PEP-LAVAL INTENSIVE GRADUATE SCHOOL IN DEVELOPMENT ECONOMICS
Measuring and Alleviating Poverty - ECN-A4405
(5-11 June 2011)

Learning Stata

by

Abdelkrim Araar
Sami Bibi
Jean-Yves Duclos

June 2011

Table of contents

List of figures

1 Introduction

1.1 What is Stata?

Stata is a general-purpose statistical software package created in 1985 by StataCorp. It is used by many businesses and academic institutions around the world. Most of its users work in research, especially in the fields of economics, sociology, political science, and epidemiology.

Stata's range of capabilities includes:

- Data management
- Statistical analysis
- Graphics
- Simulations
- Custom programming

Stata has become a very popular tool in the last 20 years to transform and process data. It comes with a large number of basic data management modules that are very efficient at transformating large datasets. The flexibility of Stata also enables programmers to provide specialized *.ado* routines to add to the power of the software.

1.2 What operating systems are compatible with Stata?

An **operating system** (**OS**) is an interface between hardware and users that is responsible for the management and coordination of activities and the sharing of the resources of a computer and that acts as a host for computing applications run on the machine. Stata can run on Windows, Mac OS X or UNIX.

2 Stata's graphical user interface

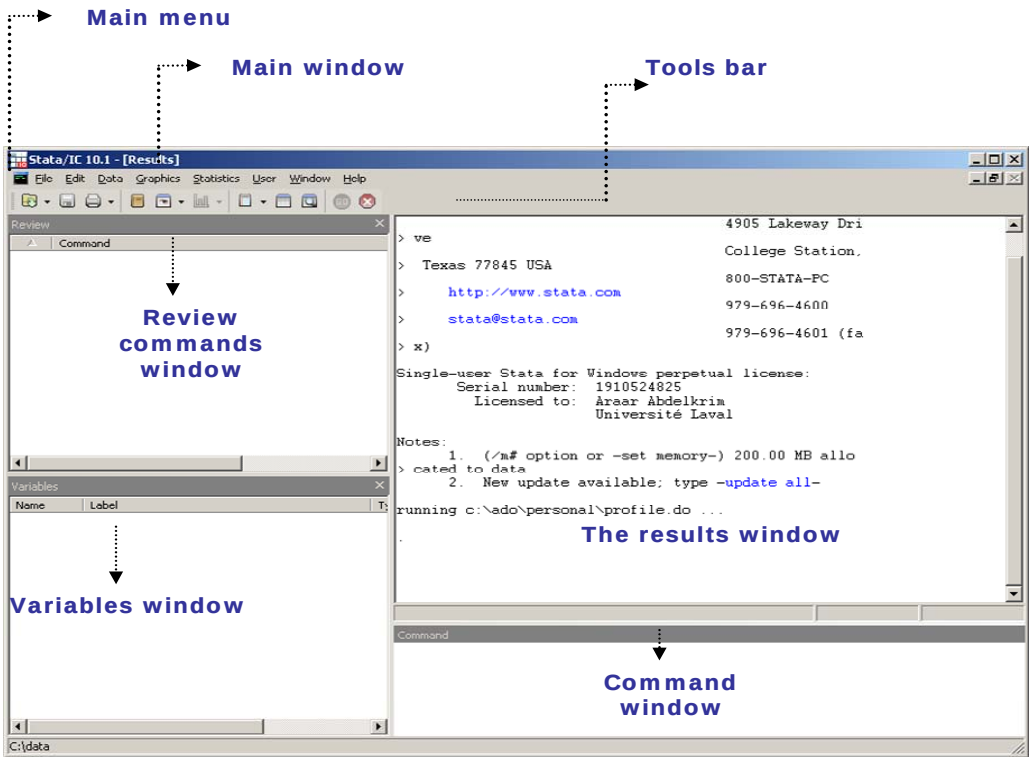
2.1 The graphical interface of Stata (version 10)

The main windows of Stata 10 are:

A- The window command
Commands are submitted to Stata in command line (only one *command line* can be executed each time).

- Page-Up : to edit the previous command;
- Page-Down : to edit the next command;
- Tab : to complete the name of the variable.

Figure 1: The graphical user interface of Stata 10



B- Review window

This window displays the command lines that were executed in the command window.

- Click on a given command line that appears in this window to copy it in the command window;
- Doubleclick on a given command line that appears in this window to execute it;
- Clicking on the left button of the mouse shows a popupmenu that allows to copy or save in a *.do file the commands that were used during a session.

C- Variables window

This window lists the names of the variables of the current datafile as well as their label names and their format.

- Click on a given variable to copy it in the window command;
- Clicking on the left button of the mouse shows a popumenu that allows to rename variables or add notes on the current datafile.

D- Results window

This window displays the results of the submitted Stata commands.

- Select a part or all of the results and click on the left button of the mouse to copy it with a text or tabulated format.

Stata's main menu also contains other items to access to dialogue boxes.

2.2 Reorganising the graphical user interface

The user can select one among two possible formats for Stata's windows settings:

1- All windows are positioned within the main window. This makes their positioning dependent on that of the main window.

Main Menu: Prefs➔Manage preferences➔ Load preferences➔ Compact window settings

2- The position of the different Stata windows is independent.

Main Menu: Prefs➔Manage preferences➔ Load preferences➔ Maximized window settings

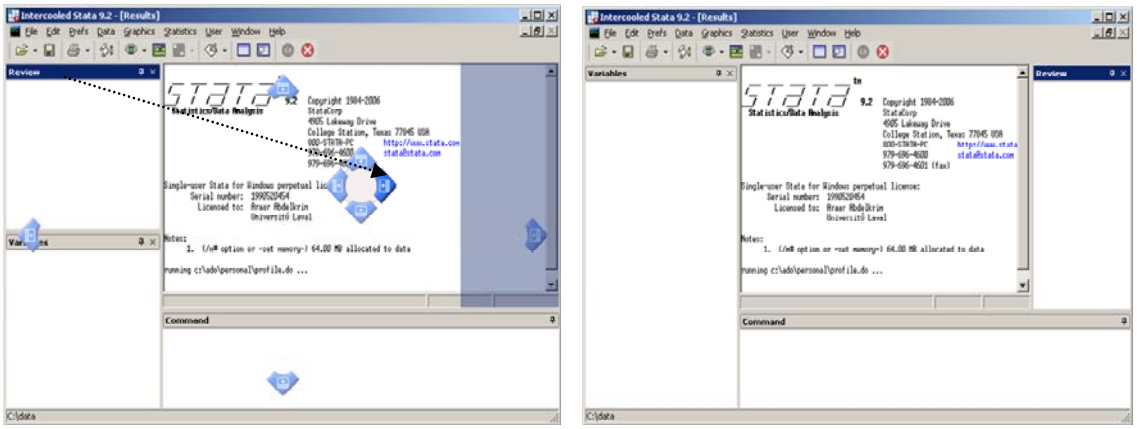
For each of the two window positioning formats, the user can reorganise the display format of the windows and save the desired format.

Example 1

Positioning the **Review window** to the left of the **Results** window**.**

Select the **Review** window and then move it by keeping the right button of the mouse pressed, as indicated in the following graph.

Figure 2: Placing the Review **window** to **the left of the** Results **window**

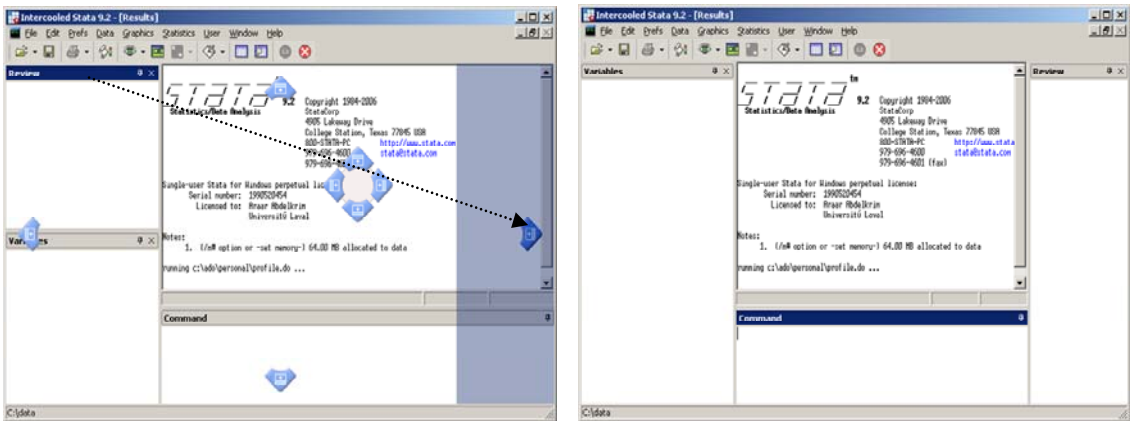


Example 2

Placing the **Review** window to the left of the main windows

Select the **Review** window and then move it by keeping the right button of the mouse pressed, as indicated in the following graph.

Figure 3: **Placing the Review window to the left of the main windows**



2.3 Main menu and dialogue boxes

Like many other softwares, Stata's main menu contains usual items such as **File,** which allows access to other sub-menus to open or to save the Stata files. Stata has improved considerably the graphical user interface and dialogue boxes in the recent years. Stata now regroups the main commands into three items: **Data**, **Graphics** and **Statistics.**

To execute the Stata commands, Stata offers three possibilities:

1. Typing the command line in the command window and clicking on **Enter** (keyboard button);
2. Executing the Stata command in a dialog box;
3. Executing a *.do file (an ASCII text file that contains a set of command lines).

To display the dialog box of a given command, two options are available.

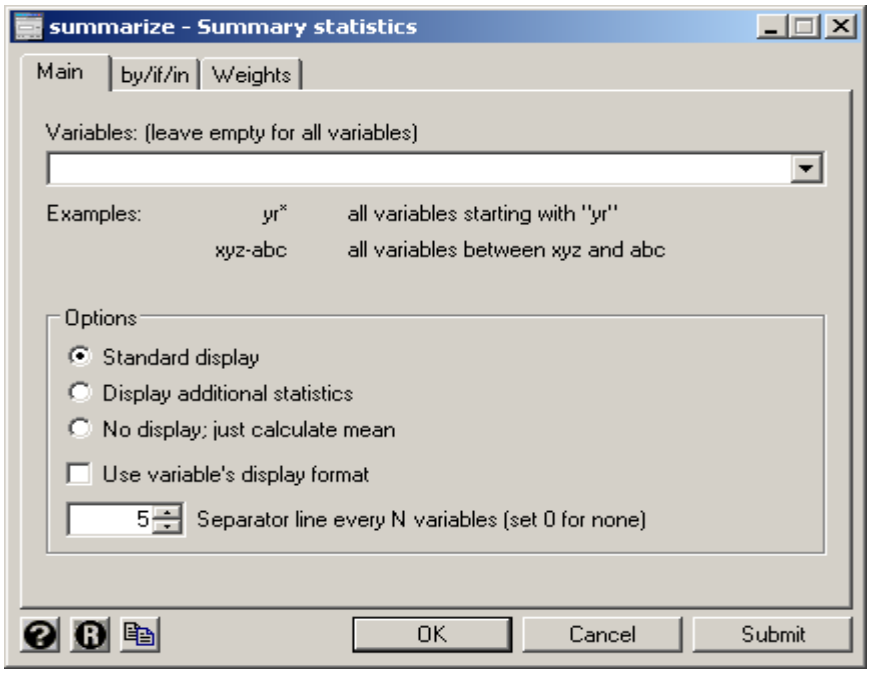
1. The first is to select the command item from Stata's main menu.
   Example
   Main menu: Statistics➔Summaries…➔ Summary statistics➔ Summary statistics

2. The second is to type the command db followed by the command of interest and then to click on **Enter.**

Example

```
db summarize
```

Figure 4: Dialogue box of the command *Summarize*



Six buttons appear in the bottom of the dialogue box. The use of these buttons is the following:

 To display the Stata help file for the command.

