Data Preparation and Management with Stata

Jim Tyson

24/01/2020

Table of Contents

# To be added

* Long to wide
* Add frames for multiple datasets

# Some conventions

Throughout I give examples like:

use "path\filename"

where path is any legal path for your operating system. In Windows this might be something like:

"c:\user\jrt\stata\accessproject\data\"

and filename is the name of a file on your system, for example:

rawdata.dta

In some code examples, I use a dummy name for a variable: myvar. If more than one is needed I use myvarone, myvartwo...myvarn.

# Motivation

## Personal motivation

When I first learned Stata, from great teachers, every learning opportunity concentrated on performing and interpreting some statistical test or creating and interpreting a model. And we created a few graphs.

When I left the classroom and began using Stata ‘in anger’ (as one so often does), I discovered that if my data were not presented to me in a nice, clean, consistent, properly classified and well labeled form, I would need to spend a long time, sometimes hours, diagnosing what was causing me trouble and working out how to fix it.

This is my contribution to saving other data analysts from that pain.

## General motivation for managing your data and variables

* the analysis is only as good as the data;
* good data management saves time and reduces stress;
* well managed data is easier to share.

In what follows I assume that you will *try to script everything* but will, like me, fall short. If you work by writing scripts in **do files** then you work is easy to adapt, easy to correct and easy to reproduce.[[1]](#footnote-23)

However the Stata ‘do file language’ is not a conventional programming language and you cannot always check and then conditionally manipulate data in the way that you might using Perl, Python or R. But the best is the enemy of the good.

# A project folder structure

Most projects benefit from having a consistent folder structure. I often have a structure like this:

1. data if this is stored locally;
2. scripts: initiation scripts - the data reading, housekeeping and cleaning scripts; analysis scripts - visualising, estimating, modelling;
3. logs;
4. reports - if possible dynamic markdown documents;
5. images.

**The raw data files are read only.**

The raw data can be in a number of formats. If possible a plain text delimited format - such as CSV - is preferred. A plain text format avoids a number of issues that can make data cleaning and set up unnecessarily complicated:

1. proprietary formats may not be readily readable in Stata;
2. some data types require careful manipulation when transferred between file formats - especially dates;
3. if your data is plain text, you can use almost any tool in preparation;
4. you can share data even if your collaborators don’t use Stata.

Cleaned data sets should only be stored temporarily - arguably not at all. The raw data file should be treated as read only. Your initiation script might clean and transform your data, producing a data set fit for analysis. Ideally this data set should not be saved - you call the analysis script over the data set in memory after its transformation. You can call a do file from **within** an do file script using a command like

do myscriptfilename

but of course change the path and file name to match your set-up.

# A quick note on preserve and restore

A good alternative to creating and saving transformed data files locally is to use Stata’s preserve command before doing any data transformation and analysis (this snapshots the pre-transformation state of the data) and use restore after analysis.

Some operations overwrite data in memory. Since we aim to always preserve original data, we do not want to overwrite data in memory and then write to file. Nor do we want a proliferation of modified data files.

If we wish to make some **temporary** change to a data set, then we should use **preserve** and **restore**. Preserve takes a snapshot of the data set in its current state. We can then process the data in anyway we like and later use restore to return to the state in the snapshot.

In some circumstances you may decide to create a file and use it temporarily. It is strongly recommended that you either:

1. delete the file after your analysis is complete (using the Stata erase command) and recreate it for any further analysis; or
2. use a **make** utility[[2]](#footnote-28) for conditional execution of scripts.

This conservative approach towards the raw data, and reluctance to create permanent files of processed data, is critical to the goal of reproducible research practices.

# A quick note on capture and quietly

When a Stata command terminates or a process finishes, Stata (usually invisibly) creates a **return code** which indicates whether the process was a success or if errors of some kind were detected.

