

String-valued series in gretl

Allin Cottrell

December 21, 2014

1 Introduction

Gretl's support for data series with string values has gone through three phases:

1. No support: we simply rejected non-numerical values when reading data from file.
2. Numeric encoding only: we would read a string-valued series from a delimited text data file (provided the series didn't mix numerical values and strings) but the representation of the data within gretl was purely numerical. We printed a "string table" showing the mapping between the original strings and gretl's encoding and it was up to the user to keep track of this mapping.
3. Preservation of string values: the string table that we construct in reading a string-valued series is now stored as a component of the dataset so it's possible to display and manipulate these values within gretl.

The third phase has now been in effect for a couple of years, with a series of gradual refinements. It's high time we documented it and exposed it to comment and criticism. This note gives an account of the status quo in gretl CVS and snapshots. It explains how to create string-valued series and describes the operations that are supported for such series. (We're not putting the documentation in the *User's Guide* just yet since we may want to change some things as people test and come up with suggestions.)

2 Creating a string-valued series

This can be done in two ways: first, by reading such a series from a suitable source file and second, by taking a suitable numerical series within gretl and adding string values using the `stringify()` function. In either case string values will now be preserved when such a series is saved in a gretl-native data file.

2.1 Reading string-valued series

The primary "suitable source" for string-valued series is a delimited text data file (but see section 6 below). Here's a little example. The following is the content of a file named `gc.csv`:

```
city,year
"Bilbao",2009
"Toruń",2011
"Oklahoma City",2013
"Berlin",2015
```

and here's a script:

```
open gc.csv --quiet
print --byobs
print city --byobs --numeric
printf "The third GC took place in %s.\n", city[3]
```

The output from the script is:

```
? print --byobs

      city      year
1      Bilbao    2009
2      Toruń     2011
3 Oklahoma C..   2013
4      Berlin    2015

? print city --byobs --numeric

      city
1         1
2         2
3         3
4         4
```

The third GC took place in Oklahoma City.

From this we can see a few things.

- By default the `print` command shows us the string values of the series `city`, and it handles non-ASCII characters provided they're in UTF-8 (but it doesn't handle longer strings very elegantly).
- The `--numeric` option to `print` exposes the numeric codes for a string-valued series.
- The syntax `seriesname[obs]` gives a string when a series is string-valued.

Suppose you want to access the numeric code for a particular string-valued observation: you can get that by “casting” the series to a vector. Thus

```
printf "The code for '%s' is %d.\n", city[3], {city}[3]
```

gives

```
The code for 'Oklahoma City' is 3.
```

The numeric codes for string-valued series are always assigned thus: reading the data file row by row, the first string value is assigned 1, the next *distinct* string value is assigned 2, and so on.

2.2 Assigning string values to an existing series

This is done via the `stringify()` function, which takes two arguments, the name of a series and an array of strings. For this to work two conditions must be met:

1. The series must have only integer values and the smallest value must be 1 or greater.
2. The array of strings must have at least n members, where n is the largest value found in the series.

The logic of these conditions is that we're looking to create a mapping as described above, from a 1-based sequence of integers to a set of strings. However, we're allowing for the possibility that the series in question is an incomplete sample from an associated population. Suppose we have a series that goes 2, 3, 5, 9, 10. This is taken to be a sample from a population that has at least 10 discrete values, 1, 2, ..., 10, and so requires at least 10 value-strings.

One aspect of `stringify()` is debatable. Right now the function returns 0 on success, otherwise an integer error code; it doesn't explicitly "fail" if the required conditions are not met, and it's up to the user to check if things went OK. Maybe it should fail on error?

Here's (a simplified version of) an example that I recently had cause to use: deriving US-style "letter grades" from a series containing percentage scores for students. Call the percentage series x , and say we want to create a series with values A for $x \geq 90$, B for $80 \leq x < 90$, and so on down to F for $x < 60$. Then we can do:

```
series grade = 1 # F, the least value
grade += x >= 60 # D
grade += x >= 70 # C
grade += x >= 80 # B
grade += x >= 90 # A
stringify(grade, strsplit("F D C B A"))
```

The way the grade series is constructed is not the most compact, but it's nice and explicit, and easy to amend if one wants to adjust the threshold values. Note the use of `strsplit()` to create an on-the-fly array of strings from a string literal; this is convenient when the array contains a moderate number of elements with no embedded spaces.

