

# **EViews 5 Command and Programming Reference**



Quantitative Micro Software

# EViews 5 Command and Programming Reference

Copyright © 1994–2004 Quantitative Micro Software, LLC

All Rights Reserved

Printed in the United States of America

This software product, including program code and manual, is copyrighted, and all rights are reserved by Quantitative Micro Software, LLC. The distribution and sale of this product are intended for the use of the original purchaser only. Except as permitted under the United States Copyright Act of 1976, no part of this product may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of Quantitative Micro Software.

## Disclaimer

The authors and Quantitative Micro Software assume no responsibility for any errors that may appear in this manual or the EViews program. The user assumes all responsibility for the selection of the program to achieve intended results, and for the installation, use, and results obtained from the program.

## Trademarks

Windows, Windows 95/98/2000/NT/Me/XP, and Microsoft Excel are trademarks of Microsoft Corporation. PostScript is a trademark of Adobe Corporation. X11.2 and X12-ARIMA Version 0.2.7 are seasonal adjustment programs developed by the U. S. Census Bureau. Tramo/Seats is copyright by Agustin Maravall and Victor Gomez. All other product names mentioned in this manual may be trademarks or registered trademarks of their respective companies.

Quantitative Micro Software, LLC

4521 Campus Drive, #336, Irvine CA, 92612-2699

Telephone: (949) 856-3368

Fax: (949) 856-2044

e-mail: [sales@eviews.com](mailto:sales@eviews.com)

web: [www.eviews.com](http://www.eviews.com)

April 15, 2004

# Table of Contents

---

<b>CHAPTER 1. INTRODUCTION .....</b>	<b>1</b>
Using Commands .....	1
Batch Program Use .....	2
How to Use this Manual .....	2
<b>CHAPTER 2. OBJECT AND COMMAND BASICS .....</b>	<b>5</b>
Object Declaration .....	5
Object Commands .....	6
Object Assignment .....	9
More on Object Declaration .....	9
Auxiliary Commands .....	10
Managing Workfiles and Databases .....	11
Managing Objects .....	13
Basic Command Summary .....	16
<b>CHAPTER 3. MATRIX LANGUAGE .....</b>	<b>23</b>
Declaring Matrices .....	23
Assigning Matrix Values .....	24
Copying Data Between Objects .....	27
Matrix Expressions .....	34
Matrix Commands and Functions .....	37
Matrix Views and Procs .....	39
Matrix Operations versus Loop Operations .....	41
Summary of Automatic Resizing of Matrix Objects .....	42
Matrix Function and Command Summary .....	44
<b>CHAPTER 4. WORKING WITH TABLES .....</b>	<b>47</b>
Creating a Table .....	47
Assigning Table Values .....	48
Customizing Tables .....	50
Labeling Tables .....	56
Printing Tables .....	57
Exporting Tables to Files .....	57
Customizing Spreadsheet Views .....	57
Table Summary .....	58

<b>CHAPTER 5. WORKING WITH GRAPHS .....</b>	<b>59</b>
Creating a Graph .....	59
Changing Graph Types .....	63
Customizing a Graph .....	64
Labeling Graphs .....	81
Printing Graphs .....	81
Exporting Graphs to Files .....	81
Graph Summary .....	82
<b>CHAPTER 6. EVIEW'S PROGRAMMING .....</b>	<b>83</b>
Program Basics .....	83
Simple Programs .....	86
Program Variables .....	88
Program Modes .....	96
Program Arguments .....	97
Control of Execution .....	99
Multiple Program Files .....	107
Subroutines .....	108
Programming Summary .....	115
<b>CHAPTER 7. STRINGS AND DATES .....</b>	<b>117</b>
Strings .....	117
Dates .....	127
<b>APPENDIX A. OBJECT, VIEW AND PROCEDURE REFERENCE .....</b>	<b>151</b>
Alpha .....	152
Coef .....	153
Equation .....	155
Graph .....	159
Group .....	161
Link .....	163
Logl .....	164
Matrix .....	166
Model .....	168
Pool .....	169
Rowvector .....	172
Sample .....	173

---

Scalar .....	174
Series .....	175
Sspace .....	177
Sym .....	180
System .....	182
Table .....	185
Text .....	186
Valmap .....	187
Var .....	187
Vector .....	190
<b>APPENDIX B. COMMAND REFERENCE .....</b>	<b>193</b>
<b>APPENDIX C. SPECIAL EXPRESSION REFERENCE .....</b>	<b>535</b>
<b>APPENDIX D. OPERATOR AND FUNCTION REFERENCE .....</b>	<b>543</b>
Operators .....	544
Basic Mathematical Functions .....	545
Time Series Functions .....	546
Descriptive Statistics .....	547
By-Group Statistics .....	549
Special Functions .....	550
Trigonometric Functions .....	553
Statistical Distribution Functions .....	554
<b>APPENDIX E. WORKFILE FUNCTIONS .....</b>	<b>559</b>
Basic Workfile Information .....	559
Dated Workfile Information .....	560
Panel Workfile Functions .....	563
<b>APPENDIX F. STRING AND DATE FUNCTION REFERENCE .....</b>	<b>565</b>
<b>APPENDIX G. MATRIX REFERENCE .....</b>	<b>581</b>
<b>APPENDIX H. PROGRAMMING LANGUAGE REFERENCE .....</b>	<b>603</b>
<b>INDEX .....</b>	<b>615</b>



# Chapter 1. Introduction

---

EViews provides you with both a Windows and a command line interface for working with your data. Almost every operation that can be accomplished using menus may also be entered into the command window, or placed in programs for batch processing. You are free to choose the mixture of techniques which best fits your particular style of work.

The *Command and Programming Reference (CPR)* documents the use of commands and programs to perform various tasks in EViews—the companion *User's Guide* describes in greater detail the general features of EViews, with an emphasis on the interactive Windows interface.

In addition to providing a basic command reference, the *Command and Programming Reference* documents the use of EViews' powerful batch processing language and advanced programming features. With EViews, you can create and store commands in programs that automate repetitive tasks, or generate a record of your research project.

## Using Commands

Commands may be used *interactively* or executed in *batch* mode.

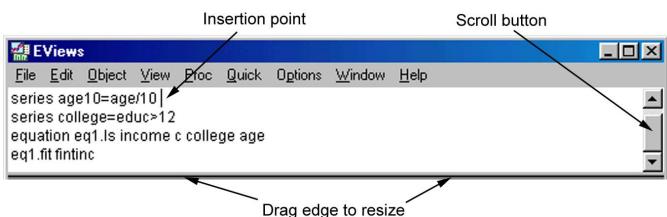
### Interactive Use

The *command window* is located just below the main menu bar at the top of the EViews window. A blinking insertion cursor in the command window indicates that keyboard focus is in the command window and that keystrokes will be entered in the window at the insertion point. If no insertion cursor is present, simply click in the command window to change the focus.

To work interactively, you will type a command into the command window, then press ENTER to execute the command. If you enter an incomplete command, EViews will open a dialog box prompting you for additional information.

A command that you enter in the window will be executed as soon as you press ENTER. The insertion point need not be at the end of the command line when you press ENTER.

EViews will execute the entire line that contains the insertion point.



When you enter a command, EViews will add it to the list of previously executed commands contained in the window. You can scroll up to an earlier command, edit it, and hit ENTER. The modified command will be executed. You may also use standard Windows copy-and-paste between the command window and any other window.

The contents of the command area may also be saved directly into a text file for later use. First make certain that the command window is active by clicking anywhere in the window, and then select **File/Save As...** from the main menu.

You may resize the command window so that a larger number of previously executed commands are visible. Use the mouse to move the cursor to the bottom of the window, hold down the mouse button, and drag the bottom of the window downwards.

We will point out that as you open and close object windows in EViews, the keyboard focus may change from the command window to the active window. If you then wish to enter a command, you will first need to click in the command window to set the focus. You can influence EViews' method of choosing keyboard focus by changing the global defaults—simply select **Options/Window and Font Options...** from the main menu, and change the **Keyboard Focus** setting as desired.

## Batch Program Use

You can assemble a number of commands into a *program*, and then execute the commands in batch mode. Each command in the program will be executed in the order that it appears in the program. Using batch programs allows you to make use of advanced capabilities such as looping and condition branching, and subroutine and macro processing. Programs also are an excellent way to document a research project since you will have a record of each step of the project.

One way to create a program file in EViews is to select **File/New/Program**. EViews will open an untitled program window into which you may enter your commands. You can save the program by clicking on the **Save** or **SaveAs** button, navigating to the desired directory, and entering a file name. EViews will append the extension “.PRG” to the name you provide.

Alternatively, you can use your favorite text (ASCII) editor to create a program file containing your commands. The commands in this program may then be executed from within EViews.

## How to Use this Manual

[Chapter 2, “Object and Command Basics”, on page 5](#) and [Appendix A, “Object, View and Procedure Reference”, on page 151](#) constitute the core of the EViews command reference:

- [Chapter 2, “Object and Command Basics”, on page 5](#) explains the basics of using commands to work with EViews objects, and provides a cross-referenced listing of basic EViews commands associated with various tasks.
- [Appendix A, “Object, View and Procedure Reference”, beginning on page 151](#) provides a cross-referenced listing of commands, views, and procedures associated with each object.

The other sections provide documentation on more advanced EViews features:

- [Chapter 3, “Matrix Language”, on page 23](#) describes the EViews matrix language and provides a summary of the available matrix operators, functions, and commands.
- [Chapter 4, “Working with Tables”, on page 47](#) documents the table object and describes the basics of working with tables in EViews.
- [Chapter 5, “Working with Graphs”, on page 59](#) describes the use of commands to work with graph objects.
- [Chapter 6, “EViews Programming”, on page 83](#) describes the basics of using programs for batch processing and documents the programming language.
- [Chapter 7, “Strings and Dates”, on page 117](#) describes the syntax and functions available for manipulating text strings and dates.

The remaining sections contain dictionary-style reference material for all of the EViews commands, functions, and operators, or more advanced material:

- [Appendix B, “Command Reference”, on page 193](#) provides a full alphabetized listing of basic commands, views and procedures. This material contains the primary reference material for working with EViews.
- [Appendix C, “Special Expression Reference”, on page 535](#) special expressions that may be used in series assignment and generation, or as terms in estimation specifications.
- [Appendix D, “Operator and Function Reference”, beginning on page 543](#) describes the operators and functions that may be used with series and (in some cases) matrix objects.
- [Appendix E, “Workfile Functions”, on page 559](#) includes functions for accessing workfile structure and date information.
- [Appendix F, “String and Date Function Reference”, on page 565](#) lists the functions used when working with strings and dates in EViews.
- [Appendix G, “Matrix Reference”, on page 581](#) is an alphabetical listing of the commands and functions associated with the EViews matrix language.

- [Appendix H, “Programming Language Reference”, on page 603](#) contains an alphabetical listing of the keywords and functions associated with the EViews programming language.

While this reference manual is not designed to be read from cover-to-cover, we recommend that before beginning extensive work using EViews commands, you spend some time with [Chapter 2, “Object and Command Basics”](#), which describes the basics of using commands to work with objects. A solid understanding of this material is important for getting the most out of EViews.

If you wish to use programs in EViews, you should, at the very least, examine the first part of [Chapter 6, “EViews Programming”](#), which describes the basics of creating, loading, and running a batch program.

## Chapter 2. Object and Command Basics

---

This chapter provides an overview of the command method of working with EViews and EViews objects. If you are new to EViews, you may find it useful to consult the *User's Guide* (especially [Chapter 1, "Introduction", on page 11](#) and [Chapter 4, "Object Basics", beginning on page 65](#)) for a more detailed introduction to EViews and a discussion of objects, their views, and procedures.

The command line interface of EViews is comprised of a set of single line commands, each of which may be classified as one of the following:

- object declarations.
- object commands.
- object assignment statements.
- auxiliary commands.

An EViews program is composed of a sequence of these commands, and may also contain the following:

- control variable assignment statements.
- program control statements.

The use of control variables and program control statements is discussed in detail in the programming guide in [Chapter 6, "EViews Programming", on page 83](#). The following sections provide an overview of the first four types of commands.

### Object Declaration

The first step is to create or declare an object. A simple declaration has the form:

`object_type object_name`

where `object_name` is the name you would like to give to the new object and `object_type` is one of the following object types:

 Alpha (p. 152)

 Model (p. 168)

 Sym (p. 180)

 Coef (p. 153)

 Pool (p. 169)

 System (p. 182)

 Equation (p. 155)

 Rowvector (p. 172)

 Table (p. 185)

 Graph (p. 159)

 Sample (p. 173)

 Text (p. 186)

 Group (p. 161) Scalar (p. 174) Valmap (p. 187) Logl (p. 164) Series (p. 175) Var (p. 187) Matrix (p. 166) Sspace (p. 177) Vector (p. 190)

For example, the declaration

```
series lgdp
```

creates a new series called LGDP, while the command

```
equation eq1
```

creates a new equation object called EQ1.

Matrix objects are typically declared with their dimension in parentheses after the object type. For example:

```
matrix(5,5) x
```

creates a  $5 \times 5$  matrix named X, while

```
coef(10) results
```

creates a 10 element coefficient vector named RESULTS.

Note that in order to create an object you must have a workfile currently open in EViews. You can open or create a workfile interactively from the File Menu, or you can use the `load` or `workfile` commands to perform the same operations inside a program. See Chapter 3, “Workfile Basics”, on page 43 of the *User’s Guide* for details.

## Object Commands

An *object command* is a command which accesses an object’s views and procedures (*procs*). Object commands have two main parts, a *display action* followed by a *view specification*. The view specification describes the view or procedure of the object to be acted upon. The display action determines what is to be done with the output from the view or procedure.

The *complete* syntax for an object command has the form:

```
action(action_opt) object_name.view_or_proc(view_proc_opt) arg_list
```

where:

*action* ..... is one of the four verbs (do, freeze, print, show)

*action\_opt* ..... an option that modifies the default behavior of the action

*object\_name* .....the name of the object to be acted upon  
*view\_or\_proc* ...the object view or procedure to be acted upon  
*view\_proc\_opt* .....an option that modifies the default behavior of the view or procedure  
*arg\_list* .....a list of view or procedure arguments, generally separated by spaces

The four possible actions are:

- `do` executes procedures without opening a window. If the object's window is not currently displayed, no output is generated. If the objects window is already open, `do` is equivalent to `show`.
- `freeze` creates a table or graph from the object view window.
- `print` prints the object view window.
- `show` displays the object view in a window.

In most cases, some of the components of the general object command are not necessary since some views and procs do not require an argument list or options.

Furthermore, you need not explicitly specify an action. If no action is provided, the `show` action is assumed for views and the `do` action is assumed for procedures. For example, when using the command to display the series view for a line graph:

```
gdp.line
```

EViews implicitly adds a `show` command:

```
show gdp.line
```

Alternatively, for the equation procedure `ls`,

```
eq1.ls cons c gdp
```

there is an implicit `do` action.

```
do eq1.ls cons c gdp
```

In some cases, you may wish to modify the default behavior by explicitly describing the action. For example:

```
print eq1.ls cons c gdp
```

both performs the implicit `do` action and then sends the output from the proc to the printer.

```
show gdp.line
print(1) group1.stats
```

```
freeze(output1) eq1.ls cons c gdp  
do eq1.forecast eq1f
```

The first example opens a window displaying a line graph of the series GDP. The second example prints (in landscape mode) descriptive statistics for the series in GROUP1. The third example creates a table named OUTPUT1 from the estimation results of EQ1 for a least squares regression of CONS on GDP. The final example executes the forecast procedure of EQ1, putting the forecasted values into the series EQ1F and suppressing any procedure output.

Of these four examples, only the first opens a window and displays output on the screen.

## Output Control

As discussed above, the display action determines the destination for view and procedure output. Here we note in passing a few extensions to these general rules.

You may request that a view be simultaneously printed and displayed on your screen by the letter “p” as an option to the object command. For example, the expression:

```
gdp.correl(24,p)
```

is equivalent to the two commands:

```
show gdp.correl(24)  
print gdp.correl(24)
```

since `correl` is a series view. The “p” option can be combined with other options, separated by commas. So as not to interfere with other option processing, we strongly recommend that the “p” option should *always be specified after any required options*.

Note that the `print` command accepts the “l” or “p” option to indicate landscape or portrait orientation. For example:

```
print(l) gdp.correl(24)
```

Printer output can be redirected to a text file or frozen output. See the `output` command in [Appendix B, “Command Reference”, on page 193](#), and the discussion in [“Print Setup” on page 919](#) of the *User’s Guide*, for details.

The `freeze` command used without options creates an untitled graph or table from a view specification:

```
freeze gdp.line
```

You also may provide a name for the frozen object in parentheses after the word `freeze`. For example,

```
freeze(figure1) gdp.bar  
names the frozen bar graph of GDP as "figure1".
```

## Object Assignment

*Object assignment statements* are commands which assign data to an EViews object. Object assignment statements have the syntax:

```
object_name = expression
```

where `object_name` identifies the object whose data is to be modified and `expression` is an expression which evaluates to an object of an appropriate type.

The nature of the assignment varies depending on what type of object is on the left hand side of the equal sign. To take a simple example, consider the assignment statement:

```
x = 5 * log(y) + z
```

where X, Y and Z are series. This assignment statement will take the log of each element of Y, multiply each value by 5, add the corresponding element of Z, and, finally, assign the result into the appropriate element of X.

## More on Object Declaration

Object declarations can often be combined with either object commands or object assignment statements to create and initialize an object in a single line. For example:

```
series lgdp = log(gdp)
```

creates a new series called LGDP and initializes its elements with the log of the series GDP. Similarly, the command

```
equation eq1.ls y c x1 x2
```

creates a new equation object called EQ1 and initializes it with the results from regressing the series Y against a constant term, the series X1 and the series X2.

Additional examples:

```
scalar elas = 2  
series tr58 = @trend(1958)  
group nipa gdp cons inv g x  
equation cnstfnc2.ls log(cons)=c(1)+c(2)*yd  
vector beta = @inverse(x*x)*(x*y)
```

An object can be declared multiple times so long as it is always declared to be of the same type. The first declaration will create the object, subsequent declarations will have no effect unless the subsequent declaration also specifies how the object is to be initialized. For example:

```
smp1 @first 1979  
series dummy = 1  
smp1 1980 @last  
series dummy=0
```

creates a series named DUMMY that has the value 1 prior to 1980 and the value 0 thereafter.

Redeclaration of an object to a different type is not allowed and will generate an error.

## Auxiliary Commands

*Auxiliary commands* are commands which are either unrelated to a particular object (*i.e.*, not views or procs), or which act on an object in a way that is generally independent of the type or contents of the object. Auxiliary commands typically follow the syntax:

```
command(option_list) argument_list
```

where `command` is the name of the view or procedure to be executed, `option_list` is a list of options separated by commas, and `argument_list` is a list of arguments generally separated by spaces.

An example of an auxiliary command is:

```
store(d=c:\newdata\db1) gdp m x
```

which will store the three objects GDP, M and X in the database named DB1 in the directory C:\NEWDATA.

There is also a set of auxiliary commands which are designed to be used interactively, and sometimes performs operations that create new untitled or unnamed objects (see “[Interactive Use Commands](#)” on page 20). For example, the command:

```
ls y c x1 x2
```

will regress the series Y against a constant term, the series X1 and the series X2, and create a new untitled equation object to hold the results.

Since these commands are designed primarily for interactive use, they will prove useful for carrying out simple tasks. Overuse of these interactive tools, or their use in programs, will make it difficult to manage your work since unnamed objects cannot be referenced by name from within a program, cannot be saved to disk, and cannot be deleted except

through the graphical Windows interface. Wherever possible, you should use named rather than untitled objects for your work. For example, we may replace the above auxiliary command with the statement:

```
equation eq1.ls y c x1 x2
```

to create the named equation object EQ1. This example uses declaration of the object EQ1 and the equation method `ls` to perform the same task as the auxiliary command above.

## Managing Workfiles and Databases

There are two types of object containers: *workfiles* and *databases*. All EViews objects must be held in an object container, so before you begin working with objects you must create a workfile or database. Workfiles and databases are described in depth in [Chapter 3, “Workfile Basics”, on page 43](#) and [Chapter 10, “EViews Databases”, beginning on page 253](#) of the *User’s Guide*.

### Managing Workfiles

To declare and create a new workfile, you may use the [`wfcreate` \(p. 503\)](#) command. You may enter the keyword `wfcreate` followed by a name for the workfile, an option for the frequency of the workfile, and the start and end dates. The workfile frequency type options are:

a	annual.
s	semi-annual.
q	quarterly.
m	monthly.
w	weekly.
d	daily (5 day week).
7	daily (7 day week).
u	undated.

For example:

```
wfcreate macrol q 1965Q1 1995Q4
```

creates a new quarterly workfile named MACRO1 from the first quarter of 1965 to the fourth quarter of 1995.

```
wfcreate cps88 u 1 1000
```

creates a new undated workfile named CPS88 with 1000 observations.

Note that if you have multiple open workfiles, the [wfselect \(p. 514\)](#) command may be used to change the active workfile.

Alternately, you may use [wfopen \(p. 504\)](#) to read a foreign data source into a new workfile.

To save your workfile, use the [wfsave \(p. 512\)](#) command by typing the keyword `wfsave` followed by a workfile name. If any part of the path or workfile name has spaces, you should enclose the entire expression in quotation marks. The active workfile will be saved in the default path under the given name. You may optionally provide a path to save the workfile in a different directory:

```
wfsave a:\mywork
```

If necessary, enclose the path name in quotations. To close the workfile, use the [close \(p. 238\)](#) command. For example:

```
close mywork
```

closes the workfile window of MYWORK.

To open a previously saved workfile, use the [wfopen \(p. 504\)](#) command. You should follow the keyword with the name of the workfile. You can optionally include a path designation to open workfiles that are not saved in the default path. For example:

```
wfopen "c:\mywork\proj1"
```

## Managing Databases

To create a new database, follow the [dbccreate \(p. 255\)](#) command keyword with a name for the new database. Alternatively, you could use the [db \(p. 254\)](#) command keyword followed by a name for the new database. The two commands differ only when the named database already exists. If you use `dbccreate` and the named database already exists on disk, EViews will error indicating that the database already exists. If you use `db` and the named database already exists on disk, EViews will simply open the existing database. Note that the newly opened database will become the default database.

For example:

```
dbccreate mydata1
```

creates a new database named MYDATA1 in the default path, opens a new database window, and makes MYDATA1 the default database.

```
db c:\evdata\usdb
```

opens the USDB database in the specified directory if it already exists. If it does not, EViews creates a new database named USDB, opens its window, and makes it the default database.

You can also use [dbopen \(p. 257\)](#) to open an existing database and to make it the default database. For example:

```
dbopen findat
```

opens the database named FINDAT in the default directory. If the database does not exist, EViews will error indicating that the specified database cannot be found.

You may use [dbrename \(p. 259\)](#) to rename an existing database. Follow the dbrename keyword by the current (old) name and a new name:

```
dbrename templ newmacro
```

To delete an existing database, use the [dbdelete \(p. 256\)](#) command. Follow the dbdelete keyword by the name of the database to delete:

```
dbdelete c:\data\usmacro
```

[dbcopy \(p. 255\)](#) makes a copy of the existing database. Follow the dbcopy keyword with the name of the source file and the name of the destination file:

```
dbcopy c:\evdata\macro1 a:\macro1
```

[dbpack \(p. 258\)](#), [dbrepair \(p. 260\)](#), and [dbrebuild \(p. 258\)](#) are database maintenance commands. See also [Chapter 10, “EViews Databases”, beginning on page 253](#) of the *User’s Guide* for a detailed description.

## Managing Objects

In the course of a program you will often need to manage the objects in a workfile by copying, renaming, deleting and storing them to disk. EViews provides a number of auxiliary commands which perform these operations. The following discussion introduces you to the use of these commands; a full description of each command is provided in [Appendix B, “Command Reference”, on page 193](#).

### Copying Objects

You may create a duplicate copy of one or more objects using the [copy \(p. 244\)](#) command. The copy command is an auxiliary command with the format:

```
copy source_name dest_name
```

where `source_name` is the name of the object you wish to duplicate, and `dest_name` is the name you want attached to the new copy of the object.

The `copy` command may also be used to copy objects in databases and to move objects between workfiles and databases.

### Copy with Wildcard Characters

EViews supports the use of wildcard characters (“?” for a single character match and “\*” for a pattern match) in destination specifications when using the `copy` and `rename` commands. Using this feature, you can copy or rename a set of objects whose names share a common pattern in a single operation. This can be useful for managing series produced by model simulations, series corresponding to pool cross-sections, and any other situation where you have a set of objects which share a common naming convention.

A destination wildcard pattern can be used only when a wildcard pattern has been provided for the source, and the destination pattern must always conform to the source pattern in that the number and order of wildcard characters must be exactly the same between the two. For example, the patterns:

Source Pattern	Destination Pattern
x*	y*
*c	b*
x*12?	yz*f?abc

conform to each other. These patterns do not:

Source Pattern	Destination Pattern
a*	b
*x	?y
x*y*	*x*y*

When using wildcards, the destination name is formed by replacing each wildcard in the destination pattern by the characters from the source name that matched the corresponding wildcard in the source pattern. Some examples should make this principle clear:

Source Pattern	Destination Pattern	Source Name	Destination Name
*_base	*_jan	x_base	x_jan
us_*	*	us_gdp	gdp
x?	x?f	x1	x1f
*_*	**f	us_gdp	usgdpf
?*f	?*_	usgdpf	us_gdp

Note, as shown in the second example, that a simple asterisk for the destination pattern does not mean to use the unaltered source name as the destination name. To copy objects between containers preserving the existing name, either repeat the source pattern as the destination pattern,

```
copy x* db1::x*
```

or omit the destination pattern entirely:

```
copy x* db1::
```

If you use wildcard characters in the source name and give a destination name without a wildcard character, EViews will keep overwriting all objects which match the source pattern to the name given as destination.

For additional discussion of wildcards, see [Appendix B, “Wildcards”, on page 921](#) of the *User’s Guide*.

## Renaming Objects

You can give an object a different name using the [rename \(p. 394\)](#) command. The `rename` command has the format:

```
rename source_name dest_name
```

where `source_name` is the original name of the object and `dest_name` is the new name you would like to give to the object.

`rename` can also be used to rename objects in databases.

You may use wildcards when renaming series. The name substitution rules are identical to those described above for `copy`.

## Deleting Objects

Objects may be removed from the workfile using the [delete \(p. 263\)](#) command. The `delete` command has the format:

```
delete name_pattern
```

where `name_pattern` can either be a simple name such as “XYZ”, or a pattern containing the wildcard characters “?” and “\*”, where “?” means to match any one character, and “\*” means to match zero or more characters. When a pattern is provided, all objects in the workfile with names matching the pattern will be deleted. [Appendix B, “Wildcards”, on page 921](#) of the *User’s Guide* describes further the use of wildcards.

`delete` can also be used to remove objects from databases.

## Saving Objects

All named objects will be saved automatically in the workfile when the workfile is saved to disk. You can store and retrieve the current workfile to and from disk using the [wfsave \(p. 512\)](#) and [wfopen \(p. 504\)](#) commands. Unnamed objects will not be saved as part of the workfile.

You can also save objects for later use by storing them in a database. The [store \(p. 465\)](#) command has the format:

```
store(option_list) object1 object2 ...
```

where `object1`, `object2`, ..., are the names of the objects you would like to store in the database. If no options are provided, the series will be stored in the current default database (see [Chapter 10](#) of the *User’s Guide* for a discussion of the default database). You can store objects into a particular database by using the option “`d = db_name`” or by prepending the object name with a database name followed by a double colon “`::`”, such as:

```
store db1::x db2::x
```

## Fetch Objects

You can retrieve objects from a database using the [fetch \(p. 279\)](#) command. The `fetch` command has the same format as the `store` command:

```
fetch(option_list) object1 object2 ...
```

To specify a particular database use the “`d =` ” option or the “`::`” extension as for `store`.

## Basic Command Summary

The following list summarizes the EViews basic commands. The full descriptions of these commands are given in [Appendix B, “Command Reference”, beginning on page 193](#).

A list of views and procedures available for each object may be found in [Appendix A, “Object, View and Procedure Reference”, on page 151](#). Commands for working with matrix

objects are listed in [Chapter 3, “Matrix Language”, on page 23](#), and EViews programming expressions are described in [Chapter 6, “EViews Programming”, beginning on page 83](#).

### Command Actions

- [\*\*do\*\*](#) ..... execute action without opening window ([p. 267](#)).
- [\*\*freeze\*\*](#) ..... create view object ([p. 290](#)).
- [\*\*print\*\*](#) ..... print view ([p. 387](#)).
- [\*\*show\*\*](#) ..... show object window ([p. 445](#)).

### Global Commands

- [\*\*cd\*\*](#) ..... change default directory ([p. 229](#)).
- [\*\*exit\*\*](#) ..... exit the EViews program ([p. 279](#)).
- [\*\*output\*\*](#) ..... redirect printer output ([p. 361](#)).
- [\*\*param\*\*](#) ..... set parameter values ([p. 383](#)).
- [\*\*rndseed\*\*](#) ..... set the seed of the random number generator ([p. 403](#)).
- [\*\*smpl\*\*](#) ..... set current workfile sample ([p. 449](#)).
- [\*\*setconvert\*\*](#) ..... set the default frequency conversion method ([p. 424](#)).

### Object Creation Commands

- [\*\*alpha\*\*](#) ..... alpha series ([p. 201](#)).
- [\*\*coef\*\*](#) ..... coefficient vector ([p. 238](#)).
- [\*\*equation\*\*](#) ..... equation object ([p. 275](#)).
- [\*\*frml\*\*](#) ..... numeric or alpha series object with a formula for auto-updating ([p. 293](#)).
- [\*\*genr\*\*](#) ..... numeric or alpha series object ([p. 295](#)).
- [\*\*graph\*\*](#) ..... graph object—create using a graph command or by merging existing graphs ([p. 303](#)).
- [\*\*group\*\*](#) ..... group object ([p. 304](#)).
- [\*\*link\*\*](#) ..... series or alpha link object ([p. 323](#)).
- [\*\*logl\*\*](#) ..... likelihood object ([p. 328](#)).
- [\*\*matrix\*\*](#) ..... matrix object ([p. 350](#)).
- [\*\*model\*\*](#) ..... model object ([p. 353](#)).
- [\*\*pool\*\*](#) ..... pool object ([p. 386](#)).
- [\*\*rowvector\*\*](#) ..... rowvector object ([p. 404](#)).
- [\*\*sample\*\*](#) ..... sample object ([p. 406](#)).
- [\*\*scalar\*\*](#) ..... scalar object ([p. 410](#)).
- [\*\*series\*\*](#) ..... numeric series ([p. 418](#)).
- [\*\*sspace\*\*](#) ..... sspace object ([p. 457](#)).

**sym** ..... sym object ([p. 471](#)).  
**system** ..... system object ([p. 472](#)).  
**table** ..... table object ([p. 457](#)).  
**text** ..... text object ([p. 482](#)).  
**valmap** ..... valmap object ([p. 499](#)).  
**var** ..... var estimation object ([p. 500](#)).  
**vector** ..... vector object ([p. 499](#)).

## Object Container, Data, and File Commands

**ccopy** ..... copy series from DRI database ([p. 229](#)).  
**cfetch** ..... fetch series from DRI database ([p. 234](#)).  
**clabel** ..... display DRI series description ([p. 237](#)).  
**close** ..... close object, program, or workfile ([p. 238](#)).  
**create** ..... create a new workfile ([p. 250](#)).  
**db** ..... open or create a database ([p. 254](#)).  
**dbcopy** ..... make copy of a database ([p. 255](#)).  
**dbcreate** ..... create a new database ([p. 255](#)).  
**dbdelete** ..... delete a database ([p. 256](#)).  
**dbopen** ..... open a database ([p. 257](#)).  
**dbpack** ..... pack a database ([p. 258](#)).  
**dbrebuild** ..... rebuild a database ([p. 258](#)).  
**dbrename** ..... rename a database ([p. 259](#)).  
**dbrepair** ..... repair a database ([p. 260](#)).  
**driconvert** ..... convert the entire DRI database to an EViews database ([p. 269](#)).  
**expand** ..... expand workfile range ([p. 279](#)).  
**fetch** ..... fetch objects from databases or databank files ([p. 279](#)).  
**hconvert** ..... convert an entire Haver Analytics database to an EViews database  
                  ([p. 305](#)).  
**hfetch** ..... fetch series from a Haver Analytics database ([p. 306](#)).  
**hlabel** ..... obtain label from a Haver Analytics database ([p. 309](#)).  
**load** ..... load a workfile ([p. 327](#)).  
**open** ..... open a program or text (ASCII) file ([p. 356](#)).  
**pageappend** ..... append observations to workfile page ([p. 364](#)).  
**pagecontract** ..... contract workfile page ([p. 366](#)).  
**pagecopy** ..... copy contents of a workfile page ([p. 367](#)).  
**pagecreate** ..... create a workfile page ([p. 369](#)).  
**pagedelete** ..... delete a workfile page ([p. 371](#)).

**pageload** .....load one or more pages into a workfile from a workfile or a foreign data source ([p. 371](#)).

**pagerename** .....rename a workfile page ([p. 372](#)).

**pagesave** .....save page into a workfile or a foreign data source ([p. 373](#)).

**pageselect** .....make specified page active ([p. 374](#)).

**pagestack** .....reshape the workfile page by stacking observations ([p. 375](#)).

**pagestruct** .....apply a workfile structure to the page ([p. 378](#)).

**pageunstack** .....reshape the workfile page by unstacking observations into multiple series ([p. 381](#)).

**range** .....reset the workfile range ([p. 391](#)).

**save** .....save workfile to disk ([p. 407](#)).

**sort** .....sort the workfile ([p. 453](#)).

**unlink** .....break links in series objects ([p. 491](#)).

**wfcreate** .....create a new workfile ([p. 503](#)).

**wfopen** .....open workfile or foreign source data as a workfile ([p. 504](#)).

**wfsave** .....save workfile to disk as a workfile or a foreign data source ([p. 512](#)).

**wfselect** .....change active workfile page ([p. 514](#)).

**workfile** .....create or change active workfile ([p. 517](#)).

**write** .....write series to a disk file ([p. 517](#)).

### Object Utility Commands

**close** .....close window of an object, program, or workfile ([p. 238](#)).

**copy** .....copy objects ([p. 244](#)).

**delete** .....delete objects ([p. 263](#)).

**rename** .....rename object ([p. 394](#)).

### Object Assignment Commands

**data** .....enter data from keyboard ([p. 252](#)).

**frml** .....assign formula for auto-updating to a numeric or alpha series object ([p. 293](#)).

**genr** .....create numeric or alpha series object ([p. 295](#)).

**rndint** .....assign random integer values to object ([p. 528](#)).

**rndseed** .....set random number generator seed ([p. 403](#)).

### Graph Commands

**area** .....area graph ([p. 207](#)).

**bar** .....bar graph ([p. 215](#)).

**errbar** ..... error bar graph ([p. 276](#)).  
**hilo** ..... high-low(-open-close) graph ([p. 307](#)).  
**line** ..... line-symbol graph ([p. 320](#)).  
**pie** ..... pie chart ([p. 384](#)).  
**scat** ..... scatterplot ([p. 412](#)).  
**spike** ..... spike graph ([p. 455](#)).  
**xy** ..... XY graph with one or more X plotted against one or more Y ([p. 528](#)).  
**xyline** ..... XY line graph ([p. 530](#)).  
**xypair** ..... XY pairs graph ([p. 528](#)).

### Table Commands

**setcell** ..... format and fill in a table cell ([p. 422](#)).  
**setcolwidth** ..... set width of a table column ([p. 423](#)).  
**setline** ..... place a horizontal line in table ([p. 439](#)).

### Programming Commands

**open** ..... open a program file ([p. 356](#)).  
**output** ..... redirects print output to objects or files ([p. 361](#)).  
**poff** ..... turns off automatic printing in programs ([p. 608](#)).  
**pon** ..... turns on automatic printing in programs ([p. 608](#)).  
**program** ..... create a new program ([p. 389](#)).  
**run** ..... read data from a foreign disk file ([p. 405](#)).  
**statusline** ..... open a file ([p. 463](#)).  
**tic** ..... reset the timer ([p. 482](#)).  
**toc** ..... display elapsed time (since timer reset) in seconds ([p. 483](#)).

### Interactive Use Commands

**arch** ..... estimate autoregressive conditional heteroskedasticity (ARCH and GARCH) models ([p. 203](#)).  
**archtest** ..... LM test for the presence of ARCH in the residuals ([p. 206](#)).  
**auto** ..... Breusch-Godfrey serial correlation Lagrange Multiplier (LM) test ([p. 212](#)).  
**binary** ..... binary dependent variable models (includes probit, logit, gompit) models ([p. 217](#)).  
**cause** ..... pairwise Granger causality tests ([p. 228](#)).  
**censored** ..... estimate censored and truncated regression (includes tobit) models ([p. 233](#)).

**chow** .....Chow breakpoint and forecast tests for structural change ([p. 236](#)).  
**coint** .....Johansen cointegration test ([p. 240](#)).  
**cor** .....correlation matrix ([p. 246](#)).  
**count** .....count data modeling (includes poisson, negative binomial and quasi-maximum likelihood count models) ([p. 248](#)).  
**cov** .....covariance matrix ([p. 250](#)).  
**cross** .....cross correlogram ([p. 251](#)).  
**fit** .....static forecast from an equation ([p. 285](#)).  
**forecast** .....dynamic forecast from an equation ([p. 287](#)).  
**gmm** .....generalized method of moments estimation ([p. 297](#)).  
**hist** .....histogram and descriptive statistics ([p. 308](#)).  
**hpf** .....Hodrick-Prescott filter ([p. 310](#)).  
**logit** .....logit (binary) estimation ([p. 328](#)).  
**ls** .....linear and nonlinear least squares regression (includes weighted least squares and ARMAX) ([p. 329](#)).  
**ordered** .....ordinal dependent variable models (includes ordered probit, ordered logit, and ordered extreme value models) ([p. 360](#)).  
**probit** .....probit (binary) estimation ([p. 388](#)).  
**reset** .....Ramsey's RESET test for functional form ([p. 397](#)).  
**seas** .....seasonal adjustment for quarterly and monthly time series ([p. 417](#)).  
**smooth** .....exponential smoothing ([p. 447](#)).  
**solve** .....solve a model ([p. 451](#)).  
**stats** .....descriptive statistics ([p. 462](#)).  
**testadd** .....likelihood ratio test for adding variables to equation ([p. 474](#)).  
**testdrop** .....likelihood ratio test for dropping variables from equation ([p. 477](#)).  
**tsls** .....linear and nonlinear two-stage least squares (TSLS) regression models (includes weighted TSLS, and TSLS with ARMA errors) ([p. 487](#)).



# Chapter 3. Matrix Language

---

EViews provides you with tools for working directly with data contained in matrices and vectors. You can use the EViews matrix language to perform calculations that are not available using the built-in views and procedures.

The following objects can be created and manipulated using the matrix command language:

- coef: column vector of coefficients to be used by equation, system, pool, logl, and sspace objects
- matrix: two-dimensional array
- rowvector: row vector
- scalar: scalar
- sym: symmetric matrix (stored in lower triangular form)
- vector: column vector

We term these objects *matrix objects* (despite the fact that some of these objects are not matrices).

## Declaring Matrices

You must declare matrix objects prior to use. Detailed descriptions of declaration statements for the various matrix objects are provided in [Appendix A, “Object, View and Procedure Reference”, on page 151](#).

Briefly, a declaration consists of the object *keyword*, followed either by size information in parentheses and the name to be given to the object, followed (optionally) by an assignment statement. If no assignment is provided, the object will be initialized to have all zero values.

The various matrix objects require different sizing information. A matrix requires the number of rows and the number of columns. A sym requires that you specify a single number representing both the number of rows and the number of columns. A vector, rowvector, or coef declaration can include information about the number of elements. A scalar requires no size information. If size information is not provided, EViews will assume that there is only one element in the object.

For example:

```
matrix(3,10) xdata
```

```
sym(9) moments
vector(11) betas
rowvector(5) xob
```

creates a  $3 \times 10$  matrix XDATA, a symmetric  $9 \times 9$  matrix MOMENTS, an  $11 \times 1$  column vector BETAS, and a  $1 \times 5$  rowvector XOB. All of these objects are initialized to zero.

To change the size of a matrix object, you can repeat the declaration statement. Furthermore, if you use an assignment statement with an existing matrix object, the target will be resized as necessary. For example:

```
sym(10) bigz
matrix zdata
matrix(10,2) zdata
zdata = bigz
```

will first declare ZDATA to be a matrix with a single element, and then redeclare ZDATA to be a  $10 \times 2$  matrix. The assignment statement in the last line will resize ZDATA so that it contains the contents of the  $10 \times 10$  symmetric matrix BIGZ.

## Assigning Matrix Values

There are three ways to assign values to the elements of a matrix: you may assign values to specific matrix elements, you may fill the matrix using a list of values, or you may perform matrix assignment.

### Element assignment

The most basic method of assigning matrix values is to assign a value for a specific row and column element of the matrix. Simply enter the matrix name, followed by the row and column indices, in parentheses, and then an assignment to a scalar value.

For example, suppose we declare the  $2 \times 2$  matrix A:

```
matrix(2,2) a
```

The first command creates and initializes the  $2 \times 2$  matrix A so that it contains all zeros. Then after entering the two commands:

```
a(1,1) = 1
a(2,1) = 4
```

we have

$$A = \begin{bmatrix} 1 & 0 \\ 4 & 0 \end{bmatrix}. \quad (3.1)$$

You can perform a large number of element assignments by placing them inside of programming loops:

```
vector(10) y
matrix (10,10) x
for !i = 1 to 10
    y(!i) = !i
    for !j = 1 to 10
        x(!i,!j) = !i + !j
    next
next
```

Note that the `fill` procedure provides an alternative to using loops for assignment.

## Fill assignment

The second assignment method is to use the `fill` procedure to assign a list of numbers to each element of the matrix in the specified order. By default, the procedure fills the matrix column by column, but you may override this behavior.

You should enter the name of the matrix object, followed by a period, the `fill` keyword, and then a *comma delimited* list of values. For example, the commands:

```
vector(3) v
v1.fill 0.1, 0.2, 0.3
matrix(2,4) x
matrix.fill 1, 2, 3, 4, 5, 6, 7, 8
```

create the matrix objects:

$$V = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}, \quad X = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix} \quad (3.2)$$

If we replace the last line with

```
matrix.fill(b=r) 1,2,3,4,5,6,7,8
```

then X is given by:

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}. \quad (3.3)$$

In some situations, you may wish to repeat the assignment over a list of values. You may use the “l” option to fill the matrix by repeatedly looping through the listed numbers until the matrix elements are exhausted. Thus,

```
matrix(3,3) y  
y.fill(l) 1, 0, -1
```

creates the matrix:

$$Y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad (3.4)$$

See [fill \(p. 282\)](#) for a complete description of the `fill` procedure.

## Matrix assignment

You can copy data from one matrix object into another using assignment statements. To perform an assignment, you should enter the name of the target matrix followed by the equal sign “=”, and then a matrix object expression. The expression on the right-hand side should either be a numerical constant, a matrix object, or an expression that returns a matrix object.

There are a variety of rules for how EViews performs the assignment that depend upon the types of objects involved in the assignment.

### Scalar values on the right-hand side

If there is a scalar on the right-hand side of the assignment, every element of the matrix object is assigned the value of the scalar.

Examples:

```
matrix(5,8) first  
scalar second  
vec(10) third  
first = 5  
second = c(2)  
third = first(3,5)
```

Since declaration statements allow for initialization, you can combine the declaration and assignment statements. Examples:

```
matrix(5,8) first = 5  
scalar second = c(2)  
vec(10) third = first(3,5)
```

### Same object type on right-hand side

If the *source* object on the right is a matrix or vector, and the *target* or *destination* object on the left is of the same type, the target will be resized to have the same dimension as the source, and every source element will be copied. For example:

```
matrix(10,2) zdata = 5  
matrix ydata = zdata  
matrix(10,10) xdata = ydata
```

declares that ZDATA is a  $10 \times 2$  matrix filled with 5's. YDATA is automatically resized to be a  $10 \times 2$  matrix and is filled with the contents of ZDATA.

Note that even though the declaration of XDATA calls for a  $10 \times 10$  matrix, XDATA is a  $10 \times 2$  matrix of 5's. This behavior occurs because the declaration statement above is equivalent to issuing the two commands:

```
matrix(10,10) xdata  
xdata = ydata
```

which will first declare the  $10 \times 10$  matrix XDATA, and then automatically resize it to  $10 \times 2$  when you fill it with the values for YDATA.

The next section discusses assignment statements in the more general case, where you are converting between object types. In some cases, the conversion is automatic; in other cases, EViews provides you with additional tools to perform the conversion.

## Copying Data Between Objects

In addition to the basic assignment statements described in the previous section, EViews provides you with a large set of tools for copying data to and from matrix objects.

At times, you may wish to move data between different types of matrix objects. For example, you may wish to take the data from a vector and put it in a matrix. EViews has a number of built-in rules which make these conversions automatically.

At other times, you may wish to move data between a matrix object and an EViews series or group object. There are a separate set of tools which allow you to convert data across a variety of object types.

### Copying data from matrix objects

Data may be moved between different types of matrix objects using assignments. If possible, EViews will resize the target object so that it contains the same information as the object on the right side of the equation.

The basic rules governing expressions of the form “Y = X” may be summarized as follows:

- Object type of Y does not change.
- The target object Y will, if possible, be resized to match the object X; otherwise, EViews will issue an error. Thus, assigning a vector to a matrix will resize the matrix, but assigning a matrix to a vector will generate an error if the matrix has more than one column.
- The data in X will be copied to Y.

Specific exceptions to the rules given above are:

- If X is a scalar, Y will keep its original size and will be filled with the value of X.
- If X and Y are both vector or rowvector objects, Y will be changed to the same type as X.

[“Summary of Automatic Resizing of Matrix Objects” on page 42](#) contains a complete summary of the conversion rules for matrix objects.

Here are some simple examples illustrating the rules for matrix assignment:

```
vector(3) x
x(1) = 1
x(2) = 2
x(3) = 3
vector y = x
matrix z = x
```

Y is now a 3 element vector because it has the same dimension and values as X. EViews automatically resizes the Z Matrix to conform to the dimensions of X so that Z is now a  $3 \times 1$  matrix containing the contents of X: Z(1,1) = 1, Z(2,1) = 2, Z(3,1) = 3.

Here are some further examples where automatic resizing is allowed:

```
vector(7) y = 2
scalar value = 4
matrix(10,10) w = value
w = y
matrix(2,3) x = 1
rowvector(10) t = 100
x = t
```

W is declared as a  $10 \times 10$  matrix of 4's, but it is then reset to be a  $7 \times 1$  matrix of 2's. X is a  $1 \times 10$  matrix of 100's.

Lastly, consider the commands:

```
vector(7) y = 2
rowvector(12) z = 3
coef(20) beta
y = z
z = beta
```

Y will be a rowvector of length 3, containing the original contents of Z, and Z will be a column vector of length 20 containing the contents of BETA.

There are some cases where EViews will be unable to perform the specified assignment because the resize operation is not defined. For example, suppose that X is a  $2 \times 2$  matrix. Then the assignment statement:

```
vector(7) y = x
```

will result in an error. EViews cannot change Y from a vector to a matrix and there is no way to assign the 4 elements of the matrix X to the vector Y. Other examples of invalid assignment statements involve assigning matrix objects to scalars or sym objects to vector objects.

### Copying data from parts of matrix objects

In addition to the standard rules for conversion of data between objects, EViews provides functions for extracting and assigning parts of matrix objects. Matrix functions are described in greater detail later in this chapter. For now, note that some functions take a matrix object and perhaps other parameters as arguments and return a matrix object.

A comprehensive list of the EViews commands and functions that may be used for matrix object conversion appears in “[Utility Functions and Commands](#)” on page 44. However, a few examples will provide you with a sense of the type of operations that may be performed.