In Stata scripts you will often see a simple line like

tostring id, force replace

which would force Stata to represent all the values in **id** as strings. If this were in a script, and **id** already contained all string values, we would not want the script to fail at that point. To avoid this happening we can **capture** the **return code** of a command and let the execution of the script continue by preceding the operation with capture

capture tostring id, force replace

This executes ignoring errors and swallowing any error output. The **return code** gives an indication of what happens when a command tries to execute. It is stored in a macro **\_rc** and can be displayed

disp \_rc

Usually a return code of 0 indicates success and anything else is some kind of error or anomaly.

As a further example, when we open a log file, we usually don’t want our script to fail because the log is already open.

Return codes from capture can also be used to create complex branching behaviour in scripts - but that is out of scope here.

The quietly keyword prevents the command following sending output to the console. We very often just accept Stata’s default output when we run commands, but this can lead to the display of a lot of screen-lines that we don’t really want to see. Using quietly either on individual lines or with {} for scope, allows us to run a process without viewing all the output and then perhaps using disp to show selective results. Thus:

`quietly summarize maths, detail  
disp r(kurtosis)  
`

or

`quietly {  
regress history maths  
predict resid, resid  
}  
summarize resid  
`

Sometimes you want to see all of the output in the console and sometimes you don’t. It’s a good idea to get more selective with quietly.

## Session logs

For your own debugging, code review and record keeping purposes, you should also log your sessions. You can include a log using command at the beginning of your scripts. The log of your session can be in either SCML (a Stata specific markup language) which is viewed with the Stata view command, or plain text which can be viewed with any text editing package, including the Stata do file editor.

There is a second kind of log file: the command log. This log contains **only** the commands issued in a session and no results. It is always in plain text format and this makes it useful for quick conversion to a rough and ready **do** script.

Log files should ideally have their own folder and a consistent file name-scheme. If you wish to create logs that are automatically named with the date you can use a command like this:

log using "$S\_DATE.log", append

or

cmdlog using "commands $S\_DATE.log", append

I have used the **append** option. This will generate a warning if the file mentioned does not yet exist, but the log will still be created.

Include lines to close logs in your scripts.

log close  
cmdlog close

# Reading your data

If you have a single source of data then you can open it with

use "path\filename"

if it is Stata data, or with one of the several import command variations if it is not Stata data. For a CSV file the command would be:

import delimited c:\filename.csv, delimiter(comma) clear

(Where we assume that the data file is in the root of your C drive.)

For an Excel file you might have

import excel c:\filename.xls, sheet("results"), firstrow case(lower)

In this example, the data are in the workbook sheet **results**, variable names are in the first row and all variable names are forced to lower case.

If the data is in more than one file, then you will need to open one file and then merge the second. The two files need to share a unique identifying column or a combination of columns that uniquely identifies a case. Suppose for example that you have a file **results** that looks like this:

surname sex class maths english history  
ADAMS 2 1 55 63 65  
ALI 2 1 52 46 35  
BAGAL 1 3 51 58 55  
BENJAMIN 1 2 59 70 68  
BLAKEMORE 2 2 56 38 40  
BUCHAN 1 3 45 62 59  
CHULANI 1 3 63 69 69

and assume that **surname** is unique. We could merge this with a second file **graded results** with a structure like

surname grade  
ADAMS a  
ALI c   
BAGAL b   
BENJAMIN a   
BLAKEMORE c   
BUCHAN b   
CHULANI a

with the stata command:

merge 1:1 surname using gradedresults

### Remote files

Increasingly analysts keep data on the net. If data is kept on a web server, then you will not usually have to worry about backing up data and you will be less likely to accidentally delete or overwrite data.

Stata provides a useful method for using data stored on a web server: webuse.

#### Setting the URL and reading the file

Before using a Stata data file stored on a web server you must set the url of the server. This is done as follows

webuse set http://somedomain/somepath/folder

Now you read the file

webuse mydata

The variables in the data set are read into memory in the normal way.