 **Reset**: To initialize dialogue box fields to their default values.

 To copy in the clipboard the syntax that will be generated after clicking on the button **OK**.

 To execute the command and close the dialogue box.

 To close the dialogue box without executing the command.

 To execute the command without closing the dialogue box. This option is useful when we plan to execute the command with different options.

Remark
By clicking on the button **Submit** or **OK** the syntax of the command is generated automatically and it appears in the window command.

Each of the following three forms of execution has its specific usefulness.

1. The use of dialog boxes generates an accurate Stata syntax when options are selected. This helps learning quickly Stata's command syntax.
2. A do file may contain a set of command lines that can form a program. Users can save this program to reuse or modify it later at their convenience.
3. More advanced Stata users can use directly the window command to generate quickly some statistical results.

2.4 The tools bar

The Stata tools bar contains a set of buttons that allow accessing quickly different tools, such as the viewer, the do file editor or the data editor

Figure 5: The Stata tools bar

To open the Stata datafile (extension `*.dta`)

To save the active datafile.

To print results, contents of the Stata viewer or graphics.

To begin, add, suspend or resume the `.log file`.

To open the Stata viewer.

To show the graphical window at the front.

To open the .do file with the Stata do editor.

To edit the opened datafile.

To visualise the data.

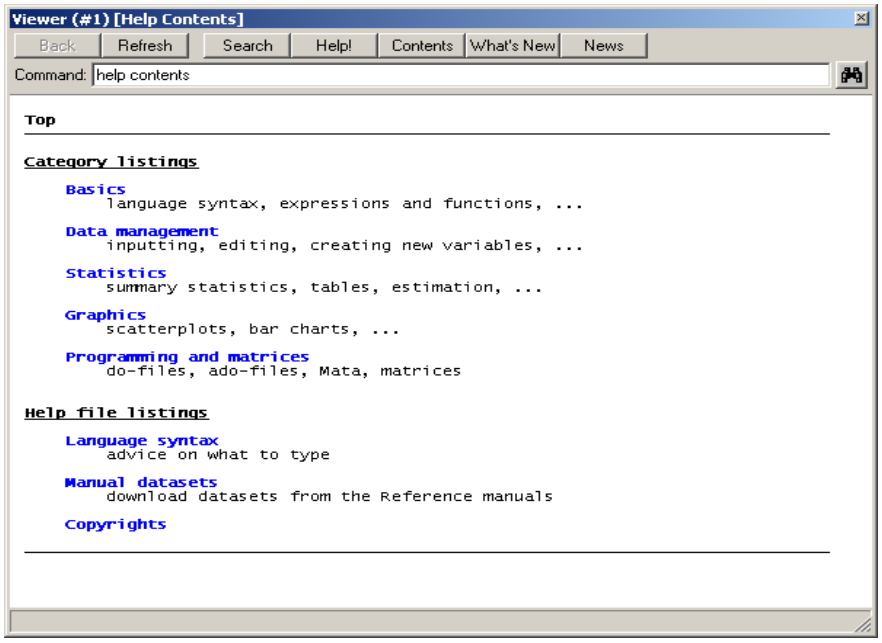
To continue the execution of the Stata program in break.

To stop the execution of the Stata program.

2.5 The Stata viewer

The Stata viewer is mainly designed to display help files for Stata commands. This viewer also allows displaying the SMCL files (Stata Markup and Control Language) as well as usual text ASCII files.

Figure 6: Stata viewer



With this tool, one can:

- Edit the Stata help files;
- Search Stata documents;
- Search the net using keywords;
- Use help on the use of a given command using the command `search`;
- Find and install new official Stata commands, which are published in the Stata Journal;
- Check for new updates of Stata;
- Edit `.log`, `.SMCL` and `ASCII` files;
- Etc.

The viewer buttons have the following functions:

Back	Return to the previous contents.
Refresh	Refresh the edited content.
Search	Search help files by keyword(s) and, optionally, on the internet.
hsearch	Search text of help files for specific words
Help!	For help on a Stata command with examples, options list, and syntax guide.
Contents	For a list of command categories, advice on language syntax, and links to datasets from the reference manuals.
What's New	Additions to Stata since release 10.0.
News	Display news about Stata.

2.6 Editing and visualizing Stata datafiles

2.6.1 The Data Editor

The data editor is a useful tool that allows entering, changing or editing data.

To open the data editor:
- Double click on the icon **Data Editor**;
- Or type the command **edit.**

The data editor has the spreadsheet format:
- Columns are the variables;
- Lines are the observations**.**

We can use the copy/past command from a sheet of other software such as Excel. One can also select a cell to change its value.

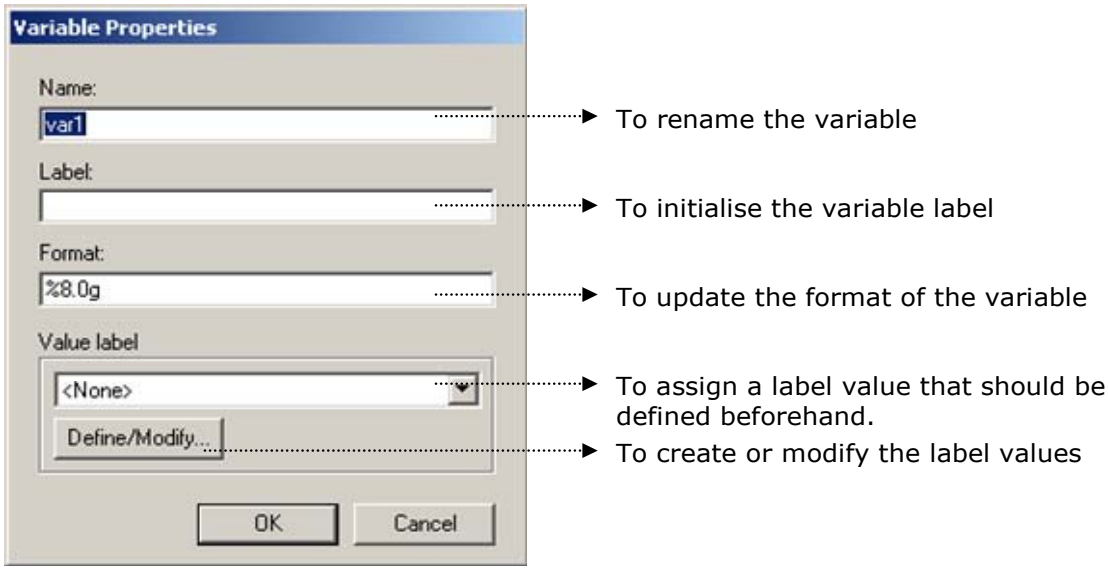
To insert the data of a new variable:
- position the cursor in the first cell of the column;
- enter the value of the cell and press Enter on the keyboard;
- enter the value of the second observation and repeat the operation until all the values of the column (variable) are inserted.

Two data types are generally used:

- Numeric (*Ex*. 1 / 1.1)
- Alphanumeric, i.e., (**string**) (*Ex*. Rural). This type of data appears in red in the data editor.

We can modify the name, the format and the label values of a variable. Starting from the data editor, doubleclick on the cell that contains the name of the variable and the following window appears:

Figure 7: Window to update the variable properties



Instead of editing all of the data, the user can edit a desired set of variables or a limited number of observations, as indicated in what follows:

edit varname	To edit one variable (ex. **edit** var1)
edit varlist	To edit a list of variables (ex. **edit** var1 var3)
edit in range	To edit a given range of observations 1 : to edit the first observation; 1/10 : to edit the first ten observations; 3/-3 : to edit all of the observations except the first two and the last two observations.
edit if exp	To edit the observations that obey the condition defined by the expression (**exp**).

We can also combine the different options:

Example

```
use   data\data1.dta, replace
edit  income age_hh in 1/20 if age_hh>30
```

The Data Editor contains the following buttons:

Figure 8: The data editor buttons

Preserve If you have made changes to data with the Data Editor, if you are satisfied with your changes, and if you are going to keep making changes, you can preserve these changes by clicking on the **Preserve** button.

Restore This reverts the dataset to the way it was when it was last preserved or when it was opened with the Data Editor.

Sort To sort the dataset in ascending order by the values of the currently selected variable.

<< Makes the currently selected variable the first variable in the data set.

>> Makes the currently selected variable the last variable in the data set.

Hide Hides the current variable from view

Delete To suppress :

- The current variable;
- The current observation ;
- All the dataset.

2.6.2 The Data Viewer

The format of the Data Viewer is similar to that of the Data Editor. However, this tool can be used only to visualise the dataset but not for changing it.

To display the Data Viewer:

- Click on the icon of the **Data Viewer**;
- Or type the command **browse.**

As for the Data Editor, we can edit a subset of variables or observations.

Example

```
use       data\data1.dta, replace
browse    income age_hh  if age_hh>40
```

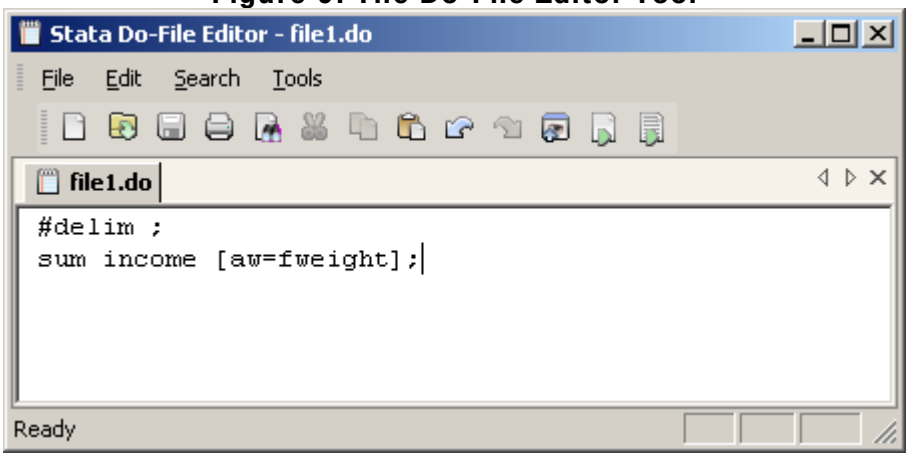
Hints

- If you would like to explore the dataset without changing it, it is safer to use the **Data Viewer** to avoid making undesired changes.
- By default, the label values of labelled variables (*e. g.*. 1 for rural area and 2 for urban area) are displayed in the Data Editor or the Data Viewer. If you wish to edit the values, you must add the option nolabel (e.g. `edit area, nolabel`).
- Whereas alphanumeric contents (string) are displayed in red, labels are displayed in blue.

