WEEK 4: VARIABLES AND BASICS OF DATA FRAMES

THOMAS ELLIOTT

1. Variables

1.1. **Storage Types.** This is a simplified discussion of storage types, I will focus on those you are most likely to encounter when doing basic data management and analysis. You can search online for more in depth discussions of storage types in R.

Like Stata, R can have different types of variables, meaning the variables contain different types of data. However, R has more types of variables than Stata. The following are the different storage types you are likely to encounter:

boolean: While Stata evaluates boolean expressions to 1=true and 0=false, R has a boolean storage type and so boolean expressions will return TRUE and FALSE. So in Stata, the expression `gen female = sex == "female"` will create a numeric dummy variable, where 1 = female and 0 = male. In R, the same expression would be `female<-sex=="female"` but female would actually contain TRUE = female and FALSE = male. However, if you use female in a function that expects female to be numeric, it will usually be converted to TRUE=1 and FALSE=0, so `mean(female)` will give the proportion of female in R as it would in Stata.

numeric, integer, double: R stores numeric variables in numeric objects. R can also distinguish between integer values versus double (or decimal) values.

complex: R can also store complex numbers — recall that complex numbers are of the form $a + bi$ where a and b are real numbers and $\sqrt{i} = -1$. R stores these numbers in their own type of storage mode.

character: Characters or strings are stored in character objects. R has built in functions for manipulating strings, though there are packages that do so in better ways.

1.2. **Naming Variables.** In R, object names can contain any valid letter, number, the period or the underscore. Objects must start with either a letter, or a dot not followed by a number (so `.dv` is a valid variable name but `.2way` is not). There are no strict naming conventions in R (to the consternation of many with a computer programming background), but it is common to use the period to separate words in a variable name (e.g. `African.American`). As in Stata, variable and object names are case-sensitive, so it is valid to have three different variables, `sex`, `Sex`, and `SEX` with different contents. Obviously, this isn't a good idea so try to avoid.

Date: January 11, 2016.

In Stata you can assign a variable label to better describe the contents of the variable. Unfortunately, there is no equivalent in R, so variable names should be as descriptive as possible without being cumbersome.

2. Data Frames

Data frames are R's objects for storing data sets. They are two-dimensional objects, with rows being interpreted as cases or observations, and columns being interpreted as variables. Columns can have different data storage types, they can even be more complex objects. In other words, you could have a column in your data frame that stores other data frames. It is strongly recommended that you do not do this, but it is possible:

```
> mydata<-data.frame(a=1:3,b=4:6)
> anotherdf<-data.frame(c=7:9,d=10:12)
> mydata$aDF=anotherdf
> mydata$aArray=as.array(c(1,6,9))
> mydata
  a b aDF.c aDF.d aArray
1 1 4     7    10      1
2 2 5     8    11      6
3 3 6     9    12      9
> names(mydata)
[1] "a"      "b"      "aDF"    "aArray"
> sapply(mydata,class)
          a           b         aDF      aArray
  "integer"   "integer" "data.frame"     "array"
```

Notice that aDF exists as one column in the data frame, even though both columns of aDF are printed when you print mydata. This is why you shouldn't have columns which are complex object types in a data frame - it makes things confusing.

As you can see from the code example above, you can define a data frame using the `data.frame()` function, and the arguments are `key=value` pairs defining variable names and variable values. Importing data from a spreadsheet will automatically store the data in a data frame.

2.1. Accessing Rows and Columns in a Data Frame.

You can access rows and columns in a data frame using indexing - a method that also works for vectors, arrays, and matrices. Indexing uses square brackets enclosing the coordinates of the data you want to extract from the data objects, separated by commas. Since data frames are two dimensional, you need two numbers to define the coordinates. The first number is the row number and the second number is the column number. In R, indexing begins at 1[1], so 1 = the first row or the first column.

```
> mydata<-data.frame(x=1:10,y=20:11)
> mydata
```

[1]Other languages, such as python begin indexing at 0, so 0 refers to the first item and 1 refers to the second item

```
    x  y
1   1 20
2   2 19
3   3 18
4   4 17
5   5 16
6   6 15
7   7 14
8   8 13
9   9 12
10 10 11
> mydata[1,1]
[1] 1
> mydata[5,2]
[1] 16
```

To select an entire row, or an entire column, leave the appropriate index blank (but still include the comma:

```
#output the fourth row
> mydata[4,]
  x  y
4 4 17
#output the second column
> mydata[,2]
 [1] 20 19 18 17 16 15 14 13 12 11
```

Notice that the row is returned as a data frame, but the column is returned as a vector. While this may seem a little confusing now, it actually makes working with data frames a lot easier. You can create subsamples of your data by selecting a range of rows and a data frame will be returned. If you are trying to recode a column, it is easier to do this if individual columns are interpreted as vectors. If you need the single column returned as a data frame (usually comes up if you are doing complex data reshaping or matrix algebra), you can specify the `drop=FALSE` argument to the index. Drop refers to dropping unnecessary dimensions.

```
> mydata[,2,drop=FALSE]
    y
1  20
2  19
3  18
4  17
5  16
6  15
7  14
8  13
9  12
10 11
```