#### Reading non-Stata files over the web.

You can read a csv file with variable names in the first row:

import delimited "https://www.ucl.ac.uk/~ccaajim/results.csv", firstrow clear

and the same for an excelfile:

import excel "https://www.ucl.ac.uk/~ccaajim/results.cxls", firstrow clear

### Dirty or untidy?

Not to push an analogy too far, we need to distinguish two things:

Tidy data:

In tidy data:

1. Each variable forms a column;
2. Each observation forms a row;
3. Each type of observational unit forms a table.

Many data analysts promote the use of tidy data. You can read about the layout and the motivation for it at [this CRAN page](https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html)(<https://tinyurl.com/tidydef>).

Clean data:

By clean data, for current purposes we mean:

1. Variables have the correct computational type;
2. Values for variables are all within the correct range;
3. No mandatory items have missing values;
4. All uniqueness constraints hold;
5. Cross field constraints hold;
6. Values recorded are, within accepted limits, accurate;
7. Data is complete within accepted limits and missing data is suitably coded;
8. Data are internally consistent.

This is a more complex matter than making data tidy.

In general having untidy data will make your analytical job more difficult and your processes less clear, while having dirty data will compromise the quality of your analysis - however carefully carried out.

## Reshaping

Data maybe either **wide** or **long** in shape. In wide format, if there is more than one measure for each subject or case, they are in separate columns. In long format, each observation is a row (and so a case may have data in more than one row)

Here is some of the data on student exam performance in wide format:

surname sex class maths english history  
ADAMS 2 1 55 63 65  
ALI 2 1 52 46 35  
BAGAL 1 3 51 58 55  
BENJAMIN 1 2 59 70 68  
BLAKEMORE 2 2 56 38 40  
BUCHAN 1 3 45 62 59  
CHULANI 1 3 63 69 69

There is only row per student and each row contains three examination scores.

Here is some of the same data in long format:

ADAMS 2 1 maths 55  
ADAMS 2 1 english 63  
ADAMS 2 1 history 65  
ALI 2 1 maths 52  
ALI 2 1 english 46  
ALI 2 1 history 55

Sometimes the software you use will require that the data be in one or other format.

### Wide to long

To reshape your data from wide to long, you need some kind of ‘stub’ in the original separate column names. for reshape to work so i rename the subject exam scores:

rename (maths english history)(mathsexam historyexam englishexam)

And then use the stub @exam in a reshape long command like this:

reshape long @exam, i(id) j(subject) string

This means **reshape the data to long format using the variable names currently ending with ‘exam’ as the values for the new variable ‘subject,’ with ‘id’ as the case identifier, and inserting the exam score into the new exam column**.

| Data | Wide | Long |  
|-----------------|--------------------------------------|---------|  
| No of rows | 30 | 90 |  
| No of variables | 7 | 6 |  
| j variable name | | subject |  
| x\_ij variables | mathsexam, englishexam, historyexam | exam |  
  
. list in 1/5  
  
 +-----------------------------------------------+  
 | id subject surname sex class exam |  
 |-----------------------------------------------|  
 1. | 1 english ALI 2 1 35 |  
 2. | 1 history ALI 2 1 46 |  
 3. | 1 maths ALI 2 1 52 |  
 4. | 2 english BLAKEMORE 2 1 40 |  
 5. | 2 history BLAKEMORE 2 1 38 |  
 +-----------------------------------------------+

### Long to wide

# Check the data

A useful technique for a general check of the data that you can eyeball in the console involves a special form of the describe command.

The procedure **overwrites** the data in memory and in your script you should precede this operation by a preserve command and follow it by restore.

preserve  
describe, replace clear  
list  
restore

This replaces the data in memory with a summary report of information about the data set.