2.7 The Do-file Editor

The Do-File Editor allows writing, editing or even executing a part or all of the Stata command lines of the current do file.

Figure 9: The Do-File Editor Tool



Buttons on the Do-File editor have the following functions:

New: Open a new do-file in a new Do-File Editor.

Open: Open a new do-file from a disk in a new Do-File Editor.

Save: Save the current do-file to disk.

Print: Print the contents of the Do-File editor window.

Find: Open the Find/Replace dialog for finding and replacing text.

Cut: Cut the selected text to the clipboard.

Copy: Copy the selected text to the clipboard.

Past: Past the selected text from the clipboard to the current do file.

Undo: Undo the last change.

Preview: Open a viewer window to display the contents of the Do-File Editor window.

Run: Run the do file command lines, showing all commands and their output. If text is highlighted, the button becomes **Run Selected Text.**

Do: Run the commands in the do files without showing any output. If text is highlighted, the button becomes **Do Selected Text.**

do files are simply text files that are saved with names with extension .do. These files contain a set of Stata command lines.

Executing a do file from the window command

Syntax

{**do**|**run**} filename [arguments] [, nostop]

The command **do** or **run** .do executes the Stata command lines of the do file with the name **filename**. The command **run** executes the command lines without displaying any output. The option **nostop** forces the execution of all command lines even if some of them contain some errors.

How to insert comments in a do file?

// Comment — The comment is written in one line.

/* Comment */ — The comment begins with **/*** and ends with ***/**. This allows writing the comment on several successive lines.

Trick

- To block the execution of a given part of a do program, the command lines of that part can be put in a comment format (/* …*/).

Usual Stata commands in do files.

#delimit	To indicate a delimiter for command lines. By default, the delimiter is the end of the command line. One can change this to be a semicolon, i.e., the character ";".
clear	This removes data and value labels from memory.
capture log close	To close the current log file if the latter is already opened
log using	To open a log file.
set more off	To avoid Stata pauses during the display of results.
set mem	To increase the memory allocated to Stata.
log off/on	To suspend or to restart writing in the log file.
save	To save the current datafile.
log close	To close the current log file.

We will review in detail these Stata commands in Section 3. We now present the use and usefulness of .log files.

2.8 Using the log command to save the executed commands and their subsequent results

The command **log** allows writing the contents of a Stata session (executed command lines and results) in an SMCL or text file. We can also use simultaneously more than one log file. The command **cmdlog** is similar to that of **log,** but with this command only the executed command lines are written in the log file. The general syntax to open a log file is:

```
log using filename [, append replace [text|smcl] name(logname)]
```

Options

filename	To specify the name of the log file (with file tree).
append	To add to the contents of the log file.
replace	To replace the log file.
[text|smcl]	To specify the format of the log file.
name()	Specifies an optional name that you may use to refer to the log while it is open.

Examples

```
log using c:\results\res1, replace
```

This command line will allow saving the executed commands and results in the file `res1.smcl` *in the path* c:\results .

```
log using c:\results\log1,  name(log1) text replace
log using c:\results\log2,  name(log2) smcl replace
```

Close, suspend or restart writing in the log file

```
log {close|off|on} [logname]
```

Close	Close the log file.
Off	Suspend temporarily writing in the log file.
On	Restart writing in the log file.

Examples
```
log close
log off log1
```

To display the status of the log file
```
log
log query [logname]
```

Examples
```
log
log query log1
```

3 The syntax of Stata commands

3.1 The general form of the syntax of Stata commands

In general, the syntax of Stata command lines takes the following form; the brackets [] are used to show the main items of a command line:

```
[prefix :] command [varlist] [=exp] [if] [in] [weight] [using filename] [, options]
```

The elements	The description
Prefix :	A prefix command that precedes the main command. *Example* **by** area : sum v1
command	Stata command. *Example* **list** *Remark* The underline part of the Stata command refers to its abbreviated form. *Example* **list or l** **describe or d**

generate or gen

varlist Names of a given list of variables.

Example
var1 var2 var3 var4

When var1 to var4 are ordered - var1 var2 var3 var4 -, we can write simply :

var1-var4

=exp Algebraical expression.

Example
gen xvar**=var1+var2**

if This option is used to indicate a given condition expressed by an algebraic expression.

Example
if rural==1

in This option is used to indicate a range of observations.

Example
in 1/10

weight This option is used to indicate the weights attributed to observations.

Example
[pweight=wvar**]**

using filename This option is used to indicate the name of the datafile.

Example
using(data1**)**

options What follows the comma are the options of the Stata command.

Example
, nolabel

Example

```
use                data\bkf94I.dta, clear
by area, sort :    summarize exppc [aweight = fw]
```

3.2 The basic Stata commands

3.2.1 The basic operating system commands

dir To list the names of files in the specified directory.

copy To copy a given file from disk or URL.

erase To delete a given file.

cd This command allows changing the working directory to the specified drive and directory.

pwd This command is equivalent to typing cd without arguments; both display the name of the current working directory.

mkdir To create a new directory.

3.2.2 Summary presentation of basic Stata commands

Version Display the installed version of Stata.

Update

- Display versions of the executable file and that of the *.ado files.
- Also allows updating Stata's program files.

Example
```
update from c:/temp
update all
```

Which Display the version of a given Stata file (`*.ado`, `*.hlp` or other).

Example
```
which svydes.ado
```

query Display system parameters.

memory Display information about the memory allocated to Stata.

set memory Allows changing the memory allocated to Stata.

Examples
```
set memory 20m
set memory 60m, permanently
```

clear Removes data and value labels from memory.

Remark
clear is equivalent in version 10.1 of Stata to:

```
drop _all
label drop _all
```

more This command causes Stata to display "—more—" and then to pause until any key is pressed.

Remarks

1. The command line « `set more off` » tells Stata not to pause or display the "—more—" message.

2. The command line « `set more on` » tells Stata to wait until a key is pressed before continuing when a "—more—" message is displayed.

#delimit

The #delimit command resets the character that marks the end of commands. It can be used only in do-files and in ado-files.

Remark
There are two ways to mark the end of a command line :

1. **#delimit** `cr` : the delimiter is automatically set to the **carriage return**.
2. **#delimit** `;` : the delimiter is automatically set to the semicolon.

count

Display the number of observations.

Remark
The command line « **count** `if exp` » displays the number of observations that respect the condition exp.

Example
```
count if age>=10
```

set obs #

To increase the number of observations.

Example
```
clear
set obs 100
```

quietly

To execute the command(s) without displaying results.

Example 1
```
quietly sum age
```

Example 2
```
qui {
     Stata command lines
    }
```

notes

This command attaches notes to the datafile in memory. These notes become a part of the datafile and are saved when the datafile is saved and retrieved when the datafile is used.

Examples

```
/* to display the notes */
notes
```

```
/* To add a note */
notes  : Ugandan Household Survey (year).

/* To add a note to the variable equi    */
notes equi  : Number of adults + 0.5 * number of children.

/* To suppress all notes  */
note drop _dta equi

/* To drop the note on the variable equi    */
note drop equi
```

list To list variables in the window of results

Examples

```
/* To list all variables */
list
/* To list variables var1, var2 and var3 */
l var1 var2 var3
/* To list observations 1 to 10 */
list in 1/10
```

Easy ways

- To use commas as the decimal separator, type the command « **set** dp comma ».
- To use dots as the decimal separator, type the command « **set** dp period ».

3.3 Arithmetic, logic and relational operators

The following table summarizes the main Stata operators that can be used in expressions.

Arithmetic	**Logic**	**Relational**
+ addition	~ Not	> greater than
- subtraction	! Not	< lower that
* multiplication	| or	>= equals or greater than
/ division	& and	<= equals or lower than
^ powered to		== equals
		~= different from
+ Text concatenation (string)		!= different from

Remarks

- To write an expression with the equivalence condition, one must use the « == » instead of « = ».

- Missing values (indicated by the dot « . » in Stata) are considered as observations with the greatest value (+infinity). Hence, the expression "size > 6" is true if the value of size is greater than six or is a missing value. To keep the observations that are greater than zero and that are not missing values, one has to use the following expression: `size > 6 & size != .`
- The arithmetic operators obey the usual order of priorities. For instance, the execution of the operator « ^ » precedes that of « + ».

3.4 Stata and mathematical functions

Stata has several predefined functions making it possible to carry out several mathematical operations starting form the current variables of the data. The following table presents some of the most used mathematical functions.

Function	**Description**
`abs(x)`	Generates the absolute value of the variable `x`
`ceil(x)`	Returns the unique integer n such that n - 1 < x < n. See also: `int(x)`, `floor(x)`, et `round(x)`.
`exp(x)`	Returns the exponential function of e^x. See `ln(x)`, `log(x)`, et `log10(x)`.
`floor(x)`	Returns the unique integer n such that n < x < n + 1. See also `int(x)`, `ceil(x)`, et `round(x)`.
`int(x)`	Returns the integer obtained by truncating x towards 0; Exp. int(5.2) = 5 and int(-5.2) = -5. The function `trunc(x)` produces the same result.
`ln(x)`	Returns the natural logarithm of ln(x). This function is the inverse of exp(x).
`max(x1, x2, , xn)`	Returns the maximum value of x1, x2, ..., xn.
`min(x1, x2, , xn)`	Returns the minimum value of x1, x2, ..., xn.
`round(x,y)` or `round(x)`	Returns x rounded in units of y or, x rounded to the nearest integer if the argument y is omitted. Example: `gen rse=round(se,0.1)`
`sign(x)`	Returns the sign of x: -1 if x < 0, 0 if x = 0, 1 if x > 0, and missing if x is missing.
`sqrt`	Returns the square root of x.
`sum(x)`	Returns the running cumulative sum of x treating missing values as zero.

3.5 The qualifiers: `by`, `if` and `in`

Most Stata commands allow the `by` prefix, which repeats the command for each of the population groups. The qualifier `by` without the `sort` option requires that the data be sorted

by varname (varlist). by and bysort are really the same command; bysort is just by with the sort option.

```
by varname : command varlist
```

It is necessary to sort the data beforehand by the variable varname. For this, there are two ways:

1. Sorting the data with the command **sort**:

```
sort  education
by    education : summarize income
```

2. Using the command bysort to sort and execute the command simultaneously.

```
bysort education : summarize income
```

With the qualifier **if,** the command is applied only with the data that obey the specified condition, situated after this qualifier.

```
command varlist if condition
```

For instance, the following command line: **summarize** income **if** education == 13, allows obtaining the descriptive income statistics for the group with education equal to 13.

The qualifier **in** allows to run the command for a specific set of observations.

```
command varlist in condition
```

Example

```
summarize income in 101/200
```

This command line makes it possible to produce descriptive income statistics when the observations are located between the 101th and 200th lines of the dataset.

