The Stata Journal

The *Stata Journal* publishes reviewed papers together with shorter notes or comments, regular columns, book reviews, and other material of interest to Stata users. Examples of the types of papers include 1) expository papers that link the use of Stata commands or programs to associated principles, such as those that will serve as tutorials for users first encountering a new field of statistics or a major new technique; 2) papers that go "beyond the Stata manual" in explaining key features or uses of Stata that are of interest to intermediate or advanced users of Stata; 3) papers that discuss new commands or Stata programs of interest either to a wide spectrum of users (e.g., in data management or graphics) or to some large segment of Stata users (e.g., in survey statistics, survival analysis, panel analysis, or limited dependent variable modeling); 4) papers analyzing the statistical properties of new or existing estimators and tests in Stata; 5) papers that could be of interest or usefulness to researchers, especially in fields that are of practical importance but are not often included in texts or other journals, such as the use of Stata in managing datasets, especially large datasets, with advice from hard-won experience; and 6) papers of interest to those who teach, including Stata with topics such as extended examples of techniques and interpretation of results, simulations of statistical concepts, and overviews of subject areas.

The *Stata Journal* is indexed and abstracted by *CompuMath Citation Index*, *Current Contents/Social and Behavioral Sciences*, *RePEc: Research Papers in Economics*, *Science Citation Index Expanded* (also known as *SciSearch*, *Scopus*, and *Social Sciences Citation Index*.

For more information on the *Stata Journal*, including information for authors, see the webpage

http://www.stata-journal.com

Subscriptions are available from StataCorp, 4905 Lakeway Drive, College Station, Texas 77845, telephone 979-696-4600 or 800-STATA-PC, fax 979-696-4601, or online at

http://www.stata.com/bookstore/sj.html

Subscription rates listed below include both a printed and an electronic copy unless otherwise mentioned.

U.S. and Canada		Elsewhere	
Printed & electronic		**Printed & electronic**	
1-year subscription	$ 98	1-year subscription	$138
2-year subscription	$165	2-year subscription	$245
3-year subscription	$225	3-year subscription	$345
1-year student subscription	$ 75	1-year student subscription	$ 99
1-year institutional subscription	$245	1-year institutional subscription	$285
2-year institutional subscription	$445	2-year institutional subscription	$525
3-year institutional subscription	$645	3-year institutional subscription	$765
Electronic only		**Electronic only**	
1-year subscription	$ 75	1-year subscription	$ 75
2-year subscription	$125	2-year subscription	$125
3-year subscription	$165	3-year subscription	$165
1-year student subscription	$ 45	1-year student subscription	$ 45

Back issues of the *Stata Journal* may be ordered online at

http://www.stata.com/bookstore/sjj.html

Individual articles three or more years old may be accessed online without charge. More recent articles may be ordered online.

http://www.stata-journal.com/archives.html

The *Stata Journal* is published quarterly by the Stata Press, College Station, Texas, USA.

Address changes should be sent to the *Stata Journal*, StataCorp, 4905 Lakeway Drive, College Station, TX 77845, USA, or emailed to sj@stata.com.

The Stata Journal (2013)
13, Number 4, pp. 699–718

Dealing with identifier variables in data management and analysis

P. Wilner Jeanty
Kinder Institute for Urban Research
and
Hobby Center for the Study of Texas
Rice University
Houston, TX
pwjeanty@rice.edu

Abstract. Identifier variables are prominent in most data files and, more often than not, are essential to fully use the information in a Stata dataset. However, rendering them in the proper format and relevant number of digits appropriate for data management and statistical analysis might pose unnerving challenges to inexperienced or even veteran Stata users. To lessen these challenges, I provide some useful tips and guard against some pitfalls by featuring two official Stata routines: the `string()` function and its elaborated wrapper, the `tostring` command. I illustrate how to use these two routines to address the difficulties caused by identifier variables in managing and analyzing data from private institutions and U.S. government agencies.

Keywords: dm0071, identifier variables, leading zeros, FIPS codes, U.S. Census Bureau, Bureau of Economic Analysis, USDA, cross-sectional data, panel data

1 Introduction

Identifier variables are essential to fully use the information in a Stata dataset. The most typical examples of identifier variables in databases provided to the public by private and governmental institutions in the United States are geographic identifiers, also known as federal information processing standards (FIPS) codes. Assigned by the National Institute of Standards and Technology, FIPS codes are crucially useful for joining geographic records or observations for different data files and databases originating from different sources.

These standardized codes identify U.S. geographic areas from aggregate levels such as states to finer levels such as census blocks. States have two-digit codes, and counties have three-digit codes; there are also codes for metropolitan areas, census tracts, and census block groups. In county-level databases maintained by various U.S. institutions, each county is identified by a five-digit FIPS code, the first two digits of which identify the state. Often, depending on the data management and analysis tasks at hand, the data analyst must generate a new identifier variable by concatenating two or more existing variables, extracting a certain number of digits from an existing variable, or converting an existing variable from numeric to string and vice versa.