Consider this data:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| surname |  | sex | class | maths | english | history |
| ADAMS |  | 2 | 1 | 55 | 63 | 65 |
| ALI |  | 2 | 1 | 52 | 46 | 35 |
| BAGAL |  | 1 | 3 | 51 | 58 | 55 |
| BENJAMIN |  | 1 | 2 | 59 | 70 | 68 |
| BLAKEMORE |  | 2 | 2 | 56 | 38 | 40 |

The data contains six variables. If we use describe we get:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| variable name | type | format | label | variable label |

|  |  |  |
| --- | --- | --- |
| Contains | data |  |
|  | obs: | 30 |
|  | vars: | 6 |
|  | size: | 480 |
|  |  |  |
| surname | str10 | %10s |
| sex | byte | %8.0g |
| class | byte | %8.0g |
| maths | int | %8.0g |
| english | byte | %8.0g |
| history | byte | %8.0g |

Sorted by:

> Note: Dataset has changed since last saved.

You can quickly check the variable names, types, labels and so on in this way. To get a summary of your variables use list after this describe operation.

Remember that this operation **overwrites the dataset in memory**. You no longer have your original data in the workspace and to use your data you must re-load the file, or you must have preserved it and restored it.

## Data type and variable format

The basic data types in Stata are:

|  |  |
| --- | --- |
| byte | smallish numbers |
| int | largeish integers |
| long | very large integers |
| float | numbers including fractional values (estimations) |
| double | longer numbers including fractional values (estimations)( |
| strN | strings of length N |
| strL | massively long strings (2000000000 characters) |

Each of these has a default display format. Stata display begin with % followed by a number indicating the width, a stop (ie .), then the decimal precision and ususally g for general format or f for fixed format.

For numbers, in general format, decimal places of a number will be displayed up to the value of width. In fixed format, the number of decimal values display is fixed by the number following `.` however wide the variable.

|  |  |
| --- | --- |
| byte | %8.0g |
| int | %8.0g |
| long | %12.0g |
| float | %9.0g |
| double | %10.0g |
| strN | %Ns |
| strL | %9s |

## Type checking

A type or data type is how the computer system or programming language knows how to treat the data. So, a different set of computations are possible with integers than with text data. Ensuring that data is recognised as the correct type helps make for safe coding.

The division between numeric data and string data (sometimes called text or character data) is one of the most obvious and fundamental. Consider that although a telephone number is a string of characters that may look like a number, we should not take the square root of someone’s telephone number or calculate ‘average telephone number’ from our data.

Stata has two non-numeric data types: short string and long string. The short string (probably the most common string type) is limited to 2054 characters. The long string can hold up to 2 billion characters.

In variable definition, Stata uses the prefix **str#** (where **#** is a number representing length) for short strings and **strL** for long strings.

Stata has five different numeric types each suitable for storing data elements of different size. They are

|  |  |  |  |
| --- | --- | --- | --- |
| Storage type | Minimum | Maximum | Bytes |
| Byte | -127 | 100 | 1 |
| Int | -32,767 | 32,740 | 2 |
| Long | -2,147,483,647 | 2,147,483,620 | 4 |
| Float | 1.70141173319 \* 10^38 | 1.70141173319 \* 10^38 | 4 |
| Double | -8.9884656743 \* 10^307 | 8.9884656743 \* 10^307 | 8 |

If you have a large data set - large enough that you experience difficulty in loading data - you should think about the storage types of your numeric data to see whether they could be stored more efficiently. There is a Stata command compress which will attempt to recast your storage types for a data set in memory to use space more efficiently.

You should also be aware that storing a value with **more space** than required can cause confusion. If you store for example the number **7** (ostensibly an integer) as a float, then the internal representation may actually be **7.0000001** or **6.9999999**. If you use a number with a boolean operator like this:

gen flag = 1 if myvar <= 7

with myvar having type float, you risk a wrong result.

