Introduction to Computing for Sociologists
Neustadtl

Stata "Gotchas"

Stata has a couple of issues that can bite you if you are not aware of them—the treatment of missing values and the precision of the data stored in variables and comparisons.

Missing Values

So far we have treated a period (".") in a numeric variable as a missing value that leads that case to be excluded from calculations and statistical analyses. Stata actually has twenty-seven numeric missing values—the period is just one, but the one most people use. The missing values .a, .b, .c, …, .z, are called "extended missing values". Extended missing values are used when it is necessary to differentiate between different types of missing data. For example survey questions sometimes allow for the following responses in addition to simply missing: "Not Applicable", "Don't Know", "Refused to Answer". These different reasons for missing data may be coded using .a, .b, and .c in case someone wanted to do an analysis of the missing responses.

Stata numeric missing values are represented by *large positive values*. The ordering is all non-missing numbers < . < .a < .b < ... < .z. The expression *sociability* > 10 is true if variable *sociability* is greater than 10 *or* missing. To exclude missing data you must specify that the value is less than ".". For instance, "`list if sociability > 10 & sociability < .`".

Stata has one string missing value, which is denoted by "" (blank).

The following Stata code creates a small dataset to illustrate how missing values work. Case four is missing on the numeric variable *y* and case five is missing on the string variable *id*:

```
/* Understanding missing values */
clear
input str1 id byte y
"a"  1
"b"  2
"c"  3
"d"  .
""   5
end
list, noobs
```

id	y
a	1
b	2
c	3
d	.
	5

```
list y if y>=5, noobs
list y if y>=5 & y <., noobs
```

```
. list y if y>=5, noobs
```

y
.
5

```
. list y if y>=5 & y <., noobs
```

y
5

The following Stata code creates four new variables using different methods:

```
gen byte y1=(y>=3)        /* Missing values are set equal to 1 */

gen byte y2=(y>=3 & y<.)  /* Missing values are set equal to 0 */

generate y3=.
replace y3=0 if y<3
replace y3=1 if y>=3 & y<.

generate byte y4=cond(y<3,0,cond(y>=3 & y<.,1,.))

list, noobs
```

```
. gen byte y1=(y>=3)        /* Missing values are set equal to 1 */

. gen byte y2=(y>=3 & y<.)  /* Missing values are set equal to 0 */

. generate y3=.
(5 missing values generated)

. replace y3=0 if y<3
(2 real changes made)

. replace y3=1 if y>=3 & y<.
(2 real changes made)

. generate byte y4=cond(y<3,0,cond(y>=3 & y<.,1,.))
(1 missing value generated)

. list, noobs
```

id	y	y1	y2	y3	y4
a	1	0	0	0	0
b	2	0	0	0	0
c	3	1	1	1	1
d	.	1	0	.	.
	5	1	1	1	1

The variable *y1* demonstrates a significant issue with missing values. This was an attempt to create a dummy variable where all values greater than or equal to 3 are set equal to one; all other values are equal to 0. Stata dutifully did this task and converted the missing value to be equal to 1 since it too is greater than or equal to 3.

The second variable *y2* takes this into account. Now, values greater than or equal to 3 and less than . (missing) are equal to 1. But, the value for case four is equal to 0! How did this happen? The Stata statement (y>=4 & y<.) is a Boolean or logical expression that evaluates to either false (0) or true (1). So, you can see that the missing value in case 4 becomes a 0 using this logic.

Look at the Stata code that created *y3* to see one way to handle missing values and logical conditions. First a new variable *y3* is created where every value is equal to missing (.). Then these values are selectively replaced to be equal to 0 (if *y* is less than 3) or 1 (if *y* is greater than or equal to three and *y* is less than missing).

Three lines of Stata code are inefficient and can be reduced to a single line of code by using the `cond` function (`help cond`) and logical expressions. The general form of this function is:

`cond(x,a,b)`

So, for example, `cond(y<3, 0, 1)` has the same result as method one (*y1*) discussed above since *x*=*y*, *a*=0, and *b*=1. The `cond()` function evaluates *x* and if *x* is true assigned the value *a*. If *x* is

false the value *b* is assigned. But, functions can be nested and here the third argument *b* is replaced with another `cond` function.