Centered on the premise that identifier variables are best stored in string format, this article features two official Stata routines—the `string()` function and its well-carved-out wrapper, the `tostring` command—to accomplish this task, which sometimes may pose unnerving challenges to inexperienced or even veteran Stata users. Cox (2002) provides a lucid discussion of these two routines and their uses but does not emphasize their relevance for dealing with identifier variables. While the gist of my article revolves around FIPS codes, the highlighted principles apply to most identifier variables. In what follows, after a succinct overview of `string()` and `tostring`, I illustrate how the two routines can be used in managing data from institutions such as the U.S. Department of Agriculture (USDA), the U.S. Census Bureau, and the Bureau of Economic Analysis, which provide data with FIPS codes to the public.

2 Overview of Stata's tostring and string()

Stata provides two commands and two functions to convert data from numeric to string format: the commands `tostring` and `decode` and the functions `string()` and `strofreal()`. Because `decode` simply creates a new string variable named `newvar` based on the "encoded" numeric variable *varname* and its value labels, it is not useful here. The `strofreal()` function is essentially a synonym for `string()`. Thus the focus here is on the `tostring` command and the `string()` function, although I will occasionally use other commands or functions to help when necessary.

The syntax for the `string()` function is `string(`*numvar*`, "%`*fmt*`")`. The first argument, *numvar*, can be either the name of a variable stored in numeric format or simply a number. The second argument, `"%`*fmt*`"`, is useful for formatting numbers, time, or dates. `tostring` relies heavily on `string()` to convert numeric variables into their string equivalents. To mimic the format argument in `string()`, `tostring` provides a `format(%`*fmt*`)` option with the default format being `"%12.0g"`. Stata carries about 10 different formats that can be used with both `string()` and `tostring` (see [D] **format**, [D] **destring**, and [D] **functions**).

Used with no format options specified, both `string()` and `tostring` will dutifully convert variables from numeric to string format. However, when the expression is greater than seven digits, using the `string()` function will result in loss of information. With `tostring`, the loss arises when the variables to be converted hold values exceeding 10 digits, even though the default format is `"%12.0g"`. To preempt loss of information in these instances, the user must specify the second argument of `string()` or the `format()` option of `tostring`. While the general format `%g` can be used to ward off the loss, it is totally inadequate for handling leading zeros. The fixed format `%f` must be used to instruct Stata that the leading zeros should be inserted before converting variables from numeric to string. This will become more clear with examples.

3 Generating identifier variables by concatenation

A five-digit FIPS code is often needed for merging. Some governmental institutions, such as the USDA or some units within the U.S. Census Bureau, publish data with separate FIPS codes for states and counties; this makes it a little difficult to join their datasets with datasets from other agencies delivering databases with a five-digit FIPS code. If you want to create a five-digit FIPS code variable, then state and county FIPS codes (most often read in as numeric in Stata) must be concatenated with leading zeros put in place before concatenation.

The need for creating a new identifier variable in this way is commonplace in many data management tasks where numeric variables have to be transformed into string variables before string functions can carry out string manipulations. For instance, researchers downloading Small Area Income and Poverty Estimates (SAIPE) data for school districts and counties from the U.S. Census Bureau have to grapple with this issue unless they have access to at least Stata 12 and elect to download SAIPE data in Excel format. SAIPE data from 1989 to 2010 can be downloaded in Excel (`.xls`) format or in text (`.txt`) format. In both file types, the state FIPS codes contain leading zeros, while the county FIPS codes do not.[1] Unless the data are in Excel format and the user is using Stata 12 or higher, when read into Stata, both variables are stored as numeric, and all leading zeros, if any, are dropped. Thus it pays to know how to obtain a five-digit FIPS code variable stored in string format by concatenating state and county FIPS codes stored in numeric format while retaining the leading zeros.

For illustration, I will use a dataset downloaded from the USDA's National Agricultural Statistics Service's website via Quick Stats.[2] A fraction of the dataset is displayed below.

1. For more details on SAIPE, see http://www.census.gov/did/www/saipe/data/index.html.
2. The USDA's Quick Stats version 2.0 is the most comprehensive tool for accessing agricultural data published by the National Agricultural Statistics Service. Using Quick Stats, researchers can query the National Agricultural Statistics Service database on the basis of commodity, location, or time period; they then can visualize the data on a map, manipulate and export the results, or save a link for future use. For more on Quick Stats, go to http://www.nass.usda.gov/Quick_Stats/.

```
. use fixit_dat2

. list state stfips county cofips in 1/6

     ┌──────────────────────────────────────────┐
     │   state   stfips        county   cofips  │
     ├──────────────────────────────────────────┤
  1. │ ALABAMA        1       AUTAUGA        1  │
  2. │ ALABAMA        1       BALDWIN        3  │
  3. │ ARIZONA        4      GREENLEE       11  │
  4. │ FLORIDA       12   PALM BEACH       99  │
  5. │ GEORGIA       13        ECHOLS      101  │
     ├──────────────────────────────────────────┤
  6. │ GEORGIA       13     EFFINGHAM      103  │
     └──────────────────────────────────────────┘
```

Now suppose you wish to generate a variable, say, **fips**, containing five-digit FIPS codes for each county by concatenating the variables **stfips** and **cofips**. You would undoubtedly want the variable to be stored in a string format. As Cox (2002) forcefully points out, code or identifier variables are better held as strings for ease of data processing. Let us put all this into context: after the **fips** variable is created, the first five observations should look like 01001, 01003, 04011, 12099, and 13101. This entails adding leading zeros to the variables **stfips** and **cofips** and converting them from numeric to string before concatenation.