We should also mention that we have a function to perform the inverse operation of `stringify()`: the `strvals()` function retrieves the array of string values from a series. (It returns an empty array if the series is not string-valued.)

3 Permitted operations?

One question that arises with string-valued series is, what are you allowed to do with them and what is banned? This may need more thought, but here we set out the current state of things.

3.1 Setting values per observation

You can set particular values in a string-valued series either by string or numeric code. For example, suppose (in relation to the example in section 2.2) that for some reason student number 31 with a percentage score of 88 nonetheless merits an A grade. We could do

```
grade[31] = "A"
```

or, if we're confident about the mapping,

```
grade[31] = 5
```

Or to raise the student's grade by one letter:

```
grade[31] += 1
```

What we're *not* allowed to do here is make a numerical adjustment that would put the value out of bounds in relation to the set of string values. For example, if we tried `grade[31] = 6` we'd get an error.

On the other hand, we *can* implicitly extend the set of string values. This wouldn't make sense for the letter grades example but it might for, say, city names. Returning to the example in section 2.1 suppose we try

```
dataset addobs 1
year[5] = 2017
city[5] = "Ancona"
```

This will work OK: we're implicitly adding another member to the string table for `city`; the associated numeric code will be the next available integer.¹

3.2 Assignment to an entire series

This is disallowed: you can't execute an assignment of any sort with the name of a string-valued series *per se* on the left-hand side. Put differently, you cannot overwrite an entire string-valued series at once. This may be debatable, but it's much the easiest way of ensuring that we never end up with a broken mapping. If anyone can come up with a really good reason for wanting to do this we might reconsider. (In that case we'd have to implement a check on each value of the series before actually carrying out the assignment, which would frankly be a pain.)

Besides assigning an out-of-bounds numerical value to a particular observation, this sort of assignment is in fact the only operation that is banned for string-valued series.

3.3 Missing values

We support one exception to the general rule, never break the mapping between strings and numeric codes for string-valued series: you can mark particular observations as missing. This is done in the usual way, e.g.,

```
grade[31] = NA
```

Note, however, that on importing a string series from a delimited text file any non-blank strings (including "NA") will be interpreted as valid values; any missing values in such a file should therefore be represented by blank cells.

3.4 Copying a string-valued series

If you make a copy of a string-valued series, as in

```
series foo = city
```

the string values are *not* copied over: you get a purely numerical series holding the codes of the original series. But if you want a full copy with the string values that can easily be arranged:

```
series citycopy = city
stringify(citycopy, strvals(city))
```

3.5 String-valued series in other contexts

String-valued series can be used on the right-hand side of assignment statements at will, and in that context their numerical values are taken. For example,

```
series y = sqrt(city)
```

will elicit no complaint and generate a numerical series 1, 1.41421, It's up to the user to judge whether this sort of thing makes any sense.

¹Admittedly there is a downside to this feature: one may inadvertently add a new string value by mistyping a string that's already present.

Similarly, it's up to the user to decide if it makes sense to use a string-valued series "as is" in a regression model, whether as regressand or regressor—again, the numerical values of the series are taken. Often this will not make sense, but sometimes it may: the numerical values may by design form an ordinal, or even a cardinal, scale (as in the "grade" example in section 2.2).

More likely, one would want to use `dummify` on a string-valued series before using it in statistical modeling. In that context `gretl`'s series labels are suitably informative. For example, suppose we have a series `race` with numerical values 1, 2 and 3 and associated strings "White", "Black" and "Other". Then the `hansl` code

```
list D = dummify(race)
labels
```

will show these labels:

```
Drace_2: dummy for race = 'Black'
Drace_3: dummy for race = 'Other'
```

Given such a series you can use string values in a sample restriction, as in

```
smp1 race == "Black" --restrict
```

(although `race == 2` would also be acceptable).

There may be other contexts that we haven't yet thought of where it would be good to have string values displayed and/or accepted on input; suggestions are welcome.