3.6 Weighting observations: weight

Most Stata commands can be executed using attributed weights. Stata allows four forms of weights:

fweights Frequency weight. It indicates the frequency of the observation (must be an integer).

`pweights` Sampling weight. It indicates the inverse of the probability that the observation is sampled.

`aweights` Analytic weights. Those weights are inversely proportional to the variance of an observation; i.e., the variance of the jth observation is assumed to be sigma^2/w_j, where w_j are the weights. Typically, the observations represent averages and the weights are the number of elements that gave rise to the average. For most Stata commands, the recorded scale of aweights is irrelevant; Stata internally rescales them to sum to N, the number of observations in the data, when it uses them.

`iweights` Importance weight. This weight has no formal statistical definition and is a "catch-all" category. The weight reflects the importance of the observation and any command that supports such weights defines exactly how such weights are treated.

Remarks

1. To estimate accurately standard errors using household surveys, it is advisable beforehand to initialize the sampling design of the survey (see the help for command **`svyset`**). Once this is done, one should use the commands that allow computing standard errors based on survey design – see the help for commands **`svy`**.

4 Stata and datasets

Stata can open only one dataset at any time. Stata holds the entire dataset in (random-access or virtual) memory. Before opening a new dataset, one has to close the opened one by using the command **`clear`**.

4.1 Opening the datafile

To load the data, Stata offers many possibilities depending on the form of the loaded data.

4.1.1 Opening Stata datafiles: The command use

If the user already has a datafile saved in Stata format (the name of the file is with extension `.dta`), the command **use** allows opening the datafile. The syntax of this command is:

`use` data\nom_du_file [, clear nolabel]

Note that:

- If the filename contains spaces, one has to add quotation marks around the name (*ex*. "data 1").
- The filename can contain the complete path of the datafile (*ex*. "c:/for1/data/data 1").

Options

`Clear` Execute `clear`, then open the datafile

`Nolabel`	Open the datafile without loading labels (label variables and label values).

Good practices

- Add the option `clear` to avoid the error: *data in memory*.
- Increase beforehand the memory allocated to Stata if you wish to open a big datafile.

4.1.2 Inserting the data manually: the command `input`

The command **`input`** allows inserting directly the data with the command window. This command is useful with a small number of observations. The general syntax of this command is:

```
input [varlist] [, automatic label]
```

Example

```
input       size weight income
            5 120 2000
            7 180 1500
            3 100 3000
            2 200 4000
            4 140 2500
end
```

Remark
It is also possible to insert directly the data with the **`Data Editor`** (**`edit`**), as is the case with the Excel sheet (see Subsection 2.6.)

4.1.3 Loading the data from ASCII or text files

There are several commands with different options that can load ASCII files. These commands are **insheet**, **infile** and **infix**.

4.1.3.1 The command `insheet`

The command `insheet` is intended for reading files created by a spreadsheet or database programs. Regardless of the origin of the file, `insheet` reads text (ASCII) files in which there are 1 observation per line and the values are separated by tabs or commas. The first line of the file can also contain the variable names. The general syntax is:

```
insheet [varlist] using filename [, options]
```

Options

`double`	override default storage type.
`tab`	tab-delimited data.
`comma`	comma-delimited data.
`delimiter("char")`	use char as delimiter.
`clear`	replace data in memory.
`no[names]`	informs Stata whether variable names are included on the first line of the file.

Example

Contents of the file « file_1.raw » *is*:

```
========================= Contents of  file_1.raw ========================
income ,age ,area
1000   ,34  ,1
3200   ,39  ,2
1700   ,40  ,1
2700   ,54  ,2
==========================================================================
```

The appropriate command line to load this file is:

```
insheet using data\file_1.raw, comma clear
```

4.1.3.2 The command infile

This command is similar to insheet but it is less restrictive about the format of the **ASCII file** (by default, .raw is the extension of the ASCII file). The syntax of this command is:

```
infile varlist using filename [if] [in] [, options]
```

Options

automatic	create value labels from the non numeric data.
byvariable(#)	organize external file by variables; # is number of variables.
clear	replace data in memory.

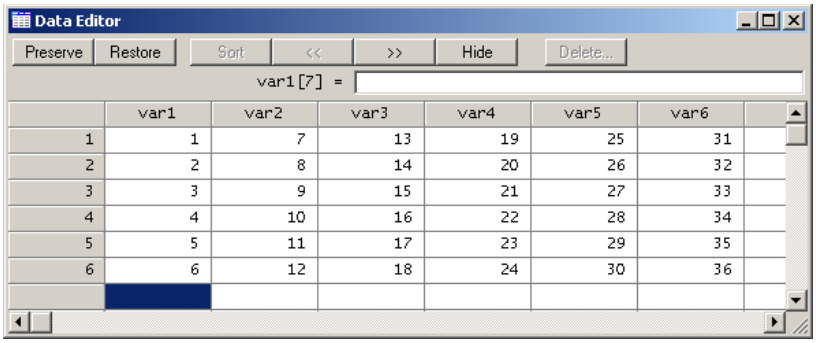
There are two general cases. The first case concerns non-formatted data with known variable delimiter (space, tabulation or semicolon, etc.).

Example 1

```
infile var1 var2 var3 var4 var5 var6 using data\file_2, clear
```

file 2.raw **The Data Editor**

```
1  7   13  19  25  31
2  8   14  20  26  32
3  9   15  21  27  33
4  10  16  22  28  34
5  11  17  23  29  35
6  12  18  24  30  36
```



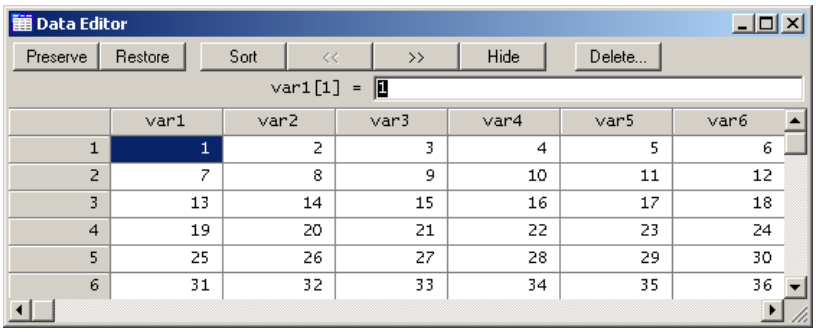
Example 2

```
infile var1-var6 using data\file_2, clear byvariable(6)
```

file 2.raw

```
1  7   13  19  25  31
2  8   14  20  26  32
3  9   15  21  27  33
4  10  16  22  28  34
5  11  17  23  29  35
6  12  18  24  30  36
```

The Data Editor



The second case concerns formatted data. This requires a dictionary to define the position of the values of each variable, as illustrated in the following example:

Example 3

The file `persons.raw` *and* `persons.dct` *must be located in the same position* (*directory*).

```
infile using data\persons
```

persons.raw

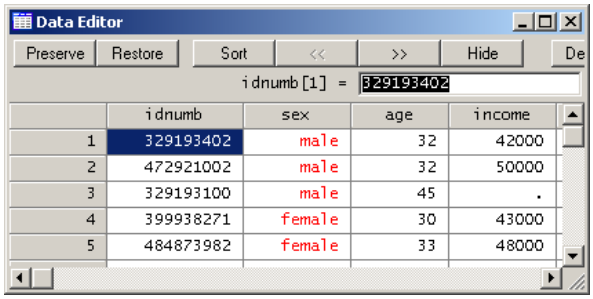
```
    329193402male  32      42000
    472921002male  32      50000
    329193100male  45
    399938271female30      43000
    484873982female33      48000
```

persons.dct

```
dictionary using data\persons.raw {
                            _column(5)
                                      long  idnumb %9f "Identification number"
                                      str6  sex    %6s "Sex"
                                      int   age    %2f "Age"

                            _column(27)
                                      float income %6f "Income"
                            }
```

After the execution of the command line « `infile using data\persons.raw` », we obtain:



4.1.3.3 The command infix

This command allows reading ASCII files with fixed format.

Syntax

infix using dfilename [if] [in] [, using(filename2) clear]

where dfilename is the dictionary file that must contain the following information:

---------- Dictionary file ----------

infix dictionary [using filename] {
* comments
  Specifications
}
(The data must be situated here)

--------- End dictionary file ---------

if dfilename is indicated without an extension, .dct is then the default extension.
if filename2 is indicated without an extension, .raw is then the default extension.

If the option **using filename2** is not used, the data are supposed to be after the line that contains the closing brace "}", which delimits the dictionary information.

Options

using(filename2)	To indicate the filename of the data.
clear	To replace data in memory

Example

infix weight 1-5 age 7-8 using data\file_4, clear

file 4.raw **The Data Editor**

60.30 30
40.20 56
45.45 80
36.10 67

	weight	age
1	60.3	30
2	40.2	56
3	45.45	80
4	36.1	67

4.2 Exporting and saving data

Stata allows saving data in several formats.

Saving a dataset in Stata format: The command save

The command `save` allows to save the data in memory in the Stata format (with extension `*.dta`).

Syntax

```
save [filename] [, save_options]
```

Remarks

- If the filename contains spaces, one has to add quotation marks around the name (*Ex*. "data 1").
- The filename can contain the complete path where the datafile must be saved (*Ex*. "c:/for1/data/data 1").

Options

`nolabel` omit value labels from the saved dataset.

`replace` overwrite existing dataset.

`orphans` save all value labels.

`emptyok` save dataset even if zero observations and zero variables.

Example

```
save data\lsms, replace nolabel
```

We save the current datafile in the directory `data` *with the name* `lsms.dta` *and without label values.*[1]

Good practices

- Add the option `replace` and the datafile will be saved even if it already exists in the same directory.
- Use a short name that indicates clearly the contents of the file.

---

[1] =

- Be sure that all variables and values of categorical variables are labeled before saving the datafile.
- You can also add notes to describe the file or the modifications made before saving the file.

4.2.1 Saving the dataset in ASCII format

It is also possible to save the current dataset in ASCII format to be imported by other software. To this end, Stata offers two possibilities depending on the desired delimiter between variables (space, tab or comma).

- The command **outfile** allows to save the data with space delimiters. The syntax is :

  outfile `varlist using filename [if] [in] [, options]`

  If the file extension is not indicated, the extension .raw is attributed by default.

- The command **outsheets** allows saving the data with tab delimiters. The option comma replaces the tab delimiter by the comma. The syntax is :

  outsheet `varlist using filename [if] [in] [, options]`

  If the file extension is not indicated, the extension .out is attributed by default. Data exported with **outsheet** can be re-imported to Stata with the command **insheet**, and data exported with **outfile** can be re-imported with the command **infile**.

4.3 Labeling variables and values of categorical variables (label)

The command **label** allows assigning labels to the datafile, to variables and to values of the categorical variables. Names of variables often do not allow a useful understanding of what the variables are. The syntax is:

To label the datafile :

```
label data  ["label"]
```

To label a given variable:

```
label variable varname ["label"]
```

To label values of a given variable, we need the following two steps:

1- Defining the label values of the categorical variable;

```
Label define lblname m1 "label_m1" m2 "label_m2"
(where m1 and m2 are integer values)
```

2- Assigning the label values to the categorical variable.

```
label values varname lblname
```

To list labels

```
label list
```

To drop all labels

```
label drop _all
```

Example

```
use          data\data1

lab drop     _all

lab def      larea       1 "rural" 2 "urban"
lab val      area        larea

lab var      hhid        "Household identifier"
lab var      area        "Household area"
lab var      income      "Household total income"
lab var      age_hh      "Age of the household head"
```

5 Descriptive and exploratory analysis of data

Stata makes it possible to inspect variables easily and to calculate simple descriptive statistics.