Using **tostring** and its **format()** option, we can conveniently convert the two numeric variables while inserting the appropriate number of leading zeros and finally perform the concatenation. Recall that quotes are not required when specifying a format with the **format()** option of a command or the **format** command itself, which is different when using the second argument of **string()**.

```
. tostring stfips, gen(stateid) format(%02.0f)
stateid generated as str2

. tostring cofips, generate(countyid) format(%03.0f)
countyid generated as str3

. generate fips = stateid + countyid

. list fips in 1/5

     ┌───────┐
     │  fips │
     ├───────┤
  1. │ 01001 │
  2. │ 01003 │
  3. │ 04011 │
  4. │ 12099 │
  5. │ 13101 │
     └───────┘
```

We can also use the `concat()` function of the `egen` command to concatenate string variables because their values will remain unchanged at the time of concatenation.

```
. drop fips
. egen fips=concat(stateid countyid)
. list fips in 1/5

      +-------+
      |  fips |
      |-------|
   1. | 01001 |
   2. | 01003 |
   3. | 04011 |
   4. | 12099 |
   5. | 13101 |
      +-------+
```

`egen`'s `concat()` function is also a wrapper of `string()` because it converts variables from numeric to string before concatenation. Similarly to `tostring`, `egen`'s `concat()` function provides a `format()` option to accommodate the format argument in `string()`. However, you may not use `concat()` and its `format()` option to directly convert `stfips` and `cofips` from numeric to string, insert the leading zeros, and finally concatenate. The problem is that two different formats must be used. The `format(%`*fmt*`)` option of any command allows only one format to be specified.

Coding

```
. egen str5 fips=concat(stfips cofips), format(%05.0f)
```

would generate a 10-digit rather than a 5-digit variable.

Using `string()` with a fixed format as the second argument, you can do everything in one line.

```
. drop fips
. generate str5 fips= string(stfips, "%02.0f") + string(cofips, "%03.0f")
. list fips in 1/5

      +-------+
      |  fips |
      |-------|
   1. | 01001 |
   2. | 01003 |
   3. | 04011 |
   4. | 12099 |
   5. | 13101 |
      +-------+
```

Largely because of its ability to deal with more than one variable at a time and to automatically determine which string type is needed, `tostring` provides some efficiency, one of the purposes for which it was written (Cox and Wernow 2000). This feature decreases when the string variables to be created require you to specify different formats. Still, you will see `tostring`'s efficiency on display in section 5.

Often we would rather let Stata decipher the string data type to be created. In that case, a variable of type `str1`, the most compact string type, would be created, and then the `replace` command would automatically lead to the promotion of the variable to the appropriate type:

```
. generate str1 fips= ""
. replace fips = string(stfips, "%02.0f") + string(cofips, "%03.0f")
```

Although identifier variables are better stored in string format, if you want the FIPS codes to be numeric, you can use the `real()` function to make them so but at the cost of losing the leading zeros (see another issue with the `real()` function in section 4.2).

```
. generate fips1 = real(fips)
```

Thus far, we have focused attention on cases where FIPS codes are read in numeric format when insheeted, which is by far the most encountered situation. Nonetheless, it may very well be the case that the variable is stored in string format but the leading zeros are left out. For instance, you may have used the user-written command `labcenswdi`, which somehow returns FIPS codes in string format but leaves out the leading zeros (Jeanty 2011).

More concretely, suppose you are dealing with counties within only one state and you have a three-digit county FIPS code variable for that state but want a five-digit FIPS code for merging. Then you would need to concatenate the state FIPS with the county FIPS while preserving the leading zeros. Suppose the variable holds the three-digit county FIPS codes for the Texas counties and is called `countyfips`. How you will obtain a five-digit FIPS code variable will depend on the storage format of `countyfips`, which can be string or numeric.

Suppose the `countyfips` variable is stored as numeric (for some reason, county FIPS codes take on only odd numbers: $1, 3, 5, \ldots$ or $001, 003, 005, \ldots$). To obtain a five-digit FIPS code, you code

```
. generate str5 fips="48" + string(countyfips, "%03.0f")
```

Now suppose `countyfips` is stored as a string. The leading zeros may or may not be in place. If the leading zeros are in place, you code

```
. generate str5 fips="48" + countyfips
```

Otherwise, you code

```
. generate str5 fips="48" + string(real(countyfips), "%03.0f")
```

Relatedly, imagine that the previously discussed variables `stfips` and `cofips` were read in string format without the leading zeros. Now suppose you wish to retain the leading zeros and keep the same variable names. You would easily code

```
. replace stfips=string(real(stfips), "%02.0f")
. replace codefips=string(real(cofips), "%03.0f")
```

Concatenating the two variables to create a five-digit FIPS code variable simply entails coding

```
. generate str5 fips=stfips+cofips
```

Or you can code everything in one line:

```
. generate str5 fips=string(real(stfips), "%02.0f") +
> string(real(cofips), "%03.0f")
```

As you can see, the way to concatenate two existing variables to generate a new identifier variable depends largely on the storage format of the existing variables and the presence or absence of leading zeros. How about generating a new identifier variable by extracting a certain number of digits from an existing string or numeric variable? This is the subject of the next section.