Suppose first that you are interested in copying data from a matrix into a vector. The following commands will copy data from M1 and SYM1 into the vectors V1, V2, V3, and V4.

```
matrix(10, 10) m1
sym(10) sym1
vector v1 = @vec(m1)
vector v2 = @columnextract(m1, 3)
vector v3 = @rowextract(m1, 4)
vector v4 = @columnextract(sym1, 5)
```

The @vec function creates a 100 element vector, V1, from the columns of M1 stacked one on top of another. V2 will be a 10 element vector containing the contents of the third col-

umn of M1 while V3 will be a 10 element vector containing the fourth row of M1. The @vec, @rowextract, and @columnextract functions also work with sym objects. V4 is a 10 element vector containing the fifth column of SYM1.

You can also copy data from one matrix into a smaller matrix using @subextract. For example:

```
matrix(20,20) m1=1  
matrix m2 = @subextract(m1,5,5,10,7)  
matrix m3 = @subextract(m1,5,10)  
matrix m4 = m1
```

M2 is a  $6 \times 3$  matrix containing a submatrix of M1 defined by taking the part of the matrix M1 beginning at row 5 and column 5, and ending at row 10 and column 7. M3 is the  $16 \times 11$  matrix taken from M1 at row 5 and column 10 to the last element of the matrix (row 20 and column 20). In contrast, M4 is defined to be an exact copy of the full  $20 \times 20$  matrix.

Data from a matrix may be copied into another matrix object using the commands colplace, rowplace, and matplace. Consider the commands:

```
matrix(100,5) m1 = 0  
matrix(100,2) m2 = 1  
vector(100) v1 = 3  
rowvector(100) v2 = 4  
matplace(m1,m2,1,3)  
colplace(m1,v1,3)  
rowplace(m1,v2,80)
```

The matplace command places M2 in M1 beginning at row 1 and column 3. V1 is placed in column 3 of M1, while V2 is placed in row 80 of M1.

### Copying data between matrix objects and other objects

The previous sections described techniques for copying data between matrix objects such as vectors, matrices and scalars. In this section, we describe techniques for copying data between matrix objects and other EViews objects such as series and groups.

Keep in mind that there are two primary differences between the ordinary series or group objects and the matrix objects. First, operations involving series and groups use information about the current workfile sample, while matrix objects do not. Second, there are important differences in the handling of missing values (NAs) between the two types of objects.

## Direct Assignment

The easiest method to copy data from series or group objects to a matrix object is to use direct assignment. Place the destination matrix object on the left side of an equal sign, and place the series or group to be converted on the right.

If you use a series object on the right, EViews will only include the observations from the current sample to make the vector. If you place a group object on the right, EViews will create a rectangular matrix, again only using observations from the current sample.

While very convenient, there are two principal limitations of this approach. First, EViews will only include observations in the current sample when copying the data. Second, observations containing missing data (NAs) for a series, or for any series in the group, are not placed in the matrix. Thus, if the current sample contains 20 observations, but the series or group contains missing data, the dimension of the vector or matrix will be less than 20. Below, we provide you with methods which allow you to override the current sample and to retain missing values.

Examples:

```
smpl 1963:3 1993:6
fetch hsf gmpyq
group mygrp hsf gmpyq
vector xvec = gmpyq
matrix xmat = mygrp
```

These statements create the vector XVEC and the two column matrix XMAT containing the non-missing series and group data from 1963:3 to 1993:6. Note that if GMPYQ has a missing value in 1970:01, and HSF contains a missing value in 1980:01, both observations for both series will be excluded from XMAT.

When performing matrix assignment, you may refer to an element of a series, just as you would refer to an element of a vector, by placing an index value in parentheses after the name. An index value *i* refers to the *i*-th element of the series from the beginning of the workfile *range*. For example, if the range of the current annual workfile is 1961 to 1980, the expression GNP(6) refers to the 1966 value of GNP. These series element expressions may be used in assigning specific series values to matrix elements, or to assign matrix values to a specific series element. For example:

```
matrix(5,10) x
series yser = nrnd
x(1,1) = yser(4)
yser(5) = x(2,3)
yser(6) = 4000.2
```

assigns the fourth value of the series YSER to X(1,1), and assigns to the fifth and sixth values of YSER, the X(2,3) value and the scalar value “4000.2”, respectively.

While matrix assignments allow you to refer to elements of series as though they were elements of vectors, you cannot generally use series in place of vectors. Most vector and matrix operations will error if you use a series in place of a vector. For example, you cannot perform a `rowplace` command using a series name.

Furthermore, note that when you are not performing matrix assignment, a series name followed by a number in parentheses will indicate that the lag/lead operator be applied to the entire series. Thus, when used in generating series or in an equation, system, or model specification, GNP(6) refers to the sixth lead of the GNP series. To refer to specific elements of the GNP series in these settings, you should use the `@elem` function.

### Copy using `@convert`

The `@convert` function takes a series or group object and, optionally, a sample object, and returns a vector or rectangular matrix. If no sample is provided, `@convert` will use the workfile sample. The sample determines which series elements are included in the matrix. Example:

```
smp1 61 90
group groupx inv gdp m1
vector v = @convert(gdp)
matrix x = @convert(groupx)
```

X is a  $30 \times 3$  matrix with the first column containing data from INV, the second column from GDP, and the third column from M1.

As with direct assignment, the `@convert` function excludes observations for which the series or any of the series in the group contain missing data. If, in the example above, INV contains missing observations in 1970 and 1980, V would be a 29 element vector while X would be a  $28 \times 3$  matrix. This will cause errors in subsequent operations that require V and X to have a common row dimension.

There are two primary advantages of using `@convert` over direct assignment. First, since `@convert` is a function, it may be used in the middle of a matrix expression. Second, an optional second argument allows you to specify a sample to be used in conversion. For example:

```
sample s1.set 1950 1990
matrix x = @convert(grp,s1)
sym y = @inverse(@inner(@convert(grp,s1)))
```

performs the conversion using the sample defined in S1.

## Copy data between Series and Matrices

EViews also provides three useful commands that perform explicit conversions between series and matrices with control over both the sample, and the handling of NAs.

`stom` (Series TO Matrix) takes a series or group object and copies its data to a vector or matrix using either the current workfile sample, or the optionally specified sample. As with direct assignment, the `stom` command excludes observations for which the series or any of the series in the group contain missing data.

Example:

```
sample smpl_cnvrt.set 1950 1995
smpl 1961 1990
group group1 gnp gdp money
vector(46) vec1
matrix(3,30) mat1
stom(gdp,vec1,smpl_cnvrt)
stom(group1,mat1)
```

While the operation of `stom` is similar to `@convert`, `stom` is a command and cannot be included in a matrix expression. Furthermore, unlike `@convert`, the destination matrix or vector must already exist and have the proper dimension.

`stomna` (Series TO Matrix with NAs) works identically to `stom`, but does not exclude observations for which there are missing values. The elements of the series for the relevant sample will map directly into the target vector or matrix. Thus,

```
smpl 1951 2000
vector(50) gvector
stom(gdp,gvector)
```

will always create a 50 element vector GVECTOR that contains the values of GDP from 1951 to 2000, including observations with NAs.

`mtos` (Matrix TO Series) takes a matrix or vector and copies its data into an existing series or group, using the current workfile sample or a sample that you provide.

Examples:

```
mtos(mat1,group1)
mtos(vec1,resid)
mtos(mat2,group1,smpl1)
```

As with `stom` the destination series or group must already exist and the destination dimension given by the sample must match that of the source vector or matrix.

## Matrix Expressions

A *matrix expression* is an expression which combines matrix objects using mathematical operators or relations, functions, and parentheses. While we discuss matrix functions in great detail below, some examples will demonstrate the relevant issues.

Examples:

```
@inner(@convert(grp,s1))  
mat1*vec1  
@inverse(mat1+mat2)*vec1  
mat1 > mat2
```

EViews uses the following rules to determine the order in which the expression will be evaluated:

- You may nest any number of pairs of parentheses to clarify the order of operations in a matrix expression.
- If you do not use parentheses, the operations are applied in the following order:
  1. Unary negation operator and functions
  2. Multiplication and division operators
  3. Addition and subtraction operators
  4. Comparison operators: “ $>=$ ”, “ $>$ ”, “ $<=$ ”, “ $<$ ”, “ $<>$ ”

Examples:

```
@inverse(mat1+mat2)+@inverse(mat3+mat4)  
vec1*@inverse(mat1+mat2)*@transpose(vec1)
```

In the first example, the matrices MAT1 and MAT2 will be added and then inverted. Similarly the matrices MAT3 and MAT4 are added and then inverted. Finally, the two inverses will be added together. In the second example, EViews first inverts MAT1 + MAT2 and uses the result to calculate a quadratic form with VEC1.

## Matrix Operators

EViews provides standard mathematical operators for matrix objects.

### Negation (-)

The unary minus changes the sign of every element of a matrix object, yielding a matrix or vector of the same dimension. Example:

```
matrix jneg = -jpos
```

### Addition (+)

You can add two matrix objects of the same type and size. The result is a matrix object of the same type and size. Example:

```
matrix(3,4) a
matrix(3,4) b
matrix sum = a + b
```

You can add a square matrix and a sym of the same dimension. The upper triangle of the sym is taken to be equal to the lower triangle. Adding a scalar to a matrix object adds the scalar value to each element of the matrix or vector object.

### Subtraction (-)

The rules for subtraction are the same as the rules for addition. Example:

```
matrix(3,4) a
matrix(3,4) b
matrix dif = a - b
```

Subtracting a scalar object from a matrix object subtracts the scalar value from every element of the matrix object.

### Multiplication (\*)

You can multiply two matrix objects if the number of columns of the first matrix is equal to the number of rows of the second matrix.

Example:

```
matrix(5,9) a
matrix(9,22) b
matrix prod = a * b
```

In this example, PROD will have 5 rows and 22 columns.

One or both of the matrix objects can be a sym. Note that the product of two sym objects is a matrix, not a sym. The @inner function will produce a sym by multiplying a matrix by its own transpose.

You can premultiply a matrix or a sym by a vector if the number of columns of the matrix is the same as the number of elements of the vector. The result is a vector whose dimension is equal to the number of rows of the matrix.

Example:

```
matrix(5,9) mat
vector(9) vec
vector res = mat * vec
```

In this example, RES will have 5 elements.

You can premultiply a rowvector by a matrix or a sym if the number of elements of the rowvector is the same as the number of rows of the matrix. The result is a rowvector whose dimension is equal to the number of columns of the matrix.

Example:

```
rowvector rres
matrix(5,9) mat
rowvector(5) row
rres = row * mat
```

In this example, RRES will have 9 elements.

You can multiply a matrix object by a scalar. Each element of the original matrix is multiplied by the scalar. The result is a matrix object of the same type and dimensions as the original matrix. The scalar can come before or after the matrix object. Examples:

```
matrix prod = 3.14159*orig
matrix xxx = d_mat*7
```

### Division (/)

You can divide a matrix object by a scalar. Example:

```
matrix z = orig/3
```

Each element of the object ORIG will be divided by 3.

### Relational Operators (=, >, >=, <, <=, <>)

Two matrix objects of the same type and size may be compared using the comparison operators (). The result is a scalar logical value. Every pair of corresponding elements is tested, and if any pair fails the test, the value 0 (FALSE) is returned; otherwise, the value 1 (TRUE) is returned.

For example, the commands:

```
if result <> value then
    run crect
endif
```

It is possible for a vector to be not greater than, not less than, and not equal to a second vector. For example:

```
vector(2) v1
vector(2) v2
v1(1) = 1
v1(2) = 2
v2(1) = 2
v2(2) = 1
```

Since the first element of V1 is smaller than the first element of V2, V1 is not greater than V2. Since the second element of V1 is larger than the second element of V2, V1 is not less than V2. The two vectors are not equal.

## Matrix Commands and Functions

EViews provides a number of commands and functions that allow you to work with the contents of your matrix objects. These commands and functions may be divided into roughly four distinct types:

1. Utility Commands and Functions
2. Matrix Algebra Functions
3. Descriptive Statistics Functions
4. Element Functions

The utility commands and functions provide support for creating, manipulating, and assigning values to your matrix objects. We have already seen the `@convert` function and the `stom` command, both of which convert data from series and groups into vectors and matrices.

The matrix algebra functions allow you to perform common matrix algebra manipulations and computations. Among other things, you can use these routines to compute eigenvalues, eigenvectors and determinants of matrices, to invert matrices, to solve linear systems of equations, and to perform singular value decompositions.

The descriptive statistics functions compute summary statistics for the data in the matrix object. You can compute statistics such as the mean, median, minimum, maximum, and variance, over all of the elements in your matrix.

The matrix element functions allow you create a new matrix containing the values of a function evaluated at each element of another matrix object. Most of the functions that are available in series expressions may be applied to matrix objects. You can compute the log-

arithm of every element of a matrix, or the cumulative normal distribution at every element of a vector.

A listing of the commands and functions is included in the matrix summary on [page 44](#). Functions for computing descriptive statistics for data in matrices are discussed in “[Descriptive Statistics](#)” on page 547. Additional details on matrix element computations are provided in “[Matrix Operators](#)” on page 34.

## Functions versus Commands

A *function* generally takes arguments, and always returns a result. Functions are easily identified by the initial “@” character in the function name.

There are two basic ways that you can use a function. First, you may assign the result to an EViews object. This object may then be used in other EViews expressions, providing you access to the result in subsequent calculations. For example:

```
matrix y = @transpose(x)
```

stores the transpose of matrix X in the matrix Y. Since Y is a standard EViews matrix, it may be used in all of the usual expressions.

Second, you may use a function as part of a matrix expression. Since the function result is used *in-line*, it will not be assigned to a named object, and will not be available for further use. For example, the command:

```
scalar z = vec1*@inverse(v1+v2)*@transpose(vec1)
```

uses the results of the @inverse and @transpose functions in forming the scalar expression assigned to Z. These function results will not be available for subsequent computations.

By contrast, a *command* takes object names and expressions as arguments, and operates on the named objects. Commands do not return a value.

Commands, which do not have a leading “@” character, must be issued alone on a line, rather than as part of a matrix expression. For example, to convert a series X to a vector V1, you would enter:

```
stom(x,v1)
```

Because the command does not return any values, it may not be used in a matrix expression.

## NA Handling

As noted above, most of the methods of moving data from series and groups into matrix objects will automatically drop observations containing missing values. It is still possible, however, to encounter matrices which contain missing values.

For example, the automatic NA removal may be overridden using the `stomna` command ([p. 464](#)). Additionally, some of the element operators may generate missing values as a result of standard matrix operations. For example, taking element-by-element logarithms of a matrix using `@log` will generate NAs for all cells containing nonpositive values.

EViews follows two simple rules for handling matrices that contain NAs. For all operators, commands, and functions, *except the descriptive statistics function*, EViews works with the full matrix object, processing NAs as required. For descriptive statistic functions, EViews automatically drops NAs when performing the calculation. These rules imply the following:

- Matrix operators will generate NAs where appropriate. Adding together two matrices that contain NAs will yield a matrix containing NAs in the corresponding cells. Multiplying two matrices will result in a matrix containing NAs in the appropriate rows and columns.
- All matrix algebra functions and commands will generate NAs, since these operations are undefined. For example, the Cholesky factorization of a matrix that contains NAs will contain NAs.
- All utility functions and commands will work as before, with NAs treated like any other value. Copying the contents of a vector into a matrix using `colplace` will place the contents, including NAs, into the target matrix.
- All of the matrix element functions will propagate NAs when appropriate. Taking the absolute value of a matrix will yield a matrix containing absolute values for non-missing cells and NAs for cells that contain NAs.
- The descriptive statistics functions are based upon the non-missing subset of the elements in the matrix. You can always find out how many values were used in the computations by using the `@OBS` function.

## Matrix Views and Procs

The object listing in [Appendix A, “Object, View and Procedure Reference”](#), on page 151 lists the various views and procs for all of the matrix objects.

## Matrix Graph and Statistics Views

All of the matrix objects, with the exception of the scalar object, have windows and views. For example, you may display line and bar graphs for each column of the  $10 \times 5$  matrix Z:

```
z.line  
z.bar(p)
```

Each column will be plotted against the row number of the matrix.

Additionally, you can compute descriptive statistics for each column of a matrix, as well as the correlation and covariance matrix between the columns of the matrix:

```
z.stats  
z.cor  
z.cov
```

EViews performs listwise deletion by column, so that each group of column statistics is computed using the largest possible set of observations.

The full syntax for the commands to display and print these views is listed in the object reference.

## Matrix input and output

EViews provides you with the ability to read and write files directly from matrix objects using the read and write procedures.

You must supply the name of the source file. If you do not include the optional path specification, EViews will look for the file in the default directory. The input specification follows the source file name. Path specifications may point to local or network drives. If the path specification contains a space, you must enclose the entire expression in double quotes “”.

In reading from a file, EViews first fills the matrix with NAs, places the first data element in the “(1,1)” element of the matrix, then continues to read the data by row or by column, depending upon the options set.

The following command reads data into MAT1 from an Excel file CPS88 in the network drive specified in the path directory. The data are read by column, and the upper left data cell is A2.

```
mat1.read(a2,s=sheet3) "\\\net1\dr_1\cps88.xls"
```

To read the same file by row, you should use the “t” option:

```
mat1.read(a2,t,s=sheet3) "\\net1\dr 1\cps88.xls"
```

To write data from a matrix, use the `write` keyword, enter the desired options, then the name of the output file. For example:

```
mat1.write mydt.txt
```

writes the data in MAT1 into the ASCII file MYDT.TXT located in the default directory.

There are many more options for controlling reading and writing of data; [Chapter 5, “Basic Data Handling”, on page 79](#) of the *User’s Guide* provides extensive discussion. See also [read \(p. 391\)](#) and [write \(p. 517\)](#).

## Matrix Operations versus Loop Operations

You can perform matrix operations using element operations and loops instead of the built-in functions and commands. For example, the inner product of two vectors may be computed by evaluating the vectors element-by-element:

```
scalar inprod1 = 0
for !i = 1 to @rows(vec1)
    inprod1 = inprod1 + vec1(!i)*vec2(!i)
next
```

This approach will, however, generally be much slower than using the built-in function:

```
scalar inprod2 = @inner(vec1,vec2)
```

You should use the built-in matrix operators rather than loop operators whenever you can. The matrix operators are always much faster than the equivalent loop operations.

There will be cases when you cannot avoid using loop operations. For example, suppose you wish to subtract the column mean from each element of a matrix. Such a calculation might be useful in constructing a fixed effects regression estimator. First, consider a slow method involving only loops and element operations:

```
matrix(2000,10) x = @convert(mygrp1)
scalar xsum
for !i = 1 to @columns(x)
    xsum = 0
    for !j = 1 to @rows(x)
        xsum = xsum+x(!j,!i)
    next
    xsum = xsum/@rows(x)
    for !j = 1 to @rows(x)
```

```
x(!j,!i) = x(!j,!i)-xsum  
next  
next
```

The loops are used to compute a mean for each column of data in X, and then to subtract the value of the mean from each element of the column. A better and much faster method for subtracting column means uses the built-in operators:

```
matrix x = @convert(mygrp1)  
vector(@rows(x)) xmean  
for !i = 1 to @columns(x)  
    xmean = @mean(@columnextract(x,!i))  
    colplace(x,@columnextract(x,!i)-xmean,!i)  
next
```

This command extracts each column of X, computes the mean, and fills the vector XMEAN with the column mean. You then subtract the mean from the column and place the result back into the appropriate column of X. While you still need to loop over the control variable *i*, you avoid the need to loop over the elements of the columns.

## Summary of Automatic Resizing of Matrix Objects

When you perform a matrix object assignment, EViews will resize, where possible, the destination object to accommodate the contents of the source matrix. This resizing will occur if the destination object type can be modified and sized appropriately and if the values of the destination may be assigned without ambiguity. You can, for example, assign a matrix to a vector and *vice versa*, you can assign a scalar to a matrix, but you cannot assign a matrix to a scalar since the EViews does not allow scalar resizing.

The following table summarizes the rules for resizing of matrix objects as a result of declarations of the form

```
object_type y = x
```

where *object\_type* is an EViews object type, or as the result of an assignment statement for Y after an initial declaration, as in:

```
object_type y  
y = x
```

Each row of the table corresponds to the specified type of the destination object, Y. Each column represents the type and size of the source object, X. Each cell of the table shows the type and size of object that results from the declaration or assignment.

<b>Object type and size for source X</b>		
<b>Object type for Y</b>	coef( $p$ )	matrix( $p, q$ )
coef( $k$ )	coef( $p$ )	<i>invalid</i>
matrix( $n, k$ )	matrix( $p, 1$ )	matrix( $p, q$ )
rowvector( $k$ )	rowvector( $p$ )	<i>invalid</i>
scalar	<i>invalid</i>	<i>invalid</i>
sym( $k$ )	<i>invalid</i>	sym( $p$ ) if $p = q$
vector( $n$ )	vector( $p$ )	<i>invalid</i>

<b>Object type and size for source X</b>		
<b>Object type for Y</b>	rowvector( $q$ )	scalar
coef( $k$ )	coef( $q$ )	coef( $k$ )
matrix( $n, k$ )	matrix( $1, q$ )	matrix( $n, k$ )
rowvector( $k$ )	rowvector( $q$ )	rowvector( $k$ )
scalar	<i>invalid</i>	scalar
sym( $k$ )	<i>invalid</i>	<i>invalid</i>
vector( $n$ )	rowvector( $q$ )	vector( $n$ )

<b>Object type and size for source X</b>		
<b>Object type for Y</b>	sym( $p$ )	vector( $p$ )
coef( $k$ )	<i>invalid</i>	coef( $p$ )
matrix( $n, k$ )	matrix( $p, p$ )	matrix( $p, 1$ )
rowvector( $k$ )	<i>invalid</i>	vector( $p$ )
scalar	<i>invalid</i>	<i>invalid</i>
sym( $k$ )	sym( $p$ )	<i>invalid</i>
vector( $n$ )	<i>invalid</i>	vector( $p$ )

For example, consider the command

```
matrix(500, 4) y = x
```

where X is a coef of size 50. The object type is given by examining the table entry corresponding to row “matrix Y” ( $n = 500, k = 4$ ), and column “coef X” ( $p = 50$ ). The entry reads “matrix( $p, 1$ )”, so that the result Y is a  $50 \times 1$  matrix.

Similarly, the command:

```
vector(30) y = x
```

where X is a 10 element rowvector, yields the 10 element rowvector Y. In essence, EViews first creates the 30 element rowvector Y, then resizes it to match the size of X, then finally assigns the values of X to the corresponding elements of Y.

## Matrix Function and Command Summary

### Utility Functions and Commands

- colplace** ..... Places column vector into matrix ([p. 581](#)).
- @columnextract**... Extracts column from matrix ([p. 582](#)).
- @columns** ..... Number of columns in matrix object ([p. 582](#)).
- @convert** ..... Converts series or group to a vector or matrix after removing NAs ([p. 583](#)).
- @explode**..... Creates square matrix from a sym ([p. 586](#)).
- @filledmatrix**..... Creates matrix filled with scalar value ([p. 586](#)).
- @filledrowvector**.. Creates rowvector filled with scalar value ([p. 587](#)).
- @filledsym** ..... Creates sym filled with scalar value ([p. 587](#)).
- @filledvector** ..... Creates vector filled with scalar value ([p. 587](#)).
- @getmaindiagonal** Extracts main diagonal from matrix ([p. 588](#)).
- @identity** ..... Creates identity matrix ([p. 588](#)).
- @implode** ..... Creates sym from lower triangle of square matrix ([p. 588](#)).
- @makediagonal** ... Creates a square matrix with ones down a specified diagonal and zeros elsewhere ([p. 591](#)).
- matplace** ..... Places matrix object in another matrix object ([p. 592](#)).
- mtos** ..... Converts a matrix object to series or group ([p. 592](#)).
- @permute**..... Permutes the rows of the matrix ([p. 594](#)).
- @resample**..... Randomly draws from the rows of the matrix ([p. 595](#)).
- @rowextract**..... Extracts rowvector from matrix object ([p. 595](#)).
- rowplace** ..... Places a rowvector in matrix object ([p. 596](#)).
- @rows** ..... Returns the number of rows in matrix object ([p. 596](#)).
- stom** ..... Converts series or group to vector or matrix after removing observations with NAs ([p. 597](#)).
- stomna**..... Converts series or group to vector or matrix without removing observations with NAs ([p. 597](#)).
- @subextract**..... Extracts submatrix from matrix object ([p. 598](#)).
- @transpose** ..... Transposes matrix object ([p. 600](#)).

- @unitvector** .....Extracts column from an identity matrix ([p. 600](#)).
- @vec** .....Stacks columns of a matrix object ([p. 600](#)).
- @vech** .....Stacks the lower triangular portion of matrix by column ([p. 601](#)).

## Matrix Algebra Functions

- @cholesky** .....Computes Cholesky factorization ([p. 581](#)).
- @cond** .....Calculates the condition number of a square matrix or sym ([p. 582](#)).
- @det** .....Calculate the determinant of a square matrix or sym ([p. 585](#)).
- @eigenvalues** .....Returns a vector containing the eigenvalues of a sym ([p. 585](#)).
- @eigenvectors** .....Returns a square matrix whose columns contain the eigenvectors of a matrix ([p. 586](#)).
- @inner** .....Computes the inner product of two vectors or series, or the inner product of a matrix object ([p. 589](#)).
- @inverse** .....Returns the inverse of a square matrix object or sym ([p. 590](#)).
- @issingular** .....Returns 1 if the square matrix or sym is singular, and 0 otherwise ([p. 590](#)).
- @kronecker** .....Computes the Kronecker product of two matrix objects ([p. 590](#)).
- @norm** .....Computes the norm of a matrix object or series ([p. 593](#)).
- @outer** .....Computes the outer product of two vectors or series, or the outer product of a matrix object ([p. 593](#)).
- @rank** .....Returns the rank of a matrix object ([p. 594](#)).
- @solvesystem** .....Solves system of linear equations,  $Mx = v$ , for  $x$  ([p. 596](#)).
- @svd** .....Performs singular value decomposition ([p. 599](#)).
- @trace** .....Computes the trace of a square matrix or sym ([p. 599](#)).

## Matrix Descriptive Statistics Functions

- @cor** .....Computes correlation between two vectors, or between the columns of a matrix ([p. 584](#)).
- @cov** .....Computes covariance between two vectors, or between the columns of a matrix ([p. 584](#)).

The remaining descriptive statistics functions that may be used with matrices and vectors are described in “[Descriptive Statistics](#)” on page 547.

## Matrix Element Functions

EViews supports matrix element versions of the following categories of functions:

Category	Matrix Element Support
<a href="#">Basic Mathematical Functions (p. 545)</a>	All, except for “@inv”
<a href="#">Special Functions (p. 550)</a>	All
<a href="#">Trigonometric Functions (p. 553)</a>	All
<a href="#">Statistical Distribution Functions (p. 554)</a>	All

# Chapter 4. Working with Tables

---

There are three types of tables in EViews: tabular views, which are tables used in the display of views of other objects, named table objects, and unnamed table objects. The main portion of this discussion focuses on the use of commands to customize the appearance of named table objects. The latter portion of the chapter describes the set of tools that may be used to customize the display characteristics of spreadsheet views of objects (see “[Customizing Spreadsheet Views](#)” beginning on page 57).

You may use EViews commands to generate custom tables of formatted output from your programs. A *table object* is an object made up of rows and columns of cells, each of which can contain either a number or a string, as well as information used to control formatting for display or printing.

[Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* describes various interactive tools for customizing table views and objects.

## Creating a Table

There are two basic ways to create a table object: by freezing an object view, or by issuing a table declaration.

### Creating Tables from an Object View

You may create a table object from another object, by combining an object view command with the [freeze \(p. 290\)](#) command. Simply follow the `freeze` keyword with an optional name for the table object, and the tabular view to be frozen. For example, since the command

```
grp6.stats
```

displays the statistics view of the group GRP6, the command

```
freeze (mytab) grp6.stats
```

creates and displays a table object MYTAB containing the contents of the previous view.

You should avoid creating unnamed tables when using commands in programs since you will be unable to refer to, or work with the resulting object using commands. If the MYTAB option were omitted in the previous example, EViews would create and display an untitled table object. This table object may be customized interactively, but may not be referred to in programs. You may, of course, assign a name to the table interactively.

Once you have created a named table object, you may use the various table object procs to further customize the appearance of your table. See “[Customizing Tables](#)” beginning on [page 50](#).

## Declaring Tables

To declare a table, indicate the number of rows and columns and provide a valid name. For example:

```
table(10,20) bestres
```

creates a table with 10 rows and 20 columns named BESTRES. You can change the size of a table by declaring it again. Re-declaring the table to a larger size does not destroy the contents of the table; any cells in the new table that existed in the original table will contain their previous values.

Tables are automatically resized when you attempt to fill a table cell outside the table’s current dimensions. This behavior is different from matrices which give an error when an out of range element is accessed.

## Assigning Table Values

You may modify the contents of cells in a table using assignment statements. Each cell of the table can be assigned either a string or a numeric value.

### Assigning Strings

To place a string value into a table cell, follow the table name by a cell location (row and column pair in parentheses), then an equal sign and a string expression.

For example:

```
table bestres
bestres(1,6) = "convergence criterion"
%strvar = "lm test"
bestres(2,6) = %strvar
bestres(2,6) = bestres(2,6) + " with 5 df"
```

creates the table BESTRES and places various string values into cells of the table.

### Assigning Numbers

Numbers can be entered directly into cells, or they can be converted to strings before being placed in the table.

Unless there is a good reason to do otherwise, we recommend that numbers be entered directly into table cells. If entered directly, the number will be displayed according to the

numerical format set for that cell; if the format is changed, the number will be redisplayed according to the new format. If the number is first converted to a string, the number will be frozen in that form and cannot be reformatted.

For example:

```
table tab1
tab1(3,4) = 15.345
tab1(4,2) = 1e-5
!ev = 10
tab1(5,1) = !ev
scalar f = 12345.67
tab1(6,2) = f
```

creates the table TAB1 and assigns numbers to various cells.

## Assignment with Formatting

The [setcell \(p. 422\)](#) command is like direct cell assignment in that it allows you to set the contents of a cell, but `setcell` also allows you to provide a set of formatting options for the cell. If you desire greater control over formatting, or if you wish to alter the format of a cell without altering its contents, you should use the tools outlined in “[Customizing Tables](#)” beginning on page 50.

The `setcell` command takes the following arguments:

- the name of the table
- the row and the column of the cell
- the number or string to be placed in the cell
- (optionally) a justification code or a numerical format code, or both

The justification codes are:

- “c” for centered (default)
- “r” for right-justified
- “l” for left-justified

The numerical format code determines the format with which a number in a cell is displayed; cells containing strings will be unaffected. The format code can either be a positive integer, in which case it specifies the number of digits to be displayed after the decimal point, or a negative integer, in which case it specifies the total number of characters to be used to display the number. These two cases correspond to the **fixed decimal** and **fixed character** fields in the number format dialog.

Note that when using a negative format code, one character is always reserved at the start of a number to indicate its sign, and if the number contains a decimal point, that will also be counted as a character. The remaining characters will be used to display digits. If the number is too large or too small to display in the available space, EViews will attempt to use scientific notation. If there is insufficient space for scientific notation (six characters or less), the cell will contain asterisks to indicate an error.

Some examples of using `setcell`:

```
setcell(tabres, 9, 11, %label)
```

puts the contents of %LABEL into row 9, column 11 of the table TABRES.

```
setcell(big_tabl, 1, 1, %info, "c")
```

inserts the contents of %INFO in BIG\_TAB1(1,1), and displays the cell with centered justification.

```
setcell(tab1, 5, 5, !data)
```

puts the number !DATA into cell (5,5) of table TAB1, with default numerical formatting.

```
setcell(tab1, 5, 6, !data, 4)
```

puts the number !DATA into TAB1, with 4 digits to the right of the decimal point.

```
setcell(tab1, 3, 11, !data, "r", 3)
```

puts the number !DATA into TAB1, right-justified, with 3 digits to the right of the decimal point.

```
setcell(tab1, 4, 2, !data, -7)
```

puts the number in !DATA into TAB1, with 7 characters used for display.

## Customizing Tables

EViews provides considerable control over the appearance of table objects, providing a variety of table procedures allowing you specify row heights and column widths, content formatting, justification, font face, size, and color, cell background color and borders. Cell merging and annotation are also supported.

### Column Width and Row Height

We begin by noting that if the contents of a cell are wider or taller than the display width or height of the cell, part of the cell contents may not be visible. You may use the [setwidth \(p. 444\)](#) and [setheight \(p. 436\)](#) table procedures to change the dimensions of a column or row of table cells.

To change the column widths for a set of columns in a table, use the `setwidth` keyword followed by a column range specification in parentheses, and a desired width.

The column range should be either a single column number or letter (e.g., “5”, “E”), a colon delimited range of columns (from low to high, e.g., “3:5”, “C:E”), or the keyword “@ALL”. The width unit is computed from representative characters in the default font for the current table (the EViews table default font at the time the table was created), and corresponds roughly to a single character. Width values may be non-integer values with resolution up to 1/10 of a unit. The default width value for columns in an unmodified table is 10.

For example, both commands

```
tab1.setWidth(2) 12  
tab1.setWidth(B) 12
```

set the width of column 2 to 12 width units, while the command

```
tab1.setWidth(2:10) 20
```

sets the widths for columns 2 through 10 to 20 width units. To set all of the column widths, use the “@ALL” keyword.

```
tab1.setWidth(@all) 20
```

Similarly, you may specify row heights using the `setheight` keyword, followed by a row specification in parentheses, and a desired row height.

Rows are specified either as a single row number (e.g., “5”), as a colon delimited range of rows (from low to high, e.g., “3:5”), or using the keyword “@ALL”. Row heights are given in height unit values, where height units are in character heights. The character height is given by the font-specific sum of the units above and below the baseline and the leading in the default font for the current table. Height values may be non-integer values with resolution up to 1/10 of a height unit. The default row height value is 1.

For example,

```
tab1.setHeight(2) 1
```

sets the height of row 2 to match the table default font character height, while

```
tab1.setHeight(2) 3.5
```

increases the row height to 3-1/2 character heights.

Similarly, the command:

```
tab1.setHeight(2:7) 1.5
```

sets the heights for rows 2 through 7 to 1-1/2 character heights.

```
tab1.setheight(@all) 2
```

sets all row heights to twice the default height.

Earlier versions of EViews supported the setting of column widths using the [setcol-width \(p. 423\)](#) command. This command, which is provided for backward compatibility, only offers a subset of the capabilities of [setwidht \(p. 444\)](#).

## Cell Formatting

A host of cell characteristics may be set using table procedures. Each procedure is designed to work on individual cells, ranges of cells, or the entire table.

### Content Formatting

Cell content formatting allows you to alter the appearance of the data in a table cell without changing the contents of the cell. Using the table proc [setformat \(p. 432\)](#), you may, for example, instruct EViews to change the format of a number to scientific or fixed decimal, or to display a date number in a different date format. These changes in display format do not alter the cell values.

To format the contents of table cells, simply follow the table name with a period and the `setformat` proc keyword, followed by a cell range specification in parentheses, and then a valid numeric or date format string. The cell range may be specified in a number of ways, including individual cells, cell rectangles, row or column ranges or the entire table. See [setformat \(p. 432\)](#) for a description of cell range specification and numeric and date format string syntax.

For example, to set the format for the fifth column of a matrix to fixed 5-digit precision, you may provide the format specification:

```
tab1.setformat(e) f.5
```

To set a format for the cell in the third row of the fifth column to scientific notation with 5 digits of precision, specify the individual cell, as in:

```
tab1.setformat(3,e) e.5  
tab1.setformat(e3) e.5
```

To specify the format for a rectangle of cells, specify the upper left and lower right cells in the rectangle. The following commands set cells in the same region to show 3-significant digits, with negative numbers in parentheses:

```
tab1.setformat(2,B,10,D) (g.3)  
tab1.setformat(r2c2:r10c4) (g.3)  
tab1.setformat(b2:d10) (g.3)
```

The rectangle of cells is delimited by row 2, column 2, and row 10, column 4.

Alternately you may provide a date format for the table cells. The command:

```
tab1.setformat (@all) "dd/MM/YY HH:MI:SS.SSS"
```

will display numeric values in the entire table using formatted date strings containing days followed by months, years, hours, minutes and seconds, to a resolution of thousandths of a second.

Note that changing the display format of a cell that contains a string will have no effect unless the cell is later changed to contain a numerical value.

### Justification and Indentation

The cell justification and indentation control the position of the table cell contents within the table cell itself.

You may use the [setjust \(p. 438\)](#) proc to position the cell contents in the cell. Simply use the `setjust` keyword, followed by a cell range specification in parentheses, and one or more keywords describing a vertical or horizontal position for the cell contents. You may use the keywords `auto`, `left`, `right`, and `center` to control horizontal positioning, and `top`, `middle`, and `bottom` to control vertical positioning. You may use the `auto` keyword to specify left justification for string cells and right justification for numeric cells.

For example,

```
tab1.setjust (@all) top left
```

sets the justification for all cells in the table to top left, while

```
tab1.setjust (2,B,10,D) center
```

horizontally centers the cell contents in the rectangle from B2 to D10, while leaving the vertical justification unchanged.

In addition, you may use [setindent \(p. 437\)](#) to specify a left or right indentation from the edge of the cell for cells that are left or right justified, respectively. You should use the `setindent` keyword followed by a cell range in parentheses, and an indentation unit, specified in 1/5 of a width unit. Indentation is only relevant for non-center justified cells.

For example:

```
tab1.setjust (2,B,10,D) left  
tab1.indent (2,B,10,D) 2
```

left-justifies, then indents the specified cells by 2/5 of a width unit from the left-hand side of the cell.

Alternatively,

```
tab2.setjust (@all) center
```

```
tab2.indent(@all) 3
```

will set the indentation for all cells in the table to 3/5 of a width unit, but this will have no effect on the center justified cells. If the cells are later modified to be left or right justified, the indentation will be used. If you subsequently issue the command

```
tab2.indent(@all) right
```

the cells will be indented 3/5 of a width unit from the right-hand edges.

## Fonts

You may specify font face and characteristics, and the font color for table cells using the [setfont \(p. 431\)](#) and [settextcolor \(p. 443\)](#) table procs.

The `setfont` proc should be used to set the font face, size, boldface, italic, strikethrough and underline characteristics for table cells. You should provide a cell range specification, and one or more font arguments corresponding to font characteristics that you wish to modify. For example:

```
tab1.setfont(3,B,10,D) "Times New Roman" +u 8pt
```

changes the text in the specified cells to Times New Roman, 8 point, underline. Similarly,

```
tab1.setfont(4,B) -b +i -s
```

adds the italic to and removes boldface and strikethrough from the B4 cell.

To set the color of your text, use `settextcolor` with a cell range specification and a color specification. Color specifications may be provided using the @RGB settings, or using one of the EViews predefined colors keywords:

```
tab1.settextcolor(f2:g10) @rgb(255, 128, 0)
```

```
tab1.settextcolor(f2:g10) orange
```

sets the text color for the specified cells to orange. See [setfillcolor \(p. 429\)](#) for a complete description of color specifications.

## Background and Borders

You may set the background color for cells using the [setfillcolor \(p. 429\)](#) table procedure. Specify the cell range and provide a color specification using @RGB settings or one of the predefined color keywords. The commands:

```
tab1.setfillcolor(R2C3:R3C6) ltgray
```

```
tab1.setfillcolor(2,C,3,F) @rgb(192, 192, 192)
```

both set the background color of the specified cells to light gray.

The [setlines \(p. 440\)](#) table proc may be used to draw borders or lines around specified table cells. If a single cell is specified, you may draw borders around the cell or a double line through the center of the cell. If multiple columns or rows is selected, you may, in addition, add borders between cells.

Follow the name of the table object with a period, the `setlines` keyword, a cell range specification, and one or more line arguments describing the lines and borders you wish to draw. For example:

```
tab1.setlines(b2:d6) +a -h -v
```

first adds all borders (“a”) to the cells in the rectangle defined by B2 and D6, then removes the inner horizontal (“h”), and inner vertical (“v”) borders. The command

```
tab1.setlines(2,b) +o
```

adds borders to the outside (“o”), all four borders, of the B2 cell.

You may also use the [setline \(p. 439\)](#) command to place double horizontal separator lines in the table. Enter the `setlines` keyword, followed by the name of the table, and a row number, both in parentheses. For example,

```
setline(bestres,8)
```

places a separator line in the eighth row of the table BESTRES. To remove the line, you must use the `setlines` proc:

```
bestres.setlines(8) -d
```

removes the double separator lines from all of the cells in the eighth row of the table.

## Cell Annotation and Merging

Each cell in a table object is capable of containing a comment. Cell comments contain text that is hidden until the mouse cursor is placed over the cell containing the comment. Comments are useful for adding notes to a table without changing the appearance of the table.

To add a comment with the [comment \(p. 242\)](#) table proc, follow the name of the table object with a period, a single cell identifier (in parentheses), and the comment text enclosed in double quotes. If no comment text is provided, a previously defined comment will be removed.

To add a comment “hello world” to the cell in the second row, fourth column, you may use the command:

```
tab1.comment(d2) "hello world"
```

To remove the comment simply repeat the command, omitting the text:

```
tab1.comment (d2)
```

In addition, EViews permits you to merge cells horizontally in a table object. To merge together multiple cells in a row or to unmerge previously merged cells, you should use the [setmerge \(p. 441\)](#) table proc. Enter the name of the table object, a period, followed by a cell range describing the cells in a single row that are to be merged.

If the first specified column is less than the last specified column (left specified before right), the cells in the row will be merged left to right, otherwise, the cells will be merged from right to left. The contents of the merged cell will be taken from the first cell in the merged region. If merging from left to right, the leftmost cell contents will be used; if merging from right to left, the rightmost cell contents will be displayed.

For example,

```
tab1.setmerge (a2:d2)
```

merges the cells in row 2, columns 1 to 4, from left to right, while

```
tab2.setmerge (d2:a2)
```

merges the cells in row 2, columns 2 to 5, from right to left. The cell display will use the leftmost cell in the first example, and the rightmost in the second.

If you specify a merge involving previously merged cells, EViews will unmerge all cells within the specified range. We may then unmerge cells by issuing the `setmerge` command using any of the previously merged cells. The command:

```
tab2.setmerge (r2c4)
```

unmerges the previously merged cells.

## Labeling Tables

Tables have a label view to display and edit information such as the graph name, last modified date, and remarks. To modify or view the label information, use the [label \(p. 317\)](#) command:

```
table11.label(r) Results from GMM estimation
```

This command shows the label view, and the “r” option appends the text “Results from GMM estimation” to the remarks field of TABLE11.

To return to the basic table view, use the `table` keyword:

```
table11.table
```

All changes made in label view will be saved with the table.

## Printing Tables

To print a table, use the [print \(p. 387\)](#) command, followed by the table object name. For example:

```
print table11
```

The `print` destination is taken from the EViews global print settings.

## Exporting Tables to Files

You may use the table [save \(p. 407\)](#) procedure to save the table to disk as a CSV (comma separated file), tab-delimited ASCII text, RTF (Rich text format), or HTML file.

You must specify a file name and an optional file type, and may also provide options to specify the cells to be saved, text to be written for NA values, and precision with which numbers should be written. RTF and HTML files also allow you to save the table in a different size than the current display. If a file type is not provided, EViews will write a CSV file.

For example:

```
tab1.save(t=csv, n="NAN") mytable
```

saves TAB1 in the default directory as a CSV file MYTABLE.CSV, with NA values translated to the text “NAN”.

Alternately, the command:

```
tab1.save(r=B2:C10, t=html, s=.5) c:\data\MyTab2
```

saves the specified cells in TAB1 as an HTML file to MYTAB2.HTM in the directory C:\DATA. The table is saved at half of the display size.

## Customizing Spreadsheet Views

A subset of the table procs for customizing table display are also available for customizing spreadsheet views of objects. You may use the [setformat \(p. 432\)](#), [setindent \(p. 437\)](#), [setjust \(p. 438\)](#), and [setwidht \(p. 444\)](#) procs to specify the spreadsheet views of numeric and alpha series objects, series objects in groups, or matrix objects.

Suppose, for example, that you wish to set the format of the spreadsheet view for series SER1. Then the commands:

```
ser1.setformat f.5  
ser1.setjust right center  
ser1.setindent 3
```

```
ser1.setWidth 10  
ser1.sheet
```

sets the spreadsheet display format for SER1 and then displays the view.

Similarly, you may set the characteristics for a matrix object using the commands:

```
mat1.setFormat f.6  
mat1.setWidth 8  
mat1.sheet
```

For group spreadsheet formatting, you must specify a column range specification. For example:

```
group1.setFormat(2) (f.7)  
group1.setWidth(2) 10  
group1.setIndent(b) 6  
group1.sheet
```

set the formats for the second series in the group, then displays the spreadsheet view.

```
group1.setWidth(@all) 10
```

sets the width for all columns in the group spreadsheet to 10.

Note that the group specified formats are used only to display series in the group and are not exported to the underlying series. Thus, if MYSER is the second series in GROUP1, the spreadsheet view of MYSER will use the original series settings, not those specified using the group procs.

## Table Summary

See “[Table](#)” (p. 185) for a full listing of formatting procs that may be used with table objects. See also the individual objects for the procs that are available for formatting spreadsheet views.

# Chapter 5. Working with Graphs

---

EViews provides an extensive set of commands to generate and customize graphs from the command line or using programs. A summary of the graph commands detailed below may be found under “[Graph](#)” (p. 159).

In addition, [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* describes graph customization in detail, focusing on the interactive method of working with graphs.

## Creating a Graph

There are three types of graphs in EViews: graphs that are views of other objects, and named or unnamed graph objects. The commands provided for customizing the appearance of your graphs are available for use with named graph objects. You may use the dialogs interactively to modify the appearance of all types of graphs.

### Displaying graphs as object views

To display a graph as a view of an object, you may form a command using the object name followed by a graph type keyword and any relevant options for that type of graph. You may use any of the following graph type command keywords: `line`, `bar`, `area`, `spike`, `errbar`, `hilo`, `pie`, `scat`, `xy`, `xyline`, or `xypair`.

For example, you may use the following two commands to display graph views of a series and a group:

```
ser2.area(n)  
grp6.xyline(b)
```

The first command plots the series SER2 as an area graph with normalized scaling. The second command provides an XY line graph view of the group GRP6, with the series plotted in pairs.

Alternatively, there are many EViews commands which display specialized statistical graph views of objects. For example, you may display a boxplot view of a group or series, using the [boxplot](#) (p. 219) or [boxplotby](#) (p. 221) procs, respectively (see “[Boxplots](#)” on page 397 of the *User’s Guide* for a detailed explanation of boxplots).

For a group object, the command

```
group01.boxplot(nofarout)
```

displays boxplots of the series in GROUP01, and does not include far outliers. Likewise,

```
income.boxplotby sex race
```

displays boxplots for the INCOME series classified by SEX and RACE.

Similarly, you may display a histogram view of a series using the [hist \(p. 308\)](#) series view command:

```
lwage.hist
```

which computes descriptive statistics and displays a histogram of the LWAGE series.

It is important to note that graph views of objects differ from graph objects in important ways:

- First, graph views of objects may not be customized using commands after they are first created. The graph commands for customizing an existing graph are designed for use only with graph objects.
- Second, while you may use interactive dialogs to customize an existing object's graph view, we caution you that there is no guarantee that the customization will be permanent. In many cases, the customized settings will not be saved with the object and will be discarded when the view changes, or if the object is closed and then reopened. For example, if you display a histogram graph view of a series, change the bar color and close the series, the color will revert to the default when the object is opened.

In contrast, graph objects may be customized extensively after they are created. Any customization of a graph object is permanent, and will be saved with the object.

In the remainder of our discussion, we will focus on the creation and customization of graph objects.

## Creating graph objects from object views

If you wish to create a graph object from another object, you should combine the object view command with the [freeze \(p. 290\)](#) command. Simply follow the `freeze` keyword with an optional name for the graph object, and the object view to be frozen. For example,

```
freeze grp6.xyline(b)
```

creates and displays an unnamed graph object of the GRP6 view showing an XY line graph with the series plotted in pairs. Note that freezing an object view will not necessarily copy the existing custom appearance settings such as line color, axis assignment, *etc.* Be sure to specify any desired options in the freeze command (*e.g.*, “`b`”). It is for this reason that we recommend that you create a graph object before performing extensive customization of a view.