5.1 Inspecting and comparing variables

The command **inspect** provides a fast synopsis of a numerical variable. It gives the number of negative, zero, and positive values; the number of integer and real values; the number of single values; the number of missing values; and it produces a small histogram. Its goal is not analytical, but it makes it possible to be familiarized quickly with unknown data. The syntax of this command is:

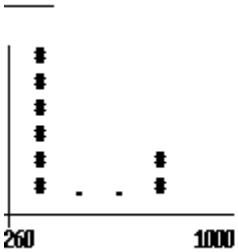
```
inspect [varlist] [if] [in]
```

Example

```
use         data\file1
insp x4
```

	x4
1.	.
2.	1000
3.	300
4.	350
5.	260

x4:



(4 unique values)

	Number of Observations		
	Total	Integers	Noninteger
Negative	-	-	-
Zero	-	-	-
Positive	4	4	-
Total	4	4	-
Missing	1		
	5		

The command **compare** reports differences and similarities between two variables:

compare varname1 varname2 [if] [in]

Example

```
use         data\file1
compare x4 y
```

	x4	y
1.	.	.
2.	1000	4000
3.	300	2100
4.	350	1750
5.	260	780

	count	difference minimum	difference average	difference maximum
x<y	4	-3000	-1680	-520
jointly defined	4	-3000	-1680	-520
jointly missing	1			
total	5			

5.2 Producing simple descriptive statistics: the commands summarize and tabstat

The command **summarize** provides descriptive statistics for numerical variables[2]. Its syntax is:

[**by** varlist :] **summarize** varlist [if] [in] [weight] [, options]

Insofar as no option is specified, Stata produces for each variable of the varlist the number of observations (Obs), the average (Mean), the standard deviation (Std. Dev.), the minimal value (Min.) and the maximum value (Max.). The option detail generates more detailed statistics, such as kurtosis and skewness (a measurement of the asymmetry of the distribution).

[2] See also the command means that can compute arithmetic, geometric and harmonic means.

Example :
The command line

```
bysort education : summarize income
```

produces the descriptive statistics of the variable income *for each of the modalities of the variable* education.

The command **tabstat** makes it possible to produce almost the same results as for **summarize**, but allows greater flexibility in the choice of the descriptive statistics.

Example:
The command line

```
tabstat income size, stats(mean, median, variance, sd, skewness)
```

produces the mean, the median, the variance, the standard deviation, and the skewness of the variables income and size.

5.3 Frequency and cross tabulations statistics: the command tabulate

The command **tabulate** produces one-way tables of frequency counts. Its syntax is:

```
[by varlist :] tabulate varname [if] [in] [weight] [, options]
```

Examples:
The command line

```
tabulate sex if strata == 5, nolabel
```

gives the frequencies of the variable sex (number of males and that of females) in the strata with value 5.

```
tabulate sex, generate(x)
```

gives the frequencies of the variable sex and generates dummy variables for each of the modalities of the variable sex.

In addition, the command **tabulate** creates crossing tables based on two categorical variables.

```
[by varlist :] tabulate var1 var2 [if] [in] [weight] [, options]
```

The option chi2 allows performing a Pearson test of independence (Null Hypothesis: independence of the crossing lines and columns).

Remarks

1. The command **tabulate** is more appropriate with categorical variables.
2. If we wish to produce frequency counts for more than one categorical variable, we can use the command **tab1**:

   tab1 varlist [if] [in] [weight] [, options]

3. If we wish to produce crossing table frequencies for more than one combination of two variables, we can use the command **tab2**:

   tab2 varlist [if] [in] [weight] [, options]

5.4 Obtaining more elaborate descriptive statistics on a given variable: the command table

The command **table** combines the commands **summarize** and **tabulate**. It provides a descriptive statistical table.

Examples:

table region

provides a table of frequencies for the variable region.

table region, contents (mean income median income)

provides the mean and median of the variable income by region.

table region education, c(mean income median size)

provides the mean of the variable income and the median of the variable size for each of the modalities of the variable region and by education level.

5.5 Analyzing the correlation between variables : the command correlate

The command **correlate** allows estimating the correlation or covariance matrix for a list of variables. The syntax of this command is:

[**by** varlist :] **correlate** varlist [if] [in] [weight] [, options]

The usual options for this command are:

Options

covariance	display covariances.
means	display means, standard deviations, minimums, and maximums of variables in addition to the matrix.

Examples

```
correlate income education size in 1/100, means
```

estimates the correlation matrix of the variables income, education and size when the observations are the 100 first observations.

```
correlate income education size, c
```

estimates the variance-covariance matrix *of the variables income, education and size*.

Remark
The command **pwcorr** displays all the pairwise correlation coefficients between the variables in varlist, or between all the variables in the dataset if varlist is not specified.

5.6 Tests on the mean and the variance of variables: the commands `ttest` and `prtest`

The command **ttest** allows performing statistical tests on estimated means or to test the equality of the estimated means of two variables. To perform the tests on the mean, the syntax is:[3]

```
ttest varname == # [if] [in] [, level(#)]
```

To compare the means of two variables, the syntax is:

```
ttest varname1 == varname2 [if] [in] [, options]
```

The command **ttest** tests the difference in means between two population groups.

```
ttest varname [if] [in], by(groupvar) [ options]
```

`by(groupvar)` specifies the group variable.

Examples

```
ttest size == 5 if region==3
```

tests if the average household size equals 5 in region 3

```
ttest income1990 == income2000
```

tests if the difference in average incomes is zero between years 1990 and 2000.

```
ttest income, by(sexe) unequal
```

tests if the difference in average incomes is zero between male and female groups.

The syntax of the command **prtest** is similar to that of the command **ttest,** but it allows performing tests on proportions. The syntax of the **prtest** command is:

[3] The option weight is not allowed with **ttest** and **prtest** commands.

```
prtest varname == p [if] [in] [, level(#)]
```

The variable varname is supposed to be a dummy variable. Moreover, when it is wanted to test if two variables have the same population proportion, the syntax is:

```
prtest varname1 == varname2 [if] [in] [, options]
```

Lastly, when the objective is to test the difference in the proportion of a variable across two groups of the population, the syntax is:

```
prtest varname [if] [in], by(groupvar) [ options]
```

The command **sdtest** resembles **ttest** but tests the variance of a variable or compares the variances of two variables. The syntax of the command to test the variance of a given variable is:

```
sdtest varname == # [if] [in] [, level(#)]
```

If the objective is to compare the variances of two variables, the syntax becomes:

```
sdtest varname1 == varname2 [if] [in] [,level(#)]
```

The command **sdtest** also makes it possible to test the difference in the variance of a given variable between two groups. In this case, the syntax is:

```
sdtest varname [if] [in], by(groupvar) [level(#)]
```

6 Manipulation of variables and observations

6.1 Types of variables

Stata variables can be numerical or alphanumeric. The numeric variables can have different formats (see the following table) according to the level of precision (and this can affect the required memory allocated to Stata). The alphanumeric variables are simply a chain of characters that form what is called a *string*.

The different types of variables in Stata are given in the following table:

Type of variables

Type	Description	Minimum	Maximum	bites
Byte	Integer	-127	127	1
Int	Integer	-32,767	32,740	2
Long	Integer	-2,147,483,647	2,147,483,647	4
Float	variable with decimal	-1.70141173319*10^38	1.70141173319*10^38	4

Double str#	variable with decimal text # characters (ASCII)	-8.9884656743*10^307 --	8.9884656743*10^307 80 (Intercolled)	8 #

The following syntax transforms an alphanumeric variable to a numerical variable:

destring varlist [, options]

The main options are `gen(var)` and `replace`. `gen(var)` generates a new variable named `var` and contains the transformed variable. `replace` deletes the alphanumeric variable and replaces it by the transformed variable. By default, Stata considers that a variable is alphanumeric when at least one of the observations contains a non-numerical character.

The command **recast** allows changing the type of the variable. Its syntax is:

recast type varlist [, force]

The option `force` forces the execution of the command even if this involves an important loss of information.

Example:
Assume that the variable `income` has the float format and we wish to transform it to be an integer (`int`) variable. This will be done by the following command line:

recast int income, force

Income (type float)	Income (type int)
25800,8	25800
30000	30000
32740	32740
35880.4	.

Remark that the fourth observation has a missing value. This is because values of type int cannot exceed 32740.[4]

6.2 Renaming and changing the display format of variables

The command **rename** allows changing the names of variables. Its syntax is the following:

rename old_name new_name

The command **format** allows changing the display format of variables. Its syntax is the following:

[4] See also the functions `round(varname)`, `floor(varname)`, `ceil(varname)`, `int(varname)` to round values of variables.

Syntax

```
format varlist %fmt
```

Some examples of formats (fmt)

fmt

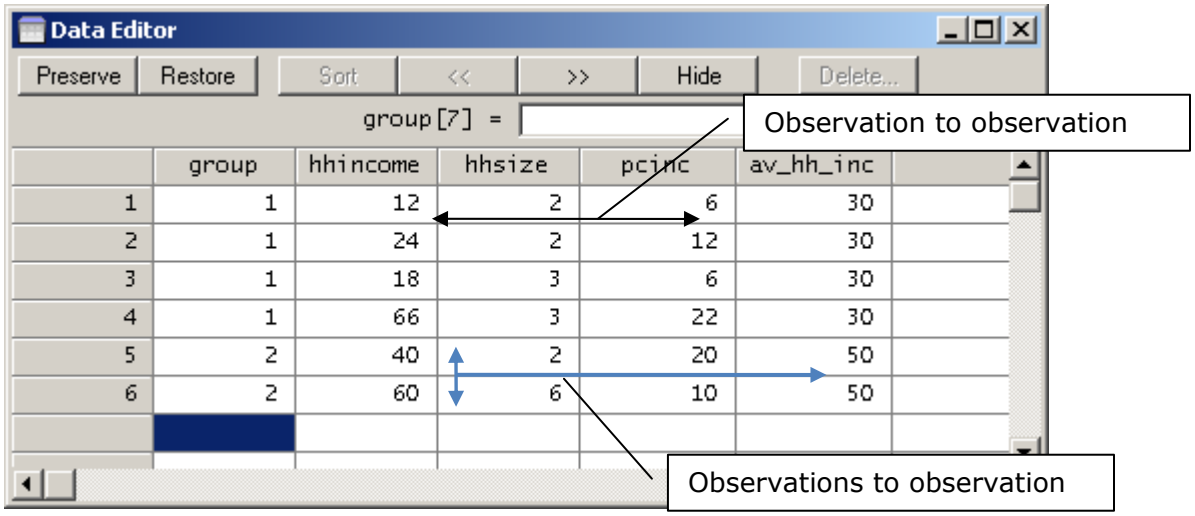
`%*&#g` general format of the numerical values.
`%*&#f` fixed format of the numerical values.
`%*&s` format of string variables.
`&`: space reserved to display the contents.
`#` : decimal precision.
`*` :(* = + or empty) ➔ right centering// (* = -) ➔ left centering.