4 Generating identifier variables by extraction

This section covers the creation of identifier variables by extracting numerical characters from either string or numeric variables.

4.1 The case of short identifier variables

Consider a dataset with the first few observations on the first four variables shown below.

```
. use fixit_dat, clear
. list in 1/5

     +-----------------------------------------------------+
     |          address           city   state         zip |
     |-----------------------------------------------------|
  1. |     PO Box 93527      Cleveland      OH   441043104 |
  2. |     PO Box 14690     Hanoverton      OH       44423 |
  3. |     PO Box 12583    Valley City      OH   442809327 |
  4. |    PO Box 297098         Rogers      OH       44455 |
  5. | 4724 Delbeach Rd         Medina      OH   442568489 |
     +-----------------------------------------------------+
```

Here **zip** is a numeric variable taking on zip codes in both five-digit and nine-digit formats. Based on the variable **zip**, imagine you want to create a numeric five-digit zip code for the purpose of merging with another dataset. Because of the lack in Stata of a function like **substr()** for numeric variables, the **zip** variable must be converted into its string equivalent before the five digits can be subtracted. One way to proceed is to use the **tostring** command to do the conversion and then apply the **substr()** and **real()** functions successively, as follows:

```
. tostring zip, gen(cutzip)
cutzip generated as str9
. generate zip1=real(substr(cutzip,1,5))
. list zip1 in 1/5

     ┌───────┐
     │  zip1 │
     ├───────┤
  1. │ 44104 │
  2. │ 44423 │
  3. │ 44280 │
  4. │ 44455 │
  5. │ 44256 │
     └───────┘
```

However, a more concise way would be to use the `string()` function with `"%9.0f"` as the second argument.

```
. generate zip2=real(substr(string(zip,"%9.0f"),1,5))
. list zip2 in 1/5

     ┌───────┐
     │  zip2 │
     ├───────┤
  1. │ 44104 │
  2. │ 44423 │
  3. │ 44280 │
  4. │ 44455 │
  5. │ 44256 │
     └───────┘
```

With the `string()` function, however, missing values would have resulted for all the nine-digit values of the zip variable if we had not specified the format `"%9.0f"` as a second argument. In addition, using a format different from the fixed type (`%f`) would do us a disservice. For example, consider

```
. generate zip3=real(substr(string(zip),1,5))
(14 missing values generated)
. list zip3 in 1/5

     ┌───────┐
     │  zip3 │
     ├───────┤
  1. │     . │
  2. │ 44423 │
  3. │     . │
  4. │ 44455 │
  5. │     . │
     └───────┘
```

The default format of `string()` truncated all the nine-digit values by converting them to scientific format as powers of 10, which caused loss of information. Against this backdrop is specifying `"%9.0f"` as a second argument. If applying the `real()`, `substr()`, and `string(n, s)` functions renders the generated variable in scientific format, you can always fix it by using the `format` command as

`format` *varname* %*fmt*

However, in the case of long identifier variables, you may still be surprised by unexpected values even after applying the `format` command. This issue is taken up below.

4.2 The case of long identifier variables

Consider the 12-digit census block group identifier variable `stfid`:

```
. use blockdata, clear
. format stfid %18.0f
. list stfid  in 1/6

            stfid

  1.   390998107003
  2.   390998108003
  3.   390998109001
  4.   390998109001
  5.   390998109001
```

Suppose you wish to extract an 11-digit census tract identifier variable from the 12-digit census block group FIPS code `stfid` for merging. To do so, you can easily apply the `string()` function with the format argument:

```
. generate tract_stid=substr(string(stfid,"%12.0f"),1,11)
. list tract_stid in 1/5

        tract_stid

  1.   39099810700
  2.   39099810800
  3.   39099810900
  4.   39099810900
  5.   39099810900
```

The extracted census tract identifier variable is stored in string format by construction. What if, for some reason, you want it to be numeric rather than string? Instinctively, you would code

```
. generate tract_numid=real(tract_stid)
. format tract_numid %12.0f
```

However, listing a few observations shows values countering expectations.

```
. list tract_numid in 1/6

     +-------------+
     | tract_numid |
     |-------------|
  1. | 39099809792 |
  2. | 39099809792 |
  3. | 39099809792 |
  4. | 39099809792 |
  5. | 39099809792 |
     +-------------+
```

The conversion process has anomalously resulted in loss of information. This is a flagrant example where `tostring` will refuse to process a conversion from numeric to string unless the `force` option is specified to explicitly convey the approval of information loss.

How about typing the following?

```
. replace tract_numid=real(substr(string(stfid,"%12.0f"),1,11))
(0 real changes made)
```

The problem, not surprisingly, remains unsolved. The primary reason is that by default, Stata creates float variables. With such a large number of digits, float variables can hold only multiples of four (Cox 2006). If you need a numeric variable of this size, the key is to specify the data storage type as `double` or `long` because the identifier variable takes on values with more than nine digits. Doubles have as many as 16 digits of accuracy.

```
. drop tract_numid
. generate double tract_numid=real(substr(string(stfid,"%12.0f"),1,11))
. format tract_numid %12.0f
. list tract_numid in 1/5

     +-------------+
     | tract_numid |
     |-------------|
  1. | 39099810700 |
  2. | 39099810800 |
  3. | 39099810900 |
  4. | 39099810900 |
  5. | 39099810900 |
     +-------------+
```

Whereas the difficulty of storing a nine-digit identifier variable in numeric format can be sidestepped, Cox (2002) provides two compelling reasons why identifier variables must be stored in string format: precision and data size. Yet the need to export your data to the ArcGIS[3] software package for mapping and spatial statistical analysis is another key reason why you might want to store your identifier variable in string format. Stata and ArcGIS communicate fairly well. Data exported using the `outsheet` command or the new Stata 12 `export excel` command are readily usable in ArcGIS with no intermediary steps or other software.