You should avoid creating unnamed graphs when using commands in programs since you will be unable to refer to, or work with the resulting object in a program. Instead, you should tell EViews to create a named object, as in

```
freeze(graph1) grp6.line
```

which creates a graph object GRAPH1 containing a line graph of the data in GRP6. Note that using the `freeze` command with a name for the graph will create the graph object and store it in the workfile without showing it. Furthermore, since we have frozen a graph type (line) that is different from our current XY line view, existing custom appearance settings will not be copied to the new graph.

Once you have created a named graph object, you may use the various graph object procs to further customize the appearance of your graph. See “[Customizing a Graph](#)” beginning on page 64.

## Creating named graph objects

There are three direct methods for creating a named graph object. First, you may use the [freeze \(p. 290\)](#) command as described in “[Creating graph objects from object views](#)” on page 60. Alternatively, you may declare a graph object using the [graph \(p. 303\)](#) command. The `graph` command may be used to create graph objects with a specific graph type or to merge existing graph objects.

### Specifying a graph by type

To specify a graph by type you should use the `graph` keyword, followed by a name for the graph, the type of graph you wish to create, and a list of series (see “[Graph Type Commands](#)” on page 159 for a list of types). If a type is not specified, a line graph will be created.

For example,

```
graph gr1 ser1 ser2  
graph gr2.line ser1 ser2
```

creates graph objects GR1 and GR2 containing the line graph view of SER1 and SER2.

Similarly,

```
graph gr3.xyline group3
```

creates a graph object GR3 containing the XY line graph view of the series in GROUP3.

Each graph type provides additional options, which may be included when declaring the graph. Among the most important options are those for controlling scaling or graph type.

The scaling options include:

- Automatic scaling (“a”), in which series are graphed using the default single scale. The default is left scale for most graphs, or left and bottom for XY graphs.
- Dual scaling without crossing (“d”) scales the first series on the left and all other series on the right. The left and right scales will not overlap.
- Dual scaling with possible crossing (“x”) is the same as the “d” option, but will allow the left and right scales to overlap.
- Normalized scaling (“n”), scales using zero mean and unit standard deviation.

For example, the commands

```
graph g1.xyline(d) unemp gdp inv  
show g1
```

create and display an XY line graph of the specified series with dual scales and no crossing.

The graph type options include:

- Stacked graph (“s”) plots the cumulative addition of the series, so the value of a series is the difference between the lines, bars, or areas.
- Mixed graph (“l”) creates a single graph in which the first series is the selected graph type (bar, area, or spike) and all remaining series are line graphs.
- Multiple graph (“m”) plots each series in a separate graph within an object view.
- Pairing (“b”), for XY graphs: By default, the first series will be plotted on the horizontal axis and all remaining series will be plotted on the vertical axis. Using the pairing option, you may choose to plot the data in pairs, where the first two series are plotted against each other, the second two series are plotted against each other, and so forth.

For example, the commands

```
group grp1 sales1 sales2  
graph grsales.bar(s) grp1  
show grsales
```

create a group GRP1 containing the series SALES1 and SALES2, then create and display a stacked bar graph GRSALES of the series in the group.

You should consult the command reference entry for each graph type for additional information, including a list of the available options (e.g., see [bar \(p. 215\)](#) for complete details on bar graphs).

## Merging graph objects

The `graph` command may also be used to merge existing named graph objects into a named multiple graph object. For example,

```
graph gr2.merge gr1 grsales
```

creates a multiple graph object GR2, combining two graph objects previously created.

## Creating unnamed graph objects

There are two ways of creating an unnamed graph object. First, you may use the `freeze` (p. 290) command as described in “[Creating graph objects from object views](#)” on page 60.

Alternatively, you may use any of the graph type keywords as a command. Follow the keyword with any available options for that type, and a list of the objects to graph. EViews will create an unnamed graph of the specified type that is not stored in the workfile. For instance,

```
line(x) ser1 ser2 ser3
```

creates a line graph with series SER1 scaled on the left axis and series SER2 and SER3 scaled on the right axis.

If you later decide to name this graph, you may do so interactively by clicking on the **Name** button in the graph button bar. Alternatively, EViews will prompt you to name or delete any unnamed objects before closing the workfile.

Note that there is no way to name an unnamed graph object in a program. We recommend that you avoid creating unnamed graphs in programs since you will be unable to use the resulting object.

## Changing Graph Types

You may change the graph type of a named graph object by following the object name with the desired graph type keyword and any options for that type. For example,

```
grsales.bar(1)
```

converts the bar graph GRSALES, created above, into a mixed bar-line graph, where SALES1 is plotted as a bar graph and SALES2 is plotted as a line graph within a single graph.

Note that special graphs, such as boxplots and histograms, do not allow you to change the graph type.

Graph options are preserved when changing graph types. This includes attributes such as line color and axis assignment, as well as objects added to the graph, such as text labels,

lines and shading. Commands to modify the appearance of named graph objects are described in “[Customizing a Graph](#)” on page 64.

Note, however, that the line and fill graph settings are set independently. Line attributes apply to line and spike graphs, while fill attributes apply to bar, area, and pie graphs. For example, if you have modified the color of a line in a spike graph, this color will not be used for the fill area if the graph is changed to an area graph.

## Customizing a Graph

EViews provides a wide range of tools for customizing the appearance of a named graph object. Nearly every display characteristic of the graph may be modified, including the appearance of lines and filled areas, legend characteristics and placement, frame size and attributes, and axis settings. In addition, you may add text labels, lines, and shading to the graph.

You may modify the appearance of a graph using dialogs or via the set of commands described below. Note that the commands are only available for graph objects since they take the form of graph procedures.

### Line characteristics

For each data line in a graph, you are able to modify color, width, pattern and symbol using the [setelem](#) (p. 426) command. Follow the command keyword with an integer representing the data element in the graph you would like to modify, and one or more keywords for the characteristic you wish to change. The list of symbol and pattern keywords supported is provided in [setelem](#) (p. 426). For a list of available color keywords and RGB settings, see the related command [setfillcolor](#) (p. 429).

To modify line color and width you should use the `lcolor` and `lwidth` keywords:

```
graph gr1.line ser1 ser2 ser3  
gr1.setelem(3) lcolor(orange) lwidth(2)  
gr1.setelem(3) lcolor(255, 128, 0) lwidth(2)
```

The first command creates a line graph GR1 with colors and widths taken from the global defaults, while the latter two commands equivalently change the graph element for the third series to an orange line 2 points wide.

Each data line in a graph may be drawn with a line, symbols, or both line and symbols. The drawing default is given by the global options, but you may elect to add lines or symbols using the `lpattern` or `symbol` keywords.

To add circular symbols to the line for element 3, you may enter

```
gr1.setelem(3) symbol(circle)
```

Note that this operation modifies the existing options for the symbols, but that the line type, color and width settings from the original graph will remain. To return to line only or symbol only in a graph in which both lines and symbols are displayed, you may turn off either symbols or patterns, respectively, by using the “none” type:

```
gr1.setelem(3) lpat(none)
```

or

```
gr1.setelem(3) symbol(none)
```

The first example removes the line from the drawing for the third series, so only the circular symbol is used. The second example removes the symbol, so only the line is used.

If you attempt to remove the lines or symbols from a graph element that contains only lines or symbols, respectively, the graph will change to show the opposite type. For example,

```
gr1.setelem(3) lpat(dash2) symbol(circle)  
gr1.setelem(3) symbol(none)  
gr1.setelem(3) lpat(none)
```

initially represents element 3 with both lines and symbols, then turns off symbols for element 3 so that it is displayed as lines only, and finally shows element 3 as symbols only, since the final command turns off lines in a line-only graph.

The examples above describe customization of the basic elements common to most graph types. [“Modifying Boxplots” on page 78](#) provides additional discussion of setelem options for customizing boxplot data elements.

## Use of color with lines and filled areas

By default, EViews automatically formats graphs to accommodate output in either color or black and white. When a graph is sent to a printer or saved to a file in black and white, EViews translates the colored lines and fills seen on the screen into an appropriate black and white representation. The black and white lines are drawn with line patterns, and fills are drawn with gray shading. Thus, the appearance of lines and fills on the screen may differ from what is printed in black and white (this color translation does not apply to boxplots).

You may override this auto choice display method by changing the global defaults for graphs. You may choose, for example, to display all lines and fills as patterns and gray shades, respectively, whether or not the graph uses color. All subsequently created graphs will use the new settings.

Alternatively, if you would like to override the color, line pattern, and fill settings for a given graph object, you may use the [options \(p. 358\)](#) graph proc.

### Color

To change the color setting for an existing graph object, you should use `options` with the `color` keyword. If you wish to turn off color altogether for all lines and filled areas, you should precede the keyword with a negative sign, as in:

```
gr1.options -color
```

To turn on color, you may use the same command with the “-” omitted.

### Lines and patterns

To always display solid lines in your graph, irrespective of the color setting, you should use `options` with the `linesolid` keyword. For example,

```
gr1.options linesolid
```

sets graph GR1 to use solid lines when rendering on the screen in color and when printing, even if the graph is printed in black and white. Note that this setting may make identification of individual lines difficult in a printed black and white graph, unless you change the widths or symbols associated with individual lines (see [“Line characteristics” on page 64](#)).

Conversely, you may use the `linepat` option to use patterned lines regardless of the color setting:

```
gr1.options linepat
```

One advantage of using the `linepat` option is that it allows you to see the pattern types that will be used in black and white printing without turning off color in your graph. For example, using the `setelem` command again, change the line pattern of the second series in GR1 to a dashed line:

```
gr1.setelem(2) lpat(dash1)
```

This command will not change the appearance of the colored lines on the screen if color is turned on and auto choice of line and fill type is set. Thus, the line will remain solid, and the pattern will not be visible until the graph is printed in black and white. To view the corresponding patterns, either turn off color so all lines are drawn as black patterned lines, or use the `linepat` setting to force patterns.

To reset the graph or to override modified global settings so that the graph uses auto choice, you may use the `lineauto` keyword:

```
gr1.options lineauto
```

This setting instructs the graph to use solid lines when drawing in color, and use line patterns and gray shades when drawing in black and white.

Note that regardless of the color or line pattern settings, you may always view the selected line patterns in the **Lines & Symbols** section of the graph options dialog. The dialog can be brought up interactively by double clicking anywhere in the graph.

## Filled area characteristics

You can modify the color, gray shade, and hatch pattern of each filled area in a bar, area, or pie graph.

To modify these settings, use [setelem \(p. 426\)](#), followed by an integer representing the data element in the graph you would like to modify, and a keyword for the characteristic you wish to change. For example, consider the commands:

```
graph mygraph.area(s) series1 series2 series3  
mygraph.setelem(1) fcolor(blue) hatch(fdiagonal) gray(6)  
mygraph.setelem(1) fcolor(0, 0, 255) hatch(fdiagonal) gray(6)
```

The first command creates MYGRAPH, a stacked area graph of SERIES1, SERIES2, and SERIES3. The latter two commands are equivalent, modifying the first series by setting its fill color to blue with a forward diagonal hatch. If MYGRAPH is viewed without color, the area will appear with a hatched gray shade of index 6.

See [setfillcolor \(p. 429\)](#) for a list of available color keywords, and [setelem \(p. 426\)](#) for gray shade indexes and available hatch keywords. Note that changes to gray shades will not be visible in the graph unless color is turned off.

## Using preset lines and fills

For your convenience, EViews provides you with a collection of preset line and fill characteristics. Each line preset defines a color, width, pattern, and symbol for a line, and each fill preset defines a color, gray shade, and hatch pattern for a fill. There are thirty line and thirty fill presets.

The global graph options are initially set to use the EViews preset settings. These global options are used when you first create a graph, providing a different appearance for each line or fill. The first line preset is applied to the first data line, the second preset is applied to the second data line, and so on. If your graph contains more than thirty lines or fills, the presets are simply reused in order.

You may customize the graph defaults in the global **Graph Options** dialog. Your settings will replace the existing EViews defaults, and will be applied to all graphs created in the future.

EViews allows you to use either the original EViews presets, or those you have specified in the global **Graph Options** dialog when setting the characteristics of an existing graph. The keyword `preset` is used to indicate that you should use the set of options from the corresponding EViews preset; the keyword `default` is used to indicate that you should use the set of options from the corresponding global graph element defaults.

For example,

```
mygraph.setelem(2) preset(3)
```

allows the second fill area in MYGRAPH to use the original EViews presets for a third fill area. In current versions of EViews, these settings include a green fill, a medium gray shade of 8, and no hatch.

Alternatively,

```
mygraph.setelem(2) default(3)
```

also changes the second area of MYGRAPH, but uses the third set of user-defined presets. If you have not yet modified your global graph defaults, the two commands will yield identical results.

When using the `preset` or `default` keywords with boxplots, the line color of the specified preset will be applied to all boxes, whiskers, and staples in the graph. See “[Modifying Boxplots](#)” on page 78 for additional information.

## Scaling and axes

There are four separate commands that may be used to modify the axes and scaling characteristics of your graphs:

- First, the [setelem \(p. 426\)](#) command with the `axis` keyword may be used to assign data elements to different axes.
- Second, the [scale \(p. 410\)](#) command modifies the axis on which your data is scaled. This is the vertical scale in most graphs, or the horizontal and vertical scales in XY graphs. You may employ the `scale` command to modify the scaling of the data itself, for example, as when you use a logarithmic scale, or to alter the scaling of the axis, as when you enable dual scaling.
- Third, the [datelabel \(p. 252\)](#) command modifies the labeling of the bottom date/time axis in time plots. Use this command to change the way date labels are formatted or to specify label frequency.

For boxplots, you should instead use the [bplabel \(p. 226\)](#) command to modify the bottom axis labeling. See “[Modifying the boxplot horizontal axis](#)” beginning on [page 80](#) for details.

- Lastly, the [axis \(p. 213\)](#) command can be used to customize the final appearance of any axes in the graph object. Use this command to modify tick marks, change the font size of labels, turn on grid or zero lines, or duplicate axes.

### Assigning data to an axis

In most cases, when a graph is created, all data elements are initially assigned to the left axis. XY graphs differ slightly in that data elements are initially assigned to either the left or bottom axis.

Once a graph is created, individual elements may generally be assigned to either the left or right axis. In XY graphs, you may reassign individual elements to either the left, right, top, or bottom axis, while in boxplots or stacked time/observation graphs all data elements must be assigned to the same vertical axis.

To assign a data element to a different axis, use the `setelem` command with the `axis` keyword. For example, the commands:

```
graph graph02.line ser1 ser2
graph02.setelem(2) axis(right)
```

first create GRAPH02, a line graph of SER1 and SER2, and then turn GRAPH02 into a dual scaled graph by assigning the second data element, SER2, to the right axis.

In this example, GRAPH02 uses the default setting for dual scale graphs by disallowing crossing, so that the left and right scales do not overlap. To allow the scales to overlap, use the `scale` command with the `overlap` keyword, as in

```
graph02.scale overlap
```

The left and right scales now span the entire axes, allowing the data lines to cross. To reverse this action and disallow crossing, use `-overlap`, (the `overlap` keyword preceded by a minus sign, “-”).

For XY graphs without pairing, the first series is generally plotted along the bottom axis, and the remaining series are plotted on the left axis. XY graphs allow more manipulation than time/observation plots, because the top and bottom axes may also be assigned to an element. For example,

```
graph graph03.xyline s1 s2 s3 s4
graph03.setelem(1) axis(top)
graph03.setelem(2) axis(right)
```

first creates an XY line graph GRAPH03 of the series S1, S2, S3, and S4. The first series is then assigned to the top axis, and the second series is moved to the right axis. Note that the graph now uses three axes: top, left, and right.

Note that the element index in the `setelem` command is not necessary for boxplots and stacked time/observation graphs, since all data elements must be assigned to the same vertical axis.

While EViews allows dual scaling for the vertical axes in most graph types, the horizontal axes must use a single scale on either the top or bottom axis. When a new element is moved to or from one of the horizontal axes, EViews will, if necessary, reassign elements as required so that there is a single horizontal scale.

For example, using the graph created above, the command

```
graph03.setelem(3) axis(bottom)
```

moves the third series to the bottom axis, forcing the first series to be reassigned from the top to the left axis. If you then issue the command,

```
graph03.setelem(3) axis(right)
```

EViews will assign the third series to the right axis as directed, with the first (next available element, starting with the first) series taking its place on the horizontal bottom axis. If the first element is subsequently moved to a vertical axis, the second element will take its place on the horizontal axis, and so forth. Note that series will never be reassigned to the right or top axis, so that series that placed on the top or right axis and subsequently reassigned will not be replaced automatically.

For XY graphs with pairing, the same principles apply. However, since the elements are graphed in pairs, there is a set of elements that should be assigned to the same horizontal axis. You can switch which set is assigned to the horizontal using the `axis` keyword. For example,

```
graph graph04.xypair s1 s2 s3 s4  
graph04.setelem(1) axis(left)
```

creates an XY graph that plots the series S1 against S2, and S3 against S4. Usually, the default settings assign the first and third series to the bottom axis, and the second and fourth series to the left axis. The second command line moves the first series (S1) from the bottom to the left axis. Since S1 and S3 are tied to the same axis, the S3 series will also be assigned to the left axis. The second and fourth series (S2 and S4) will take their place on the bottom axis.

## Modifying the data axis

The [scale \(p. 410\)](#) command may be used to change the way data is scaled on an axis. To rescale the data, specify the axis you wish to change and use one of the following keywords: `linear`, `linearzero` (linear with zero included in axis), `log` (logarithmic), `norm` (standardized). For example,

```
graph graph05.line ser1 ser2
graph05.scale(left) log
```

creates a line graph GRAPH05 of the series SER1 and SER2, and changes the left axis scaling method to logarithmic.

The interaction of the data scales (these are the left and right axes for non-XY graphs) can be controlled using `scale` with the `overlap` keyword. The `overlap` keyword controls the overlap of vertical scales, where each scale has at least one series assigned to it. For instance,

```
graph graph06.line s1 s2
graph06.setelem(2) axis(right)
graph06.scale overlap
```

first creates GRAPH06, a line graph of series S1 and S2, and assigns the second series to the right axis. The last command allows the vertical scales to overlap.

The `scale` command may also be used to change or invert the endpoints of the data scale, using the `range` or `invert` keywords:

```
graph05.scale(left) -invert range(minmax)
```

inverts the left scale of GRAPH05 (“-” indicates an inverted scale) and sets its endpoints to the minimum and maximum values of the data.

## Modifying the date/time axis

EViews automatically determines an optimal set of labels for the bottom axis of time plots. If you wish to modify the frequency or date format of the labels, you should use the [date-label \(p. 252\)](#) command. When working with boxplots, you should use [bplabel \(p. 226\)](#) instead as described in “[Modifying the boxplot horizontal axis](#)” beginning on [page 80](#).

To control the number of observations between labels, use `datelabel` with the `interval` keyword to specify a desired step size. The stand-alone step size keywords include: `auto` (use EViews' default method for determining step size), `ends` (label first and last observations), and `all` (label every observation). For example,

```
mygraph.datelabel interval(ends)
```

labels only the endpoints of MYGRAPH. You may also use a step size keyword in conjunction with a step number to further control the labeling. These step size keywords include: `obs` (one observation), `year` (one year), `m` (one month), and `q` (one quarter), where each keyword determines the units of the number specified in the step keyword. For example, to label every ten years, you may specify:

```
mygraph.datelabel interval(year, 10)
```

In addition to specifying the space between labels, you may indicate a specific observation to receive a label. The step increment will then center around this observation. For example,

```
mygraph.datelabel interval(obs, 10, 25)
```

labels every tenth observation, centered around the twenty-fifth observation.

You may also use `datelabel` to modify the format of the dates or change their placement on the axis. Using the `format` or `span` keywords,

```
mygraph02.datelabel format(yy) -span
```

formats the labels so that they display as two digit years, and disables interval spanning. If interval spanning is enabled, labels will be centered between the applicable tick marks. If spanning is disabled, labels are placed directly on the tick marks. For instance, in a plot of monthly data with annual labeling, the labels may be centered over the twelve monthly ticks (spanning enabled) or placed on the annual tick marks (spanning disabled).

### Customizing axis appearance

You may customize the appearance of tick marks, modify label font size, add grid lines, or duplicate axes labeling in your graph using [axis \(p. 213\)](#).

Follow the `axis` keyword with a descriptor of the axis you wish to modify and one or more arguments. For instance, using the `ticksin`, `minor`, and `font` keywords:

```
mygraph.axis(left) ticksin -minor font(10)
```

The left axis of MYGRAPH is now drawn with the tick marks inside the graph, no minor ticks, and a label font size of 10 point.

To add lines to a graph, use the `grid` or `zeroline` keywords:

```
mygraph01.axis(left) -label grid zeroline
```

MYGRAPH01 hides the labels on its left axis, draws horizontal grid lines at the major ticks, and draws a line through zero on the left scale.

In single scale graphs, it is sometimes desirable to display the axis labels on both the left and right hand sides of the graph. The `mirror` keyword may be used to turn on or off the display of duplicate axes. For example,

```
graph graph06.line s1 s2  
graph06.axis mirror
```

creates a line graph with both series assigned to the left axis (the default assignment), then turns on mirroring of the left axis to the right axis of the graph. Note that in the latter command, you need not specify an axis to modify, since mirroring sets both the left and right axes to be the same.

If dual scaling is enabled, mirroring will be overridden. In our example, assigning a data element to the right axis,

```
graph06.setelem(1) axis(right)
```

will override axis mirroring. Note that if element 1 is subsequently reassigned to the left scale, mirroring will again be enabled. To turn off mirroring entirely, simply precede the `mirror` keyword with a minus sign. The command

```
graph06.axis -mirror
```

turns off axis mirroring.

## Customizing the graph frame

The graph frame is used to set the basic graph proportions and display characteristics that are not part of the main portion of the graph.

### Graph size

The graph frame size and proportions may be modified using the [options \(p. 358\)](#) command. Simply specify a width and height using the `size` keyword. For example,

```
testgraph.options size(5,4)
```

resizes the frame of TESTGRAPH to  $5 \times 4$  virtual inches.

### Other frame characteristics

The `options` command can be used to remove the box around the graph frame. Use the `inbox` keyword:

```
testgraph.options -inbox
```

Note that removing the box also removes the yellow background from the graph.

EViews attempts to occupy the entire width of the graph for plotting data. If you wish to indent the data in the graph frame, use the `indent` keyword:

```
testgraph.options indent
```

## Labeling data values

Bar and pie graphs allow you to label the value of your data within the graph. Use the [options \(p. 358\)](#) command with one of the following keywords: `barlabelabove`, `barlabelinside`, or `pielabel`. For example,

```
mybargraph.options barlabelabove
```

places a label above each bar in the graph indicating its data value. Note that the label will be visible only when there is sufficient space in the graph.

## Outlining and spacing filled areas

EViews draws a black outline around each bar or area in a bar or area graph, respectively. To disable the outline, use options with the `outlinebars` or `outlineareas` keyword:

```
mybargraph.options -outlinebars
```

Disabling the outline is useful for graphs whose bars are spaced closely together, enabling you to see the fill color instead of an abundance of black outlines.

EViews attempts to place a space between each bar in a bar graph. This space disappears as the number of bars increases. You may remove the space between bars by using the `barspace` keyword:

```
mybargraph.options -barspace
```

## Modifying the Legend

A legend's location, text, and appearance may be customized. Note that special graph types such as boxplots and histograms use text objects for labeling instead of a legend. These text objects may only be modified interactively by double-clicking on the object to bring up the text edit dialog.

To change the text string of a data element for use in the legend, use the [name \(p. 354\)](#) command:

```
graph graph06.line ser1 ser2  
graph06.name(1) Unemployment  
graph06.name(2) DMR
```

The first line creates a line graph GRAPH06 of the series SER1 and SER2. Initially, the legend shows “SER1” and “SER2”. The second and third command lines change the text in the legend to “Unemployment” and “DMR”.

Note that the `name` command is equivalent to using the [setelem \(p. 426\)](#) command with the `legend` keyword. For instance,

```
graph06.setelem(1) legend(Unemployment)
graph06.setelem(2) legend(DMR)
```

produces the same results.

To remove a label from the legend, you may use `name` without providing a text string:

```
graph06.name(2)
```

removes the second label “DMR” from the legend.

For an XY graph, the `name` command modifies any data elements that appear as axis labels, in addition to legend text. For example,

```
graph xygraph.xy ser1 ser2 ser3 ser4
xygraph.name(1) Age
xygraph.name(2) Height
```

creates an XY graph named XYGRAPH of the four series SER1, SER2, SER3, and SER4. “SER1” appears as a horizontal axis label, while “SER2”, “SER3”, and “SER4” appear in the legend. The second command line changes the horizontal label of the first series to “Age”. The third line changes the second series label in the legend to “Height”.

To modify characteristics of the legend itself, use [legend \(p. 319\)](#). Some of the primary options may be set using the `inbox`, `position` and `columns` keywords. Consider, for example, the commands

```
graph graph07.line s1 s2 s3 s4
graph07.legend -inbox position(botleft) columns(4)
```

The first line creates a line graph of the four series S1, S2, S3, and S4. The second line removes the box around the legend, positions the legend in the bottom left corner of the graph window, and specifies that four columns should be used for the text strings of the legend.

When a graph is created, EViews automatically determines a suitable number of columns for the legend. A graph with four series, such as the one created above, would likely display two columns of two labels each. The `columns` command above, with an argument of four, creates a long and slender legend, with each of the four series in its own column.

You may also use the `legend` command to change the font size or to disable the legend completely:

```
graph07.legend font(10)
graph07.legend -display
```

Note that if the legend is hidden, any changes to the text or position of the legend remain, and will reappear if the legend is displayed again.

## Adding text to the graph

Text strings can be placed anywhere within the graph window. Using the [addtext \(p. 199\)](#) command,

```
graph07.addtext(t) Fig 1: Monthly GDP
```

adds the text “Fig 1: Monthly GDP” to the top of the GRAPH07 window. You can also use specific coordinates to specify the position of the upper left corner of the text. For example,

```
graph08.addtext(.2, .1, x) Figure 1
```

adds the text string “Figure 1” to GRAPH08. The text is placed 0.2 virtual inches in, and 0.1 virtual inches down from the top left corner of the graph frame. The “x” option instructs EViews to place the text inside a box.

An existing text object can be edited interactively by double-clicking on the object to bring up a text edit dialog. The object may be repositioned by specifying new coordinates in the dialog, or by simply dragging the object to its desired location.

## Adding lines and shading

You may wish to highlight or separate specific areas of your graph by adding a line or shaded area to the interior of the graph using the [draw \(p. 267\)](#) command. Specify the type of line or shade option (`line`, `dashline`, or `shade`), which axis it should be attached to (`left`, `right`, `bottom`, `top`) and its position. For example,

```
graph09.draw(line, left) 5.2
```

draws a horizontal line at the value 5.2 on the left axis. Alternately,

```
graph09.draw(shade, left) 4.8 5.6
```

draws a shaded horizontal area bounded by the values 4.8 and 5.6 on the left axis. You can also specify `color` and `line width`:

```
graph09.draw(dashline, bottom, color(blue), width(2)) 1985:1
```

draws a vertical blue dashed line of width two points at the date “1985:1” on the bottom axis. Color may be specified using one or more of the following options: `rgb(n1, n2, n3)`, `color(n1, n2, n3)`, where the arguments correspond to RGB settings, or `color(keyword)`, where `keyword` is one of the predefined color keywords.

## Using graphs as templates

After customizing a graph as described above, you may wish to use your custom settings in another graph. Using a graph template allows you to copy the graph type, line and fill settings, axis scaling, legend attributes, and frame settings of one graph into another. This enables a graph to adopt all characteristics of another graph—everything but the data itself. To copy custom line or fill settings from the global graph options, use the `preset` or `default` keywords of the `setelem` command (as described in “[Using preset lines and fills” on page 67\).](#)

### Modifying an existing graph

To modify a named graph object, use the `template` command:

```
graph10.template customgraph
```

This command copies all the appearance attributes of CUSTOMGRAPH into GRAPH10, including the graph type and axis scaling. Note that this may produce undesirable results if you are copying customized axis settings from a graph whose data does not closely match the target graph's data.

To copy text labels, lines and shading in the template graph in addition to all other option settings, use the “t” option:

```
graph10.template(t) customgraph
```

This command copies any text or shading objects that were added to the template graph using the `addtext` or `draw` commands or the equivalent steps using dialogs. Note that using the “t” option overwrites any existing text and shading objects in the target graph.

If you are using a boxplot as a template for another graph type, or vice versa, note that the graph type and boxplot specific attributes will not be changed. In addition, when the “t” option is used, vertical lines or shaded areas will not be copied between the graphs, since the horizontal scales of the two graphs are different.

### Using a template during graph creation

All graph type commands also provide a template option for use when creating a new graph. For instance,

```
graph mygraph.line(o = customgraph) ser1 ser2
```

creates the graph MYGRAPH of the series SER1 and SER2, using CUSTOMGRAPH as a template. Note that the graph type keyword (`line`, in this case) is irrelevant, as the graph type will be that of the template graph. See “[Modifying an existing graph” on page 77](#), for details on which attributes are copied from a template graph. The “o” option instructs

EViews to copy all but the text, lines, and shading of the template graph. To include these elements in the copy, use the “t” option in place of the “o” option.

When used as a graph procedure, this method is equivalent to the one described above for an existing graph:

```
graph10.template(t) customgraph  
graph10.bar(t = customgraph)
```

These two methods produce the same results. Note, again that the `bar` keyword in the second line is irrelevant.

## Arranging multiple graphs

When you create a multiple graph, EViews automatically arranges the graphs within the graph window. See “[Creating a Graph](#)” on page 59 for information on how to create a multiple graph. You may use either the “m” option during graph creation or the `merge` command.

To change the placement of the graphs, use the [align \(p. 200\)](#) command. Specify the number of columns in which to place the graphs and the horizontal and vertical space between graphs, measured in virtual inches. For example,

```
graph graph11.merge graph01 graph02 graph03  
graph11.align(2, 1, 1.5)
```

creates a multiple graph GRAPH11 of the graphs GRAPH01, GRAPH02, and GRAPH03. By default, the graphs are stacked in one column. The second command realigns the graphs in two columns, with 1 virtual inch between the graphs horizontally and 1.5 virtual inches between the graphs vertically.

## Modifying Boxplots

The appearance of boxplots can be customized using many of the commands described above. A few special cases and additional commands are described below.

### Customizing lines and symbols

As with other graph types, the `setelem` command can be used with boxplots to modify line and symbol attributes, assign the boxes to an axis, and use preset and default settings. To use the [setelem \(p. 426\)](#) command with boxplots, use a box element keyword after the command. For example,

```
boxgraph01.setelem(mean) symbol(filledcircle)
```

changes the means in the boxplot BOXGRAPH01 to filled circles. Note that all boxes within a single graph have the same attributes, and changes to appearance are applied to all boxes. For instance,

```
boxgraph01.setelem(box) lcolor(orange) lpat(dash1) lwidth(2)
```

plots all boxes in BOXGRAPH01 with an orange dashed line of width 2 points. Also note that when shaded confidence intervals are used, a lightened version of the box color will be used for the shading. In this way, the above command also changes the confidence interval shading to a light orange.

Each element in a boxplot is represented by either a line or symbol. EViews will warn you if you attempt to modify an inappropriate option (e.g., modifying the symbol of the box).

### Assigning boxes to an axis

The `setelem` command may also be used to assign the boxes to another axis:

```
boxgraph01.setelem axis(right)
```

Note that since all boxes are assigned to the same axis, the index argument specifying a graph element is not necessary.

### Using preset line colors

During general graph creation, lines and fills take on the characteristics of the user-defined presets. When a boxplot is created, the first user-defined line color is applied to the boxes, whiskers, and staples. Similarly, when you use the `preset` or `default` keywords of the `setelem` command with a boxplot, the line color of the preset is applied to the boxes, whiskers, and staples. See “[Using preset lines and fills](#) on page 67” for a description of presets.

The `preset` and `default` methods work just as they do for other graph types, although only the line color is applied to the graph. For example,

```
boxgraph01.setelem default(3)
```

applies the line color of the third user-defined line preset to the boxes, whiskers, and staples of BOXGRAPH01. Note again that `setelem` does not require an argument specifying an index, since the selected preset will apply to all boxes.

There are a number of `setelem` arguments that do not apply to boxplots. The `fill-color`, `fillgray`, and `fillhatch` option keywords are not available, as there are no custom areas to be filled. The `legend` keyword is also not applicable, as boxplots use axis text labels in place of a legend.

### Hiding boxplot elements

In addition to the `setelem` command, boxplots provide a [`setbpelem` \(p. 421\)](#) command for use in enabling or disabling specific box elements. Any element of the boxplot can be hidden, except the box itself. Use the command with a list of box elements to show or hide. For example,

```
boxgraph01.setbpelem -mean far
```

hides the means and confirms that the far outliers are shown in BOXGRAPH01.

### Modifying box width and confidence intervals

The width of the individual boxes in a boxplot can be drawn in three ways: fixed width over all boxes, proportional to the sample size, or proportional to the square root of the sample size. To specify one of these methods, use the `setbpelem` command with the `width` keyword, and one of the supported types (`fixed`, `rootn`, `n`). For example,

```
boxgraph01.setbpelem width(rootn)
```

draws the boxes in BOXGRAPH01 with widths proportional to the square root of their sample size.

There are three methods for displaying the confidence intervals in boxplots. They may be notched, shaded, or not drawn at all, which you may specify using one of the supported keywords (`notch`, `shade`, `none`). For example,

```
boxgraph01.setbpelem ci(notch)
```

draws the confidence intervals in BOXGRAPH01 as notches.

### Modifying the boxplot horizontal axis

The [bplabel](#) (p. 226) command for boxplots is similar to the `datelabel` command in time plots (see “[Modifying the date/time axis](#)” beginning on page 71). Use it to specify the frequency or alignment of labels on the horizontal axis. Provide the step method you wish to use, and optionally a step or alignment number. For example,

```
boxgraph02.bplabel interval(ends)
```

labels the first and last boxes in BOXGRAPH02. To label every fiftieth box, aligned with the tenth box:

```
boxgraph02.bplabel interval(cust, 50, 10)
```

A boxplot's horizontal scale can be modified further by placing the text labels at an angle, changing the text itself, or selectively hiding any of the labels. Using the `bplabel` command,

```
boxgraph02.bplabel angle(60) label(10, "First") -label(60)
```

sets the labels in BOXGRAPH02 to a 60 degree angle, changes the tenth box label to “First”, and hides the sixtieth label.

## Labeling Graphs

As with all EViews objects, graphs have a label view to display and edit information such as the graph name, last modified date, and remarks. To modify or view the label information, use the [label \(p. 317\)](#) command:

```
graph12.label(r) Data from CPS 1988 March File
```

This command shows the label view, and the “r” option appends the text “Data from CPS 1988 March File” to the remarks field of GRAPH12.

To return to the graph view, use the `graph` keyword:

```
graph12.graph
```

All changes made in label view will be saved when the graph is saved.

## Printing Graphs

A graph may be printed using the [print \(p. 387\)](#) command. For example,

```
print graph11 graph12
```

prints GRAPH11 and GRAPH12 on a single page.

In addition, many graph commands and graph views of objects include a print option. For example, you can create and simultaneously print a line graph GRA1 of SER1 using the “p” option:

```
graph gral.line(p) ser1
```

You should check the individual commands for availability of this option.

## Exporting Graphs to Files

You may use the [save \(p. 407\)](#) proc of a graph object to save the graph as a Windows metafile (.wmf), Enhanced Windows metafile (.emf), or Postscript file (.eps).

You must specify a file name and file type, and may also provide the file height, width, units of measurement, and color use. Postscript files also allow you to save the graph with or without a bounding box and to specify portrait or landscape orientation. For instance,

```
graph11.save(t=postscript, u=cm, w=12, -box) MyGraph1
```

saves GRAPH11 in the default directory as a Postscript file MyGraph1.eps, with a width of 12 cm and no bounding box. The height is determined by holding the aspect ratio of the graph constant. Similarly,

```
graph11.save(t=emf, u=pts, w=300, h=300, -c) c:\data\MyGraph2
```

saves GRAPH11 as an Enhanced Windows metafile MYGRAPH2.EMF. The graph is saved in black and white, and scaled to 300 × 300 points.

## Graph Summary

See “[Graph](#)” (p. 159) for a full listing of procs that may be used to customize graph objects, and for a list of the graph type commands.

# Chapter 6. EViews Programming

---

EViews' programming features allow you to create and store commands in programs that automate repetitive tasks, or generate a record of your research project.

For example, you can write a program with commands that analyze the data from one industry, and then have the program perform the analysis for a number of other industries. You can also create a program containing the commands that take you from the creation of a workfile and reading of raw data, through the calculation of your final results, and construction of presentation graphs and tables.

If you have experience with computer programming, you will find most of the features of the EViews language to be quite familiar. The main novel feature of the EViews programming language is a macro substitution language which allows you to create object names by combining variables that contain portions of names.

## Program Basics

### Creating a Program

A program is not an EViews object within a workfile. It is simply a text file containing EViews commands. To create a new program, click **File/New/Program**. You will see a standard text editing window where you can type in the lines of the program. You may also open the program window by typing `program` in the command window, followed by an optional program name. For example

```
program firstprg
```

opens a program window named FIRSTPRG. Program names should follow standard EViews rules for file names.

A program consists of a one or more lines of text. Since each line of a program corresponds to a single EViews command, simply enter the text for each command and terminate the line by pressing the ENTER key.

If a program line is longer than the current program window, EViews will autowrap the text of the line. Autowrapping alters the appearance of the program line by displaying it on multiple lines, but does not change the contents of the line. While resizing the window will change the autowrap position, the text remains unchanged and is still contained in a single line.

If you wish to have greater control over the appearance of your lines, you can manually break long lines using the ENTER key, and then use the underscore continuation character

“\_” as the last character on the line to join the multiple lines. For example, the three separate lines

```
equation eq1.ls _  
y x c _  
ar(1) ar(2)
```

are equivalent to the single line

```
equation eq1.ls y x c ar(1) ar(2)
```

formed by joining the lines at the continuation character.

## Saving a Program

After you have created and edited your program, you will probably want to save it. Press the **Save** or **SaveAs** button on the program window toolbar. When saved, the program will have the extension .PRG.

## Opening a Program

To load a program previously saved on disk, click on **File/Open/Program...**, navigate to the appropriate directory, and click on the desired name. Alternatively, from the command line, you may type `open` followed by the full program name, including the file extension .PRG. By default, EViews will look for the program in the default directory. If appropriate, include the full path to the file. The entire name should be enclosed in quotations if necessary. For example:

```
open mysp500.prg  
open "c:\mywork is here\evIEWS\myhouse.prg"
```

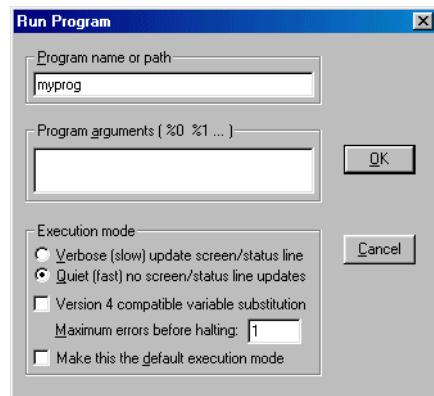
opens MYSP500.PRG in the default directory, and MYHOUSE.PRG in the directory “C:\MYWORK IS HERE\EVIEWS”.

## Executing a Program

When you enter, line by line, a series of commands in the command window, we say that you are working in *interactive mode*. Alternatively, you can type all of the commands in a program and execute or run them collectively as a batch of commands. When you execute or run the commands from a program, we say that you are in *program* (non-interactive) *mode*.

There are several ways to execute a program. The easiest method is to execute your program by pushing the **Run** button on a program window. The Run dialog opens, where you can enter the program name and supply arguments.

You may use the radio buttons to choose between **Verbose** and **Quiet** modes. In verbose mode, EViews sends messages to the status line and continuously updates the workfile window as objects are created and deleted. Quiet mode suppresses these updates, reducing the time spent writing to the screen.



If the checkbox labeled **Version 4 compatible variable substitution** is selected, EViews will use the variable substitution behavior found in EViews 4 and earlier. To support the use of alpha series, EViews 5 has changed the way that % substitution variables are evaluated in expressions. To return to EViews 4 compatible rules for substitution, you may either use this checkbox or include a “**MODE VER4**” statement in your program. See “[Version 5 Compatibility Notes](#)” on page 93 and “[Program Modes](#)” on page 96 for additional discussion.

By default, when EViews encounters an error, it will immediately terminate the program and display a message. If you enter a number into the **Maximum errors before halting** field, EViews will continue to execute the program until the maximum number of errors is reached. If there is a serious error so that it is impractical for EViews to continue, the program will halt even if the maximum number of errors is not reached. See “[Handling Execution Errors](#)” on page 106.

You may also execute a program by entering the `run` command, followed by the name of the program file:

```
run mysp500
```

or

```
run c:\eviews\myprog
```

The use of the .PRG extension is not required since EViews will automatically append one. The default run options described above are taken from the global settings, but may be overridden using command options or program options.

For example, you may use the “v” or “verbose” options to run the program in verbose mode, and the “q” or “quiet” options to run the program in quiet mode. If you include a

number as an option, EViews will use that number to indicate the maximum number of errors encountered before execution is halted. Any arguments to the program may be listed after the filename:

```
run (v, 500) mysp500
```

or

```
run (q) progarg arg1 arg2 arg3
```

Alternatively, you may modify your program to contain program options for quiet and verbose mode, and for setting version compatibility.

You can also have EViews run a program automatically upon startup by choosing **File/Run** from the menu bar of the Windows Program Manager or **Start/Run** in Windows and then typing “eviews”, followed by the name of the program and the values of any arguments. If the program has as its last line the command `exit`, EViews will close following the execution of the program.

See “[Multiple Program Files](#)” on page 107 for additional details on program execution.

## Stopping a Program

Pressing the ESC or F1 keys halts execution of a program. It may take a few seconds for EViews to respond to the halt command.

Programs will also stop when they encounter a `stop` command, when they read the maximum number of errors, or when they finish processing a file that has been executed via a `run` statement.

If you include the `exit` keyword in your program, the EViews application will close.

## Simple Programs

The simplest program is just a list of commands. Execution of the program is equivalent to typing the commands one by one into the command window.

While you could execute the commands by typing them in the command window, you could just as easily open a program window, type in the commands and click on the **Run** button. Entering commands in this way has the advantage that you can save the set of commands for later use, and execute the program repeatedly, making minor modifications each time.

Let us look at a simple example (the data series are provided in the database PROGDEMO in your EViews directory so that you can try out the program). Create a new program by typing

```
program myprog
```

in the command window. In the program window that opens for MYPROG, we are going to enter the commands to create a workfile, fetch a series from an EViews database named PROGDEMO, run a regression, compute residuals and a forecast, make a plot of the forecast, and save the results.

```
' housing analysis
workfile myhouse m 1968m3 1997m6
fetch progdemo::hsf
smpl 1968m5 1992m12
equation reg1.ls hsf c hsf(-1)
reg1.makeresid hsfres
smpl 1993m1 1997m6
reg1.forecast hsffit
freeze(hsfplot) hsffit.line
save
```

The first line of the program is a comment, as denoted by the apostrophe “'”. In executing a program, EViews will ignore all text following the apostrophe until the end of the line.

HSF is total housing units started. We end up with a saved workfile named MYHOUSE containing the HSF series, an equation object REG1, a residual and forecast series, HSFRES and HSFFIT, and a graph HSF PLOT of the forecasts.

You can run this program by clicking on **Run** and filling in the dialog box.

Now, suppose you wish to perform the same analysis, but for the S&P 500 stock price index (FSPCOM). Edit the program, changing MYHOUSE to MYSP500, and change all of the references of HSF to FSPCOM:

```
' s&p analysis
workfile mysp500 m 1968m3 1997m6
fetch progdemo::fspcom
smpl 1968m5 1992m12
equation reg1.ls fspcom c fspcom(-1)
reg1.makeresid fspcomres
smpl 1993m1 1997m6
reg1.forecast fspcomfit
freeze(fscomplot) fspcomfit.line
save
```

Click on **Run** to execute the new analysis. Click on the **Save** button to save your program file as MYPROG.PRG in the EViews directory.

Since most of these two programs are identical, it seems like a lot of typing to make two separate programs. In “[Program Arguments](#)” on page 97 we show you a method for handling these two forecasting problems with a single program. First however, we must define the notion of variables that exist solely when running programs.

## Program Variables

While you can use programs just to edit, run, and re-run collections of EViews commands, the real power of the programming language comes from the use of *program variables* and *program control statements*.

### Control Variables

*Control variables* are variables that you can use in place of numerical values in your EViews programs. Once a control variable is assigned a value, you can use it anywhere in a program that you would normally use a number.

The name of a control variable starts with an “!” mark. After the “!”, the name should be a legal EViews name of 15 characters or fewer. Examples of control variable names are:

```
!x  
!1  
!counter
```

You need not declare control variables before you refer to them, though you must assign them a value before use. Control variables are assigned in the usual way, with the control variable name on the left of an “=” sign and a numerical value or expression on the right. For example:

```
!x = 7  
!12345 = 0  
!counter = 12  
!pi = 3.14159
```

Once assigned a value, a control variable may appear in an expression. For example:

```
!counter = !counter + 1  
genr dnorm = 1/sqr(2*!pi)*exp(-1/2*epsilon^2)  
scalar stdx = x/sqr(!varx)  
smpl 1950q1+!i 1960q4+!i
```

Control variables do not exist outside of your program and are automatically erased after a program finishes. As a result, control variables are not saved when you save the workfile. You can save the values of control variables by creating new EViews objects which contain the values of the control variable.

For example, the following commands:

```
scalar stdx = sqr(!varx)
c(100) = !length
sample years 1960+!z 1990
```

use the numeric values assigned to the control variables !VARX, !LENGTH, and !Z.

## String Variables

A *string expression* or *string* is text enclosed in double quotes:

```
"gross domestic product"
"3.14159"
"ar(1) ar(2) ma(1) ma(2)"
```

A *string variable* is a variable whose value is a string of text. String variables, which may only exist during the time that your program is executing, have names that begin with a "%" symbol. String variables are assigned by putting the string variable name on the left of an "=" sign and a string expression on the right. For example, the following lines assign values to string variables:

```
%value = "value in millions of u.s. dollars"
%armas = "ar(1) ar(2) ma(1) ma(2)"
%mysample = " 83m1 96m12"
%dep = " hs"
%pi = " 3.14159"
```

You may use strings variables to help you build up command text, variable names, or other string values. EViews provides a number of operators and functions for manipulating strings; a complete list is provided in [“Strings” on page 117](#) of the *User’s Guide*.

Once assigned a value, a string variable may appear in any expression in place of the underlying string. When substituted for, the string variable will be replaced by the contents of the string variable, enclosed in double quotes.

Here is a quick example where we use string operations to concatenate the contents of three string variables.

```
!repeat = 500
%st1 = " draws from the normal"
%st2 = "Cauchy "
%st3 = @str(!repeat) + @left(%st1,16) + %st2 + "distribution"
```

In this example %ST3 is set to the value “500 draws from the Cauchy distribution”. Note the spaces before “draws” and after “Cauchy” in the string variable assignments. After string variable substitution, the latter assignment is equivalent to entering

```
%st3 = "500" + " draws from the " + "Cauchy " + "distribution"
```

Similarly, the table assignment statement

```
table1(1,1) = %st3
```

is equivalent to entering the command

```
table(1,1) = "500 draws from the Cauchy distribution"
```

One important usage of string variables is in assigning string values to alpha series. For example, we may have the assignment statement

```
%z = "Ralph"  
alpha full_name = %z + last_name
```

which is equivalent to the expression

```
alpha full_name = "Ralph" + last_name
```

We again emphasize that string variable substitution involves replacing the string variable by its string value contents enclosed in double quotes.

As with any string value, you may convert a string variable containing a number into a number by using the @val function. For example,

```
%str = ".05"  
!level = @val(%str)
```

creates a control variable !LEVEL=0.05. If the first character of the string is not a numeric character (and is not a plus or a minus sign), @val returns the value “NA”. Any characters to the right of the first non-digit character are ignored. For example,

```
%date = "04/23/97"  
scalar day = @val(@right(%date, 5))  
scalar month = @val(%date)
```

creates scalar objects DAY=23 and MONTH=4.