Example

```
clear
set obs 1
gen float  f_x = 1.1234567890123456
gen double d_x = 12345.1234567890123456
list
format f_x %10.4g
list
format f_x %20.16g
list
format d_x %20.16g
list
```

6.3 Generating new variables

There are two main commands to generate new variables. The command **generate** generates variables that require simple arithmetic computations (observation to observation). The command **egen** (*extended generate*) is more appropriate when computations are based on the whole or a part of the observations (observations to observation).



	group	hhincome	hhsize	pcinc	av_hh_inc
1	1	12	2	6	30
2	1	24	2	12	30
3	1	18	3	6	30
4	1	66	3	22	30
5	2	40	2	20	50
6	2	60	6	10	50

6.3.1 The command generate

The command **generate** generates new variables. Values of these variables are given by **= exp**.

```
generate [type] newvar[:lblname] =exp [if] [in]
```

If the type of variable is not indicated, the type of the new variable is determined automatically by the type of result returned by expression **= exp**. A variable with type float or double is generated if the result is numerical, and a string variable is generated if the result is a text.

Examples

```
use         data\data1, clear
generate    age_hh2 = age_hh*age_hh
generate    poor    = income<800  & income !=.

gen   year  = 2007  /* generates a constant variable year that equals 2007 */
gen   x1 = "poor" in 1/10 /* generates a string variable string that equals "poor" in the
      first 10 observations */
gen   x2 = (x1 == "poor")  /* x2 = 1 if x = "poor" and 0 otherwise */
gen   x3 = (income <= 500) /* x3 = 1 if income <= 500 and 0 otherwise */
gen   x4 = _n /* generates a variable with name x4 and equals the number of lines or observations
*/
gen   x5 = income[_n-1] /* x5 equals the lagged value of income */
gen   x6 = ln(income)  /* equals the logarithm of income. */
gen   x7 = sum(income) /* x7 contains the cumulative values of income */
```

Note that Stata contains many other mathematical functions that can be used to generate new variables. For this, see the help of Stata: functions.

6.3.2 The egen command

The command **egen** is an extension of the command **generate.** Its general syntax is:

```
egen [type] newvar = fcn(arguments) [if] [in] [, options]
```

Examples

```
egen  x = sum(income)  /*generates x where all values equal the sum of income */
egen  t = mean(income) /*generates t where all values equal the mean of income */
egen y = count(income), by(region) /*generates y that indicates the number of non
missing values (number of observations) of the income variable by regions*/
```

Example

```
clear
input       area  income
            1     10
            1     20
```

```
            1     30
            2     15
            2     25
            2     20
            2     90
end
lab def     larea 1 rural 2 urban
lab val     area larea
egen        inc_m      =  mean(income)
list        area income inc_m, mean( income inc_m) labvar(area) sep(0)
```

	area	income	inc_m
1.	rural	10	30
2.	rural	20	30
3.	rural	30	30
4.	urban	15	30
5.	urban	25	30
6.	urban	20	30
7.	urban	90	30
	Mean	30	30

```
bysort area:  egen  inc_m_a =  mean(income)
list   area income inc_m inc_m_a, mean( income inc_m inc_m_a) sep(7) labvar(area)
```

	area	income	inc_m	inc_m_a
1.	rural	10	30	20
2.	rural	20	30	20
3.	rural	30	30	20
4.	urban	15	30	37.5
5.	urban	25	30	37.5
6.	urban	20	30	37.5
7.	urban	90	30	37.5
	Mean	30	30	30

Note that the command **egen** has several other options. For more details, see the help of Stata for the **egen** command.

Good practices

- Use the command **egen** to generate a new variable that contains a desired estimated statistic on the whole population or for some subgroup of that population.
- Each time you generate or modify a variable, check if it contains the accurate value of the expression. This can often be done by producing summary statistics with the command **summarize**.

6.4 Changing the variable values

There are several Stata commands that can change variables.

6.4.1 The commands replace and recode

The command **replace** allows modifying the content of an already existing variable.

Syntax

```
replace oldvar =exp [if] [in] [, nopromote]
```

The option `nopromote` prevents replace from promoting the variable type to accommodate the change.

Examples

```
replace size = 6 if age > 46 & age != . /* Replace the contents of the variable size by 6 if age is higher than 46 and does not contain a missing value*/
```

```
gen x = "poor" in 1/10 /* create a variable string equal to " poor" in the first 10 observations. Consequently, x will take a missing value starting from the 11th observation if the number of observations is higher than 10.*/
```

```
replace x = "non poor" if x == . /* replace all missing values by the string "non poor" */
```

The command **recode** transforms the numerical values of a variable according to an indicated rule. The observations that do not obey the rule indicated by the command **recode** remain unchanged. The basic syntax of this command is:

```
recode varlist (rule) [(rule)] [if] [in] [, options]
```

The main rules are:

Rule	Example	Effect
# = #	3 = 1	3 recoded to 1
# # = #	2 . = 9	2 and . recoded to 9
#/# = #	1/5 = 4	1 through 5 recoded to 4
nonmissing = #	nonmiss = 8	all other nonmissing to 8
missing = #	miss = 9	all other missing to 9

Examples

For x, change 1 to 2, leave all other values unchanged, and store the results in nx

```
. recode x (1 = 2), gen(nx)
```

For x1, swap 1 and 2, and store the results in nx1

```
. recode x1 (1 = 2) (2 = 1), gen(nx1)
```

For x2, collapse 1 and 2 into 1, change 3 to 2, change 4 through 7 to 3, and store the results in nx2

```
. recode x2 (1 2 = 1) (3 = 2) (4/7 = 3), gen(nx2)
```

6.4.2 Delete variables or observations (drop and keep)

The command **drop** allows deleting variables or observations. The command **keep** specifies the set of variables or observations that must be kept.

Example 1

Assume that we want to delete observations whose variable x2 is greater than or equal to 6 (file1.dta):

	hhid	x1	x2	x3	x4	y
1.	110101	11	6	6	.	.
2.	110103	11	4	17	1000	4000
3.	210202	21	7	9	300	2100
4.	310101	31	5	0	350	1750
5.	310102	31	3	3	260	780

We can do this in either of two ways:

```
Drop
use  data\file1
drop if x2 >=  6
```

```
keep
use data\file1
keep if x2 < 6
```

Results

	hhid	x1	x2	x3	x4
1	110103	11	4	17	1000
2	310101	31	5	0	350
3	310102	31	3	3	260

Example 2

Use file1.dta and keep only observations whose variable x1 *takes the value* 31.

This may be done by one of three ways:

```
Keep
use data\file1
keep if x1 == 31
summarize x1
```

```
Drop
use  data\file1
drop if x1 !=  31
tabulate x1
```

```
drop
use  data\file1
drop if x1 < 31 | x1 > 31
tabstat x1, stats (me, sd, mi, ma)
```

Result

	hhid	x1	x2	x3	x4
1	310101	31	5	0	350
2	310102	31	3	3	260

Example 3
Starting from file2.dta:

	hhid	x1	x2	x3	x4	x5	x6
1.	110101	11	5	6	220	1100	45
2.	110102	11	3	13	430	1290	39
3.	110103	11	2	17	850	1700	32
4.	210201	21	4	0	180	720	69
5.	210202	21	5	9	340	1700	58

*Generate two files. The first (*file3.dta*) must contain the variables* hhid, x1, x2 and x3. *The second (*file4.dta*) must contain the variables* x4 and x5.

Creation of file `file3.dta`

```
use  data\file2
keep hhid x1-x3
save data\file3, replace
```

	hhid	x1	x2	x3
1	110101	11	5	6
2	110102	11	3	13
3	110103	11	2	17
4	210201	21	4	0
5	210202	21	5	9

Creation of `file4.dta`

```
use  data\file2
keep x4 x5
save data\file4, replace
```

	x4	x5
1	220	1100
2	430	1290
3	850	1700
4	180	720
5	340	1700

6.4.3 Ordering variables and sorting observations (`order` and `sort`)

The command **`order`** orders variables. For example, the command lines:

```
use   data\file3
order x1 x2 x3 hhid
```

place the variable `x1` in the first column of the datafile `file3.dta`, `x2` in the second column, `x3` in the third column, and `hhid` in the fourth column.

	x1	x2	x3	hhid
1	11	5	6	110101
2	11	3	13	110102
3	11	2	17	110103
4	21	4	0	210201
5	21	5	9	210202

The command **`sort`** orders observations in increasing values of a given variable(s). **`sort`** rearranges all lines, so all variables are rearranged. With the following example,

```
use   data\file3
sort  x2
```

Stata sorts the observations of `file3.dta` in ascending values of the variable `x2` :

	hhid	x1	x2	x3
1	110103	11	2	17
2	110102	11	3	13
3	210201	21	4	0
4	210202	21	5	9
5	110101	11	5	6

The command **`sort`** can be used with several ordering variables. For instance:

```
use   data\file3
sort  x2 hhid
```

rearranges the observations of *`file3.dta`* in ascending values of variable `x2` and in ascending values of variable `hhid` (x2 is used first and in priority to sort the data).

	hhid	x1	x2	x3
1	110103	11	2	17
2	110102	11	3	13
3	210201	21	4	0
4	110101	11	5	6
5	210202	21	5	9

The other command that sorts the data is `gsort`. This command sorts the data in ascending order (by adding "+" before the name of the variable) or in descending order (by adding "-" before the name of the variable). For instance:

```
gsort nvar
```

sorts the observations in ascending values of the variable `nvar`.

```
sort  age -income
```

sorts the observations in ascending values of the variable `age` and in descending values of the variable `income`.

The use of the option `stable` with the command **`sort`** forces the observations with the same value to keep their initial order. For instance, if we have:

x	b
3	1
1	2
1	1
1	3
2	4

and if we type the command line:

```
sort x
```

without using the option stable, we have six possible results:

```
x b | x b | x b | x b | x b | x b
1 2 | 1 2 | 1 1 | 1 1 | 1 3 | 1 3
1 1 | 1 3 | 1 3 | 1 2 | 1 1 | 1 2
1 3 | 1 1 | 1 2 | 1 3 | 1 2 | 1 1
2 4 | 2 4 | 2 4 | 2 4 | 2 4 | 2 4
3 1 | 3 1 | 3 1 | 3 1 | 3 1 | 3 1
```

If we type instead the command line:

```
sort x, stable
```

we necessarily obtain the first result.

6.4.4 The use of commands: foreach, forvalues and while

The command **foreach** is used to generate the same command line with different variables.

Syntax

```
foreach var {in|of listtype} list {
        a given command line that use the variable `var'
}
```

Example

Assume that we need to divide the variables income, exp_school, exp_housing *and* exp_food *by* 12, *to compute their monthly values. We also wish to generate the variable tot_exp (total expenditures by month).*

```
use data/ex_foreach, replace
gen tot_exp = 0
foreach var of varlist  income exp_school exp_housing  exp_food  {
                                qui  replace `var'      = `var'/12
        if ("`var'"~="income")  qui  replace  tot_exp = tot_exp +`var'
     }
```

The command **forvalues** serves to repeat the execution of the command for different numerical values.