3. ArcGIS is a registered trademark of Environmental Systems Research Institute Inc.

```
. outsheet using yourfilename.csv, names nolabel comma
```

Identifier variables generated in Stata and stored in string format can be used to join your data with basemaps or shapefiles by using a matching key variable also in string format. However, it is important to refrain from outsheeting variables holding values in scientific format. Consider a variable with the value 268398565. Without proper formatting, this number will be stored and displayed as 2.684e+08. And if outsheeted as such, the value that will be rendered is 268400000, which results in loss of information. To overcome this conundrum, you must use a `%g` or `%f` format to properly format variables holding numbers in scientific format before outsheeting the data. Users of Stata 12 or higher can now use the `export excel` command, which automatically handles such problems (see [D] **import excel**). With `export excel`, there is no need to apply any formatting for the correct values to be rendered in the Excel dataset.[4] To export data to ArcGIS using `export excel`, you code[5]

```
. export excel using yourfilename.xls, firstrow(variable) sheet("yoursheetname")
> nolabel
```

Conversely, text files generated from exporting attribute tables in ArcGIS can be readily imported into Stata with no intermediary steps or other software by using the `insheet` command.

```
. insheet using arcgisfile.txt, names clear
```

Certainly, your preference for either string or numeric will depend on your purpose, but bear in mind that in certain circumstances, Stata processes string and numeric variables differently. For instance, when sorting a string variable, Stata will sort "15" before "5", but when sorting a numeric variable, it will sort "5" before "15". One implication for a string variable taking on U.S. state or county FIPS codes is that the usual sort order of the U.S. states or counties in a dataset will get messed up. To remedy this, you must insert the leading zeros. With the leading zeros in place, Stata will sort "05" before "15", preserving the usual sort order of U.S. states and counties.

5 Importing identifier variables from a spreadsheet

Because many private and governmental institutions provide data to the public in spreadsheets, I will now address how to import long and short identifier variables from a spreadsheet. I begin with an emphasis on insheeting data in `.csv` format using the `insheet` command, available in both Stata 12 and previous Stata versions. I then highlight the advantages of importing the same data in Excel format using the new Stata 12 `import excel` command, not available in previous Stata versions. The data for this section are taken from the U.S. Census Bureau and the Bureau of Economic Analysis.

4. Note that `export excel` does not carry the format of decimal numbers onto the Excel dataset.
5. If the Stata dataset contains missing values, those values will be converted to `<Null>` in ArcGIS.

5.1 Importing long identifier variables

The U.S. Census Bureau now requires census data users to download gazetteer files containing variables on land area, water area, and geographic coordinates about the areal units for which the data were downloaded. This illustration concerns a gazetteer file containing census tract identifier variables (such as FIPS codes), land area, water area, and latitude and longitude for all census tracts in Texas.[6] A primary interest here is to import into Stata the data provided in .csv format to create a small dataset to merge with census demographic data. Yet it is ineffective for all string variables holding value characters to be converted to their numeric representations when they are imported.

At this time, there is no way to tell Stata to read in some variables as string and others as numeric when insheeting a dataset containing string variables with numerical contents. This inconvenience, far from being a downside per se, is offset by the great amount of user friendliness and freedom of action that Stata provides. If you want to keep identifier variables in string format, there are two ways to do so. The first way is to insert a new row and place some text on top of the identifier variable you want to remain as string after reading it into Stata. This can be done in Excel before insheeting the data. Then you will need to drop the inserted row once you are in Stata. The second way is to convert the numeric variables back to string format after loading the data into Stata. We will take the second route, which is a little bumpier. For now, let us bring in the data and list a few observations.

```
. insheet using Gaz_tracts_48_coord.txt, names clear
(8 vars, 5265 obs)

. list geoid aland_sqmi awater_sqmi intptlat intptlong in 1/5, abb(11)

     +--------------------------------------------------------------+
     |      geoid   aland_sqmi   awater_sqmi   intptlat   intptlong |
     |--------------------------------------------------------------|
  1. | 4.800e+10       186.606         3.037   31.97147   -95.55244 |
  2. | 4.800e+10          6.39          .115   31.73464   -95.81571 |
  3. | 4.800e+10        27.981         1.015       31.8   -95.91238 |
  4. | 4.800e+10         8.896          .038   31.78781    -95.6419 |
  5. | 4.800e+10         7.974          .128    31.7502   -95.66921 |
     +--------------------------------------------------------------+
```

The geoid variable is displayed in scientific format for taking very large values where the number of digits is greater than seven. But do not make too much of this glitch; it can easily be undone by using the format command.

```
. format geoid %11.0f
```

6. The U.S. Census Bureau provides gazetteer files for counties and lower summary levels such as census tracts, block groups, and so on. For the geographic units of interest, these files contain data on land area, water area, and latitudes and longitudes in decimal degrees. See http://www.census.gov/geo/www/gazetteer/gazette.html. Interestingly, the vintage of the geography (that is, the FIPS code or geographic identifier) in the 2010 gazetteer files is the same as that in the 2010 Census data downloads. If you download data from American FactFinder 2, you might as well download one of these files should you need the corresponding latitudes, longitudes, and land area for your geographies.