Full details on working with strings are provided in [“Strings” on page 117](#) in the *User’s Guide*.

## Replacement Variables

When working with EViews commands, you may wish to use a string variable, not simply to refer to a string value, but as an indirect way of referring to something else, perhaps a command, or a name, or portion of names for one or more underlying items.

Suppose, for example, that we assign the string variable %X the value “GDP”:

```
%x = "gdp"
```

We may be interested, not in the actual string value “gdp”, but rather in indirectly referring to an underlying object named “GDP”. For example, the series declaration

```
series %x = 300.2
```

will generate an error since the substituted expression is

```
series "gdp" = 300.2
```

and the series declaration `series` expects the *name* of a series, not a string value. In this circumstance, we would like to replace the string variable with the underlying name held in the string.

If you enclose a string variable in curly braces (“{“ and “}”) EViews will replace the expression with the name, names, or name fragment given by the string value. In this context we refer to the expression “{ %X }” as a *replacement variable* since the string variable %X is replaced in the command line by the name or names of objects to which the string refers. For example, the program line

```
equation eq1.ls { %x } c { %x } (-1)
```

would be interpreted by EViews as

```
equation eq1.ls gdp c gdp(-1)
```

Changing the contents of %X to “M1” changes the interpretation of the original line to

```
equation eq1.ls m1 c m1(-1)
```

since the replacement variable uses the name obtained from the new %X.

Similarly, when trying to find the number of valid (non-missing) observations in a series named INCOME, you may use the @obs function along with the name of the series:

```
@obs(income)
```

If you wish to use a string variable %VAR to refer to the INCOME series, you must use the replacement variable in the @OBS function, as in

```
%var = "income"
```

```
@obs({%var})
```

since you wish to refer indirectly to the object named in %VAR. Note that the expression

```
@obs(%var)
```

will return an error since @OBS requires a series or matrix object name as an argument.

Any string variable may be used as the basis of a replacement variable. Simply form your string using one or more string operations

```
%object = "group"  
%space = " "  
%reg1 = "gender"  
%reg2 = "income"  
%reg3 = "age"  
%regs = %reg1 + %space + %reg2 + %space + %reg3
```

then enclose the string variable in braces. In the expression,

```
{%object} g1 {%regs}
```

EViews will substitute the names found in %OBJECT and %REGS so that the resulting command is

```
group g1 gender income age
```

It is worth noting that replacement variables may be used as building blocks to form object names. For example, the commands

```
%b = "2"  
%c = "temp"  
series z{%b}  
matrix(2, 2) x{%b}  
vector(3) x_{%c}_y
```

declare a series named Z2, a  $2 \times 2$  matrix named X2, and a vector named X\_TEMP\_Y.

Up until now we have focused on replacement variables formed from string variables. Note however, that control variables may also be used as replacement variables. For example, the commands

```
!i = 1  
series y{!x} = nrnd  
!j = 0  
series y{!j}{!i} = nrnd
```

are equivalent to

```
series y1 = nrnd  
series y01 = nrnd
```

and will create two series Y1 and Y01 that contain a set of (pseudo-)random draws from a standard normal distribution. Note that in cases where there is no possibility of ambiguity, EViews will treat a control variable as a replacement variable, even if the braces are not provided. For example:

```
!x = 3  
series y!x = 3
```

will generate the series Y3 containing the value 3.

Replacement variables provide you with great flexibility in naming objects in your programs. We caution, however, that unless you take some care, they may cause considerable confusion. We suggest, for example, that you avoid using the same base names to refer to different objects. Consider the following program:

```
' possibly confusing commands (avoid)  
!a = 1  
series x{!a}  
!a = 2  
matrix x{!a}
```

In this small code snippet it is easy to see that X1 is the series and X2 is the matrix. But in a more complicated program, where the control variable assignment !A = 1 may be separated from the declaration by many program lines, it may be difficult to tell at a glance what kind of object X1 represents. A better approach might be to use different names for different kinds of variables:

```
!a = 1  
series ser{!a}  
!a = 2  
matrix mat{!a}
```

so that the replacement variable names are more apparent from casual examination of the program.

## Version 5 Compatibility Notes

While the underlying concepts behind string and replacement variables have not been changed since the first version EViews, beginning in EViews 5 there are two important changes in the implementation of these concepts.

### String vs. Replacement Variables

First, the use of contextual information to distinguish between the use of string and replacement variables has been eliminated.

Previously, the underlying notion that the expression “%X” refers exclusively to the string variable %X while the expression “{ %X }” refers to the corresponding replacement variable was modified slightly to fit the context in which the expression was used. In earlier versions of EViews, the string variable expression “%X” was treated as a string variable in cases where a string was expected, *but was treated as a replacement variable* in other settings.

For example, suppose that we have the string variables:

```
%y = "cons"  
%x = "income"
```

When used in settings where a string is expected, all versions of EViews treat %X and %Y as string variables. Thus, in table assignment, the command,

```
table1(2, 3) = %x + " " + %y
```

is equivalent to the expression,

```
table1(2, 3) = "cons" + " " + "income"
```

However, when string variables were used in other settings, earlier versions of EViews used the context to determine that the string variable should be treated as a replacement variable; for example, the three commands

```
equation eq1.ls %y c %x  
equation eq1.ls { %y } c { %x }  
equation eq1.ls cons c income
```

were all equivalent. Strictly speaking, the first command should have generated an error since string variable substitution would replace %Y with the double-quote delimited string “cons” and %X with the string “income”, as in

```
equation eq1.ls "cons" c "income"
```

Instead, earlier versions of EViews determined that the only valid interpretation of %Y and %X in the first command was as replacement variables so EViews simply substitutes names for %Y and %X.

Similarly, the commands

```
genr %y = %x  
genr { %y } = { %x }
```

```
genr cons = income
```

all yielded the same result, since %Y and %X were treated as replacement variables, not as string variables.

This contextual interpretation of string variables was convenient since, as seen from the examples above, it meant that users rarely needed to use braces around string variables. The EViews 5 introduction of alphanumeric series meant, however, that the existing interpretation of string variables was no longer valid. The following example clearly shows the problem:

```
alpha parent = "mother"  
%x = "parent"  
alpha temp = %x
```

Note that in the final assignment statement, the command context alone is not sufficient to determine whether %X should refer to the string variable value “parent” or to the replacement variable (alpha series) PARENT containing the string “mother”.

In the EViews 5 interpretation of string and replacement variables, users must now always use the expression “{ %X }” to refer to the substitution variable corresponding to %X so that the final line above resolves to the command

```
alpha temp = "parent"
```

To interpret the line as a replacement variable, you must enter

```
alpha temp = { %x }
```

which resolves to the command

```
alpha temp = parent
```

Under an EViews 4 and earlier interpretation of the final line, the braces would not be necessary since “%X” would be treated as a replacement variable.

### String Variables in String Expressions

The second major change in EViews 5 is that all text in a string expression is treated as a literal string. The important implication of this rule is that string variable text is no longer substituted for inside of a string expression.

Consider the assignment statements

```
%b = "mom!"  
%a = "hi %b"  
table(1, 2) = %a
```

In EViews 4 and earlier, the “%B” text in the string expression was treated as a string variable, not as literal text. Accordingly, the string variable %A contains the text “hi mom!”. One consequence of this approach was that there was no way to get the literal text of the form “%B” into a string using a program.

Beginning in EViews 5, the “%B” in the second string variable assignment is treated as literal text. The string variable %A will contain the text “hi %b”. Obtaining a %A that contains the EViews 4 result is straightforward. Simply move the first string variable %B outside of the string expression, and use the string concatenation operator:

```
%a = "hi " + %b
```

assigns the text “hi mom!” to the string variable %A.

### Version 4 Compatibility Mode

While the changes to the handling of string variables are important for extending the programming language to handle the new features of the EViews 5, we recognize that users may have a large library of existing programs which make use of the previous behavior.

Accordingly, EViews 5 provides a version 4 compatibility mode in which you may run EViews programs using the previous context sensitive handling of string and substitution variables, and the previous rules for resolving string variables in string expressions.

There are two ways to ensure that your program is run in version 4 compatibility mode. First, you may specify version 4 compatibility mode at the time the program is run. Compatibility may be set interactively from the **Run Program** dialog ([“Executing a Program” on page 84](#)) by selecting the **Version 4 compatible variable substitution** checkbox, or in a program using the “ver4” option (see [run \(p. 405\)](#)).

Alternatively, you may include “MODE VER4” statement in your program. See [“Program Modes” on page 96](#) for details.

## Program Modes

EViews provides you with the opportunity to set program execution modes at the time that the program is first run. In addition, you may use the “MODE” statement to change the execution mode of a program from within the program itself. One important benefit to using “MODE” statements is that the program can begin executing in one mode, and switch to a second mode as the program executes.

Mode options are available for turning on quiet or verbose mode, or for switching between version 4 and version 5 compatibility.

To change the mode for quiet or verbose mode, simply add a line to your program reading “MODE” followed by either the “QUIET” or the “VERBOSE” keyword, as in

```
mode quiet
```

For version 4 or 5 compatibility, you should use the keywords “VER4” or “VER5”. To use version 4 compatibility mode, you may specify

```
mode ver4
```

as a line in the body of the program.

Multiple settings may be set in a single “MODE =” line:

```
mode quiet ver4
```

and multiple mode statements may be specified in a program to change the mode as the program runs:

```
mode quiet  
[some program lines]  
mode verbose  
[additional program lines]
```

Note that setting the execution mode explicitly in a program overrides any settings specified at the time the program is executed.

## Program Arguments

*Program arguments* are special string variables that are passed to your program when you run the program. Arguments allow you to change the value of string variables every time you run the program. You may use them in any context where a string variable is appropriate. Any number of arguments may be included in a program; they will be named “%0”, “%1”, “%2”, and so on.

When you run a program that takes arguments, you will also supply the values for the arguments. If you use the **Run** button or **File/Run**, you will see a dialog box where you can type in the values of the arguments. If you use the `run` command, you should list the arguments consecutively after the name of the program.

For example, suppose we have a program named REGPROG:

```
equation eq1  
smp1 1980q3 1994q1  
eq1.ls {%-0} c {%-1} {%-1}(-1) time
```

To run REGPROG from the command line with `%0 = “lgdp”` and `%1 = “m1”`, we enter

```
run regprog lgdp m1
```

This program performs a regression of the variable LGDP, on C, M1, M1(-1), and TIME, by executing the command:

```
eq1.ls lgdp c m1 m1(-1) time
```

Alternatively, you can run this program by clicking on the **Run** button on the program window, or selecting **File/Run....** In the Run Program dialog box that appears, type the name of the program in the Program name or path field and enter the values of the arguments in the Program arguments field. For this example, type “regprog” for the name of the program, and “lgdp” and “m1” for the arguments.

Any arguments in your program that are not initialized in the `run` command or **Run Program** dialog are treated as blanks. For example, suppose you have a one-line program named REGRESS:

```
equation eq1.ls y c time {%-0} {%-1} {%-2} {%-3} {%-4} {%-5} {%-6}  
{%-7} {%-8}
```

The command,

```
run regress x x(-1) x(-2)
```

executes

```
equation eq1.ls y c time x x(-1) x(-2)
```

while the command,

```
run regress
```

executes

```
ls y c time
```

In both cases, EViews ignores arguments that are not included in your `run` command.

As a last example, we repeat our simple forecasting program from above, but use arguments to simplify our work. Suppose you have the program, MYPROG:

```
wfcreate {%-0} m 1968m3 1997m6  
fetch progdemo::{%-1}  
smpl 1968m5 1992m12  
equation reg1.ls {%-1} c {%-1}(-1)  
reg1.makeresid {%-1}res  
smpl 1993m1 1997m6  
reg1.forecast {%-1}fit
```

```
freeze({%1}plot) {%1}fit.line  
save
```

The results of running the two example programs at the start of this chapter can be duplicated by running MYPROG with arguments:

```
run myprog myhouse hsf
```

and

```
run myprog mysp500 fspcom
```

## Control of Execution

EViews provides you with several ways to control the execution of commands in your programs. Controlling execution in your program means that you can execute commands selectively or repeat commands under changing conditions. The methods for controlling execution will be familiar from other computer languages.

### IF Statements

There are many situations where you want to execute commands only if some condition is satisfied. EViews uses IF and ENDIF, or IF, ELSE, and ENDIF statements to indicate the condition to be met and the commands to be executed.

An IF statement starts with the `if` keyword, followed by an expression for the condition, and then the word `then`. You may use AND/OR statements in the condition, using parentheses to group parts of the statement as necessary.

All comparisons follow the rules outlined in “[Numeric Relational Operators](#)” beginning on [page 124](#) and “[String Relational Operators](#)” beginning on [page 119](#) of the *User’s Guide*. Note that beginning in EViews 5, all string comparisons in programs are case-sensitive. This is a change from previous versions of EViews. You may perform caseless comparison by using the `@UPPER` or `@LOWER` string functions.

If the expression is TRUE, all of the commands until the matching `endif` are executed. If the expression is FALSE, all of these commands are skipped. The expression to be tested may also take a numerical value. In this case, 0 and NA are equivalent to FALSE and any other non-zero value is TRUE. For example:

```
if !stand=1 or (!rescale=1 and !redo=1) then  
    series gnpstd = gnp/sqr(gvar)  
    series constd = cons/sqr(cvar)  
endif  
if !a>5 and !a<10 then  
    smpl 1950q1 1970q1+!a
```

```
        endif  
        if !scale then  
            series newage = age!/scale  
        endif
```

Note that in this example, all indentation is done for program clarity and has no effect on the execution of the program lines.

An IF statement may have an ELSE clause containing commands to be executed if the condition is FALSE. If the condition is true, all of the commands up to the keyword else will be executed. If the condition is FALSE, all of the commands between else and endif will be executed. For example:

```
if !scale>0 then  
    series newage = age!/scale  
else  
    series newage = age  
endif
```

IF statements may also be applied to string variables:

```
if %0=="CA" or %0=="IN" then  
    series stateid = 1  
else  
    if %0=="MA" then  
        series stateid=2  
    else  
        if %0=="IN" then  
            series stateid=3  
        endif  
    endif  
endif
```

Note that the nesting of our comparisons does not create any difficulties.

You should take care when using the IF statement with series or matrices to note that the comparison is defined on the *entire* object and will evaluate to false unless every element of the element-wise comparison is true. Thus, if X and Y are series, the IF statement

```
if x<>y then  
    [some program lines]  
endif
```

evaluates to false if any element of X is not equal to the corresponding value of Y in the default sample. If X and Y are identically sized vectors or matrices, the comparison is over all of the elements X and Y. This behavior is described in greater detail in “[Relational Operators \(=, >, >=, <, <=, <>\)](#)” on page 36.

## The FOR Loop

The FOR loop allows you to repeat a set of commands for different values of a control or string variable. The FOR loop begins with a `for` statement and ends with a `next` statement. Any number of commands may appear between these two statements.

The syntax of the FOR statement differs depending upon whether it uses control variables or string variables.

### FOR Loops with Control Variables or Scalars

To repeat statements for different values of a control variable, the FOR statement involves setting a control variable equal to an initial value, followed by the word `to`, and then an ending value. After the ending value you may include the word `step` followed by a number indicating by how much to change the control variable each time the loop is executed. If you don't include `step`, the step is assumed to be 1. For example,

```
for !j=1 to 10
    series decile{!j} = (income<level{!j})
next
```

In this example, STEP = 1 and the variable J is twice used as a replacement variable, first for the ten series declarations DECILE1 through DECILE10 and for the ten variables LEVEL1 through LEVEL10.

```
for !j=10 to 1 step -1
    series rescale{!j}=original/{!j}
next
```

In this example, the step is -1, and J is used as a replacement variable to name the ten constructed series RESCALE10 through RESCALE1 and as a scalar in dividing the series ORIGINAL.

The FOR loop is executed first for the initial value, unless that value is already beyond the terminal value. After it is executed for the initial value, the control variable is incremented by `step` and EViews compares the variable to the limit. If the limit is passed, execution stops.

One important use of FOR loops with control variables is to change the sample. If you add a control variable to a date in a `smp1` command, you will get a new date as many observa-

tions forward as the current value of the control variable. Here is a FOR loop that gradually increases the size of the sample and estimates a rolling regression:

```
for !horizon=10 to 72
    smpl 1970m1 1970m1+!horizon
    equation eq{!horizon}.ls sales c orders
next
```

One other important case where you will use loops with control variables is in accessing elements of a series or matrix objects. For example,

```
!rows = @rows(vec1)
vector cumsum1 = vec1
for !i=2 to !rows
    cumsum1(!i) = cumsum1(!i-1) + vec1(!i)
next
```

computes the cumulative sum of the elements in the vector VEC1 and saves it in the vector CUMSUM1.

To access an individual element of a series, you will need to use the @elem function and @otod to get the desired element

```
for !i=2 to !rows
    cumsum1(!i) = @elem(ser1, @otod(!i))
next
```

The @otod function returns the date associated with the observation index (counting from the beginning of the workfile), and the @elem function extracts the series element associated with a given date.

You can nest FOR loops to contain loops within loops. The entire inner FOR loop is executed for each successive value of the outer FOR loop. For example:

```
matrix(25,10) xx
for !i=1 to 25
    for !j=1 to 10
        xx(!i,!j)=(!i-1)*10+!j
    next
next
```

You should avoid changing the control variable within a FOR loop. For example, consider the commands:

```
' potentially confusing loop (avoid doing this)
```

```
for !i=1 to 25
    vector a!i
    !i=!i+10
next
```

Here, both the FOR assignment and the assignment statement within the loop change the value of the control variable I. Loops of this type are difficult to follow and may produce unintended results. If you find a need to change a control variable inside the loop, consider using a WHILE loop as explained below.

You may execute FOR loops with scalars instead of control variables. However, you must first declare the scalar, and you may not use the scalar as a replacement variable. For example,

```
scalar i
scalar sum = 0
vector (10) x
for i=1 to 10
    x(i) = i
    sum = sum + i
next
```

In this example, the scalars I and SUM remain in the workfile after the program has finished running, unless they are explicitly deleted.

### FOR Loops with String Variables

When you wish to repeat statements for different values of a string variable, you can use the FOR loop to let a string variable range over a list of string values. Give the name of the string variable followed by the list of values. For example,

```
for %y gdp gnp ndp nnp
    equation {%y}trend.ls %y c {%y}(-1) time
next
```

executes the commands

```
equation gdptrend.ls gdp c gdp(-1) time
equation gnptrend.ls gnp c gnp(-1) time
equation ndptrend.ls ndp c ndp(-1) time
equation nnptrend.ls nnp c nnp(-1) time
```

You can put multiple string variables in the same FOR statement—EViews will process the strings in sets. For example:

```
for %1 %2 %3 1955q1 1960q4 early 1970q2 1980q3 mid 1975q4
    1995q1 late
    smpl %1 %2
    equation {%3}eq.ls sales c orders
next
```

In this case, the elements of the list are taken in groups of three. The loop is executed three times for the different sample pairs and equation names:

```
smpl 1955q1 1960q4
equation earlyeq.ls sales c orders
smpl 1970q2 1980q3
equation mideq.ls sales c orders
smpl 1975q4 1995q1
equation lateeq.ls sales c orders
```

Note the difference between this construction and nested FOR loops. Here, all string variables are advanced at the same time, whereas with nested loops, the inner variable is advanced over all choices, while the outer variable is held constant. For example:

```
!eqno = 1
for %1 1955q1 1960q4
    for %2 1970q2 1980q3 1975q4
        smpl %1 %2
        'form equation name as eq1 through eq6
        equation eq{!eqno}.ls sales c orders
        !eqno=!eqno+1
    next
next
```

Here, the equations are estimated over the samples 1955Q1–1970Q2 for EQ1, 1955Q1–1980Q3 for EQ2, 1955Q1–1975Q4 for EQ3, 1960Q4–1970Q2 for EQ4, 1960Q4–1980Q3 for EQ5, and 1960Q4–1975Q4 for EQ6.

## The WHILE Loop

In some cases, we wish to repeat a series of commands several times, but only while one or more conditions are satisfied. Like the FOR loop, the WHILE loop allows you to repeat commands, but the WHILE loop provides greater flexibility in specifying the required conditions.

The WHILE loop begins with a `while` statement and ends with a `wend` statement. Any number of commands may appear between the two statements. WHILE loops can be nested.

The WHILE statement consists of the `while` keyword followed by an expression involving a control variable. The expression should have a logical (true/false) value or a numerical value. In the latter case, zero is considered false and any non-zero value is considered true.

If the expression is true, the subsequent statements, up to the matching `wend`, will be executed, and then the procedure is repeated. If the condition is false, EViews will skip the following commands and continue on with the rest of the program following the `wend` statement. For example:

```
!val = 1
!a = 1
while !val<10000 and !a<10
    smpl 1950q1 1970q1+!a
    series inc{!val} = income/!val
    !val = !val*10
    !a = !a+1
wend
```

There are four parts to this WHILE loop. The first part is the initialization of the control variables used in the test condition. The second part is the WHILE statement which includes the test. The third part is the statements updating the control variables. Finally the end of the loop is marked by the word `wend`.

Unlike a FOR statement, the WHILE statement does not update the control variable used in the test condition. You need to explicitly include a statement inside the loop that changes the control variable, or your loop will never terminate. Use the F1 key to break out of a program which is in an infinite loop.

In the example above of a FOR loop that changed the control variable, a WHILE loop provides a much clearer program:

```
!i = 1
while !i<=25
    vector a{!i}
    !i = !i + 11
wend
```

## Handling Execution Errors

By default, EViews will stop executing after encountering any errors, but you can instruct the program to continue running even if errors are encountered (see “[Executing a Program](#)” on page 84). In the latter case, you may wish to perform different tasks when errors are encountered. For example, you may wish to skip a set of lines which accumulate estimation results when the estimation procedure generated errors.

To test for and handle execution errors, you should use the `@errorcount` function to return the number of errors encountered while executing your program:

```
!errs = @errorcount
```

The information about the number of errors may be used by standard program statements to control the behavior of the program.

For example, to test whether the estimation of a equation generated an error, you should compare the number of errors before and after the command:

```
!old_count = @errorcount
equation eq1.ls y x c
!new_count = @errorcount
if !new_count > !old_count then
    [various commands]
endif
```

Here, we perform a set of commands only if the estimation of equation EQ1 incremented the error count.

## Other Tools

Occasionally, you will wish to stop a program or break out of a loop based on some conditions. To stop a program executing in EViews, use the `stop` command. For example, suppose you write a program that requires the series SER1 to have nonnegative values. The following commands check whether the series is nonnegative and halt the program if SER1 contains any negative value:

```
series test = (ser1<0)
if @sum(test) <> 0 then
    stop
endif
```

Note that if SER1 contains missing values, the corresponding value of TEST will also be missing. Since the `@sum` function ignores missing values, the program does not halt for SER1 that has missing values, as long as there is no negative value.

Sometimes, you do not wish to stop the entire program when a condition is satisfied; you just wish to exit the current loop. The `exitloop` command will exit the current `for` or `while` statement and continue running the program.

For example, suppose you computed a sequence of LR test statistics LR11, LR10, LR9, ..., LR1, say to test the lag length of a VAR. The following program sequentially carries out the LR test starting from LR11 and tells you the statistic that is first rejected at the 5% level:

```
!df = 9
for !lag = 11 to 1 step -1
    !pval = 1 - @cchisq(lr{!lag},!df)
    if !pval<=.05 then
        exitloop
    endif
next
scalar lag=!lag
```

Note that the scalar LAG has the value 0 if none of the test statistics are rejected.

## Multiple Program Files

When working with long programs, you may wish to organize your code using multiple files. For example, suppose you have a program file named `POWERS.PRG` which contains a set of program lines that you wish to use.

While you may be tempted to string files together using the `run` command, we caution you that EViews will stop after executing the commands in the referenced file. Thus, a program containing the lines

```
run powers.prg
series x = nrnd
```

will only execute the commands in the file `POWERS`, and will stop before generating the series X. This behavior is probably not what you intended.

You should instead use the `include` keyword to include the contents of a program file in another program file. For example, you can place the line

```
include powers
```

at the top of any other program that needs to use the commands in `POWERS`. `include` also accepts a full path to the program file, and you may have more than one `include` statement in a program. For example, the lines,

```
include c:\programs\powers.prg
```

```
include durbin_h  
[more lines]
```

will first execute all of the commands in C:\PROGRAMS\POWERS.PRG, then will execute the commands in DURBIN\_H.PRG, and then will execute the remaining lines in the program file.

Subroutines provide a more general, alternative method of reusing commands and using arguments.

## Subroutines

A *subroutine* is a collection of commands that allows you to perform a given task repeatedly, with minor variations, without actually duplicating the commands. You can also use subroutines from one program to perform the same task in other programs.

### Defining Subroutines

A subroutine starts with the keyword `subroutine` followed by the name of the routine and any arguments, and ends with the keyword `endsub`. Any number of commands can appear in between. The simplest type of subroutine has the following form:

```
subroutine z_square  
series x = z^2  
endsub
```

where the keyword `subroutine` is followed only by the name of the routine. This subroutine has no arguments so that it will behave identically every time it is used. It forms the square of the existing series Z and stores it in the new series X.

You can use the `return` command to force EViews to exit from the subroutine at any time. A common use of `return` is to exit from the subroutine if an unanticipated error is detected. The following program exits the subroutine if Durbin's *h* statistic for testing serial correlation with a lagged dependent variable cannot be computed (for details, see Greene, 1997, p.596, or Davidson and MacKinnon, 1993, p. 360):

```
subroutine durbin_h  
equation eqn.ls cs c cs(-1) inc  
scalar test=1-eqn.@regobs*eqn.@cov(2,2)  
' an error is indicated by test being nonpositive  
' exit on error  
if test<=0 then  
    return  
endif
```

---

```
' compute h statistic if test positive
scalar h=(1-eqn.@dw/2)*sqr(eqn.@regobs/test)
endsub
```

## Subroutine with arguments

The subroutines so far have been written to work with a specific set of variables. More generally, subroutines can take arguments. If you are familiar with another programming language, you probably already know how arguments allow you to change the behavior of the group of commands each time the subroutine is used. Even if you haven't encountered subroutines, you are probably familiar with similar concepts from mathematics. You can define a function, say

$$f(x) = x^2 \quad (6.1)$$

where  $f$  depends upon the argument  $x$ . The argument  $x$  is merely a place holder—it's there to define the function and it does not really stand for anything. Then, if you want to evaluate the function at a particular numerical value, say 0.7839, you can write

$f(0.7839)$ . If you want to evaluate the function at a different value, say 0.50123, you merely write  $f(0.50123)$ . By defining the function, you save yourself from writing out the whole expression every time you wish to evaluate it for a different value.

To define a subroutine with arguments, you start with `subroutine`, followed by the subroutine name, a left parenthesis, the arguments separated by commas, and finally a right parenthesis. Each argument is specified by listing a type of EViews object, followed by the name of the argument. Control variables may be passed by the scalar type and string variables by the string type. For example:

```
subroutine power(series v, series y, scalar p)
  v = y^p
endsub
```

This subroutine generalizes the example subroutine Z\_SQUARE. Calling POWER will fill the series given by the argument V with the power P of the series specified by the argument Y. So if you set V equal to X, Y equal to Z, and P equal to 2, you will get the equivalent of the subroutine Z\_SQUARE above. See the discussion below on how to call subroutines.

## Subroutine Placement

Your subroutine definitions should be placed, in any order, at the beginning of your program. The subroutines will not be executed until they are executed by the program using a `call` statement. For example:

```
subroutine z_square
    series x=z^2
endsub
' start of program execution
load mywork
fetch z
call z_square
```

Execution of this program begins with the `load` statement; the subroutine definition is skipped and is executed only at the last line when it is “called.”

The subroutine definitions must not overlap—after the `subroutine` keyword, there should be an `endsub` before the next `subroutine` declaration. Subroutines may call each other, or even call themselves.

Alternatively, you may wish to place frequently used subroutines in a separate program file and use an `include` statement to insert them at the beginning of your program. If, for example, you put the subroutine lines in the file `POWERS.PRG`, then you may put the line:

```
include powers
```

at the top of any other program that needs to call `Z_SQUARE` or `POWER`. You can use the subroutines in these programs as though they were built-in parts of the EViews programming language.

## Calling Subroutines

Once a subroutine is defined, you may execute the commands in the subroutine by using the `call` keyword. `call` should be followed by the name of the subroutine, and a list of any argument values you wish to use, enclosed in parentheses and separated by commas. If the subroutine takes arguments, they must all be provided in the same order as in the declaration statement. Here is an example program file that calls subroutines:

```
include powers
load mywork
fetch z gdp
series x
series gdp2
series gdp3
call z_square
call power(gdp2,gdp,2)
call power(gdp3,gdp,3)
```

The first call fills the series X with the value of Z squared. The second call creates the series GDP2 which is GDP squared. The last call creates the series GDP3 as the cube of GDP.

When the subroutine argument is a scalar, the subroutine may be called with a scalar object, a control variable, a simple number (such as “10” or “15.3”), a matrix element (such as “mat1(1,2)”) or a scalar expression (such as “!y + 25”). Subroutines that take matrix and vector arguments can be called with a matrix name, and if not modified by the subroutine, may also take a matrix expression. All other arguments must be passed to the subroutine with a simple object (or string) name referring to a single object of the correct type.

## Global and Local Variables

Subroutines work with variables and objects that are either *global* or *local*.

*Global variables* refer either to objects which exist in the workfile when the subroutine is called, or to the objects that are created in the workfile by a subroutine. Global variables remain in the workfile when the subroutine finishes.

A *local variable* is one that has meaning only within the subroutine. Local variables are deleted from the workfile once a subroutine finishes. The program that calls the subroutine will not know anything about a local variable since the local variable will disappear once the subroutine finishes and returns to the original program.

## Global Subroutines

By default, subroutines in EViews are *global*. Any global subroutine may refer to any global object that exists in the workfile at the time the subroutine is called. Thus, if Z is a series in the workfile, the subroutine may refer to and, if desired, alter the series Z. Similarly, if Y is a global matrix that has been created by another subroutine, the current subroutine may use the matrix Y.

The rules for variables in global subroutines are:

- Newly created objects are global and will be included in the workfile when the subroutine finishes.
- Global objects may be used and updated directly from within the subroutine. If, however, a global object has the same name as an argument in a subroutine, the variable name will refer to the argument and not to the global variable.
- The global objects corresponding to arguments may be used and updated by referring to the arguments.

Here is a simple program that calls a global subroutine:

```
subroutine z_square
    series x = z^2
endsub
load mywork
fetch z
call z_square
```

Z\_SQUARE is a global subroutine which has access to the global series Z. The new global series X contains the square of the series Z. Both X and Z remain in the workfile when Z\_SQUARE is finished.

If one of the arguments of the subroutine has the same name as a global variable, the argument name takes precedence. Any reference to the name in the subroutine will refer to the argument, not to the global variable. For example:

```
subroutine sqseries(series z, string %name)
    series {%name} = z^2
endsub
load mywork
fetch z
fetch y
call sqseries(y,"y2")
```

In this example, there is a series Z in the original workfile and Z is also an argument of the subroutine. Calling SQSERIES with the argument set to Y tells EViews to use the argument rather than the global Z. Upon completion of the routine, a new series Y2 will contain the square of the series Y, not the square of the series Z.

Global subroutines may call global subroutines. You should make certain to pass along any required arguments when you call a subroutine from within a subroutine. For example,

```
subroutine wgtols(series y, series wt)
    equation eq1
    call ols(eq1, y)
    equation eq2
    series temp = y/sqr(wt)
    call ols(eq2,temp)
    delete temp
endsub
subroutine ols(equation eq, series y)
    eq.ls y c y(-1) y(-1)^2 y(-1)^3
```

```
endsub
```

can be run by the program:

```
load mywork
fetch cpi
fetch cs
call wgtols(cs,cpi)
```

In this example, the subroutine WGTOLS explicitly passes along the arguments for EQ and Y to the subroutine OLS; otherwise those arguments would not be recognized by OLS. If EQ and Y were not passed, OLS would try to find a global series named Y and a global equation named EQ, instead of using EQ1 and CS or EQ2 and TEMP.

You cannot use a subroutine to change the object type of a global variable. Suppose that we wish to declare new matrices X and Y by using a subroutine NEWXY. In this example, the declaration of the matrix Y works, but the declaration of matrix X generates an error because a series named X already exists:

```
subroutine newxy
    matrix(2,2) x = 0
    matrix(2,2) y = 0
endsub
load mywork
series x
call newxy
```

EViews will return an error indicating that the global series X already exists and is of a different type than a matrix.

## Local Subroutines

All objects created by a global subroutine will be global and will remain in the workfile upon exit from the subroutine. If you include the word `local` in the definition of the subroutine, you create a local subroutine. All objects created by a local subroutine will be local and will be removed from the workfile upon exit from the subroutine. Local subroutines are most useful when you wish to write a subroutine which creates many temporary objects that you do not want to keep.

The rules for variables in local subroutines are:

- You may not use or update global objects directly from within the subroutine.
- The global objects corresponding to arguments may be used and updated by referring to the arguments.

- All other objects in the subroutine are local and will vanish when the subroutine finishes.

If you want to save results from a local subroutine, you have to explicitly include them in the arguments. For example, consider the subroutine:

```
subroutine local ols(series y, series res, scalar ssr)
    equation temp_eq.ls y c y(-1) y(-1)^2 y(-1)^3
    temp_eq.makeresid res
    ssr = temp_eq.@ssr
endsub
```

This local subroutine takes the series Y as input and returns the series RES and scalar SSR as output. The equation object TEMP\_EQ is local to the subroutine and will vanish when the subroutine finishes.

Here is an example program that calls this local subroutine:

```
load mywork
fetch hsf
equation eq1.ls hsf c hsf(-1)
eq1.makeresid rres
scalar rssr = eq1.@ssr
series ures
scalar ussr
call ols(hsf, ures, ussr)
```

Note how we first declare the series URES and scalar USSR before calling the local subroutine. These objects are global since they are declared outside the local subroutine. When we call the local subroutine by passing these global objects as arguments, the subroutine will update these global variables.

There is one exception to the general inaccessibility of global variables in local subroutines. When a global group is passed as an argument to a local subroutine, any series in the group is accessible to the local routine.

Local subroutines can call global subroutines and vice versa. The global subroutine will only have access to the global variables, and the local subroutine will only have access to the local variables, unless information is passed between the routines via arguments. For example, the subroutine

```
subroutine newols(series y, series res)
    include ols
    equation eq1.ls y c y(-1)
```

```
eq1.makeresid res
scalar rssr=eq1.@ssr
series ures
scalar ussr
call ols(y, ures, ussr)
endsub
```

and the program

```
load mywork
fetch hsf
series rres
call newols(hsf, rres)
```

produce equivalent results. Note that the subroutine NEWOLS still does not have access to any of the temporary variables in the local routine OLS, even though OLS is called from within NEWOLS.

## Programming Summary

### Support Commands

- open** .....opens a program file from disk ([p. 356](#)).
- output** .....redirects print output to objects or files ([p. 361](#)).
- poff** .....turns off automatic printing in programs ([p. 608](#)).
- pon** .....turns on automatic printing in programs ([p. 608](#)).
- program** .....declares a program.
- run** .....runs a program ([p. 405](#)).
- statusline** .....sends message to the status line ([p. 609](#)).
- tic** .....reset the timer ([p. 482](#)).
- toc** .....display elapsed time (since timer reset) in seconds ([p. 483](#)).

### Program Statements

- call** .....calls a subroutine within a program ([p. 603](#)).
- else** .....denotes start of alternative clause for IF ([p. 604](#)).
- endif** .....marks end of conditional commands ([p. 604](#)).
- endsub** .....marks end of subroutine definition ([p. 604](#)).
- exitloop** .....exits from current loop ([p. 605](#)).
- for** .....start of FOR execution loop ([p. 606](#)).
- if** .....conditional execution statement ([p. 606](#)).

**include** ..... include subroutine in programs ([p. 607](#)).  
**next** ..... end of FOR loop ([p. 607](#)).  
**return** ..... exit subroutine ([p. 609](#)).  
**step** ..... (optional) step size of a FOR loop ([p. 610](#)).  
**stop** ..... halts execution of program ([p. 611](#)).  
**subroutine** ..... declares subroutine ([p. 611](#)).  
**then** ..... part of IF statement ([p. 612](#)).  
**to** ..... upper limit of FOR loop ([p. 613](#)).  
**wend** ..... end of WHILE loop ([p. 614](#)).  
**while** ..... start of WHILE loop ([p. 614](#)).

## Support Functions

**@date** ..... string containing the current date ([p. 603](#)).  
**@errorcount** ..... number of errors encountered ([p. 605](#)).  
**@evpath** ..... string containing the directory path for the EViews executable ([p. 605](#)).  
**@isobject** ..... checks for existence of object ([p. 607](#)).  
**@tempPath** ..... string containing the directory path for EViews temporary files ([p. 611](#)).  
**@time** ..... string containing the current time ([p. 612](#)).  
**@toc** ..... calculates elapsed time (since timer reset) in seconds ([p. 613](#)).

# Chapter 7. Strings and Dates

---

## Strings

An alphanumeric *string* is a set of characters containing alphabetic (“alpha”) and numeric characters, and in some cases symbols, found on a standard keyboard.

Strings in EViews may include spaces and dashes, as well as single or double quote characters. Note also that EViews does not support unicode characters.

When entering alphanumeric values into EViews, you generally should enclose your characters in double quotes. The following are all examples of valid string input:

```
"John Q. Public"  
"Ax$23!*jFg5"  
"000-00-0000"  
"(949) 555-5555"  
"11/27/2002"  
"3.14159"
```

You should use the double quote character as an escape character for double quotes in a string. Simply enter two double quote characters to include the single double quote in the string:

```
"A double quote is given by entering two "" characters."
```

Bear in mind that strings are simply sequences of characters with no special interpretation. The string values “3.14159” and “11/27/2002” might, for example, be used to represent a number and a date, but as strings they have no such intrinsic interpretation. To provide such an interpretation, you must use the EViews tools for translating string values into numeric or date values (see “[String Information Functions](#)” on [page 122](#) and “[Translating between Date Strings and Date Numbers](#)” on [page 133](#) and).

Lastly, we note that the *empty*, or *null*, *string* (“”) has multiple interpretations in EViews. In settings where we employ strings as a building block for other strings, the null string is interpreted as a blank string with no additional meaning. If, for example, we concatenate two strings, one of which is empty, the resulting string will simply be the non-empty string.

In other settings, the null string is interpreted as a missing value. In settings where we use string values as a category, for example when performing categorizations, the null string is interpreted as both a blank string and a missing value. You may then

choose to exclude or not exclude the missing value as a category when computing a tabulation using the string values. This designation of the null string as a missing value is recognized by a variety of views and procedures in EViews and may prove useful.

Likewise, when performing string comparisons using blank strings, EViews generally treats the blank string as a missing value. As with numeric comparisons involving missing values, comparisons involving missing values will often generate a missing value. We discuss this behavior in greater detail in our discussion of “[String Comparison \(with empty strings\)](#)” on page 120.

## String Operators

The following operators are supported for strings: (1) concatenation—plus (“+”), and (2) relational—equal to (“=”), not equal to (“<>”), greater than (“>”), greater than or equal to (“>=”), less than (“<”), less than or equal to (“<=”).

### String Concatenation Operator

Given two strings, *concatenation* creates a new string which contains the first string followed immediately by the second string. You may concatenate strings in EViews using the concatenation operator, “+”. For example,

```
"John " + "Q." + " Public"  
"3.14" + "159"
```

returns the strings

```
"John Q. Public"  
"3.14159"
```

Bear in mind that string concatenation is a simple operation that does not involve interpretation of strings as numbers or dates. Note in particular that the latter entry yields the concatenated string, “3.14159”, not the sum of the two numeric values, “162.14”. To obtain numeric results, you will first have to convert your strings into a number (see “[String Information Functions](#)” on page 122).

Lastly, we note that when concatenating strings, the *empty* string is interpreted as a blank string, not as a missing value. Thus, the expression

```
"Mary " + "" + "Smith"
```

yields

```
"Mary Smith"
```

since the middle string is interpreted as a blank.

## String Relational Operators

The relational operators return a 1 if the comparison is true, and 0 if the comparison is false. In some cases, relational comparisons involving null strings will return a NA.

### *String Ordering*

To determine the ordering of strings, EViews employs the region-specific collation order as supplied by the Windows operating system using the user's regional settings. Central to the tasks of *sorting* or *alphabetizing*, the collation order is the culturally influenced order of characters in a particular language.

While we cannot possibly describe all of the region-specific collation order rules, we note a few basic concepts. First, all punctuation marks and other non alphanumeric characters, except for the hyphen and the apostrophe precede the alphanumeric symbols. The apostrophe and hyphen characters are treated distinctly so that "were" and "we're" remain close in a sorted list. Second, the collation order is case specific, so that the character "a" precedes "A". In addition, similar characters are kept close so that strings beginning with "a" are followed by strings beginning with "A", ahead of strings beginning with "b" and "B".

Typically, we determine the order of two strings by evaluating strings character-by-character, comparing pairs of corresponding characters in turn, until we find the first pair for which the strings differ. If, using the collation order, we determine the first character is less than the second character, we say that the first string is *less than* the second string and the second string is *greater than* the first. Two strings are said to be *equal* if they have the same number of identical characters.

If the two strings are identical in every character, but one of them is shorter than the other, then a comparison will indicate that the longer string is greater. A corollary of this statement is that the null string is less than or equal to all other strings.

The multi-character elements that arise in many languages are treated as single characters for purposes of comparison, and ordered using region-specific rules. For example, the "CH" and "LL" in Traditional Spanish are treated as unique characters that come between "C" and "L" and "M", respectively.

### *String Comparison (with non-empty strings)*

Having defined the notion of string ordering, we may readily describe the behavior of the relational operators for non-empty (non-missing) strings. The " = " (equal), " > = " (greater than or equal), and " < = " (less than or equal), " < > " (not equal), " > " (greater than), and " < " (less than) comparison operators return a 1 or a 0, depending on the result of the string comparison. To illustrate, the following (non region-specific) comparisons return the value 1,

```
"abc" = "abc"
```

```
"abc" <> "def"  
"abc" <= "def"  
"abc" < "abcdefg"  
"ABC" > "ABC"  
"abc def" > "abc lef"
```

while the following return a 0,

```
"AbC" = "abc"  
"abc" <> "abc"  
"aBc" >= "aB1"  
"aBC" <= "a123"  
"abc" >= "abcdefg"
```

To compare portions of strings, you may use the functions @LEFT, @RIGHT, and @MID to extract the relevant part of the string (see “[String Manipulation Functions](#)” on page 123). The relational comparisons,

```
@left("abcdef", 3) = "abc"  
@right("abcdef", 3) = "def"  
@mid("abcdef", 2, 2) = "bc"
```

all return 1.

To perform a caseless comparison, you should convert the expressions to all uppercase, or all lowercase using the @UPPER, or @LOWER functions. The comparisons,

```
@upper("abc") = @upper("aBC")  
@lower("ABC") = @lower("aBc")
```

both return 1.

To ignore leading and trailing spaces, you should use the @LTRIM, @RTRIM, and @TRIM functions remove the spaces prior to using the operator. The relational comparisons,

```
@ltrim(" abc") = "abc"  
@ltrim(" abc") = @rtrim("abc ")  
@trim(" abc ") = "abc"
```

all return 1.

### *String Comparison (with empty strings)*

Generally speaking, the relational operators treat the empty string as a missing value and return the numeric missing value NA when applied to such a string. Suppose, for example that an observation in the alpha series X contains the string “Apple”, and the correspond-

ing observation in the alpha series Y contains a blank string. All comparisons (“X = Y”, “X > Y”, “X > = Y”, “X < Y”, “X < = Y”, and “X < > Y”) will generate an NA for that observation since the Y value is treated as a missing value.

Note that this behavior differs from EViews 4.1 and earlier in which empty strings were treated as ordinary blank strings and not as a missing value. In these versions of EViews, the comparison operators always returned a 0 or a 1. The change in behavior, while regrettable, is deemed necessary to support the use of string missing values.

It is still possible to perform comparisons using the previous methods. One approach is to use the special functions @EQNA and @NEQNA for performing equality and strict inequality comparisons without propagating NAs (see [“String Information Functions” on page 122](#)). For example, you may use the expressions

```
@eqna(x, y)  
@neqna(x, y)
```

so that blanks in either X or Y are treated as ordinary string values. Using these two functions, the observation where X contains “Apple” and Y contains the “” will evaluate to 0 and 1, respectively instead of NA.

Alternatively, if you specify a relational expression involving a literal blank string, EViews will perform the test treating empty strings as ordinary string values. If, for example, you test

```
x = ""
```

or

```
x < ""
```

all of the string values in X will be tested against the string literal “”. You should contrast this behavior with the behavior for the tests “X = Y” and “X < Y” where blank values of X or Y result in an NA comparison.

Lastly, EViews provides a function for the strict purpose of testing whether or not a string value is an empty string. The @ISEMPTY function tests whether each element of the alpha series contains an empty string. The relational equality test against the blank string literal “” is equivalent to this function.

## String Functions

EViews provides a number of functions that may be used with strings or that return string values.

## String Information Functions

The following is a brief summary of the basic functions that take strings as an argument and return a number.

- `@LENGTH(str)`: returns an integer value for the length of the string *str*.

```
@length("I did not do it")
```

returns the value 15.

A shortened form of this function, `@LEN`, is also supported.

- `@INSTR(str1, str2[, int])`: finds the starting position of the target string *str2* in the base string *str1*. By default, the function returns the location of the first occurrence of *str2* in *str1*. You may provide an optional integer *int* to change the occurrence. If the occurrence of the target string is not found, `@INSTR` will return a 0.

The returned integer is often used in conjunction with `@MID` to extract a portion of the original string.

```
@instr("1.23415", "34")
```

returns the value 4, since the substring “34” appears beginning in the fourth character of the base string.

- `@VAL(str[, fmt])`: converts the string representation of a number, *str*, into a numeric value. If the string has any non-digit characters, the returned value is an NA. You may provide an optional numeric format string *fmt*.

```
@val("1.23415")
```

- `@ISEMPTY(str)`: tests for whether *str* is a blank string, returning a 1 if *str* is a null string, and 0 otherwise.

```
@isempty("1.23415")
```

returns a 0, while

```
@isempty("")
```

returns the value 1.

- `@EQNA(str1, str2)`: tests for equality of *str1* and *str2*, treating null strings as ordinary blank strings, and not as missing values. Strings which test as equal return a 1, and 0 otherwise. For example,

```
@eqna("abc", "abc")
```

returns a 1, while

```
@eqna("", "def")
```

returns a 0.

- `@NEQNA(str1, str2)`: tests for inequality of *str1* and *str2*, treating null strings as ordinary blank strings, and not as missing values. Strings which test as not equal return a 1, and 0 otherwise.

```
@neqna ("abc", "abc")  
returns a 0,  
  
@neqna ("", "def")  
returns a 1.
```

- `@DATEVAL(str[, fmt])`: converts the string representation of a date, *str*, into a date number using the optional format string *fmt*.

```
@dateval("12/1/1999", "mm/dd/yyyy")  
will return the date number for December 1, 1999 (730088) while  
  
@dateval("12/1/1999", "dd/mm/yyyy")  
will return the date number for January 12, 1999 (729765). See “Dates” beginning on page 127 for discussion of date numbers and format strings.
```

- `@DTOO(str)`: (*Date TO Obs*) converts the string representation of a date, *str*, into an observation value. Returns the scalar offset from the beginning of the workfile associated with the observation given by the date string. The string must be a valid EViews date.

```
create d 2/1/90 12/31/95  
%date = "1/1/93"  
!t = @dtoo(%date)
```

returns the value !T = 762.

Note that `@DTOO` will generate an error if used in a panel structured workfile.

## String Manipulation Functions

The following is a brief summary of the basic functions that take strings as an argument and return a string.

- `@LEFT(str, int)`: returns a string containing the *int* characters at the left end of the string *str*. If there are fewer than *int* characters, `@LEFT` will return the entire string.

```
@left("I did not do it", 5)  
returns the string "I did".
```

- `@RIGHT(str, int)`: returns a string containing the *int* characters at the right end of a string. If there are fewer than *int* characters, `@RIGHT` will return the entire string.

```
@right("I doubt that I did it", 8)
```

returns the string “I did it”.