Syntax

```
forvalues lname = range {
                         commands referring to `lname'
                        }
```

Example

Assume that we need to sum the variables var1 *to* var6.

```
generate svar=0
forvalues i = 1/3  {
                    replace svar = svar + var`i'
                   }
```

The command **while** serves to execute a command while an expression is true.

Syntax

```
while exp {
                Stata commands
          }
```

While the condition `exp` is satisfied, Stata continues the execution of the Stata commands in the braces.

Example

```
local i  = 1
while `i'<11 {
                display "`i'"
                local i = `i'+1
```

7 Combining datafiles

Stata can open only one database at a time. To clean Stata's memory, the command `clear` should be used. It is an essential operation before loading another datafile.

To use several datafiles, the simplest method consists in opening the first datafile, to use it, to close it, open thereafter the second datafile, etc. If one needs at the same time variables or observations stored in different datafiles, it is necessary to combine these datafiles and to create a new one. For this end, three main methods can be used. Each one of them answers a specific need.

7.1 Appending datafiles: vertical concatenation- (append)

The command append can be used to add new observations to the current datafile. We first open the first datafile.

use *name_of_current_file*, clear

We then use the command **append**:

append using *name_of_second_file* [*, options*]

This makes it possible to append the observations contained in the first datafile to those contained in the second datafile.

Example

Add the observations of file2.dta in file1.dta; eliminate variable x6, then sort the observations in ascending order according to variables hhid and x2, and finally save the new datafile under the name file1_2.dta.

This can be done in either of two ways:

First way

```
use data\file1
append using data\file2
drop x6
sort hhid x2, stable
```

Second way

```
use data\file2
append using data\file1
drop x6
sort hhid x2, stable
```

```
save data\file1_2, replace
```

```
save data\file1_2, replace
```

Result

	hhid	x1	x2	x3	x4	y	x5
1	110101	11	5	6	220	.	1100
2	110101	11	6	6	.	.	.
3	110102	11	3	13	430	.	1290
4	110103	11	2	17	850	.	1700
5	110103	11	4	17	1000	4000	.
6	210201	21	4	0	180	.	720
7	210202	21	5	9	340	.	1700
8	210202	21	7	9	300	2100	.
9	310101	31	5	0	350	1750	.
10	310102	31	3	3	260	780	.

Hints

- The command **cf** can be used to check whether the two files to be concatenated have the same variables with the same names (ex. `cf _all using data\file2`).

Remarks

1. If the variable y in file1.dta refers to the same thing as the variable x5 in file2.dta, the concatenation with the command **append** will contain the two variables y and x5 with missing values for each of the two variables. It is thus important to give the same name (by "rename") to variables that refer to the same thing.[5]
2. In the case in which the two datafiles are for two different years (for example 2001 for *file1.dta* and 2002 for *file2.dta*), it can be difficult to distinguish between observations that come from different years. To avoid this, one can create a variable year that contains the *survey year information*.

   The following program shows how to take into account these two remarks:

Program

```
use data\file1, clear
rename y x5
generate year = 2001
append using data\file2
drop x6
replace year = 2002 if year == .
sort hhid x2, stable
save data\file1_2, replace
```

Result

[5] Recall that the command **rename** allows changing the name of a variable.

	hhid	x1	x2	x3	x4	x5	annee
1	110101	11	5	6	220	1100	2002
2	110101	11	6	6	.	.	2001
3	110102	11	3	13	430	1290	2002
4	110103	11	2	17	850	1700	2002
5	110103	11	4	17	1000	4000	2001
6	210201	21	4	0	180	720	2002
7	210202	21	5	9	340	1700	2002
8	210202	21	7	9	300	2100	2001
9	310101	31	5	0	350	1750	2001
10	310102	31	3	3	260	780	2001

7.2 Merging datafiles: horizontal concatenation- (merge)

We may sometimes need variables that are stored in different datafiles but belong to the same sample. This is often the case with household surveys, for which the entire dataset is saved in different datafiles according to the main parts of the questionnaire, for instance, household characteristics, household expenditures, etc.

The command `merge` allows adding new variables to the current datafile. It obeys certain rules:

- There is a master datafile and a secondary datafile.
- By default, if a variable is present in the two datafiles, then values of the master datafile will remain unchanged after the merging process.
- If some variables of the secondary datafile have the same variable names in the master datafile, but the contents of the variables are different, one must change the names of these variables in one of the two datafiles before merging (for instance, by using the command **`rename)`**.

The use of the command **`merge`** involves the creation of a new variable named _merge which summarizes the result of the merging procedure. The possible values of _merge are:

- `_merge` = 1 when the data for the observation comes exclusively from the master datafile;
- `_merge` = 2 when the data for the observation comes exclusively from the secondary datafile;
- `_merge` = 3 when the data for the observation comes from the two datafiles.
-

7.2.1 Merging with one to one by observation

When the different files to be merged have the same number of observations and in the same order, they can be merged in the following way:

::: Je suprime le lien puisqu'on explique le cas:::

Program

```
use         data\file3, clear
merge       using data\file4
tabulate    _merge
```

Result

	hhid	x1	x2	x3	x4	x5	_merge
1	110101	11	5	6	220	1100	3
2	110102	11	3	13	430	1290	3
3	110103	11	2	17	850	1700	3
4	210201	21	4	0	180	720	3
5	210202	21	5	9	340	1700	3

Remarks

1. One should not sort the data before merging.
2. With observation by observation option, _merge = 3 means that the two datafiles have the same number of observations.
3. It is strongly recommended to merge by using key variables, such as a unique identifier of observations.

If there are more than two datafiles to be merged, the procedure is:

```
use         data\base_1, clear
merge using data\base_2 data\base_3 data\base_4
```

In the case of observation-by-observation merging, the variable _merge must take only a value of 3 since each observation must come from the two datafiles. If _merge is different from 3, this suggests that *observation by observation* merging is not adequate, since the merged files do not have the same observations. It is then recommended to use key variables to merge the datafiles.

7.2.2 One-to-one merging by key-variables

This procedure is useful when some of the observations are the same in the two datafiles but the others are different. In addition to the earlier rules, we have:

1. The two datafiles must contain at least one common variable. It is the key matching variable according to which the observations will be merged.
2. It is possible to use several key matching variables (example: strata, enumeration_area, etc). These variables must, however, be of the same type (numerical or alphanumeric) in the two datafiles.
3. The two datafiles, if necessary, should be sorted in ascending order of the key matching variables. Merging of several datafiles using key variables can generally be carried out as follows:

```
use   data\base_1
merge x1 x2 using data\base_2 data\base_3, unique sort
```

where `x1` and `x2` are two key matching variables and `unique` and `sort` are two among several possible options for the command **`merge`**.

The option `unique` indicates that matching variables `x1` and `x2` represents the unique observation identifiers in the master and secondary files. If this is not the case, Stata displays an error message and merging will not be carried out.[6] The `sort` indicates that the two datafiles to be merged can be sorted if necessary.

Example

Choosing file1.dta as the master datafile, change the name of the variable y by x5 and then merge this data with the file2.dta.

[6] The command **`isid`** allows checking if the key matching variables are unique identifiers.

Program

```
use data\file1, clear
* isid hhid
rename y x5
merge hhid using data\file2, unique sort
sort hhid
tabulate _merge
* drop _merge
```

Result

	hhid	x1	x2	x3	x4	x5	x6	_merge
1	110101	11	6	6	.	.	45	3
2	110102	11	3	13	430	1290	39	2
3	110103	11	4	17	1000	4000	32	3
4	210201	21	4	0	180	720	69	2
5	210202	21	7	9	300	2100	58	3
6	310101	31	5	0	350	1750	.	1
7	310102	31	3	3	260	780	.	1

Note that the values of the variables common to the two datafiles are those of the master file even if these are missing values. To update the datafile and get around this restriction, we can use the option `update` or options `update` and `replace`, as detailed in the following subsection.

7.2.3 Updating the datafiles (`merge`, `update` and `merge`, `update replace`)

Suppose we wish to update or complete a datafile (to replace old or missing values by new values for instance). In this case, suppose that the secondary datafile (using datafile) contains the new data.

When the option `update` is not followed by the option `replace`, i.e.

```
use         data\base_1
merge       x using data\base_2, update unique sort
```

only the missing values of the master file are updated. However, if the option `replace` is used with the option `update`, i.e.

```
use         data\base_1
merge       x using data\base_2, update replace unique sort
```

even the non-missing values of the master file are replaced.

With the command **`merge`** and option `update`, the possible values of the variable _merge are the following:

_merge = 1 when the data of the observation come exclusively from the master datafile.

_merge = 2 when the data of the observation come exclusively from the secondary datafile.

_merge = 3 when the data of the observation come from the two datafiles.

_merge = 4 when missing master values are updated.

_merge= 5 when old master values are updated.

Example 1

Let us suppose that the datafiles file1.dta and file2.dta refer to the same sample, but are produced by two organisations. Let us suppose that we feel more confident with the data of file1.dta but that the data of file2.dta remain useful because they can be used to replace the missing values of file1.dta and to increase the number of non-missing observations. What is the best merging strategy?

Program

```
use data\file1, clear
rename y x5
merge hhid using data\file2, update unique sort
sort hhid
tabulate _merge
* drop _merge
```

Results

	hhid	x1	x2	x3	x4	x5	x6	_merge
1	110101	11	6	6	220	1100	45	5
2	110102	11	3	13	430	1290	39	2
3	110103	11	4	17	1000	4000	32	5
4	210201	21	4	0	180	720	69	2
5	210202	21	7	9	300	2100	58	5
6	310101	31	5	0	350	1750	.	1
7	310102	31	3	3	260	780	.	1

In this case, only the missing values of `file1.dta` are replaced by the correspondent values of `file2.dta`. Thus, when the command `merge` and the option `update` are used without the option `replace`, the variable _merge takes the value of 5 for the updated values and Stata preserves the values of the Master file. Insofar as the values of the secondary file are considered more reliable, we use the command `merge` with the options `update` and `replace`, i.e.

```
merge       varlist using filename , update replace unique sort
```

Example 2

We wish to complete file1.dta with the values of file2.dta. Moreover, the values of the secondary file are considered to be more reliable.

Program

```
use data\file1, clear
rename y x5
merge hhid using data\file2, update replace unique sort
sort hhid
tabulate _merge
* drop _merge
```

Results

	hhid	x1	x2	x3	x4	x5	x6	_merge
1	110101	11	5	6	220	1100	45	5
2	110102	11	3	13	430	1290	39	2
3	110103	11	2	17	850	1700	32	5
4	210201	21	4	0	180	720	69	2
5	210202	21	5	9	340	1700	58	5
6	310101	31	5	0	350	1750	.	1
7	310102	31	3	3	260	780	.	1

In this case, the missing and non missing values of file1 are replaced by those of file2.

Example 3

Redo the example 2, but when file2.dta is considered to be the master file.

