**Intro to Stata 2**

1. **The Objectives of Stata Tutorial 2**
   - To learn data storage types
   - To learn how Stata treats missing observations
   - To learn how to use *generate* and *replace*
   - To learn how to use *egen*
   - *generate* and *egen* are two commands that are very frequently used in Stata. Both of them are very powerful and indispensable commands in your data management and research work.

2. **Preparing for Tutorial 2**
   Let's create a directory called "tutorial2" under "C:\temp\stata-tutorial\" (under which you should find the directory "tutorial1" if you did not erase it after studying Stata Tutorial 1). We continue to use auto.dta to demonstrate Stata commands. Copy auto.dta into the new directory "tutorial2" and set the directory as the working directory (If you are confused, you must go back to Tutorial 1). Please load auto.dta in Stata's memory by the command *use*. To refresh your memory, here is a description of auto.dta. This dataset is about automobiles sold in the US market. The data set contains the name of the car (make), the price (price), miles per gallon (mpg), a repair record (rep78), weight (weight) and length (length) of the car, whether the car is made by a foreign car-manufacturing company or not (foreign), and other information (headroom, trunk, turn, displacement, gear_ratio).

3. **Data storage types**
   There are two main data storage types in Stata. One type is called *string*, and the other type is called *numeric*. There are five subcategories under the *numeric* data storage: *byte*, *int*, *long*, *float*, and *double*. You do not need to know the distinctions among them for a while, but they are different in that how precisely you need to store data (1 as opposed to 1.0) and how small or large the minimum and maximum values for the variable (-50 as the minimum value as opposed to -5,000).[1]

[1]Use *search* under the *help* menu, and type "datatypes" to see more about data storage types.

*String* variables often contain identifying information, such as the name of a sample individual. As a general rule, you can consider that all variables except ones for the identifying purpose are stored as *numeric*. What is good about *string* storage of data? First, you can record alphabets which are useful for recording names. Second, string variables can contain up to 244 characteristics including numbers and blanks, so you can store a very long series of numbers and alphabets.[2] For example, the identifying information of a sample individual could be given in a 10 digit number. Although you could store this as a *numeric* variable, it is more efficient to store the id variable as *string* with 10 characters long. Efficient here means a smaller data size.

Let's see the storage type of each variable in the currently-loaded data set. Type

```
describe
```

in the command line. You see that the variable *make* is stored as *string* with 18 characters long, and the rest of the variables are stored as *numeric*. In auto.dta, *make* contains the name of a car manufacturing company and the name of a car. Check this by using the Data Browser. In the Data Browser, all data shown in red fonts are stored as *string*, and all data shown in black or blue fonts are stored as *numeric*. Note that although you see the variable *foreign* contains alphabets, it is stored as *numeric* (blue fonts). This is because the variable *foreign* has *value labels* which you do not need to know at this time but later you will learn. Also, as we learned in Tutorial 1, the command *summarize* works only on *numeric* variables. If you *summarize string* variables, Stata returns zero observations (Confirm this using auto.dta).

4. **Missing observations**

We have learned *string* and *numeric* variables. For *string* variables, a blank indicates a missing observation. The variable *make* is stored as *string*. In the Data Browser, you see no blank in the column labelled *make*, meaning that there are no missing observations in *make*. For *numeric* variables, . (dot) represents a missing observation. In some data sets, missing observations for *numeric* variables are grouped into categories, each of which is represented by .a (dot a), .b

[2]I assume you use State SE. In an older Stata before Stata 12, different versions of Stata (such as Small Stata and Stata/IC) could be more restrictive in the maximum length of *string* storage.

(dot b), .c (dot c), ..., up to .z (dot z). They are called "extended missing values," and useful if you want to distinguish one type of missing (for example, not applicable) from another (for example, no response).[3] auto.dta does not contain extended missing values but uses . (dot) to represent missing observations for *numeric* variables.

There is a very important rule about missing observations you should know. Stata treats *numeric* missing observations as very large positive values. The ordering is

all non-missing numbers< . < .a < .b < ... < .z

You will later see why this is so important.