The following example creates a new variable *y5* that is missing for both cases 4 and 5. However, the missing values are differentiated with case 4 equal to the "normal" missing period (.) and case 5 missing is equal to .a.

```
gen y5=y
replace y5=.a if y==5
list y5, noobs
list y5 if !missing(y5), noobs
list y5 if y5<=., noobs
```

```
. gen y5=y
(1 missing value generated)

. replace y5=.a if y==5
(1 real change made, 1 to missing)

. list y5, noobs

     +----+
     | y5 |
     |----|
     |  1 |
     |  2 |
     |  3 |
     |  . |
     | .a |
     +----+

. list y5 if !missing(y5), noobs

     +----+
     | y5 |
     |----|
     |  1 |
     |  2 |
     |  3 |
     +----+

. list y5 if y5<=., noobs

     +----+
     | y5 |
     |----|
     |  1 |
     |  2 |
     |  3 |
     |  . |
     +----+
```

Three different `list` commands are issued. The first simply lists all values of *y5* including the missing values. The second lists all non-missing values of *y5*. This example uses the missing() function which returns 1 (true) if any of the function arguments are missing and 0 (false) if none of the arguments are missing. This function is combined with the "not" operator ("`!`"). The `missing()` function [`help missing()`] is used to limit the listing results note that the Stata code `!missing(y5)` is read as "not missing on y5" (`help operator`). The last example lists all non-missing values of *y5* and values equal to missing ("`.`").

1. Recode or replace the missing values for the individual sociability variables *socrel, socommun, socfrend,* and *socbar* respectively to the missing values `.a`, `.b`, `.c`, and `.d`. List out the values for these variables for the records 23,414 to 23,425 and describe the pattern of missing values.

2. Describe the missing value pattern of the sociability variables *socrel, socommun, socfrend,* and *socbar*. This is a little complicated but follows these general steps:

   a. Recode each of the individual sociability variables to be equal to 0 if the case has a valid value (1 through 7) and 1 if it is missing. These variables should be called *socre1l, socommun1, socfrend1,* and *socbar1*.

   b. Create a variable equal to the sum of these four component measures called *socmiss*. Values equal to 0 mean no missing values on the four component measures; values equal to ` mean missing on one and only one variables, and so on.

   c. Use `tabulate` and `graph` hist for all values great than 0 to describe the pattern of missing values.

Precision

Programs like Stata store results in binary, and to the right of the decimal point, there is often not an exact equivalent between decimal and binary given a finite number of digits. For 0.5 there is an exact equivalent: 0.1 base 2. For 0.25 there is an exact equivalent: 0.01 base 2. For 0.125 there is an exact equivalent: 0.001 base 2. This means to the right of the binary point the powers are 2^{-1}, 2^{-2}, and so on. 0.1 base 2 is $1*2^{-1}=\frac{1}{2}$. 0 .01 base 2 is $1*2^{-2}=\frac{1}{4}$. 0.11 base 2 is $\frac{1}{2}+\frac{1}{4}=\frac{3}{4}$.

There are lots and lots of numbers less than 1 for which there is an exact binary representation. However, just because there is an exact representation in one base does not imply there is an exact representation in another. Think of the number 1/3. In base 10, it is 0.3333333 and that requires an infinite number of digits.

The difference in representing numbers in decimal (what we use) and binary (what computers use) can create some problems. The following program creates a dataset with 10 records and one variable, *x*, where every value is equal to 7.3. Note that the default storage type Stata used for this variable is float.

```
clear
set obs 10
gen x=7.3
desc
list
```

```
. clear

. set obs 10
obs was 0, now 10

. gen x=7.3

. desc

Contains data
  obs:            10
 vars:             1
 size:            80 (99.9% of memory free)
-------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
-------------------------------------------------------------------------------
x               float  %9.0g
-------------------------------------------------------------------------------
Sorted by:
     Note:  dataset has changed since last saved

. list

     +-----+
     |   x |
     |-----|
  1. | 7.3 |
  2. | 7.3 |
  3. | 7.3 |
  4. | 7.3 |
  5. | 7.3 |
     |-----|
  6. | 7.3 |
  7. | 7.3 |
  8. | 7.3 |
  9. | 7.3 |
 10. | 7.3 |
     +-----+
```

Data stored as floats have about 7 digits of accuracy. Thus, 1234567 can be stored perfectly as a float, as can 1234567e+20. The number 123456789, however, would be rounded to 123456792.