- **@MID(str1, int1[, int2]):** returns the string consisting of the characters starting from position *int1* in the string. By default, @MID returns the remainder of the string, but you may specify the optional integer *int2*, indicating the number of characters to be returned.

```
@mid("I doubt that I did it", 9, 10)
```

returns “that I did”.

```
@mid("I doubt that I did it", 9)
```

returns the string “that I did it”.

- **@INSERT(str1, str2, int):** inserts the string *str2* into the base string *str1* at the position given by the integer *int*.

```
@insert("I believe it can be done", "not ", 16)
```

returns “I believe it cannot be done”.

- **@REPLACE(str1, str2, str3[, int]):** returns the base string *str1*, with the replacement *str3* substituted for the target string *str2*. By default, all occurrences of *str2* will be replaced, but you may provide an optional integer *int* to specify the number of occurrences to be replaced.

```
@replace("Do you think that you can do it?", "you", "I")
```

returns the string “Do I think that I can do it?”, while

```
@replace("Do you think that you can do it?", "you", "I", 1)
```

returns “Do I think that you can do it?”.

- **@LTRIM(str):** returns the string *str* with spaces trimmed from the left.

```
@ltrim(" I doubt that I did it. ")
```

returns “I doubt that I did it. ”. Note that the spaces on the right remain.

- **@RTRIM(str):** returns the string *str* with spaces trimmed from the right.

```
@rtrim(" I doubt that I did it. ")
```

returns the string “ I doubt that I did it.”. Note that the spaces on the left remain.

- **@TRIM(str):** returns the string *str* with spaces trimmed from the both the left and the right.

```
@trim(" I doubt that I did it. ")
```

returns the string “I doubt that I did it.”.

- `@UPPER(str)`: returns the upper case representation of the string *str*.

```
@length("I did not do it")
```

returns the string “I DID NOT DO IT”.

- `@LOWER(str)`: returns the lower case representation of the string *str*.

```
@lower("I did not do it")
```

returns the string “i did not do it”.

## String Conversion Functions

The following functions convert numbers and date numbers into string values:

- `@STR(num[, fmt])`: returns a string representation of the number *num*. You may provide an optional numeric format string *fmt*.

```
@str(153.4)
```

returns the string “153.4”.

To create a string containing 4 significant digits and leading “\$” character, use

```
@str(-15.4435,"g$.4")
```

The resulting string is “-\$15.44”.

The expression

```
@str(-15.4435,"f7..2")
```

converts the numerical value, -15.4435, into a fixed 7 character wide decimal string with 2 digits after the decimal and comma as decimal point. The resulting string is “-15,44”. Note that there is a leading space in front of the “-” character making the string 7 characters long.

The expression

```
@str(-15.4435,"e(..2)")
```

converts the numerical value, -15.4435, into a string written in scientific notation with two digits to the right of the decimal point. The decimal point in the value will be represented using a comma and negative numbers will be enclosed in parenthesis. The resulting string is “(1,54e +01)”. A positive value will not have the parenthesis.

```
@str(15.4435,"p+.1")
```

converts the numeric value, 15.4435, into a percentage where the value is multiplied by 100. Only 1 digit will be included after the decimal and an explicit “+” will always be included for positive numbers. The resulting value after rounding is “+ 1544.4”.

- `@DATESTR(date1[, fmt])`: converts the date number *date1* to a string representation using the optional date format string, *fmt*.

```
@datestr(730088, "mm/dd/yy")
```

will return “12/1/99”,

```
@datestr(730088, "DD/mm/yyyy")
```

will return “01/12/1999”, and

```
@datestr(730088, "Month dd, yyyy")
```

will return “December 1, 1999”, and

```
@datestr(730088, "w")
```

will produce the string “3”, representing the weekday number for December 1, 1999.

See “[Dates](#)” on page 127 for additional details on date numbers and date format strings.

### Special Functions that return Strings

EViews provides a special, workfile-based function that uses the structure of the active workfile page and returns a *set* of string values representing the date identifiers associated with the observations.

- `@STRDATE(fmt)`: returns the set of workfile row dates as strings, using the date format string *fmt*. See “[Special Date Functions](#)” on page 145 for details.

In addition, EViews provides two special functions that return a string representations of the date associated with a specific observation in the workfile, or with the current time.

- `@OTOD(int)`: (*Obs TO Date*) : returns a string representation of the date associated with a single observation (counting from the start of the workfile). Suppose, for example, that we have a quarterly workfile ranging from 1950Q1 to 1990Q4. Then

```
@otod(16)
```

returns the date associated with the 16th observation in the workfile in string form, “1953Q4”.

- `@STRNOW(fmt)`: returns a string representation of the current date number (at the moment the function is evaluated) using the date format string, *fmt*.

```
@strnow("DD/mm/yyyy")
```

returns the date associated with the current time in string form with 2-digit days, months, and 4-digit years separated by a slash, “24/12/2003”.

## String Function Summary

### String Functions

- @datestr** ..... converts a date number into a string ([p. 567](#)).
- @dateval** ..... converts a string into a date number ([p. 568](#)).
- @dtoo** ..... returns observation number corresponding to the date specified by a string ([p. 569](#)).
- @eqna** ..... tests for equality of string values treating empty strings as ordinary blank values ([p. 569](#)).
- @insert** ..... inserts string into another string ([p. 570](#)).
- @instr** ..... finds the starting position of the target string in a string ([p. 570](#)).
- @isempty** ..... tests string against empty string ([p. 571](#)).
- @left** ..... returns the leftmost characters string ([p. 571](#)).
- @len, @length** .... finds the length of a string ([p. 572](#)).
- @lower** ..... returns the lowercase representation of a string ([p. 572](#)).
- @ltrim** ..... returns the string with spaces trimmed from the left end ([p. 572](#)).
- @mid** ..... returns a substring from a string ([p. 573](#)).
- @neqna** ..... tests for inequality of string values treating empty strings as ordinary blank values ([p. 574](#)).
- @otod** ..... returns a string associated with a the date or observation value ([p. 575](#)).
- @replace** ..... replaces substring in a string ([p. 575](#)).
- @right** ..... returns the rightmost characters of a string ([p. 576](#)).
- @rtrim** ..... returns the string with spaces trimmed from the right end ([p. 576](#)).
- @str** ..... returns a string representing the given number ([p. 576](#)).
- @strdate** ..... returns string corresponding to each element in workfile ([p. 577](#)).
- @strlen** ..... finds the length of a string ([p. 577](#)).
- @strnow** ..... returns a string representation of the current date ([p. 577](#)).
- @trim** ..... returns the string with spaces trimmed from the both ends ([p. 578](#)).
- @upper** ..... returns the uppercase representation of a string ([p. 578](#)).
- @val** ..... returns the number that a string represents ([p. 578](#)).

## Dates

There are a variety of places in EViews where you may work with calendar dates. For most purposes, users need not concern themselves with the intricacies of working with dates.

Simply enter your dates in familiar text notation and EViews will automatically interpret the string for you.

Those of you who wish to perform more sophisticated operations with dates will, however, need to understand some basic concepts.

In most settings, you may simply use text representations of dates, or *date strings*. For example, an EViews sample can be set to include only observations falling between two dates specified using date strings such as “May 11, 1997”, “1/10/1990” or “2001q1”. In these settings, EViews understands that you are describing a date and will interpret the string accordingly.

Date information may also be provided in the form of a *date number*. A date number is a numeric value with special interpretation in EViews as a calendar date. EViews allows you to convert date strings into date numbers which may be manipulated using a variety of tools. These tools allow you to perform standard calendar operations such as finding the number of days or weeks between two dates, the day of the week associated with a given day, or the day and time 36 hours from now.

The remainder of this section summarizes the use of dates in EViews. There are several tasks that are central to working with dates:

- Translating between date strings and date numbers.
- Translating ordinary numbers into date numbers.
- Manipulating date numbers using operators and functions.
- Extracting information from date numbers.

Before turning to these tasks, we must first provide a bit of background on the characteristics of date strings, date numbers, and a special class of strings called *date formats*, which are sometimes employed when translating between the former.

## Date Strings

*Date strings* are simply text representations of dates and/or times. Most of the conventional ways of representing days, weeks, months, years, hours, minutes, *etc.*, as text are valid date strings.

To be a bit more concrete, the following are valid date strings in EViews:

```
"December 1, 2001"  
"12/1/2001"  
"Dec/01/01 12am"  
"2001-12-01 00:00"  
"2001qIV"
```

As you can see, EViews is able to handle a wide variety of representations of your dates and times. You may use everything from years represented in 1, 2, and 4-digit Arabic form (“1”, “01”, “99”, “1999”), to month names and abbreviations (“January”, “jan”, “Jan”), to quarter designations in Roman numerals (“I” to “IV”), to weekday names and abbreviations (“Monday”, “Mon”), to 12 or 24-hour representations of time (“11:12 pm”, “23:12”). A full list of the recognized date string components is provided in [“Date Formats” on page 130](#).

It is worth noting that date string representations may be divided up into those that are *unambiguous* and those that are *ambiguous*. Unambiguous date strings have but a single interpretation as a date, while ambiguous date strings may be interpreted in multiple ways.

For example, the following dates may reasonably be deemed unambiguous:

```
"March 3rd, 1950"  
"1980Q3"  
"9:52PM"
```

while the following dates are clearly ambiguous:

```
"2/3/4"  
"1980:2"  
"02:04"
```

The first date string is ambiguous because we cannot tell which of the three fields is the year, which is the month, and which is the day, since different countries of the world employ different orderings. The second string is ambiguous since we cannot determine the period frequency within the year. The “2” in the string could, for example, refer to the second quarter, month, or even semi-annual in the year. The final string is ambiguous since it could be an example of a time of day in “hour:minute” format (2:04 am), or a date in “year:period” notation (*i.e.*, the fourth month of the year 2002) or “period:year” notation (*i.e.*, the second month of 2004).

In settings where date input is required, EViews will generally accept date string values without requiring you to provide formatting information. It is here that the importance of the distinction between ambiguous and unambiguous date strings is seen. If the date string is unambiguous, the free-format interpretation of the string as a date will produce identical results in all settings. On the other hand, if the date string is ambiguous, EViews will use the context in which the date is being used to determine the most likely interpretation of the string. You may find that ambiguous date strings are neither interpreted consistently, nor as desired.

These issues, and methods of getting around the problem of ambiguity, are explored in greater detail in [“Translating between Date Strings and Date Numbers” on page 133](#).

## Date Numbers

Date information is often held in EViews in the form of a *date number*. A date number is a double precision number corresponding to an instance in time, with the integer portion representing a specific day, and the decimal fraction representing time during the day.

The integer portion of a date number represents the number of days in the Gregorian proleptic calendar since Monday, January 1, A.D. 0001 (a “proleptic” calendar is a calendar that is applied to dates both before and after the calendar was historically adopted). The first representable day, January 1, A.D. 1 has an integer value of 0, while the last representable day, December 31, A.D. 9999, has an integer value of 3652058.

The fractional portion of the date number represents a fraction of the day, with resolution to the millisecond. The fractional values range from 0 (12 midnight) up to (but not including) 1 (12 midnight). A value of 0.25, for example, corresponds to one-quarter of the day, or 6:00 a.m.

It is worth noting that the time of day in an EViews date number is accurate up to a particular millisecond within the day, although it can always be displayed at a lower “precision” (larger unit of time). When date numbers are formatted to lower precisions, they are always rounded down to the requested precision and never rounded up. Thus, when displaying the week or month associated with a date number, EViews always rounds down to the beginning of the week or month.

## Date Formats

A *date format* (or *date format*, for short) is a string made up of text expressions that describe how components of a date and time may be encoded in a date string. Date formats are used to provide an explicit description of a date string representation, and may be employed when converting between strings or numbers and date numbers.

Before describing date formats in some detail, we consider a simple example. Suppose that we wish to use the date string “5/11/1997” to represent the date May 11, 1997. The date format corresponding to this text representation is

"mm/dd/yyyy"

which indicates that we have, in order, the following components: a one or two-digit month identifier, a “/” separator, a one or two-digit day identifier, a “/” separator, and a 4-digit year identifier.

Alternatively, we might wish to use the string “1997-May-11” to represent the same date. The date format for this string is

"yyyy-Month-dd"

since we have a four-digit year, followed by the full name of the month (with first letter capitalized), and the one or two-digit day identifier, all separated by dashes.

Similarly, the ISO 8601 representation for 10 seconds past 1:57 p.m. on this date is "1997-05-11 13:57:10". The corresponding format is

"yyyy-MM-DD HH:mi:ss"

Here, we have used the capitalized forms of "MM", "DD", and "HH" to ensure that we have the required leading zeros.

### Date Format Components

A date format may contain one or more of the following string fragments corresponding to various date components. In most cases, there are various upper and lowercase forms of the format component, corresponding either to the presence or absence of leading zeros, or to the case of the string identifiers.

The following format strings are the basic components of a date format:

#### *Years*

Year formats use either two or four digit years, with or without leading zeros. The corresponding date format strings are:

- "yyyy" or "YYYY": four digit year without/with leading zeros.
- "yy" or "YY": two digit year without/with leading zeros.
- "year" or "YEAR": synonym for "yyyy" and "YYYY", respectively.

#### *Semi-Annual*

The semi-annual format corresponds to a single digit representing the period in the year:

- "s" or "S": one digit half-year (1 or 2).

#### *Quarters*

Quarter formats allow for values entered in either standard (Arabic) or Roman numbers:

- "q" or "Q": quarter number, always without leading zeros (1 to 4).
- "qr" or "QR": quarter in Roman numerals (I to IV).

#### *Months*

Month formats may represent two-digit month values with or without leading zeros, three-letter abbreviations for the month, or the full month name. The text identifiers may be all

lowercase, all uppercase or “namecase” in which we capitalize the first letter of the month identifier. The corresponding format strings are given by:

- “mm” or “MM”: two-digit month without/with leading zeros.
- “mon”, “Mon”, or “MON”: three-letter form of month, following the case of the format string (“jan”, “Feb”, “MAR”).
- “month”, “Month”, or “MONTH”: full month name, following the case of the format string (“january”, “February”, “MARCH”).

### *Weeks*

Week of the year formats may be specified with or without leading zeros:

- “ww” or “WW”: week of year (with first week starting from Jan 1st) without/with leading zeros.

### *Days*

Day formats correspond to day of the year, business day of the year, day of the month, or day of the week, in various numeric and text representations.

- “ddd” or “DDD”: day of year without/with leading zeros.
- “bbb” or “BBB”: business day of year without/with leading zeros (only counting Monday-Friday).
- “dd” or “DD”: day of month without/with leading zeros.
- “day” or “DAY”: day of month with suffix, following the case of the format string (“1st”, “2nd”, “3RD”).
- “w” or “W”: weekday number (1-7) where 1 is Monday.
- “wdy”, “Wdy”, or “WDY”: three-letter weekday abbreviation, following the case of the format string (“Mon”, “Tue”, “WED”).
- “weekday”, “Weekday”, or “WEEKDAY”: full weekday name, following the case of the format string (“monday”, “Tuesday”, “WEDNESDAY”).

### *Time (Hours/Minutes/Seconds)*

The time formats correspond to hours (in 12 or 24 hour format), minutes, seconds, and fractional sections, with or without leading zeros and with or without the AM/PM indicator where appropriate.

- “hh” or “HH”: hour in 24-hour format without/with leading zeros.
- “hm” or “HM”: hour in 12-hour format without/with leading zeros.

- “am” or “AM”: two letter AM/PM indicator for 12-hour format, following the case of the format string.
- “a” or “A”: single letter AM/PM indicator for 12-hour format, following the case of the format string.
- “mi” or “MI”: minute, always with leading zeros.
- “ss.s”, “ss.s”, “ss.ss”, or “sssss”: seconds and tenths, hundredths, and thousandths-of-a-second, with leading zeros. The capitalized forms of these formats (“SS”, “SS.S”, ...) yield identical results.

### *Delimiters*

You may use text to delimit the date format components:

- “f” or “F”: use frequency delimiter taken from the active, regular frequency workfile page. The delimiter corresponds to the letter associated with the current workfile frequency (“a”, “m”, “q”, ..., “A”, “M”, “Q”, ...), following the case of the format string, or the colon (“:"), as determined by the Global Options setting (**Options/Dates & Frequency Conversion.../Quarterly/Monthly display**).
- “?” : used as “wildcard” single character for skipping a character formats used on date number input. Passed through to the output string.
- Other alphabetical characters are errors unless they are enclosed in square brackets e.g. “[Q]”, in which case they are passed through to the output (for example, the “standard-EViews” quarterly format is “YYYY[Q]Q”, where we use a four digit year identifier, followed by a “Q” delimiter/identifier, followed by a single digit for the quarter “1990Q2”).
- All other characters (e.g., punctuation) are passed through to the input or output without special interpretation.

### Translating between Date Strings and Date Numbers

There are times when it is convenient to work with date strings, and times when it is easier to work with date numbers.

For example, when we are describing or viewing a specific date, it is easier to use a “human readable” date string such as “2002-Mar-20”, “3/20/2002”, or “March 20, 2002 12:23 pm” than the date number 730928.515972.

Alternatively, since date strings are merely text representations of dates, working with date numbers is essential when manipulating calendar dates to find elapsed days, months or years, or to find a specific date and time 31 days and 36 hours from now.

Accordingly, translating between string representations of dates and date numbers is one of the more important tasks when working with dates in EViews. These translations occur in many places in EViews, ranging from the interpretation of date strings in sample processing, to the spreadsheet display of series containing date numbers, to the import and export of data from foreign sources.

In most settings, the translations take place automatically, without user involvement. For example, when you enter a sample command of the form

```
smpl 1990q1 2000q4
```

EViews automatically converts the date strings into a range of date numbers. Similarly, when you edit a series that contains date numbers, you typically will enter your data in the form of a date string such as

```
"2002-Mar-20"
```

which EViews will automatically translate into a date number.

In other cases, you will specifically request a translation by using the built-in EViews functions @DATESTR (to convert a date number to a string) and @DATEVAL (to convert a date string to a date number).

For example, the easiest way to identify the date 1,000 days after May 1, 2000 is first to convert the string value “May 1, 2000” into a date number using @DATEVAL, to manipulate the date number to find the value 1000 days after the original date, and finally to convert the resulting date number back into a string using @DATESTR. See “[Formatted Conversion](#)” on page 138 and “[Manipulating Date Numbers](#)” on page 142 for additional details.

All translations between dates strings and date numbers involve one of two methods:

- First, EViews may perform a *free-format conversion* in which the date format is automatically inferred from the string values, in some cases other contextual information.
- Second, EViews may perform a *formatted conversion* in which the string representation of the dates is provided explicitly via a date format.

For the most part, you should find that free-format conversion is sufficient for most needs. Nevertheless, you may find that in some cases the automatic handling of dates by EViews does not produce the desired results. If this occurs, you should either modify any ambiguous date formats, or specify an explicit formatted conversion to generate date numbers as necessary.

## Free-format Conversion

EViews will perform free-format conversions between date strings and numbers whenever: (1) there is an automatic translation between strings and numbers, or (2) when you use one of the translation functions without an explicit date format.

When converting from strings to numbers, EViews will produce a date number using the “most likely” interpretation of the date string. For the most part, you need not concern yourself with the details of the conversion, but if you require additional detail on specific topics (e.g., handling of date intervals, the implicit century cutoff for 2-digit years) see “[Free-format Conversion Details](#)” on page 147.

When converting from date numbers to strings, EViews will use the global default settings to determine the default date format, and will display all significant information in the date number.

### *Converting Unambiguous Date Strings to Numbers*

The free-format conversion of unambiguous date strings (see “[Date Strings](#)” on page 128), to numbers will produce identical results in all settings. The date string:

“March 3rd, 1950”

will be interpreted as the third day of the third month of the year A.D. 1950, and will yield the date value 711918.0. Note that the date value is the smallest associated with the given date, corresponding to 12 midnight.

Similarly, the date string:

“1980Q3”

is interpreted as the first instance in the third quarter of 1980. EViews will convert this string into the date number representing the smallest date value in that quarter, 722996.0 (12 midnight on July 1, 1980).

If we specify a time string without a corresponding day,

“9:52PM”

the day portion of the date is set to 0 (effectively, January 1, A.D. 1), yielding a value of 0.91111111 (see “[Incomplete Date Numbers](#)” on page 148) for details.

Consider also the following unambiguous date string:

“1 May 03”

While this entry may appear to be ambiguous since the “03” may reasonably refer to either 1903 or 2003, EViews resolves the ambiguity by assuming that if the two-digit year is greater than or equal to 30, the year is assumed to be from the twentieth century, otherwise the year is assumed to be from the twenty first century (see “[Two-digit Years](#)” on

[page 148](#) for discussion). Consequently free-format conversion of two-digit years will produce consistent results in all settings.

### Converting Ambiguous Date Strings to Numbers

Converting from ambiguous date strings will yield context sensitive results. In cases involving ambiguity, EViews will determine the most likely translation format by examining surrounding data or applicable settings for clues as to how the date strings should be interpreted.

The following contextual information is used in interpreting ambiguous free-form dates:

- For implicit period notation (*e.g.*, “1990:3”) the current workfile frequency is used to determine the period.
- Choosing between ambiguous “mm/dd” or “dd/mm” formats is determined by examining the values of related date strings (*i.e.*, those in the same series), user-specified date/time display formats for a series or column of a spreadsheet, or by examining the EViews global setting for date display, (**Options/Dates & Frequency Conversion.../Month/Day order in dates**).

To fix ideas, we consider a few simple examples of the use of contextual information.

If you specify an ambiguous sample string, EViews will use the context in which the sample is used, the frequency of the workfile, to determine the relevant period. For example, given the sample statement

```
smp1 90:1 03:3
```

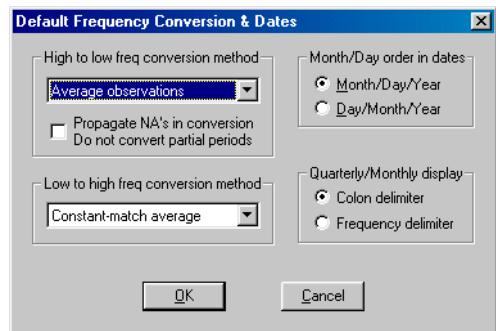
and a quarterly workfile, the sample will be set from 1990q1 to 2003q3. If the workfile is monthly, the sample will be set from January 1990 to March 2003.

Suppose instead that you are editing a series spreadsheet where your date numbers are *displayed* as dates strings using a specified format. In this setting, EViews allows you to enter your values as date strings, instead of having to enter the underlying date numbers. In this context, it is natural for EViews to use the current display format as a hint in interpreting ambiguous data. For example, if the current display format is set to “Month dd, YYYY” then an input of “2/3/4” or “@DATEVAL(“2/3/4”)” will be interpreted as February the 3rd, 2004. On the other hand, if the current display format is set to “YYYY-MM-DD” then the same input will be interpreted as the March the 4th, 2002.

In settings where an entire series is provided to an EViews procedure, EViews is able to use all of the values in the series to aid in determining the underlying data format. For example, when an alpha series is provided as a date identifier for restructuring a workfile, EViews will first scan all the values of the series in order to decide on the most likely format of all of the data before converting the string in each element into a date number. If the first observation of the series is an ambiguous “2/3/4” but a later observation is “3/20/

95" then the "2/3/4" will be interpreted as the 3rd of February 2004 since that is the only order of year, month and day that is consistent with the "3/20/95" observation.

Conversely, when generating new series values with a GENR or series assignment statement, EViews processes observation individually and is therefore unable to obtain contextual information to aid in interpreting ambiguous date strings. In this case, EViews will use the global workfile setting for the **Month/Day order in dates** to determine the ordering of the days and months in the string.



For example, when the expression

```
series dnums = @dateval("2/3/4")
```

is used to generate a series containing date values, EViews will interpret the value as February 3, 2004, if the global setting is **Month/Day/Year**, and March 2, 2004, if the global setting is **Day/Month/Year**.

### *Converting Date Numbers to Strings*

EViews provides the @DATESTR function to translate a date number to a date string. We describe the function in detail in ["Formatted Conversion" on page 138](#), but for now, simply note that @DATESTR takes an optional argument describing the date format to be used when exporting the string. If the optional argument is not provided, EViews will perform a free-format conversion.

In performing the free-format conversion, EViews examines two pieces of information. First, the global default settings for the **Month/Day order in dates** will be used to determine the ordering of days and months in the string. Next, EViews examines the date values to be translated and looks for relevant time-of-day information.

If there is no relevant time-of-day information in the date numbers (e.g., the non-integer portions are all zero), EViews writes the string corresponding to the date using either the date format

"dd/mm/yyyy"

or

"mm/dd/yyyy"

with preference given to the order favored in the global settings.

If there is relevant time-of-day information, EViews will extend the date format accordingly. Thus, if days are favored in the ordering, and relevant hours (but not minutes and seconds) information is present, EViews will use

"dd/mm/yyyy hh"

while if hours-minutes are present, the format will be

"dd/mm/yyyy hh:mi"

and so forth.

### Formatted Conversion

While the free-format conversions will generally produce the desired results, there may be times when you wish to gain control over the conversion. EViews will perform a formatted conversion between date strings and date numbers whenever you use the @DATEVAL or @DATESTR functions *with* the optional second argument specifying an explicit date format.

To convert a date string into a date number using a date format, you should use the "@DATEVAL" function with two arguments. The first argument must be a valid date string, and the second must be the corresponding date format string. If you omit the optional second argument, EViews will perform a free-format conversion.

- **@DATEVAL(*str[, fmt]*)**: takes the string *str* and evaluates it to a date number using the optional date format string, *fmt*.

A few simple examples will illustrate the wide range of string to date number conversions that are possible using @DATEVAL and a date format. The simplest format strings involve the standard month/day/year date descriptions:

```
@dateval ("12/1/1999", "mm/dd/yyyy")
```

will return the date number for December 1, 1999 (730088),

```
@dateval ("12/1/1999", "dd/mm/yyyy")
```

returns the date number for January 12, 1999 (729765). Here we have changed the interpretation of the date string from "American" to "European" by reversing the order of the parts of the format string.

Likewise, we may find the first date value associated with a given period

```
@dateval ("1999", "yyyy")
```

returns the value 729754.0 corresponding to 12 midnight on January 1, 1999, the first date value for the year 1999.

Conversion of a broad range of date strings is possible by putting together various date format string components. For example,

```
@dateval("January 12, 1999", "Month dd, yyyy")
```

returns the date number for 12 midnight on January 12, 1999 (729765), while

```
@dateval("99 January 12, 9:37 pm", "yy Month dd, hm:mi am")
```

yields the value 729765.900694 corresponding to the same date, but at 9:37 in the evening. In this example, the “hm:mi” corresponds to hours (in a 12 hour format, with no leading 0’s) and minutes, and the “am” indicates that there is an indicator for “am” and “pm”. See “[Date Strings](#)” on page 128 and “[Date Formats](#)” on page 130 for additional details.

To translate a date number to a date string using a date format, you should use the @DATESTR function with two arguments. The first argument must be a valid date number, and the second must be a date format string describing a string representation of the date.

- `@DATESTR(date_val, fmt)`: converts the date number into a string, using the optional date format *fmt*. If a format is not provided, EViews will use a default method (see “[Converting Date Numbers to Strings](#)” on page 137).

For example,

```
@datestr(730088, "mm/dd/yy")
```

will return “12/1/99”,

```
@datestr(730088, "DD/mm/yyyy")
```

will return “01/12/1999”, and

```
@datestr(730088, "Month dd, yyyy")
```

will return “December 1, 1999”, and

```
@datestr(730088, "w")
```

will produce the string “3”, representing the weekday number for December 1, 1999. See “[Date Numbers](#)” on page 130 and “[Date Formats](#)” on page 130 for additional details.

## Translating Ordinary Numbers into Date Numbers

While date information is most commonly held in the form of date strings or date numbers, one will occasionally encounter data in which a date is encoded as a (non-EViews format) numeric value or values. For example, the first quarter of 1991 may be given the numeric representation of 1991.1, or the date “August 15, 2001” may be held in the single number 8152001, or in three numeric values 8, 15, and 2001.

The `@MAKEDATE` function is used to translate ordinary numbers into date numbers. It is similar to `@DATEVAL` but is designed for cases in which your dates are encoded in one or more numeric values instead of date strings:

- `@MAKEDATE(arg1[, arg2[,arg3]], fmt)`: takes the numeric values given by the arguments *arg1*, and optionally, *arg2*, etc. and returns a date number using the required format string, *fmt*. Only a subset of all date formats are supported by `@MAKEDATE`.

If more than one argument is provided, the arguments must be listed from the lowest frequency to the highest, with the first field representing either the year or the hour.

The simplest form of `@MAKEDATE` involves converting a single number into a date or a time. The following are the supported formats for converting a single number into a date value:

- “yy” or “yyyy”: two or four-digit years.
- “yys” or “yyyys”: year\*10 + half-year.
- “yy.s” or “yyyy.s”: year + half-year/10.
- “yyq” or “yyyyq”: year\*10 + quarter.
- “yy.q” or “yyyy.q”: year + quarter/10.
- “yymm” or “yyyymm”: year\*10 + month.
- “yy.mm” or “yyyy.mm”: year + month/10.
- “yyddd” or “yyyyddd”: year\*1000 + day in year.
- “yyddd” or “yyyyddd”: year\*1000 + day in year/1000.
- “yymmdd” or “yyyymmdd”: year\*10000 + month\*100 + day in month.
- “mddyy”: month\*10000 + day in month\*100 + two-digit year.
- “mddyyyy”: month\*100000 + day in month\*10000 + four-digit year.
- “ddmmyy”: day in month\*10000 + month\*100 + two-digit year.
- “ddmmYYYY”: day in month\*1000000 + month\*10000 + four-digit year.

The following formats are supported when converting a single number into time:

- “hh”: hour in day (in 24 hour units)
- “hhmi”: hour\*100 + minute.
- “hhmiss”: hour\*10000 + minute\*100 + seconds.

Note that the `@MAKEDATE` format strings are not case sensitive, since the function requires that *all* non-leading fields must have leading zeros where appropriate. For exam-

---

ple, when using the format “YYYYMMDD”, the date March 1, 1992 must be encoded as 19920301, and not 199231, 1992031, or 1992301.

Let us consider some specific examples of @MAKEDATE conversion of a single number. You may convert a numeric value for the year into a date number using a format string to describe the year. The expressions:

```
@makedate(1999, "yyyy")
@makedate(99, "yy")
```

both return the date number 729754.0 corresponding to 12 midnight on January 1, 1999. Similarly, you may convert a numeric value into the number of hours in a day using expressions of the form,

```
@makedate(12, "hh")
```

Here, EViews will return the date value 0.5 corresponding to 12 noon on January 1, A.D. 1. While this particular date value is not intrinsically of interest, it may be combined with other date values to obtain the value for a specific hour in a particular day. For example using date arithmetic, we may add the 0.5 to the 729754.0 value (12 midnight, January 1, 1999) obtained above, yielding the date value for 12 noon on January 1, 1999. We consider these sorts of operations in greater detail in [“Manipulating Date Numbers” on page 142](#).

If your number contains “packed” date information, you may interpret the various components using @MAKEDATE with an appropriate format string. For example,

```
@makedate(199003, "yyyymm")
@makedate(1990.3, "yyyy.mm")
@makedate(1031990, "ddmmmyyy")
@makedate(30190, "mmddyy")
```

all return the value 726526.0, representing March 1, 1990.

Cases where @MAKEDATE is used to convert more than one argument into a date or time are more limited and slightly more complex. The arguments must be listed from the lowest frequency to the highest, with the first field representing either the year or the hour, and the remaining fields representing sub-periods. The valid date format strings for the multiple argument @MAKEDATE are a subset of the date format strings, with components applied sequentially to the numeric arguments:

- “yy s” or “yyyy s”: two or four-digit year and half-year.
- “yy q” or “yyyy q”: year and quarter.
- “yy mm” or “yyyy mm”: year and month.
- “yy ddd” or “yyyy ddd”: year and day in year.

- “yy mm dd” or “yyyy mm dd”: year, month, and day in month.

Similarly, the valid formats for converting multiple numeric values into a time are:

- “hh mi”: hour\*100 + minute.
- “hh mi ss”: hour\*10000 + minutes\*100 + seconds.

For convenience, the non-space-delimited forms of these format strings are also supported (e.g., “yymm”, and “hhmi”).

For example, the expressions,

```
@makedate(97, 12, 3, "yy mm dd")  
@makedate(1997, 12, 3, "yyyymmdd")
```

will return the value 729360.0 corresponding to midnight on December 3, 1997. You may provide a subset of this information so that

```
@makedate(97, 12, "yymm")
```

returns the value 729358.0 representing the earliest date and time in December of 1997 (12 midnight, December 1, 1997). Likewise,

```
@makedate(1997, 37, "yyyy ddd")
```

yields the value 729060.0 (February 6, 1997, the 37th day of the year) and

```
@makedate(14, 25, 10, "hh mi ss")
```

returns the value 0.600810185 corresponding to 2:25:10 pm on January 1, A.D. 1.

It is worth pointing out that in the examples above, the numeric arguments are entered from lowest frequency to high, as required. The following example, in which days appear before months and years, is *not* a legal specification

```
@makedate(7, 10, 98, "dd mm yy")
```

and will generate an error reporting a “Bad date format”.

Lastly, we note that limitations on the date formats supported by @MAKEDATE imply that in some cases, you are better off working with strings and the @DATEVAL function. In cases, where @MAKEDATE does not support a desired conversion, you should consider converting your numbers into strings, performing string concatenation, and then using the richer set of @DATEVAL conversions to obtain the desired date values.

## Manipulating Date Numbers

One of the most important reasons for holding your date information in the form of date numbers is so that you may perform sophisticated calendar operations.

## Date Operators

Since date values are simply double precision numbers, you may perform standard mathematical operations using these values. While many operations such as division and multiplication do not preserve the notion of a date number, you may find addition and subtraction and relational comparison of date values to be useful tools.

If, for example, you add 7 to a valid date number, you get a value corresponding to the same time exactly seven days later. Adding 0.25 adds one-quarter of a day (6 hours) to the current time. Likewise, subtracting 1 gives the previous day, while subtracting 0.5 gives the date value 12 hours earlier. Taking the difference between two date values yields the number of days between the two values.

While using the addition and subtraction operators is valid, we strongly encourage you to use the EViews specialized date functions since they allow you to perform arithmetic at various frequencies (other than days), while taking account of irregularities in the calendar (see “[Functions for Manipulating Dates](#)” on page 143).

Similarly, while you may round a date number down to the nearest integer to obtain the first instance in the day, or you may round down to a given precision to find the first instance in a month, quarter or year, the built-in functions provide a set of simple, yet powerful tools for working with dates.

Note further that all of the relational operators are valid for comparing date numbers. Thus, if two date numbers are equal, the “=”, “> =”, and “< =” relational operators all return a 1, while the “< >”, “>”, and “<” comparison operators return a 0. If two date numbers are not equal, “< >” returns a 1 and “=” returns a 0. If the first date number is less than a second date number, the corresponding first date precedes the second in calendar time.

## Functions for Manipulating Dates

EViews provides several functions for manipulating dates that take date numbers as input and return numeric values that are also date numbers. These functions may be used when you wish to find a new date value associated with a given date number, for example, a date number 3 months before or 37 weeks after a given date and time.

The functions described below all take a *time unit string* as an argument. As the name suggests, a time unit string is a character representation for a unit of time, such as a month or a year. The valid time unit string values are: “A” or “Y” (annual), “S” (semi-annual), “Q” (quarters), “MM” (months), “WW” (weeks), “DD” (days), “B” (business days), “HH” (hours), “MI” (minutes), “SS” (seconds).

There are three primary functions for manipulating a date number:

- `@DATEADD(date1, offset[, u])`: returns the date number given by *date1* offset by *offset* time units as specified by the time unit string *u*. If no time unit is specified, EViews will use the workfile regular frequency, if available.

Suppose that the value of *date1* is 730088.0 (midnight, December 1, 1999). Then we can add and subtract 10 days from the date by using the functions

```
@dateadd(730088.0, 10, "dd")
@dateadd(730088.0, -10, "dd")
```

which return 730098.0 (December 11, 1999) and (730078.0) (November 21, 1999). Note that these results could have been obtained by taking the original numeric value plus or minus 10.

The `@DATEADD` function allows for date offsets specified in various units of time. For example, to add 5 weeks to the existing date, simply specify “W” or “WW” as the time unit string, as in

```
@dateadd(730088.0, 5, "ww")
```

which returns 730123.0 (January 5, 2000).

- `@DATEDIFF(date1, date2[, u])`: returns the number of time units between *date1* and *date2*, as specified by the time unit string *u*. If no time unit is specified, EViews will use the workfile regular frequency, if available.

Suppose that *date1* is 730088.0 (December 1, 1999) and *date2* is 729754.0 (January 1, 1999), then,

```
@datediff(730088.0, 729754.0, "dd")
```

returns 334 for the number of days between the two dates. Note that this is result is simply the difference between the two numbers.

The `@DATEDIFF` function is more powerful in that it allows us to calculate differences in various units of time. For example, the expressions

```
@datediff(730088.0, 729754.0, "mm")
@datediff(730088.0, 729754.0, "ww")
```

return 11 and 47 for the number of months and weeks between the dates.

- `@DATEFLOOR(date1, u[, step])`: finds the first possible date number in the given time unit, as in the first possible date value in the current quarter, with an optional step offset.

Suppose that *date1* is 730110.5 (12 noon, December 23, 1999). Then the `@DATEFLOOR` values

```
@datefloor(730110.5, "dd")
@datefloor(730110.5, "mm")
```

yield 730110.0 (midnight, December 23, 1999) and 730088.0 (midnight, December 1, 1999), since those are the first possible date values in the current day and month. Note that the first value is simply the integer portion of the original date number, but that the latter required more complex analysis of the calendar.

Likewise, we can find the start of any corresponding period by using different time units:

```
@datefloor(730098.5, "q")
@datefloor(730110.5, "y", 1)
```

returns 730027.0 (midnight, October 1, 1999), and 729754.0 (midnight, January 1, 2000). Notice that since the latter example used an offset value of 1, the result corresponds to the first date value for the year 2000, which is the start of the following year.

## Extracting Information from Date Numbers

Given a date number you may wish to extract numeric values associated with a portion of the value. For example, you might wish to know the value of the month, the year, or the day in the year associated with a given date value. EViews provides the @DATEPART function to allow you to extract the desired information.

- `@DATEPART(date1, u)`: returns a numeric part of a date value given by *u*, where *u* is a time unit string.

Consider the *date1* date value 730110.5 (noon, December 23, 1999). The @DATEPART values for

```
@datepart(730110.5, "dd")
@datepart(730110.5, "w")
@datepart(730110.5, "ww")
@datepart(730110.5, "mm")
@datepart(730110.5, "yy")
```

are 23 (day of the month), 4 (week in the month), 52 (week in the year), 12 (month in the year), and 1999 (year), respectively.

Note that the numeric values returned from @DATEPART are not themselves date values, but may be used with @MAKEDATE to create date values.

## Special Date Functions

In addition to the functions that convert strings or numbers into date values, EViews provides the following special ways to obtain one or more date values of interest.

- `@NOW`: returns the date number associated with the current time.

- @DATE: returns the date number corresponding to every observation in the current workfile.
- @YEAR: returns the four digit year in which the current observation begins. It is equivalent to “@DATEPART(@DATE, "YYYY")”.
- @QUARTER: returns the quarter of the year in which the current observation begins. It is equivalent to “@DATEPART(@DATE, "Q")”.
- @MONTH: returns the month of the year in which the current observation begins. It is equivalent to “@DATEPART(@DATE, "MM")”.
- @DAY: returns the day of the month in which the current observation begins. It is equivalent to “@DATEPART(@DATE, "DD")”.
- @WEEKDAY: returns the day of the week in which the current observation begins, where Monday is given the number 1 and Sunday is given the number 7. It is equivalent to “@DATEPART(@DATE, "W")”.
- @STRDATE(*fmt*): returns the set of workfile row dates as strings, using the date format string *fmt*. See [“Date Formats” on page 130](#) for a discussion of date format strings.

The @DATE function will generally be used to create a series containing the date value associated with every observation, or as part of a series expression involving date manipulation. For example:

```
series y = @date
series x = @dateadd(@date, 12, "ww")
```

which generates a series containing the date values for every observation, and the date values for every observation 12 weeks from the current values.

@STRDATE should be used when you wish to obtain the date string associated with every observation in the workfile—for example, to be used as input to an alpha series. It is equivalent to using the @DATESTR function on the date number associated with every observation in the workfile.

## Free-format Conversion Formats

EViews supports the free-format conversion of a wide variety of date strings in which the string is analyzed for the most likely corresponding date.

Any of the following date string types are allowed:

### Day, month, year

- “"YYYY-MM-DD"” (IEEE, with the date enclosed in double quotes)

- “dd/mm/yy” (if American, “mm/dd/yy” instead)
- “dd/mm/yyyy” (if American, “mm/dd/yyyy” instead)
- “yyyy/mm/dd”
- “dd/month/yy”
- “dd/month/yyyy”
- “yyyy/month/dd”
- “ddmmyy” (if American, “mmddyy”)
- “ddmmmyyyy” (if American, “mmddyyyy”)

The resulting date values correspond to the first instance in the day (12 midnight).

#### Month in year

- “mon/yy”
- “mon/yyyy”
- “yy/month”
- “yyyy/month”

The results are rounded to the first instance in the month (12 midnight of the first day of the month).

#### Period in year

- “yyyy[S|Q|M|W|B|D|T|F:]period”
- “yy[S|Q|M|W|B|D|T|F:]period”

The date value is rounded to the first instance in the period in the year

#### Whole year

- “yyyy[A]”. The “A” is generally optional, but required if current WF is undated.
- “yy[A]”. The “A” is generally optional, but required if current WF is undated.

The date value is rounded to the first instance in the year (12 midnight on January 1).

### Free-format Conversion Details

Note that the following conventions may be used in interpreting ambiguous free-form dates:

## Dates and Date Intervals

A date in EViews is generally taken to represent a single point in calendar time. In some contexts, however, a date specification is used to refer to a range of values contained in a time, which can be referred to as an interval.

When a date specification is treated as an interval, the precision with which the date is specified is used to determine the duration of the interval. For example, if a full day specification is provided, such as “Oct 11 1980”, then the interval is taken to run from midnight at the beginning of the day to just before midnight at the end of the day. If only a year is specified, such as “1963”, then the interval is taken to run from midnight on the 1st of January of the year to just before midnight on the 31st of December at the end of the year.

An example where this is used is in setting the sample for a workfile. In this context, pairs of dates are provided to specify which observations in the workfile should be included in the sample. The pairs of dates are provided are processed as intervals, and the sample is defined to run from the start of the first interval to the end of the second interval. As an example, if the sample “1980q2 1980q2” is specified for a daily file, the sample will include all observations from April 1st 1980 to June 30th 1980 inclusive.

## Incomplete Date Numbers

An EViews date number can be used to represent both a particular calendar day, and a particular time of day within that day. If no time of day is specified, the time of day is set to midnight at the beginning of the day.

When no date is specified, the day portion of a date is effectively set to 1st Jan A.D. 1. For example, the date string “12 p.m.” will be translated to the date value 0.5 representing 12 noon on January 1, A.D. 1. While this particular date value is probably not of intrinsic interest, it may be combined with other information to obtain meaningful values. See [“Manipulating Date Numbers” on page 142](#)

## Two-digit Years

In general, EViews interprets years containing only two digits as belonging to either the twentieth or twenty-first centuries, depending on the value of the year. If the two digit year is greater than or equal to 30, the year is assumed to be from the twentieth century and a century prefix of “19” is added to form a four digit year. If the number is less than 30, the year is assumed to be from the twenty first century and a century prefix of “20” is added to form a four digit year.

Note that if you wish to refer to a year after 2029 or a year before 1930, you must use the full four-digit year identifier.

Because this conversion to four digit years is generally performed automatically, it is not possible to specify years less than A.D. 100 using two digit notation. Should the need ever

arise to represent these dates, such two digit years can be input directly by specifying the year as a four digit year with leading zeros. For example, the 3rd of April in the year A.D. 23 can be input as “April 3rd 0023”.

### Implicit Period Notation

In implicit period notation (*e.g.*, “1990:3”), the current workfile frequency is used to determine the period.

### American vs. European dates

When performing a free-format conversion in the absence of contextual information sufficient to identify whether data are provided in “mm/dd” or “dd/mm” format, the global workfile setting for the **Options/Dates & Frequency Conversion.../Month/Day order in dates** (“[Dates & Frequency Conversion](#) on page 915”) will be used to determine the ordering of the days and months in the string.

For example, the order of the months and years is ambiguous in the date pair:

```
1/3/91 7/5/95
```

so EViews will use the default date settings to determine the desired ordering. We caution you, however, that using default settings to define the interpretation of date strings is not a good idea since a given date string may be interpreted in different ways at different times if your settings change. You may instead use the IEEE standard format, “YYYY-MM-DD” to ensure consistent interpretation of your daily date strings. The presence of a dash in the format means that you must enclose the date in quotes for EViews to accept this format. For example:

```
smp1 "1991-01-03" "1995-07-05"
```

will always set the sample to run from January 3, 1991 and July 5, 1995.

### Time of Day

Free-format dates can also contain optional trailing time of day information which must follow the pattern:

```
hh[[[[::mi:]ss].ss]ss][am|AM|pm|PM]
```

where “[ ]” encloses optional portions of the format and “|” indicates one of a number of possibilities. In addition, either the “am” or “pm” field or an explicit minute field must be provided for the input to be recognized as a time. An hour by itself is generally not sufficient.

The time of day in an EViews date is accurate up to a particular millisecond within the day, although any date can always be displayed at a lower precision. When displaying times at

a lower precision, the displayed times are always rounded down to the requested precision, and never rounded up.

When both a day and a time of day are specified as part of a date, the two can generally be provided one after the other with the two fields separated by one or more spaces. If, however, the date is being used in a context where EViews does not permit spaces between input fields, a single letter “t” can also be used to separate the day and time so that the entire date can be contained in a single word, e.g. “1990-Jan-03T09:53”.

## Date Function Summary

### Date Functions

- @dateadd** ..... add to a date ([p. 565](#)).
- @datediff** ..... computes difference between dates ([p. 566](#)).
- @datefloor** ..... rounds date down to start of period([p. 566](#)).
- @datepart** ..... extracts part of date ([p. 567](#)).
- @datestr** ..... converts a date number into a string ([p. 567](#)).
- @dateval** ..... converts a string into a date number ([p. 568](#)).
- @dtoo**..... returns observation number corresponding to the date specified by a string ([p. 569](#)).
- @makedate** ..... converts numeric values into a date number ([p. 573](#)).
- @now**..... returns the date number associated with the current time ([p. 574](#)).
- @otod**..... returns a string associated with a the date or observation value ([p. 575](#)).
- @strdate** ..... returns string corresponding to each element in workfile ([p. 577](#)).
- @strnow** ..... returns a string representation of the current date ([p. 577](#)).

## Appendix A. Object, View and Procedure Reference

---

The following is a reference guide to the views, procedures, and data members for each of the objects found in EViews:

 Alpha (p. 153)	 Model (p. 168)	 Sym (p. 180)
 Coef (p. 153)	 Pool (p. 169)	 System (p. 182)
 Equation (p. 155)	 Rowvector (p. 172)	 Table (p. 185)
 Graph (p. 159)	 Sample (p. 173)	 Text (p. 186)
 Group (p. 161)	 Scalar (p. 174)	 Valmap (p. 187)
 Logl (p. 164)	 Series (p. 175)	 Var (p. 187)
 Matrix (p. 166)	 Sspace (p. 177)	 Vector (p. 190)

To use these views, procedures, and data members, you should list the name of the object followed by a period, and then enter the name of the view, procedure, or data member, along with any options or arguments:

```
object_name.view_name(options) arguments  
object_name.proc_name(options) arguments  
object_name.data_member
```

For example, to display the line graph view of the series object CONS, you can enter the command:

```
cons.line
```

To perform a dynamic forecast using the estimates in the equation object EQ1, you may enter:

```
eq1.forecast y_f
```

To save the coefficient covariance matrix from EQ1, you can enter:

```
sym cov1=eq1.@coefcov
```

Each of the views and procedures is documented more fully in the alphabetical listing found in [Appendix B, “Command Reference”, on page 193](#).