5. ***generate* and *replace***

The command *generate* (often abbreviated to *gen*) is used to create a new variable (or constant). As a simple example, type

gen xyz1=1

in the command line. Then, browse the data using the Data Browser. In the last non-empty column of the Data Browser, you see a new variable (constant) with the name xyz1, which is unity for all observations (rows) in the data set. For another example, type

gen xyz2=weight/length

in the command line. The new variable xyz2 shows the weight per unit length (lbs/inch) for each car in the data set. By using the *help* function, you can see how powerful the command *generate* is. Go to *search* under the *help* menu, and type "functions." In the pop-up window, click "functions" (blue fonts). You see 8 categories of functions available in Stata (math functions, density functions, and so on, each of which is clickable). In each category, you see functions you can use with the command *generate*. The simplest syntax for *generate* is:

generate *newvar=exp*

where *newvar* is whatever name you can choose for the new variable, and *exp* is any function or expression that is allowed in Stata. For example, if you need natural log of car prices, type

gen log_price=log(price)

in the command line. You can browse the data set to see that the new variable log_price has been created.

The command *replace* is equally important and is often used

[3]Extended missing values are available in Stata 8 and newer.

with *generate*. The command *replace* replaces the values of the existing variable by new values you specify. Suppose you want to create a dummy variable being equal to one for cars with prices less than \$5,000, and being equal to zero for cars with prices \$5,000 or higher. Then, you need to issue the following two commands:

```
gen affordable=0 (press return)
replace affordable=1 if price<5000 (press return)
```

The first command generates zero for all cars, and the second command replaces zero by one only for cars with prices less than \$5,000.

Next, suppose you want to create a dummy variable being equal to one for all cars with the number of repairs (rep78) less than 3 and being equal to zero for all cars with the number of repairs 3 or more. Let's call this dummy variable *reliable*. One twist with the variable *rep78* is that it contains missing observations.[4] The purpose of creating the dummy variable *reliable* is to distinguish relatively reliable cars from other cars. So, if a particular car does not contain information on the number of repairs, the dummy variable *reliable* should also indicate missing for the particular car. To do this, you need to type the following two commands:

```
gen reliable=0 if rep78<. (press return)
replace reliable=1 if rep78<3 (press return)
```

The first command creates zero only for cars with non-missing repair records. Remember that Stata treats all non-missing values as smaller values than numeric missing values . (dot). Then, the next command changes the values of *reliable* into one only for cars with the number of repairs less than 3. Please make sure to understand why the first command needs the condition "if rep78<." If you fail to include the condition "if rep78<." what problem do you see?

Finally, let's talk about how to *generate* a *string* variable and how to *replace* it. Let's start with a very simple example. Type

```
gen xyz3="aaa"
```

in the command line. This creates a *string* variable *xyz3* in which all observations (rows) are aaa. Note that whenever you create a *string* variable, you must use double quotes to tell Stata that the variable you are making is a *string* variable. You can change the contents of *xyz3*

[4]Given there are no sample cars with the number of repairs zero, . (dot) could mean zero repairs. However, we treat . (dot) as missing here.

by the command *replace*. Suppose you want to change the contents of *xyz3* into bbb only for *foreign* cars. Then, type

replace xyz3="bbb" if foreign==1

in the command line. Note again the double quotes used with bbb. Browse the data to see that all observations for *xyz3* are stored as *string* (red fonts).

Let's discuss another example of creating a *string* variable. The *string* variable *make* contains the name of a car-making company as well as the name of a car. Suppose you want to create another *string* variable that contains only the name of a car-making company. Let's call this *string* variable *company*. The first two characters of the contents of *make* are enough to tell you the car-making company for each sample car. Type

generate company=substr(make,1,2)

in the command line. *substr* is a *string* function you should check using the *help* function. *substr(make,1,2)* returns the substring of *make* starting at the 1st character for a length of 2. Browse the data to see how the new *string* variable *company* looks. Because *substr* is a *string* function, Stata can tell that the variable you are creating, *company*, is a *string* variable.

One rule that is implicit in the above discussion of *string* variables is that whenever you type the contents of a *string* variable, you must use double quotes. For example, suppose you want create a *string* variable that indicates the country of origin for each car-making company. Then, you can type
gen origin="USA" if company=="AM" | company=="Bu" | company=="Ca" | company=="Ch" | company=="Do" | company=="Fo" | company=="Li" | company=="Me" | company=="Ol" | company=="Pl" | company=="Po" (press return)
replace origin="Germany" if company=="Au" | company=="BM" | company=="VW" (press return)
replace origin="Japan" if company=="Da" | company=="Ho" | company=="Ma" | company=="Su" | company=="To" (press return)
and so on.

6. ***egen***

The command *egen* (extensions to *generate*) also creates a new variable (or constant). *generate* and *egen* handle different functions, so you must use *generate* for some functions and *egen* for other functions. As your experience in Stata accumulates, you will soon be able to tell which command, *generate* or *egen*, is appropriate to your case at hand. Soon you will see how useful *egen* is.