4 String-valued series and functions

User-defined `hansl` functions can deal with string-valued series, although there are a few points to note.

If you supply such a series as an argument to a `hansl` function its string values will be accessible within the function (this is quite new in CVS). One can test whether a given series `arg` is string-valued as follows:

```
if nelem(strvals(arg)) > 0
  # yes
else
  # no
endif
```

Now suppose one wanted to put something like the code that generated the `grade` series in section 2.2 into a function. That can be done, but *not* in the form of a function that directly returns the desired series—that is, something like

```
function series letter_grade (series x)
  series grade
  # define grade based on x and stringify it, as shown above
  return grade
end function
```

Unfortunately the above will not work: the caller will get the `grade` series OK but it won't be string-valued. At first sight this may seem to be a bug but I think it's defensible as a consequence of the way series work in `gretl`.

The point is that series have, so to speak, two grades of existence. They can exist as fully-fledged members of a dataset, or they can have a fleeting existence as simply anonymous arrays of numbers that are of the same length as dataset series. Consider the statement

```
series rootx1 = sqrt(x+1)
```

On the right-hand side we have the “series” `x+1`, which is called into existence as part of a calculation but has no name and cannot have string values. Similarly, consider

```
series grade = letter_grade(x)
```

The return value from `letter_grade()` is likewise an anonymous array,² incapable of holding string values *until* it gets assigned to the named series `grade`. The solution is to define `grade` as a series, at the level of the caller, before calling `letter_grade()`, as in

```
function void letter_grade (series x, series *grade)
  # define grade based on x and stringify it
  # this version will work!
end function

# caller
...
series grade
letter_grade(x, &grade)
```

5 Nothing very fancy

As you’ll see from the account above, we don’t offer any very fancy facilities for string-valued series. We’ll read them from suitable sources and we’ll create them natively via `stringify`—and we’ll try to ensure that they retain their integrity—but we don’t, for example, take the specification of a string-valued series as a regressor as an implicit request to include the dummification of its distinct values. Besides laziness, this reflects the fact that in `gretl` a string-valued series *may* be usable “as is”, depending on how it’s defined; you can use `dummi fy` if you need it.

Personally, I don’t have much inclination to pursue this sort of thing. However, I do think that having introduced such series we ought to be ready to show the string values wherever it’s useful to do so. As mentioned above, we’re open to suggestions for improvement in this regard.

6 Other import formats

In section 2.1 we illustrated the reading of string-valued series with reference to a delimited text data file. `Gretl` can also handle some other sources of string-valued data.

We should first point out that `gretl`’s spreadsheet importers (for `xls`, `xlsx`, `gnnumeric` and `ods` files) *cannot* handle string-valued variables (although they can deal with “observation marker” strings in the first column). This is not a serious limitation since all spreadsheet programs can write out data as delimited text, which is very much easier to process than the native spreadsheet formats.

6.1 Stata files

`Stata` supports two relevant sorts of variables: (1) those that are of “string type” and (2) variables of one or other numeric type that have “value labels” defined. Neither of these is exactly equivalent to what we call a “string-valued series” in `gretl`.

`Stata` variables of string type have no numeric representation; their values are literally strings, and that’s all. `Stata`’s numeric variables with value labels do not have to be integer-valued and their

²A proper named series, with string values, existed while the function was executing but it ceased to exist as soon as the function was finished.

least value does not have to be 1; however, you can't define a label for a value that is not an integer. Thus in Stata you can have a series that comprises both integer and non-integer values, but only the integer values can be labeled.³

This means that on import to gretl we can readily handle variables of string type from Stata's `dta` files. We give them a 1-based numeric encoding; this is arbitrary but does not conflict with any information in the `dta` file. On the other hand, in general we're not able to handle Stata's numeric variables with value labels; currently we report the value labels to the user but do not attempt to store them in the gretl dataset. We could check such variables and import them as string-valued series if they satisfy the criteria stated in section 2.2 but we don't at present.

6.2 SAS and SPSS files

Gretl is able to read and preserve string values associated with variables from SAS "export" (`xpt`) files, and also from SPSS `sav` files. Such variables seem to be on the same pattern as Stata variables of string type.

³Verified in Stata 12.