` access the `population` value for Nebraska. And then we assign the correct value 1.8 to it. The underlying idea is that we select the observations that we want to transform and replace them.

The dataset is also useful to think about variable types. Consider the variable `region`: it is a vector of characters, but we would like to consider it as a discrete variable in which we would represent it as a value (a number) with a label (the region name). That's a `factor` in R.

```
states$region <- factor(states$region)
states$region
levels(states$region)
```

Depending on who you talk to, they may hate them or love them.

Input/Output

We can write our dataset to disk in comma-separated format using the `write.csv` function.

```
write.csv(states, file="states.csv")
```

Note the named argument to indicate the filename. We could have specified any other folder/directory by passing a path. In the previous call, the file will be written to our current working directory. We can see which one it is with:

```
getwd()
```

```
## [1] "H:/teaching/westraining-R/intro-to-R/src"
```

and we can use the `setwd()` to set it.

Unsurprisingly, we can read our dataset back using `read.csv`.

```
states <- read.csv("states.csv")
states
```

What about other delimiters and even formats? For other delimiters, we can use the more general function `read.table` and `write.table` that allows us to specify which delimiter we want to use. Actually, `read.csv` is just `read.table` with a predefined delimiter

```
states <- read.table("states.csv", sep=",")
```

and we could have, for instance, defined that our input is tab separated by specifying

```
states <- read.table("states.csv", sep="\t")
```

The most common format for R uses the extension `.RData` (or `RDS`), using the functions `save` (`saveRDS`) and `load` (`readRDS`):

```
save(states, file="states.RData")
load("states.RData")
states
```

But R can also read (and sometimes write) data in other binary formats: data coming from Stata, SAS, SPSS, or even Excel. The functions to handle these foreign formats are provided in the `foreign` package that comes with `R` but a much better alternative is the `haven` package which gives us access to functions like `read_sas` to read SAS sas7bdat files or `read_stata` to read Stata dta files.

This package is not provided with R but we need instead to install it and loaded, which gives us a good opportunity to look at importing new functionality into R.

We first need to install the library using:

```
install.packages("haven")
```

```
## Warning: package 'haven' is in use and will not be installed
```

The function will hit a CRAN mirror, download the file for the package that we want, and perform the installation routine (which includes a number of checks). Now the package is available in our system, we can load it into our session:

```
library(haven)
```

For instance, we can now load an *uncompressed* SAS file using

```
path <- system.file("examples", "iris.sas7bdat", package="haven")
read_sas(path)
```

```
## # A tibble: 150 x 5
##    Sepal_Length Sepal_Width Petal_Length Petal_Width Species
##           <dbl>       <dbl>        <dbl>       <dbl>   <chr>
##  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
##  7          4.6         3.4          1.4         0.3  setosa
##  8          5.0         3.4          1.5         0.2  setosa
##  9          4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## # ... with 140 more rows
```

or we could read a Stata file using

```
path <- system.file("examples", "iris.dta", package="haven")
read_stata(path)
```

```
## # A tibble: 150 x 5
##    sepallength sepalwidth petallength petalwidth species
##          <dbl>      <dbl>       <dbl>      <dbl>   <chr>
##  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
##  7         4.6        3.4         1.4        0.3  setosa
##  8         5.0        3.4         1.5        0.2  setosa
##  9         4.4        2.9         1.4        0.2  setosa
## 10         4.9        3.1         1.5        0.1  setosa
## # ... with 140 more rows
```

As a matter of fact, one of the good things about `R` is that it interacts nicely with many other programs. For instance, it can read Excel spreadsheets in several different ways. One option is `readxl` library, which does not require Java. So first, like before, we need to install the package and load it:

```
install.packages("readxl")
```

```
## Warning: package 'readxl' is in use and will not be installed
```

```
library(readxl)
```

Now we will be able to call a function like `read_excel`

```
read_excel(path)
```

```
## # A tibble: 3 x 3
##      B3    C3    D3
##   <chr> <chr> <chr>
## 1    B4    C4    D4
## 2    B5    C5    D5
## 3    B6    C6    D6
```

Each of these function has a number of options (Do we want to use a catalog file for SAS? In which sheet is the data in our Excel file? How to deal with user generated missing values in SPSS?) and of course we can interact with many other formats (`feather` for data exchange with Python, fixed width files, hd5, ...), connections (PostgreSQL, MongoDB, Microsoft Access, SQL Server, ...), streams (for real-time analysis), ... but this is sufficient for now. Let's move on to doing something with the data.