Another feature of indexing is you can select multiple rows and/or columns at once. You can either do this with slicing, which is indicating a range of rows with the notation `s:e` where s is the row/column you want to start and e is the row/column you want to end. Notice I used this notation when I created the data frame in the example above, and also notice that if s is greater than e, R will return the range in reverse order (so y above is 11-20 but in reverse order). You can also select rows and columns by passing a vector of indices you want to select with the `c()` function.

```
#Slice out rows 1 through 4
> mydata[1:4,]
  x  y
1 1 20
2 2 19
3 3 18
4 4 17
#Slice out columns 1 through 2
> mydata[,1:2]
    x  y
1   1 20
2   2 19
3   3 18
4   4 17
5   5 16
6   6 15
7   7 14
8   8 13
9   9 12
10 10 11
#Slice out rows 1 through 4 and columns 1 through 2
> mydata[1:4,1:2]
  x  y
1 1 20
2 2 19
3 3 18
4 4 17
#Select rows 1, 3, 6, and 8
> mydata[c(1,3,6,8),]
  x  y
1 1 20
3 3 18
6 6 15
8 8 13
#Slice our rows 2 through 5 but in reverse order
> mydata[5:2,]
  x  y
5 5 16
4 4 17
```

```
3 3 18
2 2 19
```

Another way to select columns in a data frame is using the `$` operator. The syntax is `dataframe$column.name`, which will return the column as a vector, similar to the way indexing a single column returned a vector:

```
> mydata$y
 [1] 20 19 18 17 16 15 14 13 12 11
```

You can also do assignment with this notation, creating new variables or overwriting existing ones:

```
> mydata$z<-c(30:39)
> mydata
    x  y  z
1   1 20 30
2   2 19 31
3   3 18 32
4   4 17 33
5   5 16 34
6   6 15 35
7   7 14 36
8   8 13 37
9   9 12 38
10 10 11 39
```

And since columns are returned as vectors, you can use indexing on the columns to both return data points and change them:

```
> mydata$z[3]
[1] 32
> mydata$z[3]<-99
> mydata
    x  y  z
1   1 20 30
2   2 19 31
3   3 18 99
4   4 17 33
5   5 16 34
6   6 15 35
7   7 14 36
8   8 13 37
9   9 12 38
10 10 11 39
```

3. Drop/Keep/Filtering

In Stata, `drop` and `keep` allow you to both drop variables and cases, depending on how you use them. Unfortunately, dropping columns in R is not as straightforward as in Stata (without using third party packages — more on that later). One way to drop columns or rows is to using indexing but putting a negative in front of the columns/rows you want to drop:

```
#Remove the first column
> mydata[,-1]
    y  z
1  20 30
2  19 31
3  18 99
4  17 33
5  16 34
6  15 35
7  14 36
8  13 37
9  12 38
10 11 39
#remove rows 3 through 6
> mydata[-c(3:6),]
    x  y  z
1   1 20 30
2   2 19 31
7   7 14 36
8   8 13 37
9   9 12 38
10 10 11 39
```

Notice that in order to do negative slicing, you need to wrap the slice in the `c()` function. This is because without `c()`, R interprets the slice as -3 through 6, which throws an error.

Dropping columns using column names (which is often much easier than trying to figure out the index number of a column, especially for very large datasets) is not as easy. To understand how this works, it is worth stepping back to investigate another feature of indexing. When indexing, you can pass the row or column indexes of the data you are interested in, but you can also pass a boolean vector equal to the number of rows or columns. R interprets TRUE in the boolean vector as wanting to keep the item at the index, and FALSE as dropping the item at the index. So to drop column names, we need to construct a boolean vector that is TRUE for all columns except those we want to drop. To do this, we'll need to access a list of all column names in the data frame, which we can do with the `names()` function.

```
> names(mydata)
[1] "x" "y" "z"
> mydata[,names(mydata)!="y"]
    x  z
```

```
1   1 30
2   2 31
3   3 99
4   4 33
5   5 34
6   6 35
7   7 36
8   8 37
9   9 38
10 10 39
```

The `names()` function returns a character vector containing the names of the columns of the data frame. In the next command, I construct a boolean statement that evaluates to TRUE for columns I want to keep and FALSE to columns I want to drop. To drop multiple columns, a useful operator is the `%in%` operator. This operator compares values on the left hand side to see if it appears anywhere in a vector on the right hand side. We can use this to pass a character vector containing the names we want to drop:

```
> mydata[,!names(mydata)%in%c("x","z"),drop=FALSE]
    y
1  20
2  19
3  18
4  17
5  16
6  15
7  14
8  13
9  12
10 11
```

We could exclude the `!` to list out the columns we want to KEEP instead.