Using the `format` command at this point is as important as specifying the `format()` option when `tostring` is invoked below. I now list a few observations to give you a sense of the values taken by the `geoid` variable.

```
. list geoid in 1/5

     +-------------+
     |       geoid |
     |-------------|
  1. | 48001950100 |
  2. | 48001950401 |
  3. | 48001950402 |
  4. | 48001950500 |
  5. | 48001950600 |
     +-------------+
```

If you want to keep the identifier variable as numeric, you can stop right there. But the goal is to obtain a string identifier variable from `geoid`. There are two prominent ways to proceed: keep the same variable name or use a new name such as `fips`. In the first case, `tostring` can do it all in one call:

```
. tostring geoid, replace force format(%11.0f)
geoid was double now str11
```

Let us pause for a moment. Why do we need to specify three options here? The answer is obvious. Specifying the `force` option explicitly conveys agreement on conversion from numeric to string, even if the conversion is potentially irreversible. By specifying the `replace` option, you confirm in essence that your old variable should be replaced with a new one. The `format()` option—the most important one here—is to forestall any loss of information. The default format used by `tostring`, `%12.0g`, is ineffective at preventing the loss of information.

The line of code above works flawlessly, but Cox (2011) rightly recommends using the underlying function directly when `tostring`'s `force` option has to be invoked. To perform the conversion, you could directly use the `string()` function, the workhorse of `tostring`. However, if you want to keep the same variable name, using the `string()` function requires some intermediary steps.

```
. insheet using Gaz_tracts_48_coord.txt, names clear
(8 vars, 5265 obs)
. generate ngeoid=string(geoid, "%11.0f")
. drop geoid
. rename ngeoid geoid
```

As seen before, `tostring` obviates those intermediary steps. On the other hand, should a different name be needed for the `geoid` variable, then `tostring` and the `string()` function will involve the same number of calls. Whether to use `string()` or `tostring` in this instance is essentially immaterial and depends on personal preference. For example, suppose you want the new name to be `fips`. To use `string()`, you code

```
. insheet using Gaz_tracts_48_coord.txt, names clear
(8 vars, 5265 obs)
. generate fips =string(geoid, "%11.0f")
. drop geoid
```

A similar call to `tostring()` is

```
. insheet using Gaz_tracts_48_coord.txt, names clear
(8 vars, 5265 obs)
. tostring geoid, format(%11.0f) force gen(fips)
fips generated as str11
. drop geoid
```

Note here the use of the `gen()` option rather than `replace`.

5.2 Importing short identifier variables

This example shows another downside of insheeting a spreadsheet in `.csv` format: the deletion of the leading zeros in the insheeted dataset.[7] This problem is inherent in identifier variables being automatically converted from string to numeric when a dataset is insheeted. In the case of short identifier variables, the reverse conversion is much simpler. Consider a simple five-digit county FIPS code downloaded with transfer payment data from the website of the Bureau of Economic Analysis.

```
. insheet using transfer2009_csv.csv, names clear
(3 vars, 3138 obs)
. list fips in 1/5

     +------+
     | fips |
     |------|
  1. | 1001 |
  2. | 1003 |
  3. | 1005 |
  4. | 1007 |
  5. | 1009 |
     +------+
```

7. This problem is commonly encountered by 2010 U.S. Census data users. See the following frequently asked question at https://ask.census.gov/faq.php?id=5000&faqld=1647: American FactFinder: How do I replace the leading zeros in my database compatible (`.csv`) download when opening the download in Microsoft Excel (that is, GEO ID2)?

Converting back to string format and inserting leading zeros with the same variable name entail a simple call to `tostring` with the `replace` option. Also needed is the `format()` option to retain the leading zeros.

```
. tostring fips, replace format("%05.0f")
fips was long now str5
. list fips in 1/5

       ┌───────┐
       │  fips │
       ├───────┤
    1. │ 01001 │
    2. │ 01003 │
    3. │ 01005 │
    4. │ 01007 │
    5. │ 01009 │
       └───────┘
```

Notice that here I did not specify the `force` option: not doing so is unharmful.

5.3 Using the new import excel command in Stata 12 and higher

Earlier, I asserted that there is no way to tell Stata to treat some variables with numerical content as string and others as numeric when insheeting a `.csv` data file. This is true in Stata 12 and under. Yet a startling difference exists between Stata 12 and previous Stata versions when it comes to importing spreadsheets.