Another basic command for type checking in Stata is ds with the option detail. This will return a list of variables in your data and their current storage types. The output is substantially the same as describe, replace. For example:

. ds, detail  
  
 storage display value  
variable name type format label variable label  
----------------------------------------------------------------------------  
surname str10 %10s   
sex byte %8.0g   
class byte %8.0g   
maths int %8.0g   
english byte %8.0g   
history byte %8.0g

There is a useful option on ds , has. This option allows you to specify **which type** of variables you wish to list in the results, so:

. ds, detail has(type byte)  
  
 storage display value  
variable name type format label variable label  
--------------------------------------------------------------------------  
sex byte %8.0g   
class byte %8.0g   
english byte %8.0g   
history byte %8.0g

(You should confirm that you understand why **maths** is not included in this list).

You can also use ds, has(format %8.0g) (or indeed any format type) to check for variables with a specific format. You can now visually examine the type and format information for your variables for correctness.

### Coerce to string or numeric

#### Destring and back again

Data may not always be read in to Stata correctly. If after checking you have a variable which should be numeric but is stored as a string, you have to convert if. The command to convert from string to numeric has this format

destring originalstringdata, replace

The converse is achieved with tostring.

tostring originalnumericdata, generate(newstringdata)

In each case you can either **replace** the original values in place, or **generate** a new variable to hold the new values.

### Coercing to integer

If you have a variable of type float and you wish to convert it to integer values, then assuming that rounding is appropriate you should generate a new variable of type integer with the following code:

gen int myvari = round(myvar)

where I have chosen myvari as the new variable name. As well as round(), you can use the functions floor() and ceiling().

## Range checking

The strategy for range checking is to create a semaphore or alert variable if a value is not in range for any case.

Consider the continuous variables in this example data

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| surname |  | sex | class | maths | english | history |
| ADAMS |  | 2 | 1 | 55 | 63 | 65 |
| ALI |  | 2 | 1 | 52 | 46 | 35 |
| BAGAL |  | 1 | 3 | 51 | 58 | 55 |
| BENJAMIN |  | 1 | 2 | 59 | 70 | 68 |
| BLAKEMORE |  | 2 | 2 | 56 | 38 | 40 |

Suppose that we know that the scores **maths**, english\*\* and **history** are examinations scores out of 100. If any score is less than zero or more than 100, we will consider it out of range and we will have to find a strategy to deal with it.

So we will write a do file to generate a new variable to act as our semaphore - outofrange - with a value zero for all cases. Then create a list of relevant variables (here the three continuous variables **maths, english** and **history**) and use a conditional replace statement to replace the zeros with 1 for values out of range.

We will exploit the inbuilt Stata function **inrange()** which has this syntax:

inrange(myvar, minimum, maximum)

This function returns true when myvar is between the minimum and maximum values. So the code might look like this:

capture drop outofrange  
  
local continuous maths english history  
  
gen outofrange = 0  
  
foreach var of local continuous{  
  
 replace outofrange = 1 if !inrange(`var',0,100)  
  
}

We use capture in the drop statement to ignore any error if outofrange does not yet exist (that is, for example, if we are running this routine for the first time).

We can now use

list if outofrange == 1

to examine any cases with values out of range.

The question remains what you should **do** with these cases, but that is a theoretical matter. You must decide.

## Factor variables

### Labelling values (and variables)

If we consider a typical factor variable like ethnicity. **The Office of National Statistics** in the United Kingdom has a list of eighteen options for respondent self identification of ethnic group. Let’s look only the **White** sub-category. Here respondents have the choices:

1. English/Welsh/Scottish/Northern Irish/British
2. Irish
3. Gypsy or Irish Traveller
4. Any other White background, please describe

For the purposes of entering, storing and manipulating data it makes good sense to use the numeric code assigned to each of these groups. However, in reporting (and often just eyballing the data) it is good practice to provide ‘human friendly’ labels for the values.

This is done with the Stata label command.

There are two steps involved in labeling values for a factor variable. First define a set (we call it *ethnicityl* here) of label-code equivalences. Like this:

label define ethnicityl 1 "English,Welsh,Scottish,Northern Irish,British' 2 "Irish" 3 "Gypsy or Irish Traveller" 4"Anyother White background"

here **ethnicityl** is a handle for this label definition and is followed by pairs of code and label.