We can use this boolean indexing to filter out rows by some column attribute. For example, we could filter out our data to exclude x values less than five:

```
> mydata[mydata$x>=5,]
    x  y  z
5   5 16 34
6   6 15 35
7   7 14 36
8   8 13 37
9   9 12 38
10 10 11 39
```

Since these are boolean expressions, we can string together as many as we want with and (`&`) and or (`|`) operators:

```
> mydata[mydata$x>=5&(mydata$y<=12|mydata$z<36),]
    x  y  z
```

```
5   5 16 34
6   6 15 35
9   9 12 38
10 10 11 39
```

3.1. **Dropping and Filtering with Dplyr.** As I've mentioned before, dplyr is a package that makes data wrangling in R much easier, and this is one of those contexts that dplyr really shines. Dplyr comes with several functions to keep and drop columns, as well as keep, drop, and filter rows. I'll briefly detail them here, but I would recommend reading over the relevant vignettes that dplyr comes with (type `browseVignettes("dplyr")` into R to see the available vignettes for dplyr).

First, dplyr comes with the `select()` command, which allows you to select, and deselect, columns from your data frame. The syntax for use is `select(data,col1,col2,...)`. On other words, you first pass the data frame you want to work with, then a comma separated list of columns you want to keep.

```
library(dplyr)
> select(mydata,x,z)
    x  z
1   1 30
2   2 31
3   3 99
4   4 33
5   5 34
6   6 35
7   7 36
8   8 37
9   9 38
10 10 39
```

You can also put a `-` before the column name and that column will be dropped.

```
> select(mydata,-y)
    x  z
1   1 30
2   2 31
3   3 99
4   4 33
5   5 34
6   6 35
7   7 36
8   8 37
9   9 38
10 10 39
```

Select even allows you to slice with column *names*.

```
> select(mydata,x:z)
```

```
    x  y  z
1   1 20 30
2   2 19 31
3   3 18 99
4   4 17 33
5   5 16 34
6   6 15 35
7   7 14 36
8   8 13 37
9   9 12 38
10 10 11 39
```

Dplyr also comes with the function `filter()` which allows you to filter out rows based on some criteria.

```
> filter(mydata,x>4)
   x  y  z
1  5 16 34
2  6 15 35
3  7 14 36
4  8 13 37
5  9 12 38
6 10 11 39
```

Notice that we can specify the column name we want to sort on directly. This is because `filter` (as well as most other functions in dplyr) knows to look for the variable names in mydata. This makes specifying variable names much easier and more readable, compared to filtering using the indexing method we used above. The second argument in the `filter()` command is just a boolean of the same length as the number of rows in your data frame, in which TRUE indicates the row should be kept and FALSE indicates the row should be dropped.

4. Renaming Variables

Renaming variables is another thing Stata does better than base R. To rename a variable in base R, you need to modify the object returned by `names()`. This requires knowing what the index number for the column we want to change is.

```
> names(mydata)[1]<-"a"
> mydata
    a  y  z
1   1 20 30
2   2 19 31
3   3 18 99
4   4 17 33
5   5 16 34
6   6 15 35
7   7 14 36
```

```
8   8 13 37
9   9 12 38
10 10 11 39
```

So in the code above, we replace the first name in the names object, which corresponds to the name of the first column in the data frame, with "a."

4.1. **Renaming with Dplyr.** This is another function that is much more straightforward with dplyr. The package comes with a `rename()` function, which is very intuitive to use. The syntax for use is `rename(data,newname=oldname)`.

```
> rename(mydata,a=x)
    a  y  z
1   1 20 30
2   2 19 31
3   3 18 99
4   4 17 33
5   5 16 34
6   6 15 35
7   7 14 36
8   8 13 37
9   9 12 38
10 10 11 39
```

5. Importing Data

R comes with many functions to import data from various sources. The most common source will be a spreadsheet. Unfortunately, R does not come with a function to import excel files directly (though there are some third party packages that do so, I haven't been impressed with any of them to date). So you'll need to resave any excel files you want to import as .csv files. To import a csv file, you will use the `read.csv()` function.

```
> gdp<-read.csv("gdp.csv")
> gdp
  year  gdp pop
1 1995 15.4 8.2
2 1996 16.1 8.4
3 1997 16.8 8.7
4 1998 17.3 9.0
5 1999 17.5 9.1
```

`read.csv()` has many optional arguments to modify how it imports the data so check out its help file to see the different ways you can modify the function. One thing I will point out is that, by default, `read.csv()` will import any columns with character values as factors (we will talk more about factors next week), rather than as characters. This can make working with those columns more difficult if you are expecting them to be characters. As a result, I almost always include the argument `stringsAsFactors=FALSE` when I import csv files.

5.1. **Importing From Other Sources.** R comes with a package called `foreign` which includes many functions to import data from other statistical packages. For example, to import a data file created by spss, you can use the function `read.spss()`. The library does include a `read.dta()` function for importing Stata files, unfortunately it only works with data files created with Stata 12 and earlier.

There is a third party package, `readstata13`, for importing Stata 13 and above stata data files, and includes the `read.dta13()` function to do so. I have not had a chance to try this out myself, so I don't know how well it imports data.