Recall the four main problems encountered when using the `insheet` command. First, string variables with numerical contents get converted to numeric. Second, all leading zeros are dropped. Third, identifier variables with large values get converted to scientific format. Fourth, numerical variables with 1,000-separator commas become characters. Against these shortcomings is the new `import excel` command in Stata 12 and higher. Interestingly, `import excel` leaves identifier variables utterly intact upon reading Excel files. `import excel` can decipher for itself whether a variable holding value characters is string or numeric. Thus with the advent of `import excel`, there is no need to instruct Stata whether a variable with numerical characters should be stored as string or numeric.

Consider `import excel` on the same gazetteer and transfer data files used earlier but now saved in Excel rather than `.csv` or `.txt` format.

```
. import excel Gaz_tracts_48_coord.xls, firstrow case(lower) clear
. list geoid in 1/5

       ┌─────────────┐
       │       geoid │
       ├─────────────┤
    1. │ 48001950100 │
    2. │ 48001950401 │
    3. │ 48001950402 │
    4. │ 48001950500 │
    5. │ 48001950600 │
       └─────────────┘
```

It does not matter whether you are dealing with long or short identifier variables.

```
. import excel transfer2009_xls, firstrow case(lower) clear
. list fips in 1/5

     +-------+
     |  fips |
     |-------|
  1. | 01001 |
  2. | 01003 |
  3. | 01005 |
  4. | 01007 |
  5. | 01009 |
     +-------+
```

In both examples, note the use of the `case` option to convert the variable names to lowercase; `import excel`, by default, preserves the case.

If you are working with identifier variables and have access to at least Stata 12, then you should save your data in Excel format. Even better is if you have access to Excel 2007 or 2010, in which case it would be wise to save the data in `.xlsx` format to take advantage of the flexibility of the `import excel` command. If the data provider offers both `.csv` and Excel formats, you have access to both Stata 12 and Excel 2007 or 2010, and, most importantly, the two data files only differ by their formats, then choose the Excel format. As indicated earlier, SAIPE provides data in both `.csv` and `.xls` formats. At this time, Stata documentation lacks technical details with regard to the size of a dataset that can be insheeted via the `insheet` command. Documentation on `import excel`, on the other hand, is inordinately terse and self-contained (see [D] **import excel**). An `.xls` worksheet may contain as many as 65,536 rows and 256 columns. The string size limit is 255 characters. The size limits for an `.xlsx` worksheet are 1,048,576 rows by 16,384 columns with a string size limit of 32,767 characters. Thus it is worth saving your data under the `.xlsx` format whenever possible. If your Stata flavor can load more variables and observations than allowed by `import excel`, you always have the option of importing your data bit by bit and then piecing the bits together afterward.[8]

6 Creating your own identifier variables

Often and for a number of reasons, you might need to generate your own identifier variable, numeric or string. Even more important might be the need to create a unique identification number (ID) for elements within panels. In such a case, an ID is required not only for each group or panel but also for each element of each panel. Two key system variables in Stata, `_n` and `_N`, upon which I will expand later, prove indispensable. To create a unique ID for each observation in a cross-section dataset, you code

```
. generate uniqid=_n
```

The variable `uniqid`, stored in numeric format, will take on values ranging from 1 to N, where N is the current number of observations. There are many reasons why you

8. Type `help limits` to know the limits of your Stata flavor.

might want to keep `uniqid` numeric. For instance, several Stata commands require a numeric ID variable. You might also want data points to be equally spaced on the x axis when graphing, a job perfect for a numeric ID variable. But chances are that you might also want this variable to be stored in string format. Again the `string()` function is handy.

```
. generate uniqid=string(_n)
```

Because this is a string variable, adding the leading zeros makes it a little more appealing. Doing so, however, requires some thinking because the number of leading zeros to be inserted must be in accord with the string variable length or the number of characters it contains.

```
. local slen=length(string(_N))
. generate str`slen´ uniqid=string(_n,"%0`slen´.0f")
```

As you can see, adding the leading zeros hinges upon knowing a priori the number of observations in your dataset. In the case of a panel or repeated-observation dataset, you first need to create an ID for each group and then an ID for each observation within each group. You might have a variable, say, `myidvar`, taking on repeated values on which you want to base the group or panel ID numbers. If that is the case, a simple way to proceed is to code

```
. egen groupid=group(myidvar)
. by groupid, sort: generate obsid=_n
```

where `myidvar` is the variable containing the elements for which a unique ID is needed.

Nonetheless, the notion that you have in your dataset an existing variable on which you can base the groups might be untenable. Instead, all you might want to do is group observations using a unique group ID and a unique observation ID. To do this, you will first have to decide how many groups you need and the number of elements in each group. Once you decide, the `egen` command and its `seq()` function are all you need to get the job done. Suppose you have a dataset of `N` observations and you want to create `n1` groups of `n2` observations so that $\texttt{n1} \times \texttt{n2} = N$. To assign an ID to the groups and the group elements, you code

```
. egen groupid=seq(), from(1) to(n1) block(n2)
. by groupid, sort: generate obsid=_n
```

These two variables, of course, will be numeric.