Here is one example. Suppose you want to create a variable that shows deviations from the mean price. We expect that *reliable* cars are more expensive than other cars, where *reliable* cars are ones with the number of repairs less than 3. We want deviations from the mean price separately for *reliable* cars and other cars, where the mean price is separately calculated for *reliable* cars and other cars. You can perform this seemingly very complicated task in two commands by utilizing *egen*. Type in the command line

```
bysort reliable: egen group_mean=mean(price) (press return)
gen mean_deviation=price−group_mean (press return)
```

The first command creates the group-specific means of *price*, and the second command calculates deviations from the group-specific means. Browse the data to see what actually occurs in the data set. Stata created three group-specific means—one for *reliable* cars, one for non-*reliable* cars, and one for cars with missing repair records.

There are many other powerful functions that come with *egen*. Use the *help* function and look up *egen*.

7. **Button for "Do current file" and Button for "Run current file"**

Here, I would like to let you know convenient facilities available inside of a *do*-file window. In the top part of a *do*-file window, you see multiple buttons with illustrated pictures inside. The first button from the right is to "Execute (do)" (point to the button and you see a boxed remark "Execute (do)"). If you click this button, Stata executes all commands inside of the *do* file. This is equivalent to typing "do *filename*" (without double quotes) in the command line, but clicking the button is much faster. Further, you can execute a part of the *do* file by highlighting the part you want to execute and clicking the "Execute (do)" button.

The second button from the right is to "Execute Quietly (run)" (point to the button and you see a boxed remark "Execute Quietly (run)"). The only difference between "Execute (do)" and "Execute Quietly (run)" is that "run" executes commands without showing the results in the Results window. (However, if Stata encounters an error, Stata shows an error message in the Results window.) Everything else is the same. If you do not need to see returned results, you can execute a part of the *do* file by highlighting the part you want to execute and clicking the "Execute Quietly (run)" button. These "do" and "run" buttons are very convenient because you can easily check whether your commands work as you expect.

8. **Tutorial 2 Homework**

   (a) First, check my answer to Stata Exercise 1 to learn how to use asterisk (*) to make comments inside of a *do* file. In this exercise for Stata Tutorial 2, you must use asterisk (*) to indicate the question numbers i. through v. in your *do* file. You will be penalized for not including the question numbers in your *do* file. Create and run a *do* file that implements the following tasks:

      i. Recently, you find that the existing *mpg* contains errors. You must make the following changes. For Honda Civic, *mpg* should be 24 instead of 28. For VW Rabbit, *mpg* should be 27 instead of 25. For Chev. Nova and Peugeot 604, data are so unreliable that *mpg* for these cars should be missing. Create a new variable (Let's call it *new_mpg*) that incorporates these changes.

      ii. The gas price has been increasing rapidly, and suppose you expect that the gas price will continue to increase in the future. You are interested in buying a car with good miles per gallon. Create a dummy variable (*efficient*) that is equal to one if miles per gallon is more than 25 and zero otherwise. You must use *new_mpg* to create *efficient*.

      iii. Create a new variable called *median_deviation* that shows deviations from the median price of all sample cars.

      iv. Separately for *reliable* cars and other cars, create a variable called *more_miles* that shows differences from the minimum of *new_mpg* within the group (we previously defined *reliable* in this tutorial). You must create separate minimums of *new_mpg* for *reliable* cars and other cars. Note both *reliable* and *new_mpg* contain missing observations. Browse the data set to see how Stata treats missing observations.

v. Create an indicator variable (*price_group*) that is equal to 1 if *median_deviation* is less than $-500$, 2 if *median_deviation* is $-500$ or larger but smaller than 500, and 3 if *median_deviation* is 500 and larger.

(b) If you have questions, stop by one of TA's. Stata Exercise 2 is due at the beginning of the class on April 18 (Wednesday). No late assignment will be accepted. You need to turn in a hard copy of your result file (convert your *smcl* file into a *txt* file). You MUST create your result file from a *do* file. (To create an output to be submitted as homework, you must complete a do file first and then run the do file to create a smcl file. Please do not work interactively with Stata to create an output to be submitted as homework.) If you do not follow this instruction, you get zero credits from this assignment. Please do not include error commands in your *do* file. Please use asterisk (*) to put your name and ID at the beginning of your *do* file.

End of Tutorial 2