Now, let's try to list some specific examples to demonstrate the problem:

```
list if x==7.3

list if x==float(7.3)
```

```
. list if x==7.3

.
. list if x==float(7.3)

     +-----+
     |   x |
     |-----|
  1. | 7.3 |
  2. | 7.3 |
  3. | 7.3 |
  4. | 7.3 |
  5. | 7.3 |
     |-----|
  6. | 7.3 |
  7. | 7.3 |
  8. | 7.3 |
  9. | 7.3 |
 10. | 7.3 |
     +-----+
```

When the command `list if x==7.3` was issued no values of *x* satisfied the equality! The `float()` function rounds its argument (7.3 in this example) to float accuracy. Now, the two values are equal (within rounding) and the `list` command does as we expect.

The following Stata code displays three values to 14 decimal places: 1) the contents of variable *x* for the first record, 2) the number of 7.3 (as a double), and 3) the number of 7.3 represented in float accuracy.

```
di %16.14f x
di %16.14f 7.3
di %16.14f float(7.3)
```

```
. di %16.14f x
7.30000019073486

. di %16.14f 7.3
7.30000000000000

. di %16.14f float(7.3)
7.30000019073486
```

Note that the first and second numbers are not equal (so the `list` command fails). But, the first and third numbers are identical (so the `list` command works). Another solution is to store your data as double (more precision than float), but this typically just wastes memory by increasing the storage requirements of your data. In a simple test a dataset that stored numbers as double was 1.8 times larger than the same numbers stored as floats.

3. There are no problems for you to complete, just be careful with precision issues.

Big Numbers

The precision issue can also bite you when you have large numbers. Large numbers are often used as case identifiers (ID's). Consider the following:

```
clear
input id
123456789
123456790
123456791
123456792
123456793
123456794
123456795
123456796
end
list, noobs
```

```
. list, noobs

        id
  1.23e+08
  1.23e+08
  1.23e+08
  1.23e+08
  1.23e+08
  1.23e+08
  1.23e+08
  1.23e+08
```

```
format id %9.0f
list, noobs
```

```
. format id %9.0f

. list, noobs

        id
 123456792
 123456792
 123456792
 123456792
 123456792
 123456792
 123456792
 123456800
```

First we read in some data and display them. The values are difficult to read because they are displayed in scientific notation. We can use the `format` command to tell Stata that we would like it to display the values with 9 values before the decimal place and with no values after the decimal place. Now we can see that the values were not stored they way we anticipated. The problem is that *id* is stored as float which can only store an integer value with up to 7 digits of accuracy (the *id* values are 9 digits).

There are two solutions 1) change the data storage type to long or double, or 2) store your identification numbers as strings (if they have no numerical importance). If identification numbers are integers and take 9 digits or less, store them as longs; otherwise, store them as doubles (doubles have 16 digits of accuracy). The following Stata code demonstrates both of these solutions:

```
input long id
123456789
123456790
123456791
123456792
123456793
123456794
123456795
123456796
end

format id %9.0f
list, noobs
```

```
. list, noobs

        id
 123456789
 123456790
 123456791
 123456792
 123456793
 123456794
 123456795
 123456796
```

```
input str9 id
123456789
123456790
123456791
123456792
123456793
123456794
123456795
123456796
end

list, noobs
```

```
. list, noobs

        id
 123456789
 123456790
 123456791
 123456792
 123456793
 123456794
 123456795
 123456796
```

4. There are no problems for you to complete, just be careful with precision issues and large numbers.