You can check what label definitions exist in your data with label dir.

The next step is to apply **ethnicityl** to the values of some variable in our data:

label values myvars ethnicityl

### Defining and managing factors

Stata (unlike for example R or SPSS) does not require that you **define** a variable as a factor or categorical variable. Rather it is when you run analysis that you will need to specify whether some variable should be treated as categorical and its levels transformed (on the fly) to indicators (‘dummy variables’).

To check the range of factor variables you can use inlist(). For example

capture drop foutofrange  
  
gen foutofrange = 0  
  
replace foutofrange = 1 if !inlist(myfactorvar,1,2)

As before we can now list to see the results:

list if foutofrange == 1

### Generating dummy binary variables for factors

Using a factor variable with the *i* prefix, for example

regress maths i.stream

avoids the necessity of creating dummy binary variables. A dummy binary variable recodes each level of a categorical variable into a new 0/1 variable. For example if we have a variable eyecolour with levels

eyecolour  
blue  
brown  
green  
other

We can recode this as four variables, each taking a value 0 (false) or 1 (true)

|  |  |  |  |
| --- | --- | --- | --- |
| ecblue | ecbrown | ecgreen | ecother |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

This can be done with gen and replace and if. However, a much simple way to achieve this dummy coding is with tabulate:

tabulate eyecolour, generate(ec)

Which produces

|  |  |  |  |
| --- | --- | --- | --- |
| eyecolour | Freq. | Percent | Cum. |
| blue | 6 | 20.00 | 20.00 |
| brown | 5 | 16.67 | 36.67 |
| green | 6 | 20.00 | 56.67 |
| other | 13 | 43.33 | 100.00 |
| Total | 30 | 100.00 |  |

and createes four new variables, for example:

|  |  |  |  |
| --- | --- | --- | --- |
| ec1 | ec2 | ec3 | ec4 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

By default, Stata will use the original level names to create labels for these new variables of the form eyecolour==blue etc.

# Missing values

Stata can represent up to twenty seven distinct levels of missing values. These are represented as **.**, and **.a**…**.z**.

The default treatment of cases with missing values is to drop them from analysis (listwise deletion).

If your data has codes for missing values before you import it, then you can use replace to recode the data so that Stata recognises the missing values. Suppose for example that your missing codes are

|  |  |
| --- | --- |
| Code | Value |
| -99 | Missing completely at random |
| -98 | Missing at random |
| -97 | Missing not at random |

You can use a command like:

replace myvar = .a if myvar == -99 replace myvar = .b if myvar == -98 replace myvar = .c if myvar == -97

and then apply a label set like

label define formissing .a "MCAR" .b "MAR" .c "MNAR"

and finally apply the set of labels **formissing** to your variable:

label values myvar formissing

This method works, but you will have to do it a single variable at a time (or use a foreach loop). Perhaps better is to use the command mvdecode like this

mvdecode myvarone myvartwo, mv(-99 = .a  -98 = .b  -97 = .c)

So encoding two variables in one instruction. Indeed if your missing value code scheme is uniform across variables you can use the construct **\_all** to simply apply the encoding to all variables in the data.

mvencode \_all, mv(-99 = .a  -98 = .b  -97 = .c)

While a few missing values may not critically impact on your analysis and may allow for simply dropping cases, it will be worthwhile thinking about how to encode missing values. Not only because incomplete data may require special handling, but also because data maybe missing for different reasons. Consider for example a survey that asks respondents for their date of birth: no answer is very different from an answer ‘unknown.’

Supposing that you wish to exclude all missing values from some procedure (maybe a plot, for example), then you can use an if condition with a comparison operator like

if myvar >= .

## Outliers

We should remember that an outlier is not an error, nor even a ‘bad’ data point. There is no reason to suppose that one should always exclude outliers from analysis: to do so is a statistical decision. Moreover, automatic detection of outliers is inherently limited and crude.