## Alpha

Alpha (alphanumeric) series. An EViews alpha series contains a set of observations on a variable containing string values.

### Alpha Declaration

- `alpha` ..... declare alpha series ([p. 201](#)).
- `frm1` ..... create alpha series object with a formula for auto-updating ([p. 293](#)).
- `genr` ..... create alpha or numeric series object ([p. 295](#)).

To declare an alpha series, use the keyword `alpha`, followed by a name, and optionally, by an “=” sign and a valid series expression:

```
alpha y  
alpha x = "initial strings"
```

If there is no assignment, the series will be initialized to contain blank values, “”.

### Alpha Views

- `freq` ..... one-way tabulation ([p. 291](#)).
- `label` ..... label information for the alpha ([p. 317](#)).
- `sheet` ..... spreadsheet view of the alpha ([p. 444](#)).

### Alpha Procs

- `displayname` ..... set display name ([p. 266](#)).
- `map` ..... assign or remove value map setting ([p. 349](#)).
- `setformat` ..... set the display format for the alpha series spreadsheet ([p. 432](#)).
- `setindent` ..... set the indentation for the alpha series spreadsheet ([p. 437](#)).
- `setjust` ..... set the justification for the alpha series spreadsheet ([p. 438](#)).
- `setwidht` ..... set the column width for alpha series spreadsheet ([p. 444](#)).

### Alpha Data Members

- (i) ..... *i*-th element of the alpha series from the beginning of the workfile (when used on the left-hand side of an assignment, or when the element appears in a table or string variable assignment).

### Alpha Element Functions

- `@elem(ser, j)` ..... function to access the *j*-th observation of the alpha series, where *j* identifies the date or observation.

### Alpha Examples

```
alpha val = "initial string"
```

initializes an alpha series VAL using a string literal.

If FIRST is an alpha series containing first names, and LAST is an alpha containing last names, then:

```
alpha name = first + " " + last
```

creates an alpha series containing the full names.

## Coef

Coefficient vector. Coefficients are used to represent the parameters of equations and systems.

### Coef Declaration

`coef` .....declare coefficient vector ([p. 238](#)).

There are two ways to create a coef. First, enter the `coef` keyword, followed by a name to be given to the coefficient vector. The dimension of the coef may be provided in parentheses after the keyword:

```
coef alpha  
coef(10) beta
```

If no dimension is provided, the resulting coef will contain a single element.

You may also combine a declaration with an assignment statement. If you do not provide an explicit assignment statement, new coeffs are initialized to zero.

See also [param \(p. 383\)](#) for information on initializing coefficients, and the entries for each of the estimation objects ([equation](#), [pool](#), [sspace](#), [system](#), and [var](#)) for additional methods of accessing coefficients.

### Coef Views

`area` .....area graph of the coefficient vector ([p. 207](#)).

`bar` .....bar graph of coefficient vector plotted against the coefficient index ([p. 215](#)).

`label` .....label view ([p. 317](#)).

`line` .....line graph of coefficient vector plotted against the coefficient index ([p. 320](#)).

`sheet` .....spreadsheet view of the coefficient ([p. 444](#)).

`spike` .....spike graph ([p. 455](#)).

`stats`.....descriptive statistics ([p. 462](#)).

## Coef Procs

**displayname** ..... set display name ([p. 266](#)).  
**fill** ..... fill the elements of the coefficient vector ([p. 282](#)).  
**read** ..... import data into coefficient vector ([p. 391](#)).  
**setformat** ..... set the display format for the coefficient vector spreadsheet  
                 ([p. 432](#)).  
**setindent** ..... set the indentation for the coefficient spreadsheet ([p. 437](#)).  
**setjust** ..... set the justification for the coefficient spreadsheet ([p. 438](#)).  
**setWidth** ..... set the column width for the coefficient spreadsheet ([p. 444](#)).  
**write** ..... export data from coefficient vector ([p. 517](#)).

## Coef Data Members

(i) ..... *i*-th element of the coefficient vector. Simply append “(i)” to the matrix name (without a “.”).

## Coef Examples

The coefficient vector declaration:

```
coef(10) coef1=3
```

creates a 10 element coefficient vector COEF1, and initializes all values to 3.

Suppose MAT1 is a  $10 \times 1$  matrix, and VEC1 is a 20 element vector. Then:

```
coef mycoef1=coef1
coef mycoef2=mat1
coef mycoef3=vec1
```

create, size, and initialize the coefficient vectors MYCOEF1, MYCOEF2 and MYCOEF3.

Coefficient elements may be referred to by an explicit index. For example:

```
vector(10) mm=beta(10)
scalar shape=beta(7)
```

fills the vector MM with the value of the tenth element of BETA, and assigns the seventh value of BETA to the scalar SHAPE.

## Equation

Equation object. Equations are used for single equation estimation, testing, and forecasting.

### Equation Declaration

**equation** .....declare equation object ([p. 275](#)).

To declare an equation object, enter the keyword **equation**, followed by a name:

```
equation eq01
```

and an optional specification:

```
equation r4cst.ls r c r(-1) div  
equation wcd.ls q=c(1)*n^c(2)*k^c(3)
```

### Equation Methods

**arch** .....autoregressive conditional heteroskedasticity (ARCH and GARCH) ([p. 203](#)).

**binary** .....binary dependent variable models (includes probit, logit, gompit) models ([p. 217](#)).

**censored** .....censored and truncated regression (includes tobit) models ([p. 233](#)).

**count** .....count data modeling (includes poisson, negative binomial and quasi-maximum likelihood count models) ([p. 248](#)).

**gmm** .....generalized method of moments ([p. 297](#)).

**logit** .....logit (binary) estimation ([p. 328](#)).

**ls** .....linear and nonlinear least squares regression (includes weighted least squares and ARMAX) models ([p. 329](#)).

**ordered** .....ordinal dependent variable models (includes ordered probit, ordered logit, and ordered extreme value models) ([p. 360](#)).

**probit** .....probit (binary) estimation ([p. 388](#)).

**tsls** .....linear and nonlinear two-stage least squares (TSLS) regression models (includes weighted TSLS, and TSLS with ARMA errors) ([p. 487](#)).

### Equation Views

**archtest** .....LM test for the presence of ARCH in the residuals ([p. 206](#)).

**arma** .....Examine ARMA structure of estimated equation ([p. 210](#)).

**auto** .....Breusch-Godfrey serial correlation Lagrange Multiplier (LM) test ([p. 212](#)).

**cellipse** ..... Confidence ellipses for coefficient restrictions ([p. 231](#)).  
**chow** ..... Chow breakpoint and forecast tests for structural change ([p. 236](#)).  
**coefcov** ..... coefficient covariance matrix ([p. 239](#)).  
**correl** ..... correlogram of the residuals ([p. 246](#)).  
**correlnsq** ..... correlogram of the squared residuals ([p. 247](#)).  
**derivs** ..... derivatives of the equation specification ([p. 264](#)).  
**garch** ..... conditional standard deviation graph (only for equations estimated using ARCH) ([p. 295](#)).  
**grads** ..... examine the gradients of the objective function ([p. 302](#)).  
**hist** ..... histogram and descriptive statistics of the residuals ([p. 308](#)).  
**label** ..... label information for the equation ([p. 317](#)).  
**means** ..... descriptive statistics by category of the dependent variable (only for binary, ordered, censored and count equations) ([p. 350](#)).  
**output** ..... table of estimation results ([p. 361](#)).  
**predict** ..... prediction (fit) evaluation table (only for binary and ordered equations) ([p. 387](#)).  
**representations** ..... text showing specification of the equation ([p. 395](#)).  
**reset** ..... Ramsey's RESET test for functional form ([p. 397](#)).  
**resids** ..... display, in tabular form, the actual and fitted values for the dependent variable, along with the residuals ([p. 399](#)).  
**results** ..... table of estimation results ([p. 400](#)).  
**rls** ..... recursive residuals least squares (only for non-panel equations estimated by ordinary least squares, without ARMA terms) ([p. 400](#)).  
**testadd** ..... likelihood ratio test for adding variables to equation ([p. 474](#)).  
**testdrop** ..... likelihood ratio test for dropping variables from equation ([p. 477](#)).  
**testfit** ..... performs Hosmer and Lemeshow and Andrews goodness-of-fit tests (only for equations estimated using binary) ([p. 479](#)).  
**wald** ..... Wald test for coefficient restrictions ([p. 502](#)).  
**white** ..... White test for heteroskedasticity ([p. 514](#)).

### Equation Procs

**displayname** ..... set display name ([p. 266](#)).  
**fit** ..... static forecast ([p. 285](#)).  
**forecast** ..... dynamic forecast ([p. 287](#)).  
**makederivs** ..... make group containing derivatives of the equation specification ([p. 335](#)).

**makegarch** ..... create conditional variance series (only for ARCH equations)  
[\(p. 336\)](#).

**makegrads** ..... make group containing gradients of the objective function  
[\(p. 337\)](#).

**makelimits** ..... create vector of estimated limit points (only for ordered models)  
[\(p. 341\)](#).

**makemodel** ..... create model from estimated equation [\(p. 341\)](#).

**makeregs** ..... make group containing the regressors [\(p. 342\)](#).

**makeresids** ..... make series containing residuals from equation [\(p. 342\)](#).

**updatecoefs** ..... update coefficient vector(s) from equation [\(p. 493\)](#).

## Equation Data Members

### Scalar Values

**@aic** ..... Akaike information criterion.

**@cofcov(i,j)** ..... covariance of coefficient estimates  $i$  and  $j$ .

**@coefs(i)** .....  $i$ -th coefficient value.

**@dw** ..... Durbin-Watson statistic.

**@f** .....  $F$ -statistic.

**@hq** ..... Hannan-Quinn information criterion.

**@jstat** .....  $J$ -statistic — value of the GMM objective function (for GMM).

**@logl** ..... value of the log likelihood function.

**@meandep** ..... mean of the dependent variable.

**@ncoef** ..... number of estimated coefficients.

**@r2** ..... R-squared statistic.

**@rbar2** ..... adjusted R-squared statistic.

**@regobs** ..... number of observations in regression.

**@schwarz** ..... Schwarz information criterion.

**@sddep** ..... standard deviation of the dependent variable.

**@se** ..... standard error of the regression.

**@ssr** ..... sum of squared residuals.

**@stderrs(i)** ..... standard error for coefficient  $i$ .

**@tstats(i)** .....  $t$ -statistic value for coefficient  $i$ .

**c(i)** .....  $i$ -th element of default coefficient vector for equation (if applicable).

### Vectors and Matrices

**@cofcov** ..... covariance matrix for coefficient estimates.

@coefs..... coefficient vector.  
@stderrs ..... vector of standard errors for coefficients.  
@tstats ..... vector of *t*-statistic values for coefficients.

## Equation Examples

To apply an estimation method (proc) to an existing equation object:

```
equation ifunc  
ifunc.ls r c r(-1) div
```

To declare and estimate an equation in one step, combine the two commands:

```
equation value.tsls log(p) c d(x) @ x(-1) x(-2)  
equation drive.logit ifdr c owncar dist income  
equation countmod.count patents c rdd
```

To estimate equations by list, using ordinary and two-stage least squares:

```
equation ordinary.ls log(p) c d(x)  
equation twostage.tsls log(p) c d(x) @ x(-1) x(-2)
```

You can create and use other coefficient vectors:

```
coef(10) a  
coef(10) b  
equation eq01.ls y=c(10)+b(5)*y(-1)+a(7)*inc
```

The fitted values from EQ01 may be saved using,

```
series fit = eq01.@coefs(1) + eq01.@coefs(2)*y(-1) +  
eq01.@coefs(3)*inc
```

or by issuing the command:

```
eq01.fit fitted_vals
```

To perform a Wald test:

```
eq01.wald a(7)=exp(b(5))
```

You can save the *t*-statistics and covariance matrix for your parameter estimates:

```
vector eqstats=eq01.@tstats  
matrix eqcov=eq01.@coeffcov
```

## Graph

Graph object. Specialized object used to hold graphical output.

### Graph Declaration

- freeze** .....freeze graphical view of object ([p. 290](#)).
- graph** .....create graph object using graph command or by merging existing graphs ([p. 303](#)).

Graphs may be created by declaring a graph using one of the graph commands described below, or by freezing the graphical view of an object. For example:

```
graph myline.line ser1  
graph myscat.scat ser1 ser2  
graph myxy.xyline grp1
```

declare and create the graph objects MYLINE, MYSCAT and MYXY. Alternatively, you can use the `freeze` command to create graph objects:

```
freeze(myline) ser1.line  
group grp2 ser1 ser2  
freeze(myscat) grp2.scat  
freeze(myxy) grp1.xyline
```

which are equivalent to the declarations above.

### Graph Type Commands

- area** .....area graph ([p. 207](#)).
- bar** .....bar graph ([p. 215](#)).
- errbar** .....error bar graph ([p. 276](#)).
- hilo** .....high-low(-open-close) graph ([p. 307](#)).
- line** .....line-symbol graph ([p. 320](#)).
- pie** .....pie chart ([p. 384](#)).
- scat** .....scatterplot ([p. 412](#)).
- spike** .....spike graph ([p. 455](#)).
- xy** .....XY graph with one or more X plotted against one or more Y ([p. 528](#)).
- xyline** .....XY line graph ([p. 530](#)).
- xypair** .....XY pairs graph ([p. 528](#)).

### Graph View

- label** .....label information for the graph ([p. 317](#)).

## Graph Procs

**addtext**..... place arbitrary text on the graph ([p. 199](#)).  
**align**..... align the placement of multiple graphs ([p. 200](#)).  
**axis**..... sets axis display characteristics for the graph ([p. 213](#)).  
**bplabel**..... specify labeling of bottom-axis in boxplots ([p. 226](#)).  
**datelabel**..... controls labeling of the bottom date/time axis in time plots ([p. 252](#)).  
**dates**..... controls labeling of the bottom date/time axis in time plots ([p. 254](#)).  
**draw**..... draw lines and shaded areas on the graph ([p. 267](#)).  
**legend**..... control the appearance and placement of legends ([p. 319](#)).  
**merge**..... merge graph objects ([p. 351](#)).  
**metafile**..... save graph to a Windows metafile ([p. 352](#)).  
**name**..... change the series name for legends or axis labels ([p. 354](#)).  
**options**..... change the option settings of the graph ([p. 358](#)).  
**save**..... save graph to a Windows metafile or PostScript file ([p. 407](#)).  
**scale**..... set the axis scaling for the graph ([p. 410](#)).  
**setbpelem**..... set options for element of a boxplot graph ([p. 421](#)).  
**setelem**..... set individual line, bar and legend options for each series in the graph ([p. 426](#)).  
**template**..... use template graph ([p. 473](#)).

## Graph Examples

You can declare your graph:

```
graph abc.xyline(m) unemp gnp inf  
graph bargraph.bar(d,1)unemp gnp
```

Alternately, you may freeze any graphical view:

```
freeze (mykernel) ser1.kdensity
```

You can change the graph type,

```
graph mygraph.line ser1  
mygraph.hist
```

or combine multiple graphs:

```
graph xyz.merge graph1 graph2
```

## Group

Group of series. Groups are used for working with collections of series objects (series, alphas, links).

### Group Declaration

**group**.....create a group object ([p. 304](#)).

To declare a group, enter the keyword `group`, followed by a name, and optionally, a list of series or expressions:

```
group salesvrs  
group nipa cons(-1) log(inv) g x
```

Additionally, a number of object procedures will automatically create a group.

### Group Views

**area** .....area graph of the series in the group ([p. 207](#)).  
**bar** .....single or multiple bar graph view of all series ([p. 215](#)).  
**boxplot** .....boxplot of each series in the group ([p. 219](#)).  
**cause** .....pairwise Granger causality tests ([p. 228](#)).  
**cdfplot** .....distribution (cumulative, survivor, quantile) graphs ([p. 230](#)).  
**coint** .....Johansen cointegration test ([p. 240](#)).  
**cor** .....correlation matrix between series ([p. 246](#)).  
**correl** .....correlogram of the first series in the group ([p. 246](#)).  
**cov** .....covariance matrix between series ([p. 250](#)).  
**cross** .....cross correlogram of the first two series ([p. 251](#)).  
**dtable** .....dated data table ([p. 270](#)).  
**errbar** .....error bar graph view ([p. 276](#)).  
**freq** .....frequency table  $n$ -way contingency table ([p. 291](#)).  
**hilo** .....high-low(-open-close) chart ([p. 307](#)).  
**kerfit** .....scatter of the first series against the second series with kernel fit ([p. 316](#)).  
**label** .....label information for the group ([p. 317](#)).  
**line** .....single or multiple line graph view of all series ([p. 320](#)).  
**linefit** .....scatter of the first series against the second series with regression line ([p. 322](#)).  
**nnfit** .....scatter of the first series against the second series with nearest neighbor fit ([p. 355](#)).

- pie** ..... pie chart view ([p. 384](#)).  
**pcomp** ..... principal components analysis ([p. 383](#)).  
**qqplot** ..... quantile-quantile plots ([p. 389](#)).  
**scat** ..... single scatter diagram of the series in the group ([p. 412](#)).  
**scatmat** ..... matrix of all pairwise scatter plots ([p. 414](#)).  
**sheet** ..... spreadsheet view of the series in the group ([p. 444](#)).  
**spike** ..... spike graph ([p. 455](#)).  
**stats** ..... descriptive statistics ([p. 462](#)).  
**testbtw** ..... tests of equality for mean, median, or variance, between series in group ([p. 475](#)).  
**uroot** ..... unit root test on the series in the group ([p. 494](#)).  
**xy** ..... XY graph with one or more X series plotted against one or more Y ([p. 528](#)).  
**xyline** ..... XY line graph ([p. 530](#)).  
**xypair** ..... XY pairs graph ([p. 528](#)).

### Group Procs

- add** ..... add one or more series to the group ([p. 196](#)).  
**displayname** ..... set display name ([p. 266](#)).  
**drop** ..... drop one or more series from the group ([p. 270](#)).  
**resample** ..... resample from rows of group ([p. 395](#)).  
**setformat** ..... set the display format in the group spreadsheet for the specified series ([p. 432](#)).  
**setindent** ..... set the indentation in the group spreadsheet for the specified series ([p. 437](#)).  
**setjust** ..... set the justification in the group spreadsheet for the specified series ([p. 438](#)).  
**setWidth** ..... set the column width in the group spreadsheet for the specified series ([p. 444](#)).

### Group Data Members

- (i) ..... *i*-th series in the group. Simply append “(i)” to the group name (without a “.”).  
**@comobs** ..... number of observations in the current sample for which each series in the group has a non-missing value (observations in the common sample).  
**@count** ..... number of series in the group.

@minobs ..... number of non-missing observations in the current sample for the shortest series in the group.  
@maxobs ..... number of non-missing observations in the current sample for the the longest series in the group.  
@seriesname(i) .... string containing the name of the *i*-th series in the group.

## Group Examples

To create a group G1, you may enter:

```
group g1 gdp income
```

To change the contents of an existing group, you can repeat the declaration, or use the add and drop commands:

```
group g1 x y  
g1.add w z  
g1.drop y
```

The following commands produce a cross-tabulation of the series in the group, display the covariance matrix, and test for equality of variance:

```
g1.freq  
g1.cov  
g1.testbtw(var,c)
```

You can index selected series in the group:

```
show g1(2).line  
series sum=g1(1)+g1(2)
```

To create a scalar containing the number of series in the group, use the command:

```
scalar nsers=g1.@count
```

## Link

Link object. Series or alpha link used to frequency converted or match merge data from another workfile page.

Once created, links may be used just like “[Series](#)” (p. 175) or “[Alpha](#)” (p. 152) objects.

### Link Declaration

[link](#) ..... link object declaration (p. 323).

To declare a link object, enter the keyword `link`, followed by a name:

```
link newser
```

and an optional link specification:

```
link altser.linkto(c=obs,nacat) indiv::x @src ind1 ind2 @dest  
ind1 ind2
```

### Link Procs

[linkto](#)..... specify link object definition ([p. 324](#)).

## Logl

Likelihood object. Used for performing maximum likelihood estimation of user-specified likelihood functions.

### Logl Declaration

[logl](#)..... likelihood object declaration ([p. 328](#)).

To declare a logl object, use the `logl` keyword, followed by a name to be given to the object.

### Logl Method

[ml](#)..... maximum likelihood estimation ([p. 352](#)).

### Logl Views

[append](#)..... add line to the specification ([p. 202](#)).

[cellipse](#)..... Confidence ellipses for coefficient restrictions ([p. 231](#)).

[checkderivs](#)..... compare user supplied and numeric derivatives ([p. 235](#)).

[coefcov](#)..... coefficient covariance matrix ([p. 239](#)).

[grads](#)..... examine the gradients of the log likelihood ([p. 302](#)).

[label](#)..... label view of likelihood object ([p. 317](#)).

[output](#)..... table of estimation results ([p. 361](#)).

[results](#)..... estimation results ([p. 400](#)).

[spec](#)..... likelihood specification ([p. 454](#)).

[wald](#)..... Wald coefficient restriction test ([p. 502](#)).

### Logl Procs

[displayname](#)..... set display name ([p. 266](#)).

[makegrads](#)..... make group containing gradients of the log likelihood ([p. 337](#)).

[makemodel](#)..... make model ([p. 341](#)).

[updatecoefs](#)..... update coefficient vector(s) from likelihood ([p. 493](#)).

## Logl Statements

The following statements can be included in the specification of the likelihood object. These statements are optional, except for “@logl” which is required. See [Chapter 22, “The Log Likelihood \(LogL\) Object”, on page 655](#) of the *User’s Guide* for further discussion.

- @byeqn ..... evaluate specification by equation.
- @byobs ..... evaluate specification by observation (default).
- @deriv ..... specify an analytic derivative series.
- @derivstep ..... set parameters to control step size.
- @logl ..... specify the likelihood contribution series.
- @param ..... set starting values.
- @temp ..... remove temporary working series.

## Logl Data Members

### *Scalar Values (system data)*

- @aic ..... Akaike information criterion.
- @cofcov(i,j) ..... covariance of coefficients  $i$  and  $j$ .
- @coefs(i) ..... coefficient  $i$ .
- @hq ..... Hannan-Quinn information criterion.
- @logl ..... value of the log likelihood function.
- @ncoefs ..... number of estimated coefficients.
- @regobs ..... number of observations used in estimation.
- @sc ..... Schwarz information criterion.
- @stderrs(i) ..... standard error for coefficient  $i$ .
- @tstats(i) .....  $t$ -statistic value for coefficient  $i$ .
- coef\_name(i) .....  $i$ -th element of default coefficient vector for likelihood.

### *Vectors and Matrices*

- @cofcov ..... covariance matrix of estimated parameters.
- @coefs ..... coefficient vector.
- @stderrs ..... vector of standard errors for coefficients.
- @tstats ..... vector of  $t$ -statistic values for coefficients.

## Logl Examples

To declare a likelihood named LL1:

```
logl ll1
```

To define a likelihood function for OLS (not a recommended way to do OLS!):

```
ll1.append @logl logl1
```

```
ll1.append res1 = y-c(1)-c(2)*x  
ll1.append logl1 = log(@dnorm(res1/@sqrt(c(3))))-log(c(3))/2
```

To estimate LL1 by maximum likelihood (the “showstart” option displays the starting values):

```
ll1.ml(showstart)
```

To save the estimated covariance matrix of the parameters from LL1 as a named matrix COV1:

```
matrix cov1=ll1.@cofcov
```

## Matrix

Matrix (two-dimensional array).

### Matrix Declaration

**matrix**..... declare matrix object ([p. 350](#)).

There are several ways to create a matrix object. You can enter the **matrix** keyword (with an optional row and column dimension) followed by a name:

```
matrix scalarmat  
matrix(10,3) results
```

Alternatively, you can combine a declaration with an assignment statement, in which case the new matrix will be sized accordingly.

Lastly, a number of object procedures create matrices.

### Matrix Views

**area** ..... area graph of the columns in the matrix ([p. 207](#)).

**bar** ..... single or multiple bar graph of each column against the row index ([p. 215](#)).

**cor** ..... correlation matrix by columns ([p. 246](#)).

**cov** ..... covariance matrix by columns ([p. 250](#)).

**errbar** ..... error bar graph view ([p. 276](#)).

**hilo** ..... high-low(-open-close) chart ([p. 307](#)).

**label** ..... label information for the matrix ([p. 317](#)).

**line** ..... single or multiple line graph of each column by the row index ([p. 320](#)).

**pie** ..... pie chart view ([p. 384](#)).

**scat** ..... scatter diagrams of the columns of the matrix ([p. 412](#)).

**sheet** .....spreadsheet view of the matrix ([p. 444](#)).  
**spike** .....spike graph ([p. 455](#)).  
**stats**.....descriptive statistics by column ([p. 462](#)).  
**xy**.....XY graph with one or more X columns plotted against one or more Y ([p. 528](#)).  
**xyline** .....XY line graph ([p. 530](#)).  
**xypair** .....XY pairs graph ([p. 528](#)).

## Matrix Procs

**displayname** .....set display name ([p. 266](#)).  
**fill** .....fill the elements of the matrix ([p. 282](#)).  
**read**.....import data from disk ([p. 391](#)).  
**setformat** .....set the display format for the matrix spreadsheet ([p. 432](#)).  
**setindent**.....set the indentation for the matrix spreadsheet ([p. 437](#)).  
**setjust**.....set the justification for the matrix spreadsheet ([p. 438](#)).  
**setWidth** .....set the column width in the matrix spreadsheet([p. 444](#)).  
**write** .....export data to disk ([p. 517](#)).

## Matrix Data Members

**(i,j)** .....(i,j)-th element of the matrix. Simply append “(i, j)” to the matrix name (without a “.”).

## Matrix Examples

The following assignment statements create and initialize matrix objects,

```
matrix copymat=results  
matrix covmat1=eq1.@coefcov  
matrix(5,2) count  
count.fill 1,2,3,4,5,6,7,8,9,10
```

as does the equation procedure:

```
eq1.mak ecoefcov covmat2
```

You can declare and initialize a matrix in one command:

```
matrix(10,30) results=3  
matrix(5,5) other=results1
```

Graphs and covariances may be generated for the columns of the matrix,

```
copymat.line  
copymat.cov
```

and statistics computed for the rows of a matrix:

```
matrix rowmat=@transpose(copymat)
rowmat.stats
```

You can use explicit indices to refer to matrix elements:

```
scalar diagsum=cov1(1,1)+cov1(2,2)+cov(3,3)
```

## Model

Set of simultaneous equations used for forecasting and simulation.

### Model Declaration

**model** ..... declare model object ([p. 353](#)).

Declare an object by entering the keyword **model**, followed by a name:

```
model mymod
```

declares an empty model named MYMOD. To fill MYMOD, open the model and edit the specification view, or use the **append** view. Note that models are not used for estimation of unknown parameters.

See also the section on model keywords in “[Text View](#)” on page 783 of the *User’s Guide*.

### Model Views

**block** ..... display model block structure ([p. 219](#)).

**eqs** ..... view of model organized by equation ([p. 275](#)).

**label** ..... view or set label information for the model ([p. 317](#)).

**msg** ..... display model solution messages ([p. 354](#)).

**text** ..... show text showing equations in the model ([p. 482](#)).

**trace** ..... view of trace output from model solution ([p. 484](#)).

**vars** ..... view of model organized by variable ([p. 501](#)).

### Model Procs

**addassign** ..... assign add factors to equations ([p. 196](#)).

**addinit** ..... initialize add factors ([p. 197](#)).

**append** ..... append a line of text to a model ([p. 202](#)).

**control** ..... solve for values of control variable so that target matches trajectory ([p. 243](#)).

**displayname** ..... set display name ([p. 266](#)).

**exclude** ..... specifies (or merges) excluded series to the active scenario ([p. 278](#)).

---

**makegraph**.....make graph object showing model series ([p. 338](#)).  
**makegroup** .....make group out of model series and display dated data table  
([p. 339](#)).  
**merge** .....merge objects into the model ([p. 351](#)).  
**override** .....specifies (or merges) override series to the active scenario  
([p. 363](#)).  
**scenario** .....set the active, alternate, or comparison scenario ([p. 415](#)).  
**solve** .....solve the model ([p. 451](#)).  
**solveopt** .....set solve options for model ([p. 452](#)).  
**spec** .....display the text specification view ([p. 454](#)).  
**unlink** .....break links in specification ([p. 491](#)).  
**update** .....update model specification ([p. 492](#)).

## Model Examples

The commands:

```
model mod1
mod1.append y=324.35+x
mod1.append x=-234+7.3*z
mod1.solve(m=100,c=.008)
```

create, specify, and solve the model MOD1.

The command:

```
mod1(g).makegraph gr1 x y z
```

plots the endogenous series X, Y, and Z, in the active scenario for model MOD1.

## Pool

Pooled time series, cross-section object. Used when working with data with both time series and cross-section structure.

### Pool Declaration

**pool**.....declare pool object ([p. 386](#)).

To declare a pool object, use the **pool** keyword, followed by a pool name, and optionally, a list of pool members. Pool members are short text identifiers for the cross section units:

```
pool mypool
pool g7 _can _fr _ger _ita _jpn _us _uk
```

## Pool Methods

- ls** ..... estimate linear regression models including cross-section weighted least squares, and fixed and random effects models ([p. 329](#)).
- tsls** ..... linear two-stage least squares (TSLS) regression models ([p. 487](#)).

## Pool Views

- cellipse** ..... Confidence ellipses for coefficient restrictions ([p. 231](#)).
- coefcov** ..... coefficient covariance matrix ([p. 239](#)).
- describe** ..... calculate pool descriptive statistics ([p. 265](#)).
- label** ..... label information for the pool object ([p. 317](#)).
- output** ..... table of estimation results ([p. 361](#)).
- representations** ..... text showing equations in the model ([p. 395](#)).
- residcor** ..... residual correlation matrix ([p. 398](#)).
- residcov** ..... residual covariance matrix ([p. 398](#)).
- resids** ..... table or graph of residuals for each pool member ([p. 399](#)).
- results** ..... table of estimation results ([p. 400](#)).
- sheet** ..... spreadsheet view of series in pool ([p. 444](#)).
- uroot** ..... unit root test on a pool series ([p. 494](#)).
- wald** ..... Wald coefficient restriction test ([p. 502](#)).

## Pool Procs

- add** ..... add cross section members to pool ([p. 196](#)).
- define** ..... define cross section identifiers ([p. 262](#)).
- delete** ..... delete pool series ([p. 263](#)).
- displayname** ..... set display name ([p. 266](#)).
- drop** ..... drop cross section members from pool ([p. 270](#)).
- fetch** ..... fetch series into workfile using a pool ([p. 279](#)).
- genr** ..... generate pool series using the “?” ([p. 295](#)).
- makegroup** ..... create a group of series from a pool ([p. 339](#)).
- makemodel** ..... creates a model object from the estimated pool ([p. 341](#)).
- makeresids** ..... make series containing residuals from pool ([p. 342](#)).
- makestats** ..... make descriptive statistic series ([p. 347](#)).
- makesystem** ..... creates a system object from the pool for other estimation methods ([p. 348](#)).
- read** ..... import pool data from disk ([p. 391](#)).
- store** ..... store pool series in database/bank files ([p. 465](#)).
- updatecoefs** ..... update coefficient vector from pool ([p. 493](#)).

**write** .....export pool data to disk ([p. 517](#)).

## Pool Data Members

### *String Values*

**@idname(i)** .....*i*-th cross-section identifier.

### *Scalar Values*

**@aic** .....Akaike information criterion.

**@cofcov(i,j)** .....covariance of coefficients *i* and *j*.

**@coefs(i)** .....coefficient *i*.

**@dw** .....Durbin-Watson statistic.

**@effects(i)** .....estimated fixed or random effect for the *i*-th cross-section member  
(only for fixed or random effects).

**@f** .....*F*-statistic.

**@logl** .....log likelihood.

**@meandep** .....mean of the dependent variable.

**@ncoef** .....total number of estimated coefficients.

**@ncross** .....total number of cross sectional units.

**@ncrossest** .....number of cross sectional units in last estimated pool equation.

**@r2** .....R-squared statistic.

**@rbar2** .....adjusted R-squared statistic.

**@regobs** .....total number of observations in regression.

**@schwarz** .....Schwarz information criterion.

**@sddep** .....standard deviation of the dependent variable.

**@se** .....standard error of the regression.

**@ssr** .....sum of squared residuals.

**@stderrs(i)** .....standard error for coefficient *i*.

**@totalobs** .....total number of observations in the pool. For a balanced sample  
this is “@regobs\*@ncrossest”.

**@tstats(i)** .....*t*-statistic value for coefficient *i*.

**c(i)** .....*i*-th element of default coefficient vector for the pool.

### *Vectors and Matrices*

**@cofcov** .....covariance matrix for coefficients of equation.

**@coefs** .....coefficient vector.

**@effects** .....vector of estimated fixed or random effects (only for fixed or ran-  
dom effects estimation).

**@stderrs** .....vector of standard errors for coefficients.

**@tstats** ..... vector of *t*-statistic values for coefficients.

## Pool Examples

To read data using the pool object:

```
mypool1.read(b2) data.xls x? y? z?
```

To delete and store pool series you may enter:

```
mypool1.delete x? y?  
mypool1.store z?
```

Descriptive statistics may be computed using the command:

```
mypool1.describe(m) z?
```

To estimate a pool equation using least squares and to access the *t*-statistics, enter:

```
mypool1.ls y? c z? @ w?  
vector tstat1 = mypool1.@tstats
```

## Rowvector

Row vector. (One dimensional array of numbers).

### Rowvector Declaration

**rowvector** ..... declare rowvector object ([p. 404](#)).

There are several ways to create a rowvector object. First, you can enter the `rowvector` keyword (with an optional dimension) followed by a name:

```
rowvector scalarmat  
rowvector(10) results
```

The resulting rowvector will be initialized with zeros.

Alternatively, you may combine a declaration with an assignment statement. The new vector will be sized and initialized accordingly:

```
rowvector(10) y=3  
rowvector z=results
```

### Rowvector Views

**area** ..... area graph of the vector ([p. 207](#)).

**bar** ..... bar graph of each column (element) of the data against the row index ([p. 215](#)).

**errbar** ..... error bar graph view ([p. 276](#)).

**label** .....label information for the rowvector ([p. 317](#)).  
**line** .....line graph of each column (element) of the data against the row index ([p. 320](#)).  
**scat** .....scatter diagrams of the columns of the rowvector ([p. 412](#)).  
**sheet** .....spreadsheet view of the vector ([p. 444](#)).  
**spike** .....spike graph ([p. 455](#)).  
**stats** .....(trivial) descriptive statistics ([p. 462](#)).

## Rowvector Procs

**displayname** .....set display name ([p. 266](#)).  
**fill** .....fill elements of the vector ([p. 282](#)).  
**read** .....import data from disk ([p. 391](#)).  
**setformat** .....set the display format for the vector spreadsheet ([p. 432](#)).  
**setindent** .....set the indentation for the vector spreadsheet ([p. 437](#)).  
**setjust** .....set the justification for the vector spreadsheet ([p. 438](#)).  
**setWidth** .....set the column width in the vector spreadsheet ([p. 444](#)).  
**write** .....export data to disk ([p. 517](#)).

## Rowvector Data Members

(i) ..... $i$ -th element of the vector. Simply append “(i)” to the matrix name (without a “.”).

## Rowvector Examples

To declare a rowvector and to fill it with data read from an Excel file:

```
rowvector(10) mydata  
mydata.read(b2) thedata.xls
```

To access a single element of the vector using direct indexing:

```
scalar result1=mydata(2)
```

The rowvector may be used in standard matrix expressions:

```
vector transdata=@transpose(mydata)
```

## Sample

Sample of observations. Description of a set of observations to be used in operations.

### Sample Declaration

**sample** .....declare sample object ([p. 406](#)).

To declare a sample object, use the keyword `sample`, followed by a name and a sample string:

```
sample mysample 1960:1 1990:4  
sample altsample 120 170 300 1000 if x>0
```

## Sample Procs

`set` ..... reset the sample range ([p. 420](#)).

## Sample Example

To change the observations in a sample object, you can use the `set` proc:

```
mysample.set 1960:1 1980:4 if y>0  
sample thesamp 1 10 20 30 40 60 if x>0  
thesamp.set @all
```

To set the current sample to use a sample, enter a `smp1` statement, followed by the name of the sample object:

```
smp1 mysample  
equation eq1.ls y x c
```

## Scalar

**Scalar (single number).** A scalar holds a single numeric value. Scalar values may be used in standard EViews expressions in place of numeric values.

### Scalar Declaration

`scalar` ..... declare scalar object ([p. 410](#)).

To declare a scalar object, use the keyword `scalar`, followed by a name, an “=” sign and a scalar expression or value.

Scalar objects have no views or procedures, and do not open windows. The value of the scalar may be displayed in the status line at the bottom of the EViews window.

### Scalar Examples

You can declare a scalar and examine its contents in the status line:

```
scalar pi=3.14159  
scalar shape=beta(7)  
show shape
```

or you can declare a scalar and use it in an expression:

```
scalar inner=@transpose(mydata)*mydata
```

```
series x=1/@sqrt(inner)*y
```

## Series

Series of numeric observations. An EViews series contains a set of observations on a numeric variable.

### Series Declaration

```
frml .....create numeric series object with a formula for auto-updating  
          (p. 293).  
genr .....create numeric series object (p. 295).  
series .....declare numeric series object (p. 418).
```

To declare a series, use the keyword `series` or `alpha` followed by a name, and optionally, by an “=” sign and a valid numeric series expression:

```
series y  
genr x=3*z
```

If there is no assignment, the series will be initialized to contain NAs.

### Series Views

```
area .....area graph of the series (p. 207).  
bar .....bar graph of the series (p. 215).  
bdstest .....BDS independence test (p. 217).  
boxplotby .....boxplot by classification (p. 221).  
cdfplot .....distribution (cumulative, survivor, quantile) functions (p. 230).  
correl .....correlogram, autocorrelation and partial autocorrelation functions  
          (p. 246).  
edftest .....empirical distribution function tests (p. 272).  
freq .....one-way tabulation (p. 291).  
hist .....descriptive statistics and histogram (p. 308).  
kdensity .....kernel density estimate (p. 315).  
label .....label information for the series (p. 317).  
line .....line graph of the series (p. 320).  
qqplot .....quantile-quantile plot (p. 389).  
seasplot .....seasonal line graph (p. 418).  
sheet .....spreadsheet view of the series (p. 444).  
spike .....spike graph (p. 455).  
statby .....statistics by classification (p. 457).  
stats .....descriptive statistics table (p. 308).
```

- testby**..... equality test by classification ([p. 476](#)).
- teststat**..... simple hypothesis tests ([p. 481](#)).
- uroot**..... unit root test on an ordinary or panel series ([p. 494](#)).

### Series Procs

- displayname**..... set display name ([p. 266](#)).
- fill** ..... fill the elements of the series ([p. 282](#)).
- hpf**..... Hodrick-Prescott filter ([p. 310](#)).
- map** ..... assign or remove value map setting ([p. 349](#)).
- resample** ..... resample from the observations in the series ([p. 395](#)).
- seas** ..... seasonal adjustment for quarterly and monthly time series ([p. 417](#)).
- setconvert** ..... set default frequency conversion method ([p. 424](#)).
- setformat** ..... set the display format for the series spreadsheet ([p. 432](#)).
- setindent** ..... set the indentation for the series spreadsheet ([p. 437](#)).
- setjust** ..... set the justification for the series spreadsheet ([p. 438](#)).
- setwidth** ..... set the column width in the series spreadsheet ([p. 444](#)).
- smooth** ..... exponential smoothing ([p. 447](#)).
- tramoseats** ..... seasonal adjustment using Tramo/Seats ([p. 484](#)).
- x11** ..... seasonal adjustment by Census X11 method for quarterly and monthly time series ([p. 520](#)).
- x12** ..... seasonal adjustment by Census X12 method for quarterly and monthly time series ([p. 522](#)).

### Series Data Members

- (i) ..... *i*-th element of the series from the beginning of the workfile (when used on the left-hand side of an assignment, or when the element appears in a matrix, vector, or scalar assignment).

### Series Element Functions

- @elem(ser, j)** ..... function to access the *j*-th observation of the series SER, where *j* identifies the date or observation.

### Series Examples

You can declare a series in the usual fashion:

```
series b=income*@mean(z)  
series blag=b(1)
```

Note that the last example above involves a series expression so that  $B(1)$  is treated as a one-period lead of the entire series, not as an element operator. In contrast:

```
scalar blag1=b(1)
```

evaluates the first observation on  $B$  in the workfile.

Once a series is declared, views and procs are available:

```
a.qqplot
a.statby(mean, var, std) b
```

To access individual values:

```
scalar quarterlyval = @elem(y, "1980:3")
scalar undatedval = @elem(x, 323)
```

## Sspace

**State space object.** Estimation and evaluation of state space models using the Kalman filter.

### Sspace Declaration

**sspace** .....create sspace object ([p. 457](#)).

To declare a sspace object, use the `sspace` keyword, followed by a valid name.

### Sspace Method

**ml** .....maximum likelihood estimation or filter initialization ([p. 352](#)).

### Sspace Views

**cellipse** .....Confidence ellipses for coefficient restrictions ([p. 231](#)).

**coefcov** .....coefficient covariance matrix ([p. 239](#)).

**endog** .....table or graph of actual signal variables ([p. 274](#)).

**grads** .....examine the gradients of the log likelihood ([p. 302](#)).

**label** .....label information for the state space object ([p. 317](#)).

**output** .....table of estimation results ([p. 361](#)).

**residcor** .....standardized one-step ahead residual correlation matrix ([p. 398](#)).

**residcov** .....standardized one-step ahead residual covariance matrix ([p. 398](#)).

**resids** .....one-step ahead actual, fitted, residual graph ([p. 399](#)).

**results** .....table of estimation and filter results ([p. 400](#)).

**signalgraphs** .....display graphs of signal variables ([p. 446](#)).

**spec** .....text representation of state space specification ([p. 454](#)).

**statefinal** .....display the final values of the states or state covariance ([p. 460](#)).

**stategraphs** ..... display graphs of state variables ([p. 459](#)).  
**stateinit** ..... display the initial values of the states or state covariance ([p. 461](#)).  
**structure** ..... examine coefficient or variance structure of the specification  
([p. 467](#)).  
**wald** ..... Wald coefficient restriction test ([p. 502](#)).

## Sspace Procs

**append** ..... add line to the specification ([p. 202](#)).  
**displayname** ..... set display name ([p. 266](#)).  
**forecast** ..... perform state and signal forecasting ([p. 287](#)).  
**makeendog** ..... make group containing actual values for signal variables ([p. 335](#)).  
**makefilter** ..... make new Kalman Filter([p. 336](#)).  
**makegrads** ..... make group containing the gradients of the log likelihood ([p. 337](#)).  
**makemodel** ..... make a model object containing equations in sspace ([p. 341](#)).  
**makesignals** ..... make group containing signal and residual series ([p. 344](#)).  
**makestates** ..... make group containing state series ([p. 345](#)).  
**sspace** ..... declare sspace object ([p. 457](#)).  
**updatecoefs** ..... update coefficient vector(s) from sspace ([p. 493](#)).

## Sspace Data Members

### Scalar Values

**@coefcov(i,j)** ..... covariance of coefficients  $i$  and  $j$ .  
**@coefs(i)** ..... coefficient  $i$ .  
**@eqregobs(k)** ..... number of observations in signal equation  $k$ .  
**@sddep(k)** ..... standard deviation of the signal variable in equation  $k$ .  
**@ssr(k)** ..... sum-of-squared standardized one-step ahead residuals for equation  $k$ .  
**@stderrs(i)** ..... standard error for coefficient  $i$ .  
**@tstats(t)** .....  $t$ -statistic value for coefficient  $i$ .

### Scalar Values (system level data)

**@aic** ..... Akaike information criterion for the system.  
**@hq** ..... Hannan-Quinn information criterion for the system.  
**@logl** ..... value of the log likelihood function.  
**@ncoefs** ..... total number of estimated coefficients in the system.  
**@neqns** ..... number of equations for observable variables.  
**@regobs** ..... number of observations in the system.  
**@sc** ..... Schwarz information criterion for the system.

`@totalobs` .....sum of “`@eqregobs`” from each equation.

#### *Vectors and Matrices*

`@coefcov` .....covariance matrix for coefficients of equation.

`@coefs` .....coefficient vector.

`@stderrs` .....vector of standard errors for coefficients.

`@tstats` .....vector of *t*-statistic values for coefficients.

#### *State and Signal Results*

The following functions allow you to extract the filter and smoother results for the estimation sample and place them in matrix objects. In some cases, the results overlap those available thorough the `sspace` procs, while in other cases, the matrix results are the only way to obtain the results.

Note also that since the computations are only for the estimation sample, the one-step-ahead predicted state and state standard error values *will not* match the final values displayed in the estimation output. The latter are the predicted values for the first out-of-estimation sample period.

`@pred_signal` .....matrix or vector of one-step ahead predicted signals.

`@pred_sigmalcov` .....matrix where every row is the `@vech` of the one-step ahead predicted signal covariance.

`@pred_signalse` .....matrix or vector of the standard errors of the one-step ahead predicted signals.

`@pred_err` .....matrix or vector of one-step ahead prediction errors.

`@pred_errcov` .....matrix where every row is the `@vech` of the one-step ahead prediction error covariance.

`@pred_errcovinv` .....matrix where every row is the `@vech` of the inverse of the one-step ahead prediction error covariance.

`@pred_errse` .....matrix or vector of the standard errors of the one-step ahead prediction errors.

`@pred_errstd` .....matrix or vector of standardized one-step ahead prediction errors.

`@pred_state` .....matrix or vector of one-step ahead predicted states.

`@pred_statecov` .....matrix where each row is the `@vech` of the one-step ahead predicted state covariance.

`@pred_statese` .....matrix or vector of the standard errors of the one-step ahead predicted states.

`@pred_stateerr` .....matrix or vector of one-step ahead predicted state errors.

`@curr_err` .....matrix or vector of filtered error estimates.

`@curr_gain` .....matrix or vector where each row is the `@vec` of the Kalman gain.

@curr\_state ..... matrix or vector of filtered states.  
@curr\_statecov .... matrix where every row is the @vech of the filtered state covariance.  
@curr\_statese ..... matrix or vector of the standard errors of the filtered state estimates.  
@sm\_signal ..... matrix or vector of smoothed signal estimates.  
@sm\_signalcov .... matrix where every row is the @vech of the smoothed signal covariance.  
@sm\_signalse ..... matrix or vector of the standard errors of the smoothed signals.  
@sm\_signalerr ..... matrix or vector of smoothed signal error estimates.  
@sm\_signalerrcov matrix where every row is the @vech of the smoothed signal error covariance.  
@sm\_signalerrse.. matrix or vector of the standard errors of the smoothed signal error.  
@sm\_signalerrstd. matrix or vector of the standardized smoothed signal errors.  
@sm\_state ..... matrix or vector of smoothed states.  
@sm\_statecov ..... matrix where each row is the @vech of the smoothed state covariances.  
@sm\_statese ..... matrix or vector of the standard errors of the smoothed state.  
@sm\_stateerr ..... matrix or vector of the smoothed state errors.  
@sm\_stateerrcov.. matrix where each row is the @vech of the smoothed state error covariance.  
@sm\_stateerrse.... matrix or vector of the standard errors of the smoothed state errors.  
@sm\_stateerrstd... matrix or vector of the standardized smoothed state errors .  
@sm\_crosserrcov . matrix where each row is the @vec of the smoothed error cross-covariance.

## Sspace Examples

The one-step-ahead state values and variances from SS01 may be saved using:

```
vector ss_state=ss01.@pred_state  
matrix ss_statecov=ss01.@pred_statecov
```

## Sym

Symmetric matrix (symmetric two-dimensional array).