In a panel-data setting, an interesting yet unnerving task is to generate a string identifier variable where each member of a group carries the ID number of the group and where the leading zeros are in place for both groups and group members. To proceed, we will build on the foundation laid above.

```
. quietly tabulate groupid
. local lnf=length(string(`r(r)´))
. by groupid, sort: generate nb=_N
. quietly summarize nb
. local mln=length(string(`r(max)´))
. by goupid, sort: generate obsid=string(groupid, "%0`lnf´.0f") +
> string(_n, "%0`mln´.0f")
```

It is worth understanding these lines of code. Beginning with the second line, the `string()` function feeds on one of the results, `r(r)`, which is the number of rows or, in this case, groups returned by the `tabulate` command. `length()` counts how many digits that number contains. Because `length()` is essentially a string function, `string()` is used to convert the number from numeric to string. The third line counts how many elements are in each group. Assuming the number of elements is the same for each group, lines 4 and 5 count the number of digits contained in that number. Built on the hindsight gained from the previous sections, line 6 creates an identifier variable by concatenating group ID and member ID after inserting leading zeros in both. Admittedly, things might become obfuscated in an unbalanced panel-data setting.

Also noteworthy is the heavy reliance on the system variables `_n` and `_N`. `_n` acts like an observation counter or marker. Stata thinks of it as the observation number of the current observation. `_N` typically indicates the total number of observations in the current dataset, including missing ones, or the number of observations in the current `by()` group. It stands out as the subscript of the last observation in a dataset or in the current `by()` group. In essence, typing `display _N` will display the number of observations in the dataset currently loaded in memory. These two crucially important built-in variables deserve due acquaintance for serious data management tasks. For more details, see [U] **13.4 System variables (_variables)** and [U] **13.7 Explicit subscripting**.

7 Checking identifier variables for duplicates

As stressed before, one of the many reasons for maintaining identifier variables is to merge datasets obtained from different sources: to link data to geographic information system databases such as geographic boundary files or topologically integrated geographic encoding and referencing line shapefiles representing geographic features such as roads, rivers, and nonvisible legal boundaries; selected point features such as hospitals; or selected areas such as parks.[9] Before you embark on such an endeavor, it is good practice to ensure that your identifier variable is unique, be it in cross-sectional or panel-data settings. Note that duplicate FIPS code elements in a dataset are very unlikely.

9. Topologically integrated geographic encoding and referencing LINE shapefiles for 2010 are available for download from the U.S. Census Bureau's website at http://www.census.gov/cgi-bin/geo/shapefiles2010/main.

To check for duplicates using the variable `myidvar` in a cross-sectional dataset, you type

```
. duplicates list myidvar
```

If duplicates are present, Stata will display their occurrence. Otherwise, Stata will respond with the message `0 observations are duplicates`, an indication that only a one-to-one or a one-to-many merge can be done using the key variable `myidvar`.

To check for duplicates within panels in a panel dataset, you invoke the `duplicates` command by listing the panel identifier and then the observation identifier. This idea can be extended to census tracts within counties or block groups within census tracts.

```
. duplicates list panelidvar obsidvar
```

Once identified, duplicates can be easily dropped. To drop all but the first occurrence of each group of duplicated observations, you code

```
. duplicates drop myidvar
```

or in a panel dataset,

```
. duplicates drop panelidvar obsidvar
```

If you are a Stata user who cannot afford to be without data from U.S. government agencies, the `string()` function and its elaborated wrapper, the `tostring` command, deserve to be part of your repertoire.

8 Conclusion

Identifier variables are key to data management and analysis. Importing and exporting them from and to other statistical software packages and rendering them to the proper format and relevant number of digits might be at times challenging for inexperienced or even veteran Stata users. Arguably, the new `import excel` and `export excel` commands provided in Stata 12 or higher can relieve Stata users from most of the data management burdens inherent in importing or exporting identifier variables when the data format is of the type `.xls` or `.xlsx`. Many private and governmental institutions continue to deliver data to the public in `.csv` format. Dealing with identifier variables when the data are in this format involves many unwieldy challenges. Featuring Stata's `string()` function and its discreetly carved-out wrapper, the `tostring` command, this article addressed most, if not all, of those challenges. The principles highlighted here will enable better and efficient management of data about U.S. geographic areas.

9 Acknowledgments

The author thanks an anonymous reviewer and Deborah Perez for their useful comments on this article.

10 References

Cox, N. J. 2002. Speaking Stata: On numbers and strings. *Stata Journal* 2: 314–329.

———. 2006. Stata tip 33: Sweet sixteen: Hexadecimal formats and precision problems. *Stata Journal* 6: 282–283.

———. 2011. Speaking Stata: Fun and fluency with functions. *Stata Journal* 11: 460–471.

Cox, N. J., and J. B. Wernow. 2000. dm80: Changing numeric variables to string. *Stata Technical Bulletin* 56: 8–12. Reprinted in *Stata Technical Bulletin Reprints*, vol. 10, pp. 24–28. College Station, TX: Stata Press.

Jeanty, P. W. 2011. Managing the U.S. Census 2000 and World Development Indicators databases for statistical analysis in Stata. *Stata Journal* 11: 589–604.

About the author

P. Wilner Jeanty is a research scientist for the Kinder Institute for Urban Research and the Hobby Center for the Study of Texas, Rice University. Jeanty has broad interests in urban and regional economics, environmental economics, development economics, and applied econometrics. His research applies statistical theory and modeling techniques from the fields of spatial, environmental, and regional economics, including the applications of spatial econometrics, geographic information systems, and econometrics of nonmarket valuation and panel data. Jeanty also has experience in survey design and implementation and in statistical software programming.