Let us assume for the moment Tukey’s 1977 characterisation of an outlier: a value beyond the whiskers, where the upper whisker is at

and the lower is at

.

tabstat maths, statistics(p25 p50 p75 iqr) save  
  
matrix statsc=r(StatTotal)  
  
local p75 = statsc[3,1]  
local p25 = statsc[1,1]  
local IQR = statsc[4,1]   
  
  
disp "`p75'"  
  
disp "`IQR'"  
  
local upperwhisker = `p75' + 1.5\* `IQR'   
local lowerwhisker = `p25' - 1.5\* `IQR'  
  
disp "`upperwhisker'"  
disp "`lowerwhisker'"  
  
list surname maths if maths >= `upperwhisker'  
list surname maths if maths <= `lowerwhisker'

But it is very important not to rely on the output of this (or any other computation of extreme values) automatically. It is critical that you consider carefully what constitutes an extreme or otherwise interesting value and decide what to do about them. Graphical approaches such as scatter and box plots are often more conducive to understanding the data.

Notice also that this only detects univariate outliers - you can have combinations of values for a case that together make the case of interest or problematic and this procedure would not detect those.

With small data samples and a bi-variate relationship, it can be useful to create a scatter plot, run a regression analysis and then examine the leverage against residuals plot. So, imagining for a moment data with 30 rows like the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| surname | sex | class | maths | english | history |
| ADAMS | 2 | 1 | 55 | 2 | 99 |
| ALI | 2 | 1 | 52 | 46 | 35 |
| BAGAL | 1 | 3 | 51 | 58 | 55 |
| BENJAMIN | 1 | 2 | 59 | 70 | 68 |
| BLAKEMORE | 2 | 2 | 56 | 38 | 40 |

We can investigate any very influential points with this code:

scatter english history  
reg history english  
capture gen id=\_n  
lvr2plot, mlabel(id)  
predict l, leverage  
hilo l english history id, high show(5)

This allows us to consider any outliers and their effect.

## Duplicates

## Constraints I: uniqueness

The simplest uniqueness constraint can be illustrated by considering records that have a unique id - for example patient id in a clinical trial. However, uniqueness constraints can be complex: it may be some combination of values that must be unique.

Look at **contract** in Stata help.

contract varlist if in weight , options

contract replaces the dataset in memory with a new dataset consisting of all combinations of varlist that exist in the data and a new variable that contains the frequency of each combination. As usual with processes that replace data, you should probably wrap this in preserve and restore.

## Constraints II: cross value constraints

If students grades are based on level of marks, then ensure that the correct relationship holds. How to check? Write code that correctly generates grade from marks and then compare with your original data. *Do not assume that your calculation is always correct - if you discover an anomaly you may need to investigate rather than just assume.* For example a students grade may have improved due to some mitigation process.

# Creating new variables in your data

Think carefully about **when** you create any derived variables: before or after checking? If you create them before checking you may introduce errors in your derived variables, but if you create them after checking then you may want to check that the derived variables are then also checked for correctness etc.

## Aggregate or summary variables

Once you have your data, you may need to create some variables derived from the raw data. One of the most useful Stata commands for this task is egen.

Let’s take a simple example. Look at the head of our original data:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| surname |  | sex | class | maths | english | history |
| ADAMS |  | 2 | 1 | 55 | 63 | 65 |
| ALI |  | 2 | 1 | 52 | 46 | 35 |
| BAGAL |  | 1 | 3 | 51 | 58 | 55 |
| BENJAMIN |  | 1 | 2 | 59 | 70 | 68 |
| BLAKEMORE |  | 2 | 2 | 56 | 38 | 40 |

Suppose that you would like too calculate for each student a measure of their overall examination performance. We can calculate the average score as

(maths + english + history)/3

These sorts of summaries of data **by case** (rather than by variable) are better created with egen.