### Sym Declaration

**sym** ..... declare sym object ([p. 471](#)).

Declare by providing a name after the `sym` keyword, with the optionally specified dimension in parentheses:

```
sym(10) symmatrix
```

You may optionally assign a scalar, a square matrix or another `sym` in the declaration. If the square matrix is not symmetric, the `sym` will contain the lower triangle. The `sym` will be sized and initialized accordingly.

## Sym Views

- `area` .....area graph of the columns of the matrix ([p. 207](#)).
- `bar` .....single or multiple bar graph of each column against the row index ([p. 215](#)).
- `cor` .....correlation matrix by columns ([p. 246](#)).
- `cov` .....covariance matrix by columns ([p. 250](#)).
- `errbar` .....error bar graph view ([p. 276](#)).
- `hilo` .....high-low(-open-close) chart ([p. 307](#)).
- `label` .....label information for the symmetric matrix ([p. 317](#)).
- `line` .....single or multiple line graph of each column against the row index ([p. 320](#)).
- `pie` .....pie chart view ([p. 384](#)).
- `scat` .....scatter diagrams of the columns of the `sym` ([p. 412](#)).
- `sheet` .....spreadsheet view of the symmetric matrix ([p. 444](#)).
- `spike` .....spike graph ([p. 455](#)).
- `stats` .....descriptive statistics by column ([p. 462](#)).
- `xy` .....XY graph with one or more X columns plotted against one or more Y ([p. 528](#)).
- `xyline` .....XY line graph ([p. 530](#)).
- `xypair` .....XY pairs graph ([p. 528](#)).

## Sym Procs

- `displayname` .....set display name ([p. 266](#)).
- `fill` .....fill the elements of the matrix ([p. 282](#)).
- `read` .....import data from disk ([p. 391](#)).
- `setformat` .....set the display format for the `sym` spreadsheet ([p. 432](#)).
- `setindent` .....set the indentation for the `sym` spreadsheet ([p. 437](#)).
- `setjust` .....set the justification for the `sym` spreadsheet ([p. 438](#)).
- `setWidth` .....set the column width in the `sym` spreadsheet ([p. 444](#)).
- `write` .....export data to disk ([p. 517](#)).

## Sym Data Members

**(i,j)** .....  $(i,j)$ -th element of the matrix. Simply append “(i,j)” to the matrix name (without a “.”).

## Sym Examples

The declaration:

```
sym results(10)  
results=3
```

creates the  $10 \times 10$  matrix RESULTS and initializes each value to be 3. The following assignment statements also create and initialize sym objects:

```
sym copymat=results  
sym covmat1=eq1.@coefcov  
sym(3,3) count  
count.fill 1,2,3,4,5,6,7,8,9,10
```

Graphs, covariances, and statistics may be generated for the columns of the matrix:

```
copymat.line  
copymat.cov  
copymat.stats
```

You can use explicit indices to refer to matrix elements:

```
scalar diagsum=cov1(1,1)+cov1(2,2)+cov(3,3)
```

## System

System of equations for estimation.

### System Declaration

**system** ..... declare system object ([p. 472](#)).

Declare a system object by entering the keyword `system`, followed by a name:

```
system mysys
```

To fill a system, open the system and edit the specification view, or use `append`. Note that systems are not used for simulation. See “[Model](#)” ([p. 168](#)).

### System Methods

**3sls** ..... three-stage least squares ([p. 194](#)).

**fiml** ..... full information maximum likelihood ([p. 284](#)).

**gmm** ..... generalized method of moments ([p. 297](#)).

**ls** .....ordinary least squares ([p. 329](#)).  
**sur** .....seemingly unrelated regression ([p. 468](#)).  
**tsls** .....two-stage least squares ([p. 487](#)).  
**wls** .....weighted least squares ([p. 515](#)).  
**wtsls** .....weighted two-stage least squares ([p. 519](#)).

## System Views

**cellipse** .....Confidence ellipses for coefficient restrictions ([p. 231](#)).  
**coefcov** .....coefficient covariance matrix ([p. 239](#)).  
**derivs** .....derivatives of the system equations ([p. 264](#)).  
**endog** .....table or graph of endogenous variables ([p. 274](#)).  
**grads** .....examine the gradients of the objective function ([p. 302](#)).  
**label** .....label information for the system object ([p. 317](#)).  
**output** .....table of estimation results ([p. 361](#)).  
**residcor** .....residual correlation matrix ([p. 398](#)).  
**residcov** .....residual covariance matrix ([p. 398](#)).  
**resids** .....residual graphs ([p. 399](#)).  
**results** .....table of estimation results ([p. 400](#)).  
**spec** .....text representation of system specification ([p. 454](#)).  
**wald** .....Wald coefficient restriction test ([p. 502](#)).

## System Procs

**append** .....add a line of text to the system specification ([p. 202](#)).  
**displayname** .....set display name ([p. 266](#)).  
**makeendog** .....make group of endogenous series ([p. 335](#)).  
**makemodel** .....create a model from the estimated system ([p. 341](#)).  
**makeresids** .....make series containing residuals from system ([p. 342](#)).  
**updatecoefs** .....update coefficient vector(s) from system ([p. 493](#)).

## System Data Members

### *Scalar Values (individual equation data)*

**@coefcov(i, j)** .....covariance of coefficients  $i$  and  $j$ .  
**@coefs(i)** .....coefficient  $i$ .  
**@dw(k)** .....Durbin-Watson statistic for equation  $k$ .  
**@eqncoef(k)** .....number of estimated coefficients in equation  $k$ .  
**@eqregobs(k)** .....number of observations in equation  $k$ .  
**@meandep(k)** .....mean of the dependent variable in equation  $k$ .

@ncoef(k) ..... total number of estimated coefficients in equation  $k$ .  
@r2(k) ..... R-squared statistic for equation  $k$ .  
@rbar2(k) ..... adjusted R-squared statistic for equation  $k$ .  
@sddep(k) ..... standard deviation of dependent variable in equation  $k$ .  
@se(k) ..... standard error of the regression in equation  $k$ .  
@ssr(k) ..... sum of squared residuals in equation  $k$ .  
@stderrs(i) ..... standard error for coefficient  $i$ .  
@tstats(i) .....  $t$ -statistic for coefficient  $i$ .  
c(i) .....  $i$ -th element of default coefficient vector for system (if applicable).

#### *Scalar Values (system level data)*

@aic ..... Akaike information criterion for the system (if applicable).  
@detresid ..... determinant of the residual covariance matrix.  
@hq ..... Hannan-Quinn information criterion for the system (if applicable).  
@jstat .....  $J$ -statistic — value of the GMM objective function (for GMM estimation).  
@logl ..... value of the log likelihood function for the system (if applicable).  
@ncoefs ..... total number of estimated coefficients in system.  
@neqn ..... number of equations.  
@regobs ..... number of observations in the sample range used for estimation (“@regobs” will differ from “@eqregobs” if the unbalanced sample is non-overlapping).  
@schwarz ..... Schwarz information criterion for the system (if applicable).  
@totalobs ..... sum of “@eqregobs” from each equation.

#### *Vectors and Matrices*

@coefcov ..... covariance matrix for coefficients of equation.  
@coefs ..... coefficient vector.  
@stderrs ..... vector of standard errors for coefficients.  
@tstats ..... vector of  $t$ -statistic values for coefficients.

#### **System Examples**

To estimate a system using GMM and to create residual series for the estimated system:

```
sys1.gmm(i,m=7,c=.01,b=v)
sys1.makeresids consres incres saveres
```

To test coefficients using a Wald test:

---

```
sys1.wald c(1)=c(4)
```

To save the coefficient covariance matrix:

```
sym covs=sys1.@coefcov
```

## Table

Table object. Formatted two-dimensional table for output display.

### Table Declaration

[freeze](#) .....freeze tabular view of object ([p. 290](#)).

[table](#) .....create table object ([p. 457](#)).

To declare a table object, use the keyword `table`, followed by an optional row and column dimension, and then the object name:

```
table onelement
table(10,5) outtable
```

If no dimension is provided, the table will contain a single element.

Alternatively, you may declare a table using an assignment statement. The new table will be sized and initialized, accordingly:

```
table newtable=outtable
```

Lastly, you may use the `freeze` command to create tables from tabular views of other objects:

```
freeze(newtab) ser1.freq
```

### Table Views

[label](#) .....label information for the table object ([p. 317](#)).

[sheet](#) .....view the table ([p. 444](#)).

### Table Procs

[comment](#) .....adds or removes a comment in a table cell ([p. 242](#)).

[displayname](#) .....set display name ([p. 266](#)).

[save](#) .....save table as CSV, tab-delimited ASCII text, RTF, or HTML file on disk ([p. 407](#)).

[setfillcolor](#) .....set the fill (background) color of a set of table cells ([p. 429](#)).

[setfont](#) .....set the font for the text in a set of table cells ([p. 431](#)).

[setformat](#) .....set the display format of a set of table cells ([p. 432](#)).

[setheight](#) .....set the row height in a set of table cells ([p. 436](#)).

**setindent** ..... set the indentation for a set of table cells ([p. 437](#)).  
**setjust** ..... set the justification for a set of table cells ([p. 438](#)).  
**setlines** ..... set the line characteristics and borders for a set of table cells ([p. 440](#)).  
**setmerge** ..... merge or unmerge a set of table cells ([p. 441](#)).  
**settextcolor** ..... set the text color in a set of table cells ([p. 443](#)).  
**setwidth** ..... set the column width for a set of table cells ([p. 444](#)).

## Table Data Members

**(i,j)** ..... the  $(i,j)$ -th element of the table, formatted as a string.

## Table Commands

**setcell** ..... format and fill in a table cell ([p. 422](#)).  
**setcolwidth** ..... set width of a table column ([p. 423](#)).  
**setline** ..... place a horizontal line in table ([p. 439](#)).

## Table Examples

```
table(5,5) mytable  
%strval = mytable(2,3)  
mytable(4,4) = "R2"  
mytable(4,5) = @str(eq1.@r2)
```

## Text

Text object.

Object for holding arbitrary text information.

### Text Declaration

**text** ..... declare text object ([p. 482](#)).

To declare a text object, use the keyword `text`, followed by the object name:

```
text mytext
```

### Text Views

**label** ..... label information for the valmap object ([p. 317](#)).  
**text** ..... view contents of text object ([p. 482](#)).

## Text Examples

```
text mytext  
[add text to the object]
```

```
mytext.text
```

## Valmap

Valmap (value map).

### Valmap Declaration

**valmap** .....declare valmap object ([p. 499](#)).

To declare a valmap use the keyword **valmap**, followed by a name

```
valmap mymap
```

### Valmap Views

**label** .....label information for the valmap object ([p. 317](#)).

**sheet** .....view table of map definitions ([p. 444](#)).

**stats**.....summary of map definitions ([p. 462](#)).

**usage**.....list of series and alphas which use the map ([p. 499](#)).

### Valmap Procs

**append** .....append a definition to a valmap ([p. 202](#)).

**displayname** .....set display name ([p. 266](#)).

### Valmap Examples

```
valmap b  
b.append 0 no  
b.append 1 yes
```

declares a valmap B, and adds two map definitions, mapping 0 to “no” and 1 to “yes”.

```
valmap txtmap  
txtmap append "NM" "New Mexico"  
txtmap append CA California  
txtmap append "RI" "Rhode Island"
```

declares the valmap TXTMAP and adds three definitions.

## Var

Vector autoregression and error correction object.

### Var Declaration

**var** .....declare var estimation object ([p. 500](#)).

To declare a var use the keyword `var`, followed by a name and, optionally, by an estimation specification:

```
var finvar  
var empvar.ls 1 4 payroll hhold gdp  
var finec.ec(e,2) 1 6 cp div r
```

## Var Methods

- `ec` ..... estimate a vector error correction model ([p. 271](#)).
- `ls` ..... estimate an unrestricted VAR ([p. 329](#)).

## Var Views

- `arlm` ..... serial correlation LM test ([p. 209](#)).
- `arroots` ..... inverse roots of the AR polynomial ([p. 212](#)).
- `coint` ..... Johansen cointegration test ([p. 240](#)).
- `correl` ..... residual autocorrelations ([p. 246](#)).
- `decomp` ..... variance decomposition ([p. 260](#)).
- `endog` ..... table or graph of endogenous variables ([p. 274](#)).
- `impulse` ..... impulse response functions ([p. 311](#)).
- `jbera` ..... residual normality test ([p. 313](#)).
- `label` ..... label information for the var object ([p. 317](#)).
- `laglen` ..... lag order selection criteria ([p. 318](#)).
- `output` ..... table of estimation results ([p. 361](#)).
- `qstats` ..... residual portmanteau tests ([p. 390](#)).
- `representations` ..... text describing var specification ([p. 395](#)).
- `residcor` ..... residual correlation matrix ([p. 398](#)).
- `residcov` ..... residual covariance matrix ([p. 398](#)).
- `resids` ..... residual graphs ([p. 399](#)).
- `results` ..... table of estimation results ([p. 400](#)).
- `testexog` ..... exogeneity (Granger causality) tests ([p. 478](#)).
- `testlags` ..... lag exclusion tests ([p. 480](#)).
- `white` ..... White heteroskedasticity test ([p. 514](#)).

## Var Procs

- `append` ..... append restriction text ([p. 202](#)).
- `cleartext` ..... clear restriction text ([p. 237](#)).
- `displayname` ..... set display name ([p. 266](#)).
- `makecoint` ..... make group of cointegrating relations ([p. 334](#)).
- `makeendog` ..... make group of endogenous series ([p. 335](#)).

---

**makemodel** .....make model from the estimated var ([p. 341](#)).  
**makeresids** .....make residual series ([p. 342](#)).  
**makesystem** .....make system from var ([p. 348](#)).  
**svar** .....estimate structural factorization ([p. 469](#)).

## Var Data Members

### *Scalar Values (individual level data)*

**@eqlogl(k)** .....log likelihood for equation  $k$ .  
**@eqncoef(k)** .....number of estimated coefficients in equation  $k$ .  
**@eqregobs(k)** .....number of observations in equation  $k$ .  
**@meandep(k)** .....mean of the dependent variable in equation  $k$ .  
**@r2(k)** .....R-squared statistic for equation  $k$ .  
**@rbar2(k)** .....adjusted R-squared statistic for equation  $k$ .  
**@sddep(k)** .....std. dev. of dependent variable in equation  $k$ .  
**@se(k)** .....standard error of the regression in equation  $k$ .  
**@ssr(k)** .....sum of squared residuals in equation  $k$ .  
**a(i,j)** .....adjustment coefficient for the  $j$ -th cointegrating equation in the  $i$ -th equation of the VEC (where applicable).  
**b(i,j)** .....coefficient of the  $j$ -th variable in the  $i$ -th cointegrating equation (where applicable).  
**c(i,j)** .....coefficient of the  $j$ -th regressor in the  $i$ -th equation of the var, or the coefficient of the  $j$ -th first-difference regressor in the  $i$ -th equation of the VEC.

### *Scalar Values (system level data)*

**@aic** .....Akaike information criterion for the system.  
**@detresid** .....determinant of the residual covariance matrix.  
**@hq** .....Hannan-Quinn information criterion for the system.  
**@logl** .....log likelihood for system.  
**@ncoefs** .....total number of estimated coefficients in the var.  
**@neqn** .....number of equations.  
**@regobs** .....number of observations in the var.  
**@sc** .....Schwarz information criterion for the system.  
**@svarcvgtype** .....Returns an integer indicating the convergence type of the structural decomposition estimation: 0 (convergence achieved), 2 (failure to improve), 3 (maximum iterations reached), 4 (no convergence—structural decomposition not estimated).  
**@svaroverid** .....over-identification LR statistic from structural factorization.

**@totalobs** ..... sum of “@eqregobs” from each equation (“@regobs\*@neqn”).

#### Vectors and Matrices

**@coefmat** ..... coefficient matrix (as displayed in output table).

**@coefse** ..... matrix of coefficient standard errors (corresponding to the output table).

**@cointse** ..... standard errors of cointegrating vectors.

**@cointvec** ..... cointegrating vectors.

**@impfact** ..... factorization matrix used in last impulse response view.

**@lrrsp** ..... accumulated long-run responses from last impulse response view.

**@lrrspse** ..... standard errors of accumulated long-run responses.

**@residcov** ..... (sym) covariance matrix of the residuals.

**@svaramat** ..... estimated A matrix for structural factorization.

**@svarbmat** ..... estimated B matrix for structural factorization.

**@svarcovab** ..... covariance matrix of stacked A and B matrix for structural factorization.

**@svarrcov** ..... restricted residual covariance matrix from structural factorization.

### Var Examples

To declare a var estimate a VEC specification and make a residual series:

```
var finec.ec(e,2) 1 6 cp div r  
finec.makeresids
```

To estimate an ordinary var, to create series containing residuals, and to form a model based upon the estimated var:

```
var empvar.ls 1 4 payroll hhold gdp  
empvar.makeresids payres hholdres gdpres  
empvar.makemodel(inmdl) cp fcp div fdiv r fr
```

To save coefficients in a scalar:

```
scalar coef1=empvar.b(1,2)
```

## Vector

**Vector.** (One dimensional array of numbers).

#### Vector Declaration

**vector** ..... declare vector object ([p. 499](#)).

There are several ways to create a vector object. Enter the `vector` keyword (with an optional dimension) followed by a name:

```
vector scalarmat  
vector(10) results
```

Alternatively, you may declare a vector using an assignment statement. The vector will be sized and initialized, accordingly:

```
vector(10) myvec=3.14159  
vector results=vec1
```

## Vector Views

**area** .....area graph of the vector ([p. 207](#)).  
**bar** .....bar graph of data against the row index ([p. 215](#)).  
**label** .....label information for the vector object ([p. 317](#)).  
**line** .....line graph of the data against the row index ([p. 320](#)).  
**sheet** .....spreadsheet view of the vector ([p. 444](#)).  
**spike** .....spike graph ([p. 455](#)).  
**stats** .....descriptive statistics ([p. 462](#)).

## Vector Procs

**displayname** .....set display name ([p. 266](#)).  
**fill** .....fill elements of the vector ([p. 282](#)).  
**read** .....import data from disk ([p. 391](#)).  
**setformat** .....set the display format for the vector spreadsheet ([p. 432](#)).  
**setindent** .....set the indentation for the vector spreadsheet ([p. 437](#)).  
**setjust** .....set the justification for the vector spreadsheet ([p. 438](#)).  
**setWidth** .....set the column width for the vector spreadsheet ([p. 444](#)).  
**write** .....export data to disk ([p. 517](#)).

## Vector Data Members

(i) ..... $i$ -th element of the vector. Simply append “(i)” to the matrix name (without a “.”).

## Vector Examples

To declare a vector and to fill it with data read in from an Excel file:

```
vector(10) mydata  
mydata.read(b2) thedata.xls
```

To access a single element of the vector using direct indexing:

```
scalar result1=mydata(2)
```

The vector may be used in standard matrix expressions:

```
rowvector transdata=@transpose(mydata)
scalar inner=@transpose(mydata) *mydata
```

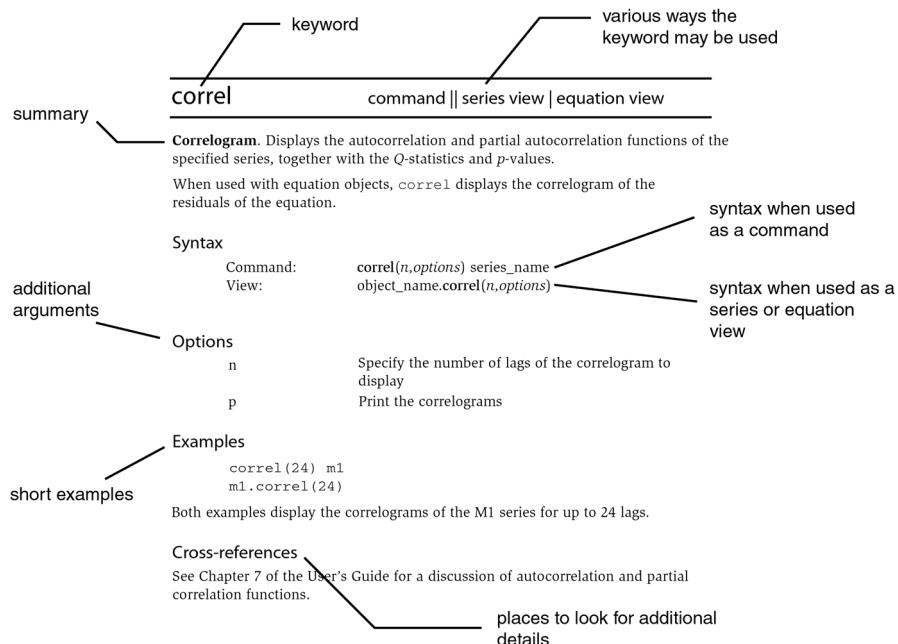
## Appendix B. Command Reference

---

This chapter provides a complete alphabetical listing of the commands, views, and procedures in EViews.

Each entry is comprised of a keyword, followed by a listing of the ways in which the keyword may be used. The entry also provides a summary of behavior, as well as a description of the syntax. If appropriate, we also provide a listing of additional arguments, short examples, and cross-references.

For example, the listing for `correl` is given by:



Note that the keyword `correl` may be used in three distinct ways: as a command to compute the correlogram of a specified series, as a series view to display the correlogram for a series object, and as an equation view to display the correlogram of the residuals from the estimated equation. The syntax for each of these uses is documented separately.

In addition to the dictionary-style alphabetical listing provided here, there are several other places in the *Command and Programming Reference* that contain reference material:

- “Basic Command Summary” beginning on page 16, provides a listing of commands commonly used when managing objects in workfiles and databases.

- In “[Matrix Function and Command Summary](#)” on page 44, we list the basic functions and commands used when working with matrix objects.
- In “[Programming Summary](#)” on page 115, we list the commands, functions and keywords used in EViews programming and string processing.
- [Appendix A, “Object, View and Procedure Reference”](#), beginning on page 151, lists all of the views and procedures classified by object, making it easy to see which views and procedures are available for a given object.
- [Appendix C, “Special Expression Reference”](#), beginning on page 535 lists special expressions that may be used in series assignment and generation, or as terms in estimation specifications.
- [Appendix D, “Operator and Function Reference”](#), on page 543 contains a complete listing of the functions and mathematical operators used in forming series expressions, and in performing matrix element operations.
- [Appendix E, “Workfile Functions”](#), on page 559 documents special functions that return information about the workfile and the observation identifiers.
- [Appendix F, “String and Date Function Reference”](#), on page 565 lists functions for working with strings and date numbers.
- [Appendix G, “Matrix Reference”](#), beginning on page 581 documents the matrix, vector, scalar, and string functions and commands.
- [Appendix H, “Programming Language Reference”](#), on page 603 lists the commands and keywords used in the programming language. These entries may only be used in batch programs.

<b>3sls</b>	<a href="#">System Method</a>
-------------	-------------------------------

Estimate a system of equations by three-stage least squares.

### Syntax

System Method:    `system_name.3sls(options)`

### Options

i                Iterate simultaneously over the weighting matrix and coefficient vector.

s                Iterate sequentially over the weighting matrix and coefficient vector.

<i>o</i> ( <i>default</i> )	Iterate the coefficient vector to convergence following one-iteration of the weighting matrix.
<i>c</i>	One step (iteration) of the coefficient vector following one-iteration of the weighting matrix.
<i>m = integer</i>	Maximum number of iterations.
<i>c = number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<i>l = number</i>	Set maximum number of iterations on the first-stage coefficient estimation to get the one-step weighting matrix.
<i>showopts / -showopts</i>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<i>deriv = keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
<i>p</i>	Print estimation results.

## Examples

```
sys1.3sls(i)
```

Estimates SYS1 by the 3SLS method, iterating simultaneously on the weighting matrix and the coefficient vector.

```
nlsys.3sls(showopts,m=500)
```

Estimates NLSYS by 3SLS with up to 500 iterations. The “showopts” option displays the starting values and other estimation options.

## Cross-references

See [Chapter 23, “System Estimation”, on page 679](#) of the *User’s Guide* for discussion of system estimation.

**add**[Group Proc](#) | [Pool Proc](#)

Add series to a group or add cross section members to a pool.

**Syntax**

Group Proc:      `group_name.add arg1 [arg2 arg3 ...]`

Pool Proc:      `pool_name.add id1 [id2 id3 ...]`

List the names of series or group of series to add to the group, or list the cross-section identifiers to add to the pool.

**Examples**

```
dummy.add d11 d12
```

Adds the two series D11 and D12 to the group DUMMY.

```
countries.add us gr
```

Adds US and GR as cross-section members of the pool object COUNTRIES.

**Cross-references**

See “[Groups](#)” on page 85 of the *User’s Guide* for additional discussion of groups. “[Cross-section Identifiers](#)” on page 811 of the *User’s Guide* discusses pool identifiers.

See also [drop](#) (p. 270).

**addassign**[Model Proc](#)

Assign add factors to equations.

**Syntax**

Model Proc:      `model_name.addassign(options) equation_spec`

where *equation\_spec* identifies the equations for which you wish to assign add factors. You may either provide a list of endogenous variables, or you can use one of the following shorthand keywords:

`@all`      All equations.

`@stochastic`      All stochastic equations (no identities).

`@identity`      All identities.

The options identify the type of add factor to be used, and control the assignment behavior for equations where you have previously assigned add factors. `addassign` may be called multiple times to add different types of add factors to different equations. `addassign` may also be called to remove existing add factors.

## Options

i	Intercept shifts (default).
v	Variable shift.
n	None—remove add factors.
c	Change existing add factors to the specified type—if the “c” option is not used, only newly assigned add factors will be given the specified type.

## Examples

```
m1.addassign(v) @all
```

assigns a variable shift to all equations in the model.

```
m1.addassign(c, i) @stochastic
```

changes the stochastic equation add factors to intercept shifts.

```
m1.addassign(v) @stochastic
m1.addassign(v) y1 y2 y3
m1.addassign(i) @identity
```

assigns variable shifts to the stochastic equations and the equations for Y1, Y2, and Y3, and assigns intercept shifts to the identities.

## Cross-references

See “[Using Add Factors](#)” on page 786 of the *User’s Guide*. See also [Chapter 26, “Models”, beginning on page 761](#) of the *User’s Guide* for a general discussion of models.

See [addinit](#) (p. 197).

<b>addinit</b>	<a href="#">Model Proc</a>
----------------	----------------------------

Initialize add factors.

## Syntax

Model Proc:      `model_name.addinit(options) equation_spec`

where *equation\_spec* identifies the equations for which you wish to initialize the add factors. You may either provide a list of endogenous variables, or you may use one of the following shorthand keywords:

@all	All equations
@stochastic	All stochastic equations (no identities)
@identity	All identities

The options control the type of initialization and the scenario for which you want to perform the initialization. `addinit` may be called multiple times to initialize various types of add factors in the different scenarios.

## Options

v = <i>arg</i> ( <i>default</i> = “z”)	Initialize add factors: “z” (set add factor values to zero), “n” (set add factor values so that the equation has no residual when evaluated at actuals), “b” (set add factors to the values of the baseline; override = actual).
s = <i>arg</i> ( <i>default</i> = “a”)	Scenario selection: “a” (set active scenario add factors), “b” (set baseline scenario/actuals add factors), “o” (set active scenario override add factors).

## Examples

```
m1.addinit(v=b) @all
```

sets all of the add factors in the active scenario to the values of the baseline.

```
m1.addinit(v=z) @stochastic  
m1.addinit(v=n) y1 y1 y2
```

first sets the active scenario stochastic equation add factors to zero, and then sets the Y1, Y2, and Y3 equation residuals to zero (evaluated at actuals).

```
m1.addinit(s=b, v=z) @stochastic
```

sets the baseline scenario add factors to zero.

## Cross-references

See “[Using Add Factors](#)” on page 786 of the *User’s Guide*. See also [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a general discussion of models.

See also [addassign \(p. 196\)](#).

**addtext****Graph Proc**

Place text in graphs.

### Syntax

Graph Proc:      `graph_name.addtext(options) text`

Follow the `addtext` keyword with the text to be placed in the graph.

### Options

`font = n`      Set the size of the font.

The following options control the position of the text:

`t`      Top (above the graph and centered).

`l`      Left rotated.

`r`      Right rotated.

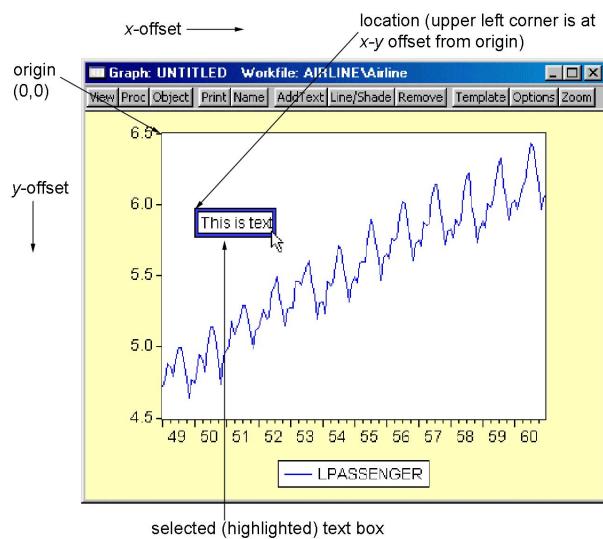
`b`      Below and centered.

`x`      Enclose text in box.

To place text within a graph, you can use explicit coordinates to specify the position of the upper left corner of the text.

Coordinates are set by a pair of numbers  $h, v$  in virtual inches. Individual graphs are always  $4 \times 3$  virtual inches (scatter diagrams are  $3 \times 3$  virtual inches) or a user-specified size, regardless of their current display size.

The origin of the coordinate is the upper left hand corner of the graph. The first number  $h$  specifies how many virtual inches to offset to the right from the origin. The second number  $v$  specifies how many



virtual inches to offset below the origin. The upper left hand corner of the text will be placed at the specified coordinate.

Coordinates may be used with other options, but they must be in the first two positions of the options list. Coordinates are overridden by other options that specify location.

When `addtext` is used with a multiple graph, the text is applied to the whole graph, not to each individual graph.

### Examples

```
freeze(g1) gdp.line  
g1.addtext(t) Fig 1: Monthly GDP (78.1-95.12)
```

places the text “Fig1: Monthly GDP (78.1-95.12)” centered above the graph G1.

```
g1.addtext(.2,.2,X) Seasonally Adjusted
```

places the text “Seasonally Adjusted” in a box within the graph, slightly indented from the upper left corner.

### Cross-references

See also [legend \(p. 319\)](#).

<b>align</b>	<a href="#">Graph Proc</a>
--------------	----------------------------

Align placement of multiple graphs.

### Syntax

Graph Proc:      `graph_name.align(n,h,v)`

### Options

You must specify three numbers (each separated by a comma) in parentheses in the following order: the first number *n* is the number of columns in which to place the graphs, the second number *h* is the horizontal space between graphs, and the third number *v* is the vertical space between graphs. Spacing is specified in virtual inches.

### Examples

```
mygraph.align(3,1.5,1)
```

aligns MYGRAPH with graphs placed in three columns, horizontal spacing of 1.5 virtual inches, and vertical spacing of 1 virtual inch.

```
var var1.ls 1 4 m1 gdp
```

---

```
freeze(impgra) var1.impulse(m, 24) gdp @ gdp m1
impgra.align(2,1,1)
```

estimates a VAR, freezes the impulse response functions as multiple graphs, and realigns the graphs. By default, the graphs are stacked in one column, and the realignment places the graphs in two columns.

### Cross-references

For a detailed discussion of customizing graphs, see [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide*.

See also [graph \(p. 303\)](#).

<b>alpha</b>	<a href="#">Object Declaration</a>
--------------	------------------------------------

Declare an alpha series object.

The `alpha` command creates and optionally initializes an alpha series or modifies an existing series.

### Syntax

Command:	<code>alpha ser_name</code>
Command:	<code>alpha ser_name = formula</code>

The `alpha` command should be followed by either the name of a new series, or an explicit or implicit expression for generating a series. If you create a series and do not initialize it, the series will be filled with the blank string “”.

### Examples

```
alpha x = "initial value"
```

creates a series named X filled with NAs.

Once an alpha is declared, you need not include the `alpha` keyword prior to entering the formula (alternatively, you may use [genr \(p. 295\)](#)). The following example generates an alpha series named VAL that takes value “Low” if either INC is less than or equal to 5000 or EDU is less than 13, and “High” otherwise:

```
alpha val
val = @recode(inc<=5000 or edu<13, "High", "Low")
```

If FIRST and LAST are alpha series containing first and last names, respectively, the commands:

```
alpha name = first + " " + last  
genr name = name + " " + last  
create an alpha series containing the full names.
```

### Cross-references

See “[Alpha Series](#)” on page 145 of the *User’s Guide* for additional discussion. “[Alpha](#)” (p. 152) provides details on the alpha object.

See also [genr](#) (p. 295).

append	<a href="#">Logl Proc</a>   <a href="#">Model Proc</a>   <a href="#">Sspace Proc</a>   <a href="#">System Proc</a>   <a href="#">Valmap Proc</a>   <a href="#">Var Proc</a>
--------	--

Append a specification line to a logl, model, sspace, system, valmap, or var.

### Syntax

Object Proc:	object_name.append <i>text</i>
Var Proc:	var_name.append( <i>options</i> ) <i>text</i>

Type the text to be added after the append keyword. *For vars, you must specify the restrictions type option.*

### Options for Vars

One of the following options is required when using append as a var proc:

svar	Text for identifying restrictions for structural VAR.
coint	Text for restrictions on the cointegration relations and/ or adjustment coefficients.

### Examples

```
model macro2  
macro2.merge eq_m1  
macro2.merge eq_gdp  
macro2.append assign @all f  
macro1.append @trace gdp  
macro2.solve
```

The first line declares a model object. The second and third lines merge existing equations into the model. The fourth and fifth line appends an assign statement and a trace of GDP to the model. The last line solves the model.

---

```

system macro1
macro1.append cons=c(1)+c(2)*gdp+c(3)*cons(-1)
macro1.append inv=c(4)+c(5)*tb3+c(6)*d(gdp)
macro1.append gdp=cons+inv+gov
macro1.append inst tb3 gov cons(-1) gdp(-1)
macro1.gmm
show macro1.results

```

The first line declares a system. The next three lines append the specification of each endogenous variable in the system. The fifth line appends the list of instruments to be used in estimation. The last two lines estimate the model by GMM and display the estimation results.

```

vector(2) svec0=
sspace1.append @mprior svec0

```

appends a line in the state space object SSPACE1 instructing EViews to use the zero vector SVEC0 as initial values for the state vector.

## Cross-references

See “[System Estimation Methods](#)” on page 680, and “[Models](#)” on page 761, and “[Value Maps](#)” on page 155 of the *User’s Guide* for details. See also [cleartext \(p. 237\)](#).

arch	<a href="#">Command</a>    <a href="#">Equation Method</a>
------	--

Estimate generalized autoregressive conditional heteroskedasticity (GARCH) models.

## Syntax

Command:	<code>arch(p,q,options) y [x1 x2 x3] [@ p1 p2 [@ t1 t2]]</code>
Command:	<code>arch(p,q,options) y = expression [@ p1 p2 [@ t1 t2]]</code>
Equation Method:	<code>eq_name.arch(p,q,options) y [x1 x2 x3] [@ p1 p2 [@ t1 t2]]</code>
Equation Method:	<code>eq_name.arch(p,q,options) y = expression [@ p1 p2 [@ t1 t2]]</code>

The ARCH command or method estimates a model with  $p$  ARCH terms and  $q$  GARCH terms. Note the order of the arguments in which the ARCH and GARCH terms are entered, which gives precedence to the ARCH term.

The maximum value for  $p$  or  $q$  is 9; values above will be set to 9. The minimum value for  $p$  is 1. The minimum value for  $q$  is 0. If either  $p$  or  $q$  is not specified, EViews will assume a corresponding order of 1. Thus, a GARCH(1, 1) is assumed by default.

After the “ARCH” keyword, specify the dependent variable followed by a list of regressors in the mean equation.

By default, no exogenous variables (except for the intercept) are included in the conditional variance equation. If you wish to include variance regressors, list them after the mean equation using an “@”-sign to separate the mean from the variance equation.

When estimating component ARCH models, you may specify exogenous variance regressors for the permanent and transitory components. After the mean equation regressors, first list the regressors for the permanent component, followed by an “@”-sign, then the regressors for the transitory component. A constant term is always included as a permanent component regressor.

## Options

egarch	Exponential GARCH.
parch[ = <i>arg</i> ]	Power ARCH. If the optional <i>arg</i> is provided, the power parameter will be set to that value, otherwise the power parameter will be estimated.
cgarch	Component (permanent and transitory) ARCH.
asy = <i>integer</i> (default = 1)	Number of asymmetric terms in the Power ARCH or EGARCH model. The maximum number of terms allowed is 9.
thrsh = <i>integer</i> (default = 0)	Number of threshold terms for GARCH and Component models. The maximum number of terms allowed is 9. For Component models, “thrsh” must take a value of 0 or 1.
archm = <i>arg</i>	ARCH-M (ARCH in mean) specification with the conditional standard deviation (“archm = sd”), the conditional variance (“archm = var”), or the log of the conditional variance (“archm = log”) entered as a regressor in the mean equation.
tdist [= <i>number</i> ]	Estimate the model assuming that the residuals follow a conditional Student’s <i>t</i> -distribution (the default is the conditional normal distribution). Providing the optional number greater than two will fix the degrees of freedom to that value. If the argument is not provided, the degrees of freedom will be estimated.

---

ged [ <i>= number</i> ]	Estimate the model assuming that the residuals follow a conditional GED (the default is the conditional normal distribution). Providing a positive value for the optional argument will fix the GED parameter. If the argument is not provided, the parameter will be estimated.
h	Bollerslev-Wooldridge robust quasi-maximum likelihood (QML) covariance/standard errors. Not available when using the “tdist” or “ged” options.
z	Turn of backcasting for both initial MA innovations and initial variances.
b	Use Berndt-Hall-Hausman (BHHH) as maximization algorithm. The default is Marquardt.
m = <i>integer</i>	Set maximum number of iterations.
c = <i>scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
s	Use the current coefficient values in “C” as starting values (see also <a href="#">param (p. 383)</a> ).
s = <i>number</i>	Specify a number between zero and one to determine starting values as a fraction of preliminary LS estimates (out of range values are set to “s = 1”).
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
deriv = <i>keyword</i>	Set derivative method. The argument <i>keyword</i> should be a one-letter string (“f” or “a” corresponding to fast or accurate numeric derivatives, respectively).
p	Print estimation results.

## Saved results

Most of the results saved for the `ls` command are also available after ARCH estimation; see [ls \(p. 329\)](#) for details.

## Examples

```
arch(4, 0, m=1000, h) sp500 c
```

estimates an ARCH(4) model with a mean equation consisting of the series SP500 regressed on a constant. The procedure will perform up to 1000 iterations, and will report Bollerslev-Wooldridge robust QML standard errors upon completion.

The commands:

```
c = 0.1  
equation arcl.arch(thrsh=1, s, mean=var) @pch(nys) c ar(1)
```

estimate a TARCH(1, 1)-in-mean specification with the mean equation relating the percent change of NYS to a constant, an AR term of order 1, and a conditional variance (GARCH) term. The first line sets the default coefficient vector to 0.1, and the “s” option uses these values as coefficient starting values.

The command:

```
arch(1, 2, asy=0, parch=1.5, ged=1.2) dlog(ibm)=c(1)+c(2)*  
dlog(sp500) @ r
```

estimates a symmetric Power ARCH(2, 1) (autoregressive GARCH of order 2, and moving average ARCH of order 1) model with GED errors. The power of model is fixed at 1.5 and the GED parameter is fixed at 1.2. The mean equation consists of the first log difference of IBM regressed on a constant and the first log difference of SP500. The conditional variance equation includes an exogenous regressor R.

Following estimation, we may save the estimated conditional variance as a series named GARCH1.

```
arcl.makegarch garch1
```

## Cross-references

See [Chapter 20, “ARCH and GARCH Estimation”, on page 585](#) of the *User’s Guide* for a discussion of ARCH models. See also [garch \(p. 295\)](#) and [makegarch \(p. 336\)](#).

archtest	<a href="#">Command</a>    <a href="#">Equation View</a>
----------	--

Test for autoregressive conditional heteroskedasticity (ARCH).

Carries out Lagrange Multiplier (LM) tests for ARCH in the residuals.

## Syntax

Command:      **archtest**(*options*)  
 Equation View:    eq\_name.**archtest**(*options*)

## Options

You must specify the order of ARCH for which you wish to test. The number of lags to be included in the test equation should be provided in parentheses after the **arch** keyword.

*Other Options:*

p	Print output from the test.
---	-----------------------------

## Examples

```
ls output c labor capital
archtest(4)
```

Regresses OUTPUT on a constant, LABOR, and CAPITAL, and tests for ARCH up to order 4.

```
equation eq1.arch sp500 c
eq1.archtest(4)
```

Estimates a GARCH(1,1) model with mean equation of SP500 on a constant and tests for additional ARCH up to order 4. Note that when performing an **archtest** after an **arch** estimation, EViews uses the standardized residuals (the residual of the mean equation divided by the estimated conditional standard deviation) to form the test.

## Cross-references

See “[ARCH LM Test](#)” on page 566 of the *User’s Guide* for further discussion of testing ARCH and [Chapter 20, “ARCH and GARCH Estimation”, on page 585](#) of the *User’s Guide* for a general discussion of working with ARCH models in EViews.

area	<a href="#">Command</a>   <a href="#">Coef View</a>   <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a>   <a href="#">Series View</a>   <a href="#">Sym View</a>   <a href="#">Vector View</a>
------	---

Display area graph view of the object or change existing graph object type to area graph.

Create area or filled line graph from one or more series, or from each column of a matrix object.

## Syntax

Command: **area(options) arg1 [arg2 arg3 ...]**

Object View: **object\_name.area(options)**

Graph Proc: **graph\_name.area(options)**

## Options

### *Template and printing options*

**o = graph\_name** Use appearance options from the specified graph.

**t = graph\_name** Use appearance options and copy text and shading from the specified graph.

**p** Print the area graph.

### *Scale options*

**a (default)** Automatic single scale.

**d** Dual scaling with no crossing. The first series is scaled on the left and all other series are scaled on the right.

**x** Dual scaling with possible crossing. See the “d” option.

**n** Normalized scale (zero mean and unit standard deviation). May not be used with the “s” option.

**s** Stacked area graph. Each area represents the cumulative total of the series listed. The difference between areas corresponds to the value of a series. May not be used with the “l” option.

**l** Area graph for the first series listed and a line graph for all subsequent series. May not be used with the “s” option.

**m** Plot areas in multiple graphs (will override the “s” or “l” options). Not for use with an existing graph object.

### *Panel options*

The following options apply when graphing panel structured data.

<b>panel = arg</b> (default taken from global set- tings)	Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-sec- tion), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quan- tiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “med- mxmn” (plot median, max and min).
--	---

## Examples

```
group g1 ser1 ser2 ser3
g1.area(s)
```

defines a group G1 containing the three series SER1, SER2 and SER3, then plots a stacked area graph of the series in the group.

```
area(l, o=gra1) s1 gdp cons
```

creates an area graph of series S1, together with line graphs of GDP and CONS. The graph uses options from graph GRA1 as a template.

## Cross-references

The graph object is described in greater detail in “[Graph](#)” (p. 159). See [Chapter 14, Graphs, Tables, and Text Objects](#), on page 403 of the *User’s Guide* for a detailed discussion of graphs in EViews, and “[Graph Templates](#)” on page 410 of the *User’s Guide* for a discussion of graph templates. See [graph](#) (p. 303) for graph declaration and other graph types.

arlm	<a href="#">Var View</a>
------	--------------------------

Perform multivariate residual serial correlation LM test using an estimated Var.

## Syntax

Var View:            `var_name.arlm(h, options)`

You must specify the highest order of lag,  $h$ , for which to test.

## Options

name = <i>arg</i>	Save LM statistics in named matrix object. The matrix has $h$ rows and one column.
p	Print test output.

## Examples

```
var var1.ls 1 6 lgdp lm1 lcpi  
show var1.arlm(12, name=lmout)
```

The first line declares and estimates a VAR with 6 lags. The second line displays the serial correlation LM tests for lags up to 12 and stores the statistics in a matrix named LMOUT.

## Cross-references

See “[Diagnostic Views](#)” on page [708](#) of the *User’s Guide* for other VAR diagnostics. See also [qstats \(p. 390\)](#) for related multivariate residual autocorrelation Portmanteau tests.

<b>arma</b>	<a href="#">Equation View</a>
-------------	-------------------------------

Examine ARMA structure of estimated equation.

Provides diagnostic graphical and tabular views that aid you in assessing the structure of the ARMA component of an estimated equation. The view is currently available only for equations specified by list and estimated by least squares that include at least one AR or MA term. There are three views types available: roots, correlogram, and impulse response.

## Syntax

Object View:      `eq_name.arma(type = arg [,options])`

where `eq_name` is the name of an equation object specified by list, estimated by least squares, and contains at least one ARMA term.

## Options

type = <i>arg</i>	Required “ <code>type = </code> ” option selects the type of ARMA structure output: “root” displays the inverse roots of the AR/MA characteristic polynomials, “acf” displays the second moments (autocorrelation and partial autocorrelation) for the data in the estimation sample and for the estimated model, “imp” displays the impulse responses.
-------------------	---

t	Displays the table view of the results for the view specified by the “type = ” option. By default, EViews will display a graphical view of the ARMA results.
hrz = <i>arg</i>	Specifies the maximum lag length for “type = acf”, and the maximum horizon (periods) for “type = imp”.
imp = <i>arg</i>	Specifies the size of the impulse for the impulse response (“type = imp”) view. By default, EViews will use the regression estimated standard error.
save = <i>arg</i>	Stores the results as a matrix object with the specified name. The matrix holds the results roughly as displayed in the table view of the corresponding type. For “type = root”, roots for the AR and MA polynomials will be stored in separate matrices as NAME_AR and NAME_MA, where “NAME” is the name given by the “save = ” option.
p	Print the table or graph output.

## Examples

```
eq1.arma(type=root, save=root)
```

displays and saves the ARMA roots from the estimated equation EQ1. The roots will be placed in the matrix object ROOT.

```
eq1.arma(type=acf, hrz=25, save=acf)
```

computes the second moments (autocorrelation and partial autocorrelations) for the observations in the sample and the estimated model. The results are computed for a 25 period horizon. We save the results in the matrix object ACF.

```
eq1.arma(type=imp, hrz=25, save=imp)
```

computes the 25 period impulse-response function implied by the estimated ARMA coefficients. EViews will use the default 1 standard error of the estimated equation as the shock, and will save the results in the matrix object IMP.

## Cross-references

See “[ARMA Structure](#)” on page 496 of the *User’s Guide* for details.

arroots	<a href="#">Var View</a>
---------	--------------------------

Inverse roots of the characteristic AR polynomial.

### Syntax

Var View: `var_name.arroots(options)`

### Options

<code>name = arg</code>	Save roots in named matrix object. Each root is saved in a row of the matrix, with the first column containing the real, and the second column containing the imaginary part of the root.
<code>graph</code>	Plots the roots together with a unit circle. The VAR is stable if all of the roots are inside the unit circle.
<code>p</code>	Print table of AR roots.

### Examples

```
var var1.ls 1 6 lgdp lm1 lcpi  
var1.arroots(graph)
```

The first line declares and estimates a VAR with 6 lags. The second line plots the AR roots of the estimated VAR.

```
var var1.ls 1 6 lgdp lm1 lcpi  
'store roots  
freeze(tab1) var1.arroots(name=roots)
```

The first line declares and estimates a VAR with 6 lags. The second line stores the roots in a matrix named ROOTS, and the table view as a table named TAB1.

### Cross-references

See “[Diagnostic Views](#)” on page 708 of the *User’s Guide* for other VAR diagnostics.

auto	<a href="#">Command</a>    <a href="#">Equation View</a>
------	--

Compute serial correlation LM (Lagrange multiplier) test.

Carries out Breusch-Godfrey Lagrange Multiplier (LM) tests for serial correlation in the estimation residuals.

## Syntax

Command: **auto(*order, options*)**

Equation View: **eq\_name.auto(*order, options*)**

You must specify the order of serial correlation for which you wish to test. You should specify the number of lags in parentheses after the `auto` keyword, followed by any additional options.

In command form, `auto` tests the residuals from the default equation.