Program

```
use data\file2, clear
rename x5 y
merge hhid using data\file1, update unique sort
rename y x5
sort hhid
tabulate _merge
* drop _merge
```

Result

	hhid	x1	x2	x3	x4	x5	x6	_merge
1	110101	11	5	6	220	1100	45	5
2	110102	11	3	13	430	1290	39	1
3	110103	11	2	17	850	1700	32	5
4	210201	21	4	0	180	720	69	1
5	210202	21	5	9	340	1700	58	5
6	310101	31	5	0	350	1750	.	2
7	310102	31	3	3	260	780	.	2

8 Managing databases with Stata

Stata contains several useful commands to organize databases. Data structure can differ from one datafile to another for many practical reasons. In distributive analysis, we often use income-expenditures household surveys. These files contain information on the socio-demographic characteristics of household members. Two household survey file types are usually found. The first has lines containing information on households, with a unique identifier for each household and other variables such as total household income. The second contains information on individual household members. Each line contains information on only one member of each surveyed household, along with the unique identifier of that household. Variables of that second file type concern the individual, such as the age of the individual, his education level, etc.

8.1 The command `collapse`

The command **`collapse`** aggregates the dataset. For instance, starting from the individual file, we can generate a household file. The syntax of that command is:

```
collapse clist [if] [in] [weight] [, options]
```

Options

by(varlist) specifies the key variables over which the aggregation will be preformed. If this option is not specified, the resulting dataset will contain 1 observation. If it is specified, varlist may refer to either string or numerical variables.

Example

In this example, we assume that the individual file is as follows:

Individual file

hhid	income	size	exp_fact	age
11	300	3	10	34

11	0	3	10	29
11	0	3	10	4
13	260	2	20	34
13	0	2	20	35
16	780	4	12	69
16	140	4	12	45
16	0	4	12	13
16	0	4	12	16

Also, assume that the household head is situated on the first line of household members observations. The aim is to generate a datafile that takes the following from:

Household file

hhid	**pc_inc**	**size**	**ex_fact**	**age_hh**
11	100	3	10	34
13	130	2	20	45
16	230	4	12	69

The variables of the household file are defined as follows:

hhid	The household identifier
pc_inc	Per capita income
size	Household size
exp_fact	The expansion factor (the sampling weight)
Age_h	Age of the household head

The variable `hhid` is the one to be used for regrouping observations in our example. The averages of variables `income` and `size` and `exp_fact` by household are equivalent to the variables pc_inc, `size` and `exp_fact` in the household file. For the age of the household head, we start by generating a variable that equals the age of the household head.

```
use         data\ex_collapse_ind, clear
by hhid:    gen age_hh = age[1]
collapse    (mean) income size exp_fact age_hh, by(hhid)
rename      income pc_inc
lab var     hhid     "Household indentifier"
lab var     pc_inc    "Per capita income"
lab var     size     "Household size"
lab var     exp_fact  "Sampling weight"
lab var     age_hh "Age of household head"
save        c:\data\ex_collapse_household.dta, replace
```

8.2 The command expand

The command **`expand`** makes it possible to increase or replace each observation by `n` copies, where `n` is an integer value. If the expression indicating `n` is lower than 1 or if it is a missing value, it is interpreted as being equal to 1. The syntax is:

```
expand [=]exp [if] [in]
```

Example
Using the file ex_collapse_household.dta, generate a new file with the two variables hhid *and* f_exp, *which corresponds to that of the individual level*.

```
use          data/ex_collapse_household.dta, clear
expand       size
sort         hhid
keep         hhid exp_fact
```

8.3 The command reshape

The command **reshape long** makes it possible to convert a database of a "wide" format to "long" format and the command reshape wide makes it possible to make the opposite operation.

```
reshape long stubnames, i(varlist) j(varname) [options]
```

```
reshape wide stubnames, i(varlist) j(varname) [options]
```

Example:

Consider the following two databases. The first has a wide format and the second a long one.

Base 1: Wide format

	hhid	income1980	income1990	income2000	sex
1	1	5000	5500	6000	0
2	2	2000	2200	3300	1
3	3	3000	200	1000	0

Base 2: Long format

	hhid	year	income	sex
1	1	1980	5000	0
2	1	1990	5500	0
3	1	2000	6000	0
4	2	1980	2000	1
5	2	1990	2200	1
6	2	2000	3300	1
7	3	1980	3000	0
8	3	1990	200	0
9	3	2000	1000	0

To transform the format of the first file to that of the second, we use the command **reshape long** as follows:

```
use data/ex_reshape_l, replace
reshape long income, i(hhid) j(year)
```

To transform the format of the second base to that of the first base, we use the command **reshape wide** as follows:

```
use data/ex_reshape_w, replace
reshape wide income, i(hhid) j(year)
```

8.4 The command contract

The command **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. Its syntax is:

```
contract varlist [if] [in] [weight] [, options]
```

The option freq(varname) specifies the name of variable of frequencies. If this option is not used the name by default will be _freq.

Example :

Assume that we have the following household file:

Household file

hhid	income	size	age
11	100	3	34
13	130	2	45
16	230	4	69
20	130	2	45
24	100	3	34
33	130	2	45

The execution of the command

```
contract income age, freq(w_freq)
```

produces the following result:

income	age	w_freq
100	34	2
130	45	3
230	69	1
130	45	2

9 The basics of Stata programming

In addition to writing sets of Stata command lines and saving them into text files with extension .do, Stata also enables programmers to provide specialized *.ado* (an automatic do-file) routines to add to the power of the software.

Stata ado files usually serve to perform precise tasks using some predefined input. For instance, the command `mean` estimates the mean of a variable and displays the result. The minimum required input information for this command is the name of the variable whose mean will be estimated.

9.1 Local and global macros and scalars

In Stata, a macro may contain many elements that are a combination of alphanumeric characters (more than 8000 characters in all versions of Stata). A local macro is usually defined in a do or ado file. A global macro may be initialized at any Stata execution level and continues to exist until explicitly dropped by the user or at the end of a Stata session.

Example 1: Local macros

```
local lcountry CAM UGA BOT SAF
dis "`lcountry'"
local count 0
foreach c of local lcountry {
local count = `count'+1
display "Country  `count': `c'"
}
```

The displayed results are:

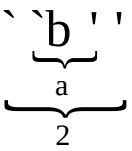
```
Country  1: CAM
Country  2: UGA
Country  3: BOT
Country  4: SAF
```

In the above example, the local macro `lcountry` contains a list of names of four African countries.

☐ Remark that we do not use the character "=" to assign the value to the local macro. This practice is not recommended. Indeed, using the "=" character will force Stata to evaluate the assigned content. In Stata, the local macro may be considered as an alias that contains the name and implicitly the value.

Example 2: Local macros

```
local a = 2
local b   a
dis "the name  of macro b is  : `b'    "
dis "the value of macro b is : ``b''   "
```



The displayed results are:

```
. dis "the name  of macro b is  : `b'    "
the name  of macro b is  : a

. dis "the value of macro b is : ``b''   "
the value of macro b is : 2
```

- Remark that we need to put the name of the local macro between the two specific characters: (`) – left tick character- and (') - the apostrophe- to invoke its content.

Example 3: Global macros

```
global nsqpi square_of_pi
global square_of_pi sqrt(_pi)
dis “$nsqpi”
dis $$nsqpi
```

The displayed results are:

```
. dis "$nsqpi"
square_of_pi

. dis $$nsqpi
1.7724539
```

- Remark that we need to precede the name of a global macro with the $ character to invole its content.
- Scalars are typically used to store numerical values or numerical results. In contrast to local or global macros, we do not need to precede the scalar with a special character to refer to its value.

Example 4: scalars

```
scalar pi = _pi
dis pi
```

The displayed results are:

```
. scalar pi = _pi

.
. dis pi
3.1415927
```

9.2 The Stata program

This section discusses a more ambitious task, namely, how to write our own Stata program. First note that an ado file that is saved in the Stata ado paths can be executed until the redefinition of a new command.

9.2.1 Defining and storing new Stata programs

The first step in designing a new Stata ado file is to write a text file that contains the contents of the program and to save it in some Stata ado path folder (for instance c:/ado/personal) with the same name as that of the program and with the extension *.ado.

==========Contents of the file c:/ado/personal/myprog.ado==========

```
*! myprog v1.0.1 UNDP 16April2010
capture program drop myprog
program define myprog
version 10.0
args nvar
quietly sum `nvar'
dis "The mean of `nvar' equals:" %16.3f `r(mean)'
end
```

1- The first line: `*! myprog v1.0.1 UNDP 16April2010` is used to show information on the command or the program (version, authors, dates, etc.).
2- The command line `capture program drop myprog` is equivalent to ask Stata to try to drop the program with name myprog. This avoids the error of defining a program that is already defined.
3- The command line `program define myprog` is used to define the new program with name `myprog` and to mark its beginning.
4- The command line `version 10.0` is used to indicate the minimum required version of Stata to run the new command.
5- The command line args `nvar` is used to indicate the arguments of the inputs to the program. This program estimates and displays the mean of a given variable. The minimum required information is the name of this variable.
6- The command line `end` marks the end of the program.

9.2.2 The syntax of the program

The definition of the syntax of the new program allows to Stata to parse the content of the command line and to catch the inputted information (name of variables, options, etc.). The general form of the syntax is as follows:

```
command [varlist] [=exp] [if] [in]  , options
```

Example 4: the syntax of the program

```
*! myprog v1.0.2 UNDP 17April2010
capture program drop myprog
program define myprog
version 10.0
syntax varlist(min=1) [if] [in]
foreach var of varlist `varlist' {
quietly sum `var' `if' `in'
dis "The mean of `var' equals:" %16.3f `r(mean)'
}
end
```

The command line - **syntax varlist(min=1) [if] [in]** – shows the desired form of the syntax of the new command `myprog`. After typing the command, the user can indicate a list of variables to estimate their means. Also, the program allows to restrict the observations to be used by the qualifiers if and in.

9.2.3 The ouputs of the program

The outputs of the program may take different forms, such as:

- displaying results in the results window;
- generating a new variable;
- drawing a specific graph;
- storing the results as scalars and matrices;

The option `rclass` allows returning results in scalar or macro formats.
Example 5: the return list

```
*! myprog v1.0.3 UNDP 17April2010
capture program drop myprog
program define myprog, rclass
version 10.0
syntax varlist(min=1) [if] [in]
foreach var of varlist `varlist' {
quietly sum `var' `if' `in'
dis "The mean of `var' equals:" %16.3f `r(mean)'
return scalar m_`var' = `r(mean)'
}
return local var `varlist'
end
```

9.2.4 Making the program byable

The option `byable` allows running the command over each population group.
Example 6: the return list

```
*! myprog v1.0.4 UNDP 17April2010
capture program drop myprog
program define myprog, rclass byable(recall) sortpreserve
version 10.0
syntax varlist(min=1) [if] [in]
```

```
foreach var of varlist `varlist' {
marksample touse
quietly sum `var' if `touse'
dis "The mean of `var' equals to:" %16.3f `r(mean)'
return scalar m_`var' = `r(mean)'
}
return local var `varlist'
end
```