First, creating the average examination score for each student.

egen avxm = rowmean(maths english history)

Now, use **cut** to break this into groups of scores. I have chosen groups as from zero to less than 50, from 50 to less than 60 and from 60 and above. The icodes keyword creates numeric code names (starting at 0) for each resulting group:

egen stream = cut(avxm), at(0,50,60,101) icodes

To check that the result is correct we can run this line:

table stream, contents(min avxm max avxm)

which will show us the lowest and highest data points in each group for our data.

|  |  |  |
| --- | --- | --- |
| stream | min(avxm) | max(avxm) |
| 0 | 41.66667 | 49.66667 |
| 1 | 51.66667 | 57.66667 |
| 2 | 60.33333 | 67 |

# Dates

Stata dates are **elapsed dates** with a reference date **0** of January 1, 1960. So, dates are represented internally as integers counting back or forward from this date.

Any other true date representation is a formatting of this representation.

This means that data you import into Stata which might look like, for example

20-12-2022

may not be correctly recognised as a date by Stata.

There are Stata functions to convert dates to elapsed dates and to format dates.

Let’s imagine a file of data that looks like this:

dateone datetwo  
01-01-2000 01-01-2000  
03-02-2001 03-02-2005  
01-01-2000 01-01-1960

First, I will assume that the data are in two columns of a table in a single Excel worksheet and import them with this command:

import excel https://www.ucl.ac.uk/~ccaajim/datesstataexample.xlsx, firstrow

(You can use this command if you want to grab this small sheet from my website).

Now, how has Stata recognised the data? Here is the output of desc

The display format And now, let’s try some simple mathematics with these dates:

gen interval = datetwo - dateone

Stat creates a variable of type float, with format %9.0g.

Since, this interval is expressed in elapsed days, we can convert to years by dividing by 365.25

## Converting strings to dates

Stata will convert many different string types to dates. The command is date. For example, suppose that we have a variable idate with a single row

Jan, 31, 2001

we can convert this with

gen newidate = date(idate, "MDY")

This will generate a float that is the elapsed date for January 31st, 2001.

The string in quotes in date() is called the mask. It is used to specify in what order the *month*, *day* and *year* elements occur in the input string.

## Extracting parts of dates

From any date variable, you can extract the *month*, *day* and *year* elements with the functions month(\_date variable\_),day(\_date variable\_) and year(\_year variable\_).

For example,

gen yearalone = year(date1)

gen dayalone = day(date1)

gen monthalone = month(date1)

# Exporting tables to Office or LaTeX format from scripts

Dynamic reporting using dyndoc or markstat is a great way to work with Stata. However, not everyone wants to take on the whole workflow of producing documents programmatically, it is useful to have a script that analyses your data and exports tables of results as you go.

One of the simplest ways to do this is to use collect export in your scripts after you run a routine. Suppose that you create a simple cross tabulation, for example

table teacher class

You can immediately export the result to excel with

collect export teacherclass.xls

or

collect export teacherclass.tex, tableonly

The option tableonly prevents the tex output including a full document environment, restricting it to the tabular float code.

So this will create an Excel or tex file with the results of table. You can use .docx, .html, and .pdf files (see the Stata documentation for the full list) as well as .xls(x) and .tex.

Unfortunately, it is not possible to append all your outputs to one file with collect export (it is allowed for some formats but not others), nor can you label the ouput easily.

A more sophisticated method which allows you to label the exported content and collect all ouput in one document uses the putdocx or putexcel. Since this is outside the central topic of this guide (ie data management), an example do file that illustrates the process is provided in putdocxexample.do.

# References

Baker, Peter. 2020. “Using *GNU Make* to Manage the Workflow of Data Analysis Projects.” *Journal of Statistical Software* 94 (Code Snippet 1). <https://doi.org/10.18637/jss.v094.c01>.

1. [If you are not familiar with do files, you will need to read this help page before going any further](https://www.stata.com/manuals13/gsw13.pdf) [↑](#footnote-ref-23)
2. This is not the place to describe the use of makefiles, but the topic is eminently googleable. A good reference for general use is Baker (2020) [↑](#footnote-ref-28)