## Options

**p**

Print output from the test.

## Examples

To regress OUTPUT on a constant, LABOR, and CAPITAL, and test for serial correlation of up to order four you may use the commands:

```
ls output c labor capital
auto(4)
```

The commands:

```
output(t) c:\result\artest.txt
equation eq1.ls cons c y y(-1)
eq1.auto(12, p)
```

perform a regression of CONS on a constant, Y and lagged Y, and test for serial correlation of up to order twelve. The first line redirects printed tables/text to the ARTEST.TXT file.

## Cross-references

See “[Serial Correlation LM Test](#)” on page 479 of the *User’s Guide* for further discussion of the Breusch-Godfrey test.

<b>axis</b>	<a href="#">Graph Proc</a>
-------------	----------------------------

Sets axis display characteristics for the graph.

## Syntax

Graph Proc: **graph\_name.axis(*axis\_id*) *options\_list***

The *axis\_id* parameter identifies which of the axes the proc modifies. If no option is specified, the proc will modify all of the axes. *axis\_id* may take on one of the following values:

left / l	Left vertical axis.
right / r	Right vertical axis.
bottom / b	Bottom axis.
top / t	Top axis.
all / a	All axes.

## Options

The options list may include any of the following options:

grid / -grid	[Draw / Do not draw] grid lines.
zeroline / -zeroline	[Draw / Do not draw] a line at zero on the data scale.
ticksout	Draw tickmarks outside the graph axes.
ticksin	Draw tickmarks inside the graph axes.
ticksboth	Draw tickmarks both outside and inside the graph axes.
ticksnone	Do not draw tickmarks.
minor / -minor	[Allow / Do not allow] minor tick marks.
label / -label	[Place / Do not place] labels on the axes.
font( <i>arg</i> )	Set font size of labels, where <i>arg</i> is a numeric font size.
mirror / -mirror	[Label / Do not label] both left and right axes with duplicate axes (single scale graphs only).

Note that the default settings are taken from the Global Defaults.

## Examples

```
graph1.axis(r) zeroline -minor font(12)
```

draws a horizontal line through the graph at zero on the right axis, removes minor ticks, and changes the font size of the right axis labels to 12 point.

```
graph2.axis -mirror
```

turns of mirroring of axes in single scale graphs.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graph options.

See also [scale \(p. 410\)](#), [datelabel \(p. 252\)](#), [options \(p. 358\)](#) and [setelem \(p. 426\)](#).

<b>bar</b>	<a href="#">Command</a>    <a href="#">Coef View</a>   <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a>   <a href="#">Series View</a>   <a href="#">Sym View</a>   <a href="#">Vector View</a>
------------	--

Display bar graph of object, or change existing graph object type to bar graph.

Create bar graph from one or more series or from each column of a matrix object.

Note: when the individual bars in a bar graph become too thin to be distinguished, the graph will automatically be converted into an area graph (see [area \(p. 207\)](#)).

## Syntax

Command:      **bar**(*options*) *arg1* [*arg2 arg3 ...*]

Object View:    *object\_name*.**bar**(*options*)

Graph Proc:    *graph\_name*.**bar**(*options*)

When used as a graph proc, **bar** changes the graph type to a bar graph.

## Options

### *Template and printing options*

<i>o = graph_name</i>	Use appearance options from the specified graph.
-----------------------	--

<i>t = graph_name</i>	Use appearance options and copy text and shading from the specified graph.
-----------------------	--

<b>p</b>	Print the bar graph.
----------	----------------------

### *Scale options*

<b>a (default)</b>	Automatic single scale.
--------------------	-------------------------

<b>d</b>	Dual scaling with no crossing. The first series is scaled on the left, and all other series are scaled on the right.
----------	--

<b>x</b>	Dual scaling with possible crossing. See the “d” option.
----------	--

<b>n</b>	Normalized scale (zero mean and unit standard deviation). May not be used with the “s” option.
----------	--

s	Stacked bar graph. Each bar represents the cumulative total of the series listed. The difference between bars corresponds to the value of the corresponding stacked series. May not be used with the “l” option.
l	Bar graph for the first series and a line graph for all subsequent series. May not be used with the “s” option.
m	Plot bars in multiple graphs. Will override the “s” and the “l” options. Not for use with an existing graph object.

### *Panel options*

The following options apply when graphing panel structured data:

panel = <i>arg</i> (default taken from global set- tings)	Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “med-mxmn” (plot median, max and min).
--	---

### Examples

Plot a bar graph of POP together with line graphs of GDP and CONS:

```
bar(l, x, o=mybar1) pop gdp cons
```

The bar graph is scaled on the left, while the line graphs are scaled on the right. The graph uses options from graph MYBAR1 as a template.

```
group mygrp oldsales newsales  
mygrp.bar(s)
```

The first line defines a group of series and the second line displays a stacked bar graph view of the series in the group.

## Cross-references

See “[Graph Templates](#)” on page 410 of the *User’s Guide* for a discussion of graph templates.

See [graph](#) (p. 303) for graph declaration and additional graph types.

<b>bdstest</b>	<a href="#">Series View</a>
----------------	-----------------------------

Perform BDS test for independence.

The BDS test is a Portmanteau test for time-based dependence in a series. The test may be used for testing against a variety of possible deviations from independence, including linear dependence, non-linear dependence, or chaos.

## Syntax

Series View:      `series_name.bds(options)`

## Options

<code>m = arg</code> ( <i>default</i> = “p”)	Method for calculating $\epsilon$ : “p” (fraction of pairs), “v” (fixed value), “s” (standard deviations), “r” (fraction of range).
<code>e = number</code>	Value for calculating $\epsilon$ .
<code>d = integer</code>	Maximum dimension.
<code>b = integer</code>	Number of repetitions for bootstrap <i>p</i> -values. If option is omitted, no bootstrapping is performed.
<code>o = arg</code>	Name of output vector for final BDS <i>z</i> -statistics.
<code>p</code>	Print output.

## Cross-references

See “[BDS Test](#)” on page 317 of the *User’s Guide* for additional discussion.

<b>binary</b>	<a href="#">Command</a>    <a href="#">Equation Method</a>
---------------	--

Estimate binary dependent variable models.

Estimates models where the binary dependent variable Y is either zero or one (probit, logit, gompit).

## Syntax

Command: **binary(*options*)** *y* *x1* [*x2* *x3* ...]

Equation Method: **eq\_name.binary(*options*)** *y* *x1* [*x2* *x3* ...]

## Options

<b>d = <i>arg</i></b> <i>(default = "n")</i>	Specify likelihood: normal likelihood function, probit ("n"), logistic likelihood function, logit ("l"), Type I extreme value likelihood function, Gompit ("x").
<b>q (<i>default</i>)</b>	Use quadratic hill climbing as the maximization algorithm.
<b>r</b>	Use Newton-Raphson as the maximization algorithm.
<b>b</b>	Use Berndt-Hall-Hausman (BHHH) for maximization algorithm.
<b>h</b>	Quasi-maximum likelihood (QML) standard errors.
<b>g</b>	GLM standard errors.
<b>m = <i>integer</i></b>	Set maximum number of iterations.
<b>c = <i>scalar</i></b>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<b>s</b>	Use the current coefficient values in C as starting values.
<b>s = <i>number</i></b>	Specify a number between zero and one to determine starting values as a fraction of EViews default values (out of range values are set to "s = 1").
<b>showopts / -showopts</b>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<b>p</b>	Print results.

## Examples

To estimate a logit model of Y using a constant, WAGE, EDU, and KIDS, and computing QML standard errors, you may use the command:

```
binary(d=l,h) y c wage edu kids
```

Note that this estimation uses the default global optimization options. The commands:

```
param c(1) .1 c(2) .1 c(3) .1
```

```
equation probit1.binary(s) y c x2 x3
```

estimate a probit model of Y on a constant, X2, and X3, using the specified starting values.  
The commands:

```
coef beta_probit = probit1.@coefs
matrix cov_probit = probit1.@coefcov
```

store the estimated coefficients and coefficient covariances in the coefficient vector  
BETA\_PROBIT and matrix COV\_PROBIT.

### Cross-references

See “[Binary Dependent Variable Models](#)” on page 605 of the *User’s Guide* for additional discussion.

<b>block</b>	<a href="#">Model View</a>
--------------	----------------------------

Display the model block structure view.

Show the block structure of the model, identifying which blocks are recursive and which blocks are simultaneous.

### Syntax

Model View:      `model_name.block(options)`

### Options

<code>p</code>	Print the block structure view.
----------------	---------------------------------

### Cross-references

See “[Block Structure View](#)” on page 782 of the *User’s Guide* for details. Chapter 26 of the *User’s Guide* provides a general discussion of models.

See also [eqs](#) (p. 275), [text](#) (p. 482) and [vars](#) (p. 501) for alternative representations of the model.

<b>boxplot</b>	<a href="#">Group View</a>
----------------	----------------------------

Display boxplot of each series in the group.

Create a boxplot graph view containing boxplots for each series in the group.

## Syntax

Group View:      `group_name.boxplot(options)`

Follow the group name with a period, the keyword, and any options. The default settings are to display fixed width boxplots for each series using individual samples, with all basic elements drawn (mean, med, staples, whiskers, near outliers, far outliers), and with shading representing the approximate confidence intervals for the median.

## Options

### *Options to control initial display*

<code>nomean</code>	Do not display means.
<code>nomed</code>	Do not display medians.
<code>nostaple</code>	Do not display staples.
<code>nowhisk</code>	Do not display whiskers.
<code>nonearout</code>	Do not display near outliers.
<code>nofarout</code>	Do not display far outliers.
<code>width = arg</code> <i>(default = “fixed”)</i>	Boxplot width: “fixed” (fixed width boxplots), “n” (width proportional to number of observations), “rootn” (width proportional to square root of number of observations).
<code>ci = arg</code> <i>(default = “shade”)</i>	95 % confidence interval for median: “none” (do not display), “shade” (display interval as shaded area), “notch” (display interval using notched boxes).

### *Options to control calculation of boxplots*

<code>b</code>	Balance sample.
<code>q = arg</code> <i>(default = “r”)</i>	Compute quantiles using the specified definition: “b” (Blom), “r” (Rankit-Cleveland), “o” (simple fraction), “t” (Tukey), “v” (van der Waerden).

### *Other options*

<code>p</code>	Print the view.
----------------	-----------------

## Examples

`grp1.boxplot`

displays default boxplots for all of the series in the group GRP1.

```
grp2.boxplot(b, nowhisk)
```

displays boxplots for the series in GRP2 after balancing the sample. The boxplots will not have whiskers drawn.

## Cross-references

See “[Boxplots](#)” on page 397 of the *User’s Guide* for additional discussion.

See also [boxplotby](#) (p. 221) and [stats](#) (p. 462). For customization of a boxplot graph object, see [setelem](#) (p. 426) and [setbpelem](#) (p. 421).

<b>boxplotby</b>	<a href="#">Series View</a>
------------------	-----------------------------

Display the boxplots of a series classified into categories.

Create a boxplot graph view containing boxplots of the elements of a series classified into categories by one or more series.

## Syntax

Series View:      `series_name.boxplotby(options) classifier_names`

Follow the series name with a period, the keyword, and a name (or a list of names) for the series or groups by which to classify. The default settings are to display fixed width boxplots for each category, with all basic elements drawn (mean, med, staples, whiskers, near outliers, far outliers), and with shading representing the approximate confidence intervals for the median.

## Options

*Options to control display*

nomean	Do not display means.
nomed	Do not display medians.
nostaple	Do not display staples.
nowhisk	Do not display whiskers.
nonearout	Do not display near outliers.
nofarout	Do not display far outliers.
nolabel	Do not display axis labels.

width = <i>arg</i> (default = "fixed")	Boxplot width: "fixed" (fixed width boxplots), "n" (width proportional to number of observations), "rootn" (width proportional to square root of number of observations).
ci = <i>arg</i> (default = "shade")	95 % confidence interval for median: "none" (do not display), "shade" (display interval as shaded area), "notch" (display interval using notched boxes).

#### Options to control calculation of boxplots

dropna (default), keepna	[Drop/Keep] NA as a category.
total	Create category for entire series.
q = <i>arg</i> (default = "r")	Compute quantiles using the specified definition: "b" (Blom), "r" (Rankit-Cleveland), "o" (simple fraction), "t" (Tukey), "v" (van der Waerden).

#### Options to control binning

v = <i>integer</i> (default = 100)	Bin categories if classification series take on more than the specified number of distinct values.
nov	Do not bin based on the number of values of the classification series.
a = <i>number</i> (default = 2)	Bin categories if average cell count is less than the specified number.
noa	Do not bin based on the average cell count.
b = <i>integer</i> (default = 5)	Set maximum number of binned categories.

#### Other options

p	Print the view.
---	-----------------

#### Examples

```
wage.boxplotby sex race
```

displays boxplots for the series WAGE categorized by the values of SEX and RACE.

---

```
income.boxplotby(total, keepna, width=n) sex race
```

displays boxplots for INCOME classified by SEX and RACE, with missing values in the classifier series treated as categories, and an additional boxplot drawn for the entire sample of observations. The boxes will be drawn with variable widths proportional to the number of observations in each category.

## Cross-references

See “[Boxplots](#)” on page 397 of the *User’s Guide*.

See also [boxplot \(p. 219\)](#), and [statby \(p. 457\)](#). For customization of a boxplot graph object, see [setelem \(p. 426\)](#) and [setbpelem \(p. 421\)](#).

<b>bpf</b>	<a href="#">Series Proc</a>
------------	-----------------------------

Compute and display the band-pass filter of a series.

Computes, and displays a graphical view of the Baxter-King fixed length symmetric, Christiano-Fitzgerald fixed length symmetric, or the Christiano-Fitzgerald full sample asymmetric band-pass filter of the series.

The view will show the original series, the cyclical component, and non-cyclical component in a single graph. For non time-varying filters, a second graph will show the frequency responses.

## Syntax

Series Proc:      `series_name.bpf(options) {cyc_name}`

Follow the bpf keyword with any desired options, and the optional name to be given to the cyclical component. If you do not provide *cyc\_name*, the filtered series will be named BPFILTER## where ## is a number chosen to ensure that the name is unique.

## Options

<code>type = arg</code> ( <i>default</i> = “bk”)	Specify the type of band-pass filter: “bk” is the Baxter-King fixed length symmetric filter, “cffix” is the Christiano-Fitzgerald fixed length symmetric filter, “cfasym” is the Christiano-Fitzgerald full sample asymmetric filter.
---	---

<code>low = number,</code> <code>high = number</code>	Low ( $P_L$ ) and high ( $P_H$ ) values for the cycle range to be passed through (specified in periods of the workfile frequency).  Defaults to the workfile equivalent corresponding to a range of 1.5–8 years for semi-annual to daily workfiles; otherwise sets “low = 2”, “high = 8”.  The arguments must satisfy $2 \leq P_L < P_H$ . The corresponding frequency range to be passed through will be $(2\pi/P_H, 2\pi/P_L)$ .
<code>lag = integer</code>	Fixed lag length (positive integer). Sets the fixed lead/lag length for fixed length filters (“type = bk” or “type = cffix”). Must be less than half the sample size. Defaults to the workfile equivalent of 3 years for semi-annual to daily workfiles; otherwise sets “lag = 3”.
<code>iorder = {0,1}</code> ( <i>default</i> = 0)	Specifies the integration order of the series. The default value, “0” implies that the series is assumed to be (covariance) stationary; “1” implies that the series contains a unit root.  The integration order is only used in the computation of Christiano-Fitzgerald filter weights (“type = cffix” or “type = cfasymp”). When “iorder = 1”, the filter weights are constrained to sum to zero.
<code>detrend = arg</code> ( <i>default</i> = “n”)	Detrending method for Christiano-Fitzgerald filters (“type = cffix” or “type = cfasymp”).  You may select the default argument “n” for no detrending, “c” to demean, or “t” to remove a constant and linear trend.  You may use the argument “d” to remove drift, if the option “iorder = 1” is also specified.
<code>nogain</code>	Suppresses plotting of the frequency response (gain) function for fixed length symmetric filters (“type = bk” or “type = cffix”). By default, EViews will plot the gain function.
<code>noncyc = arg</code>	Specifies a name for a series to contain the non-cyclical series (difference between the actual and the filtered series). If no name is provided, the non-cyclical series will not be saved in the workfile.

w = arg

Store the filter weights as an object with the specified name. For fixed length symmetric filters (“type = bk” or “type = cfix”), the saved object will be a matrix of dimension  $1 \times (q + 1)$  where  $q$  is the user-specified lag length order. For these filters, the weights on the leads and the lags are the same, so the returned matrix contains only the one-sided weights. The filtered series  $z_t$  may be computed as:

$$z_t = \sum_{c=1}^{q+1} w(1, c)y_{t+1-c} + \sum_{c=2}^{q+1} w(1, c)y_{t+c-1}$$

for  $t = q + 1, \dots, n - q$ .

For time-varying filters, the weight matrix is of dimension  $n \times n$  where  $n$  is the number of non-missing observations in the current sample. Row  $r$  of the matrix contains the weighting vector used to generate the  $r$ -th observation of the filtered series, where column  $c$  contains the weight on the  $c$ -th observation of the original series. The filtered series may be computed as:

$$z_t = \sum_{c=1}^T w(r, c)y_c \quad r = 1, \dots, T$$

where  $y_t$  is the original series and  $w(r, c)$  is the  $(r, c)$  element of the weighting matrix. By construction, the first and last rows of the weight matrix will be filled with missing values for the symmetric filter.

p

Print the graph.

## Examples

Suppose we are working in a quarterly workfile and we issue the following command:

```
lgdp.bpf(type=bk, low=6, high=32) cyc0
```

EViews will compute the Baxter-King band-pass filter of the series LGDP. The periodicity of cycles extracted ranges from 6 to 32 quarters, and the filtered series will be saved in the workfile in CYC0. The BK filter uses the default lag of 12 (3 years of quarterly data).

Since this is a fixed length filter, EViews will display both a graph of the cyclical/original/non-cyclical series, as well as the frequency response (gain) graph. To suppress the latter graph, we could enter a command containing the “nogain” option:

```
lgdp.bpf (type=bk,low=6,high=32,lag=12,nogain)
```

In this example, we have also overridden the default by specifying a fixed lag of 12 (quarters). Since we have omitted the name for the cyclical series, EViews will create a series with a name like BPFILTER01 to hold the results.

To compute the asymmetric Christiano-Fitzgerald filter, we might enter a command of the form:

```
lgdp.bpf (type=cfasym,low=6,high=32,noncyc=non1,weight=wm) cyc0
```

The cyclical components are saved in CYC0, the non-cyclical in NON1, and the weighting matrix in WM.

### Cross-references

See “[Frequency \(Band-Pass\) Filter](#)” on page [346](#). See also [hpf](#) (p. 310).

<b>bplabel</b>	<a href="#">Graph Proc</a>
----------------	----------------------------

Specify labeling of the bottom axis in boxplots.

Sets options that are specific to the appearance of the bottom axis of boxplots.

Additional options that affect the appearance of the axis may be set using [axis](#) (p. 213) using the “bottom” option. These options include tick control, label and font options, and grid lines.

### Syntax

Graph Proc:      graph\_name.**bplabel** *option\_list*

The *option\_list* may contain one or more of the following:

interval( <i>step_size</i> [, <i>steps</i> ] [, <i>align_id</i> ]) ( <i>default step_size</i> = “auto”)	<p>Specify the label frequency: <i>step_size</i> can take on one of the following values: “auto” (automatic), “ends” (only label endpoints; first and last boxplot), “cust” (custom labels that use <i>align_id</i> and <i>steps</i> to determine which labels to display).</p> <p><i>steps</i> is an optional number indicating the number of steps between labels (a single step is defined as one box). The default value is automatically determined.</p> <p><i>align_id</i> is an optional integer representing a box number to be used as a base. The base box will always receive a label, and subsequent labeling will be determined using the specified <i>steps</i>. The default value is 1.</p> <p>Note: the <i>align_id</i> may lie above the current number of boxes.</p>
angle( <i>arg</i> )	Rotates the text label to the specified angle, where <i>arg</i> can be either a number, measured from -90 to 90 degrees, or “auto” (or “a”) for automatically determined angling. The angle is measured in 15 degree increments, counterclockwise from the horizontal axis.
label( <i>id</i> , “ <i>text</i> ”) / -label( <i>id</i> , “ <i>text</i> ”)	<p>[Show / Hide] the label of the <i>id</i> box. <i>text</i> is a new label for the box, and should be enclosed in quotes.</p> <p>You may optionally precede the “label” keyword with “+”.</p>

## Examples

```
graph01.bplabel interval(ends) angle(45)
```

will display the endpoint labels only at a 45 degree angle.

```
graph01.bplabel interval(cust, 10, 2) label(2, "North")
```

displays labels on every tenth box, centered around the second. The label for the second box is also changed to “North”.

## Cross-references

See “[Boxplots](#)” on page 397 of the *User’s Guide* for a description of boxplots.

See [setelem](#) (p. 426) to modify line and symbol attributes. See also [setbpelem](#) (p. 421), [options](#) (p. 358), [axis](#) (p. 213), and [scale](#) (p. 410).

cause	<a href="#">Command</a>    <a href="#">Group View</a>
-------	---

Granger causality test.

Performs pairwise Granger causality tests between (all possible) pairs of the listed series or group of series.

### Syntax

Command:      `cause(n, options) ser1 ser2 ser3`

Group View:    `group_name.cause(n, options)`

In command form, you should list the series or group of series for which you wish to test for Granger causality.

### Options

You must specify the number of lags *n* to use for the test by providing an integer in parentheses after the keyword. Note that the regressors of the test equation are a constant and the specified lags of the pair of series under test.

#### *Other options:*

p	Print output of the test.
---	---------------------------

### Examples

To compute Granger causality tests of whether GDP Granger causes M1 and whether M1 Granger causes GDP, you may enter the command:

```
cause(4) gdp m1
```

The regressors of each test are a constant and four lags of GDP and M1.

The commands:

```
group macro m1 gdp r  
macro.cause(12,p)
```

print the result of six pairwise Granger causality tests for the three series in the MACRO group. The regressors of each test are a constant and twelve lags of the two series under test (and do not include lagged values of the third series in the group).

### Cross-references

See “[Granger Causality](#)” on page 376 of the *User’s Guide* for a discussion of Granger’s approach to testing hypotheses about causality.

See also [var \(p. 500\)](#).

<b>ccopy</b>	<a href="#">Command</a>
--------------	-------------------------

Copy one or more series from the DRI Basic Economics database to EViews data bank (.DB) files.

*You must have the DRI database installed on your computer to use this feature.*

### Syntax

Command:      **ccopy** *series\_name*

Type the name of the series or wildcard expression for series you want to copy after the **ccopy** keyword. The data bank files will be stored in the default directory with the same name as the series names in the DRI database. You can supply path information to indicate the directory for the data bank files.

### Examples

The command:

`ccopy lhur`

copies the DRI series LHUR to LHUR.DB file in the default path directory.

`ccopy b:gdp c:\nipadata\gnet`

copies the GDP series to GDP.DB file in the B: drive and the GNET series to the GNET.DB file in C:\NIPADATA.

### Cross-references

See also [cfetch \(p. 234\)](#), [clabel \(p. 237\)](#), [store \(p. 465\)](#), [fetch \(p. 279\)](#).

<b>cd</b>	<a href="#">Command</a>
-----------	-------------------------

Change default directory.

The **cd** command changes the current default working directory. The current working directory is displayed in the “Path = ...” message in the bottom right of the EViews window.

### Syntax

Command:      **cd** *path\_name*

## Examples

To change the default directory to “SAMPLE DATA” in the A: drive, you may issue the command:

```
cd "a:\sample data"
```

Notice that the directory name is surrounded by double quotes. If your name does not contain spaces, you may omit the quotes. The command:

```
cd a:\test
```

changes the default directory to A:\TEST.

Subsequent save operations will save into the default directory, unless you specify a different directory at the time of the operation.

## Cross-references

See [Chapter 3, “Workfile Basics”, on page 43](#) of the *User’s Guide* for further discussion of basic operations in EViews.

See also [wfsave \(p. 512\)](#), [pagesave \(p. 373\)](#), and [save \(p. 407\)](#).

## cdfplot

[Group View](#) | [Series View](#)

### Empirical distribution functions.

Displays empirical cumulative distribution functions, survivor functions, and quantiles functions with standard errors.

### Syntax

Object View:      series\_name.cdfplot(*options*)

Note: due to the potential for extended computation times, standard errors will only be computed if there are fewer than 2500 observations.

### Options

c (default)	Plot the empirical CDF.
s	Plot the empirical survivor function.
q	Plot the empirical quantiles.
a	Plot all CDF, survivor, and quantiles.
n	Do not include standard errors.

<code>q = arg</code> <code>(default = "r")</code>	Compute quantiles using the definition: "b" (Blom), "r" (Rankit-Cleveland), "o" (simple fraction), "t" (Tukey), "v" (van der Waerden).
<code>o = arg</code>	Output results to matrix. Each column of the results matrix corresponds to one of the values used in plotting: (in order) evaluation points, the value, standard errors. If there are multiple graphs, all of the columns for the first graph are presented prior to the columns for the subsequent graph.
<code>p</code>	Print the distribution function(s).

## Examples

To plot the empirical cumulative distribution function of the series LWAGE:

```
lwage.cdfplot
```

## Cross References

See [Chapter 13, “Statistical Graphs from Series and Groups”, on page 379](#) of the *User’s Guide* for a discussion of empirical distribution graphs.

See [qqplot \(p. 389\)](#).

**cellipse**

[Equation View](#) | [Logl View](#) | [Pool View](#) | [Sspace View](#) | [System View](#)

Confidence ellipses for coefficient restrictions.

The `cellipse` view displays confidence ellipses for pairs of coefficient restrictions for an estimation object.

## Syntax

Object View:      `object_name.cellipse(options) restrictions`

Enter the object name, followed by a period, and the keyword `cellipse`. This should be followed by a list of the coefficient restrictions. Joint (multiple) coefficient restrictions should be separated by commas.

## Options

ind = <i>arg</i>	Specifies whether and how to draw the individual coefficient intervals. The default is “ind = line” which plots the individual coefficient intervals as dashed lines. “ind = none” does not plot the individual intervals, while “ind = shade” plots the individual intervals as a shaded rectangle.
size = <i>number</i> (default = 0.95)	Set the size (level) of the confidence ellipse. You may specify more than one size by specifying a space separated list enclosed in double quotes.
dist = <i>arg</i>	Select the distribution to use for the critical value associated with the ellipse size. The default depends on estimation object and method. If the parameter estimates are least-squares based, the $F(2, n - 2)$ distribution is used; if the parameter estimates are likelihood based, the $\chi^2(2)$ distribution will be employed. “dist = f” forces use of the $F$ -distribution, while “dist = c” uses the $\chi^2$ distribution.
p	Print the graph.

## Examples

The two commands:

```
eq1.cellipse c(1), c(2), c(3)  
eq1.cellipse c(1)=0, c(2)=0, c(3)=0
```

both display a graph showing the 0.95-confidence ellipse for C(1) and C(2), C(1) and C(3), and C(2) and C(3).

```
pool1.cellipse(dist=c, size="0.9 0.7 0.5") c(1), c(2)
```

displays multiple confidence ellipses (contours) for C(1) and C(2).

## Cross-references

See “[Confidence Ellipses](#)” on page 554 of the *User’s Guide* for discussion.

See also [wald](#) (p. 502).

censored	<a href="#">Command</a>    <a href="#">Equation Method</a>
----------	--

### Estimation of censored and truncated models.

Estimates models where the dependent variable is either censored or truncated. The allowable specifications include the standard Tobit model.

### Syntax

Command: **censored**(*options*) *y x1 x2 x3*

Equation Method: **eq\_name.censored**(*options*) *y x1 x2 x3*

### Options

<i>l = number</i> ( <i>default</i> = 0)	Set value for the left censoring limit.
<i>r = number</i> ( <i>default</i> = none)	Set value for the right censoring limit.
<i>l = series_name, i</i>	Set series name of the indicator variable for the left censoring limit.
<i>r = series_name, i</i>	Set series name of the indicator variable for the right censoring limit.
<i>t</i>	Estimate truncated model.
<i>d = arg</i> ( <i>default</i> = "n")	Specify error distribution: normal ("n"), logistic ("l"), Type I extreme value ("x").
<i>q</i> ( <i>default</i> )	Use quadratic hill climbing as the maximization algorithm.
<i>r</i>	Use Newton-Raphson as the maximization algorithm.
<i>b</i>	Use Berndt-Hall-Hausman for maximization algorithm.
<i>h</i>	Quasi-maximum likelihood (QML) standard errors.
<i>g</i>	GLM standard errors.
<i>m = integer</i>	Set maximum number of iterations.
<i>c = scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.

s	Use the current coefficient values in C as starting values.
s = number	Specify a number between zero and one to determine starting values as a fraction of EViews default values (out of range values are set to “s = 1”).
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
p	Print results.

## Examples

The command:

```
censored(h) hours c wage edu kids
```

estimates a censored regression model of HOURS on a constant, WAGE, EDU, and KIDS with QML standard errors. This command uses the default normal likelihood, with left-censoring at HOURS = 0, no right censoring, and the quadratic hill climbing algorithm.

## Cross-references

See [Chapter 21, “Discrete and Limited Dependent Variable Models”, on page 605](#) of the *User’s Guide* for discussion of censored and truncated regression models.

<b>cfetch</b>	<a href="#">Command</a>
---------------	-------------------------

Fetch a series from the DRI Basic Economics database into a workfile.

`cfetch` reads one or more series from the DRI Basic Economics Database into the active workfile. *You must have the DRI database installed on your computer to use this feature.*

## Syntax

Command:      `cfetch series_name`

## Examples

```
cfetch lhur gdp gnet
```

reads the DRI series LHUR, GDP, and GNET into the current active workfile, performing frequency conversions if necessary.

## Cross-references

EViews' automatic frequency conversion is described in “[Frequency Conversion](#)” beginning on page 107 of the *User’s Guide*.

See also [ccopy](#) (p. 229), [clabel](#) (p. 237), [store](#) (p. 465), [fetch](#) (p. 279).

<b>chdir</b>	<a href="#">Command</a>
--------------	-------------------------

Change default directory.

See [cd](#) (p. 229).

<b>checkderivs</b>	<a href="#">Log View</a>
--------------------	--------------------------

Check derivatives of likelihood object.

Displays a table containing information on numeric derivatives and, if available, the user-supplied analytic derivatives.

### Syntax

Log View:      `logl_name.checkderiv(options)`

### Options

**p**      Print the table of results.

### Examples

`ll1.checkderiv`

displays a table that evaluates the numeric derivatives of the logl object LL1.

## Cross-references

See [Chapter 22, “The Log Likelihood \(LogL\) Object”, on page 655](#) for a general discussion of the likelihood object and the “@deriv” statement.

See also [grads](#) (p. 302) and [makegrads](#) (p. 337).

**chow**[Command](#) || [Equation View](#)

Chow test for stability.

Carries out Chow breakpoint or Chow forecast tests for parameter constancy.

### Syntax

Command:      **chow**(*options*) *obs1* [*obs2 obs3 ...*]

Equation View:    *eq\_name.chow*(*options*) *obs1* [*obs2 obs3 ...*]

You must provide the breakpoint observations (using dates or observation numbers) to be tested. To specify more than one breakpoint, separate the breakpoints by a space.

### Options

f                Chow forecast test. For this option, you must specify a single breakpoint to test (default performs breakpoint test).

p                Print the result of test.

### Examples

The commands:

```
ls m1 c gdp cpi ar(1)  
chow 1970Q1 1980Q1
```

perform a regression of M1 on a constant, GDP, and CPI with first order autoregressive errors, and employ a Chow breakpoint test to determine whether the parameters before the 1970's, during the 1970's, and after the 1970's are "stable".

To regress the log of SPOT on a constant, the log of P\_US, and the log of P\_UK, and to carry out the Chow forecast test starting from 1973, enter the commands:

```
equation ppp.ls log(spot) c log(p_us) log(p_uk)  
ppp.chow(f) 1973
```

### Cross-references

See “[Chow's Breakpoint Test](#)” on page 568 of the *User's Guide* for further discussion.

See also [rls \(p. 400\)](#).

<b>clabel</b>	<a href="#">Command</a>
---------------	-------------------------

Display a DRI Basic Economics database series description.

`clabel` reads the description of a series from the DRI Basic Economics Database and displays it in the status line at the bottom of the EViews window.

Use this command to verify the contents of a given DRI database series name. *You must have the DRI database installed on your computer to use this feature.*

### Syntax

Command:      **clabel** series\_name

### Examples

```
clabel lhur
```

displays the description of the DRI series LHUR on the status line.

### Cross-references

See also [ccopy \(p. 229\)](#), [cfetch \(p. 234\)](#), [read \(p. 391\)](#), [fetchn \(p. 279\)](#).

<b>cleartext</b>	<a href="#">Var Proc</a>
------------------	--------------------------

Clear restriction text from a var object.

### Syntax

Var Proc:      var\_name.cleartext(arg)

You must specify the text type you wish to clear using one of the following arguments:

svar	Clear text of identifying restrictions for a structural VAR.
------	--

coint	Clear text of restrictions on the cointegration relations and/or adjustment coefficients.
-------	---

### Examples

```
var1.cleartext(svar)
var1.append(svar) @lr2(@u1) = 0
```

The first line clears the structural VAR identifying restrictions in VAR1. The next line specifies a new long-run restriction for a structural factorization.

## Cross-references

See [Chapter 24, “Vector Autoregression and Error Correction Models”, on page 705](#) of the *User’s Guide* for a discussion of VARs.

See also [append \(p. 202\)](#).

<b>close</b>	<a href="#">Command</a>
--------------	-------------------------

Close object, program, or workfile.

Closing an object eliminates its window. If the object is named, it is still displayed in the workfile as an icon, otherwise it is deleted. Closing a program or workfile eliminates its window and removes it from memory. If a workfile has changed since you activated it, you will see a dialog box asking if you want to save it to disk.

## Syntax

Command:      **close** *object\_name*

## Examples

```
close gdp graph1 table2  
closes the three objects GDP, GRAPH1, and TABLE2.
```

```
lwage.hist  
close lwage
```

opens the LWAGE window and displays the histogram view of LWAGE, then closes the window.

## Cross-references

See [Chapter 1, “Introduction”, on page 11](#) of the *User’s Guide* for a discussion of basic control of EViews.

<b>coef</b>	<a href="#">Object Declaration</a>
-------------	------------------------------------

Declare a coefficient (column) vector.

## Syntax

Command:      **coef**(*n*) *coef\_name*

Follow the **coef** keyword with the number of coefficients in parentheses, and a name for the object. If you omit the number of coefficients, EViews will create a vector of length 1.

## Examples

```
coef(2) slope
ls lwage = c(1)+slope(1)*edu+slope(2)*edu^2
```

The first line declares a coef object of length 2 named SLOPE. The second line estimates a least squares regression and stores the estimated slope coefficients in SLOPE.

```
arch(2,2) sp500 c
coef beta = c
coef(6) beta
```

The first line estimates a GARCH(2,2) model using the default coef vector C (note that the “C” in an equation specification refers to the constant term, a series of ones.) The second line declares a coef object named BETA and copies the contents of C to BETA (the “C” in the assignment statement refers to the default coef vector). The third line resizes BETA to “chop off” all elements except the first six. Note that since EViews stores coefficients with equations for later use, you will generally not need to perform this operation to save your coefficient vectors.

## Cross-references

See “[Coef](#)” on page 153 for a full description of the coef object. See also [vector \(p. 501\)](#).

<b>coefcov</b>	<a href="#">Equation View</a>   <a href="#">Logl View</a>   <a href="#">Pool View</a>   <a href="#">Sspace View</a>   <a href="#">System View</a>
----------------	---

**Coefficient covariance matrix.**

Displays the covariances of the coefficient estimates for objects containing an estimated equation or equations.

## Syntax

Object View:      object\_name.coefcov(*options*)

## Options

<b>p</b>	Print the coefficient covariance matrix.
----------	--

## Examples

The set of commands:

```
equation eq1.ls lwage c edu edu^2 union
eq1.coefcov
```

declares and estimates equation EQ1 and displays the coefficient covariance matrix in a window. To store the coefficient covariance matrix as a sym object, use “@cofcov”:

```
sym eqcov = eq1.@cofcov
```

### Cross-references

See also [coef](#) (p. 238) and [spec](#) (p. 454).

coint	<a href="#">Command</a>    <a href="#">Group View</a>   <a href="#">Var View</a>
-------	--

Johansen's cointegration test.

### Syntax

- Command:      `coint(test_option,n,option) y1 y2 [y3 ...]`  
Command:      `coint(test_option,n,option) y1 y2 [y3 ...] [@ x1 x2 x3 ...]`  
Group View:    `group_name.coint(test_option,n,option)`  
Var View:      `var_name.coint(test_option,n,option) [@ x1 x2 x3 ...]`

In command form, you should enter the `coint` keyword followed by a list of series or group names for you wish to test for cointegration. Each name should be separated by a space. To use exogenous variables, such as seasonal dummy variables, in the test, list the names after an “@”-sign.

When used as a group or var view, `coint` tests for cointegration among the series in the group or var. By default, if the var object contains exogenous variables, the cointegration test will use those exogenous variables; however, if you explicitly list the exogenous variables with an “@”-sign, then only the listed variables will be used in the test.

The EViews 5 output for cointegration tests now displays *p*-values for the rank test statistics. These *p*-values are computed using the response surface coefficients as estimated in MacKinnon, et. al. (1999). The 0.05 critical values are now based on the response surface coefficients from MacKinnon-Haug-Michelis. *Note: the reported critical values assume no exogenous variables other than an intercept and trend.*

### Options

You must specify the test option followed by the number of lags *n*. You must choose one of the following six test options:

- |   |  |
|---|--|
| a | No deterministic trend in the data, and no intercept or trend in the cointegrating equation. |
|---|--|

b	No deterministic trend in the data, and an intercept but no trend in the cointegrating equation.
c	Linear trend in the data, and an intercept but no trend in the cointegrating equation.
d	Linear trend in the data, and both an intercept and a trend in the cointegrating equation.
e	Quadratic trend in the data, and both an intercept and a trend in the cointegrating equation.
s	Summarize the results of all 5 options (a-e).

*Other Options:*

restrict	Impose restrictions as specified by the <code>append</code> ( <code>coint</code> ) proc.
<code>m = integer</code>	Maximum number of iterations for restricted estimation (only valid if you choose the <code>restrict</code> option).
<code>c = scalar</code>	Convergence criterion for restricted estimation. (only valid if you choose the <code>restrict</code> option).
<code>save = mat_name</code>	Stores test statistics as a named matrix object. The <code>save =</code> option stores a $(k + 1) \times 4$ matrix, where $k$ is the number of endogenous variables in the VAR. The first column contains the eigenvalues, the second column contains the maximum eigenvalue statistics, the third column contains the trace statistics, and the fourth column contains the log likelihood values. The $i$ -th row of columns 2 and 3 are the test statistics for rank $i - 1$ . The last row is filled with NAs, except the last column which contains the log likelihood value of the unrestricted (full rank) model.
<code>cvtype = ol</code>	Display 0.05 and 0.01 critical values from Osterwald-Lenum (1992). This option reproduces the output from version 4. The default is to display critical values based on the response surface coefficients from MacKinnon-Haug-Michelis (1999). Note that the argument on the right side of the equals sign are letters, not numbers 0-1.

`cysize = arg`  
*(default = 0.05)*      Specify the size of MacKinnon-Haug-Michelis (1999) critical values to be displayed. The size must be between 0.0001 and 0.9999; values outside this range will be reset to the default value of 0.05. This option is ignored if you set “`cvtype = ol`”.

`p`      Print output of the test.

## Examples

```
coint(s, 4) gdp m1 tb3
```

summarizes the results of the Johansen cointegration test among the three series GDP, M1, and TB3 for all five specifications of trend. The test equation uses lags of up to order four.

```
var1.coint(c, 12) @
```

carries out the Johansen test for the series in the var object named VAR1. The “@”-sign without a list of exogenous variables ensures that the test does not include any exogenous variables in VAR1.

## Cross-references

See “[Cointegration Test](#)” on page [723](#) of the *User’s Guide* for details on the Johansen test.

See also [ec](#) (p. [271](#)).

<b>comment</b>	<a href="#">Table Proc</a>
----------------	----------------------------

Adds or removes a comment in a table cell.

## Syntax

Table Proc:      `table_name.comment(cell_arg) [comment_arg]`

where `cell_arg`, which identifies the cell, can take one of the following forms:

`cell`      Cell identifier. You can reference cells using either the column letter and row number (e.g., “A1”), or by using “R” followed by the row number followed by “C” and the column *number* (e.g., “R1C2”).

`row[,] col`      Row number, followed by column letter or number (e.g., “2,C”, or “2,3”), separated by “,”.

and where `comment_arg` is a string expression enclosed in double quotes. If `comment_arg` is omitted, a previously defined comment will be removed.

## Examples

To add a comment, “hello world”, to the cell in the second row, fourth column, you may use one of the following:

```
tab1.comment(d2) "hello world"
tab1.comment(r2c4) "hello world"
tab1.comment(2,d) "hello world"
tab1.comment(2,4) "hello world"
```

To remove a comment, simply omit the *comment\_arg*:

```
tab1.comment(d2)
```

clears the comment (if present) from the second row, fourth column.

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”](#). See also [set-lines \(p. 440\)](#) and [setmerge \(p. 441\)](#).

<b>control</b>	<a href="#">Model Proc</a>
----------------	----------------------------

Solve for values of control variable so that the target series matches a trajectory.

## Syntax

Model Proc:      `model_name.control control_var target_var trajectory`

Specify the name of the control variable, followed by the target variable, and then the trajectory you wish to achieve for the target variable. EViews will solve for the values of the control so that the target equals the trajectory over the current workfile sample.

## Examples

```
m1.control myvar targetvar trajvar
```

will put into MYVAR the values that lead the solution of the model for TARGETVAR to match TRAJVAR for the workfile sample.

## Cross-references

See [“Solve Control for Target” on page 800](#) of the *User’s Guide*. See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a general discussion of models.

**copy****Command**

Copy an object. Duplicate objects within and across workfiles and databases.

### Syntax

Command:      **copy(options) src\_object\_name dest\_object\_name**

To make copies of the object within the active workfile, list the name of the original object, followed by the name for the copy. To make copies of objects within a database, you can precede the name of the object with the database name and a double colon “::”.

You may also copy objects between a workfile and a database, or between two databases. Simply prefix the object name with the database name and double colon “::”.

You can copy several objects with one command by using the wild card characters “?” (to match any single character) and “\*” (to match zero or more characters).

### Options

When copying a group object from a workfile to a database:

**g = arg**

Group copy from workfile to database: “s” (copy group definition and series as separate objects), “t” (copy group definition and series as one object), “d” (copy series only as separate objects), “l” (copy group definition only).

When copying a group object from a database to a workfile:

**g = arg**

Group copy from database to workfile: “b” (copy both group definition and series), “d” (copy only the series), “l” (copy only the group definition).

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *low* to *high* frequency:

**c = arg**

Low to high conversion methods: “r” (constant match average), “d” (constant match sum), “q” (quadratic match average), “t” (quadratic match sum), “i” (linear match last), “c” (cubic match last).

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *high* to *low* frequency:

<code>c = arg</code>	<p><i>High to low conversion methods removing NAs:</i> “a” (average of the nonmissing observations), “s” (sum of the nonmissing observations), “f” (first nonmissing observation), “l” (last nonmissing observation), “x” (maximum nonmissing observation), “m” (minimum nonmissing observation).</p> <p><i>High to low conversion methods propagating NAs:</i> “an” or “na” (average, propagating missings), “sn” or “ns” (sum, propagating missings), “fn” or “nf” (first, propagating missings), “ln” or “nl” (last, propagating missings), “xn” or “nx” (maximum, propagating missings), “mn” or “nm” (minimum, propagating missings).</p>
----------------------	--

Note that if no conversion method is specified, the series specific default conversion method or the global settings will be employed.

## Examples

```
copy good_equation best_equation
```

makes a copy of GOOD\_EQUATION at names it BEST\_EQUATION.

```
copy gdp usdat::gdp
```

copies GDP in the current workfile to the series GDP in the USDAT database. To copy GDP in the default database to the database named MACRO1 with the new name, “GDP\_US”, use the command:

```
copy ::gdp macrol::gdp_us
```

The command:

```
copy ::gdp ::gdp2
```

makes a backup copy of GDP in the default database with a different name “GDP2”.

```
copy gd* findat::
```

makes a duplicate of all objects in the current workfile with name starting with “GD” to the database named FINDAT.

## Cross-references

See “[Copying Objects](#)” on page 262 of the *User’s Guide* for a discussion of copying and moving objects.

See also [fetch \(p. 279\)](#), [setconvert \(p. 424\)](#), and [store \(p. 465\)](#).

<b>cor</b>	<a href="#">Command</a>    <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Sym View</a>
------------	--

Correlation matrix.

### Syntax

Command:      `cor(options) ser1 ser2 ser3`

Group View:    `group_name.cor(options)`

Matrix View:   `matrix_name.cor(options)`

In command form, EViews creates an untitled group from the listed series, then displays the correlation matrix view for that group. When used as a matrix view, `cor` displays the correlation matrix computed from the columns of the matrix.

### Options

i                Compute correlations using pairwise samples (default is to use the common sample).

p                Print the correlation matrix.

### Examples

```
cor height weight age
```

displays a 3 by 3 correlation matrix for the three series HEIGHT, WEIGHT, and AGE.

```
group mygroup height weight age  
mygroup.cor
```

displays the equivalent view using the group MYGROUP.

### Cross-references

See also [cov \(p. 250\)](#), [@cor \(p. 584\)](#), and [@cov \(p. 584\)](#).

<b>correl</b>	<a href="#">Equation View</a>   <a href="#">Group View</a>   <a href="#">Series View</a>   <a href="#">Var View</a>
---------------	---

Display autocorrelation and partial correlations.

Displays the autocorrelation and partial correlation functions of the specified series, together with the  $Q$ -statistics and  $p$ -values associated with each lag.

When used with equation objects, `correl` displays the correlogram of the residuals of the equation.

## Syntax

Object View:      `object_name.correl(n, options)`

You must specify the largest lag *n* to use when computing the autocorrelations.

## Options

<code>p</code>	Print the correlograms.
----------------	-------------------------

### *Var View Options:*

<code>graph</code>	Display correlograms (graphs).
<code>byser</code>	Display autocorrelations in tabular form, by series.
<code>bylag</code>	Display autocorrelations in tabular form, by lag.

## Examples

```
m1.correl(24)
```

Displays the correlograms of the M1 series for up to 24 lags.

## Cross-references

See “Autocorrelations (AC)” on page 314 and “Partial Autocorrelations (PAC)” on page 315 of the *User’s Guide* for a discussion of autocorrelation and partial correlation functions, respectively.

See also [correlsq \(p. 247\)](#).

<b>correlsq</b>	<a href="#">Equation View</a>
-----------------	-------------------------------

Correlogram of squared residuals.

Displays the autocorrelation and partial correlation functions of the squared residuals from an estimated equation, together with the *Q*-statistics and *p*-values associated with each lag.

## Syntax

View:      `equation_name.correl(n, options)`

## Options

<code>n</code>	Specify the number of lags of the correlograms to display.
----------------	--

p Print the correlograms.

## Examples

```
eq1.correl(24)
```

displays the correlograms of the squared residuals of EQ1 up to 24 lags.

## Cross-references

See “[Autocorrelations \(AC\)](#)” on page 314 and “[Partial Autocorrelations \(PAC\)](#)” on page 315 of the *User’s Guide* for a discussion of autocorrelation and partial correlation functions, respectively.

See also [correl](#) (p. 246).

<b>count</b>	<a href="#">Command</a>    <a href="#">Equation Method</a>
--------------	--

Estimates models where the dependent variable is a nonnegative integer count.

## Syntax

Command: `count(options) y x1 x2 x3`

Equation Method: `eq_name.count(options) y x1 x2 x3`

Follow the `count` keyword by the name of the dependent variable and a list of regressors.

## Options

<code>d = arg</code> <code>(default = "p")</code>	Likelihood specification: Poisson likelihood (“p”), normal quasi-likelihood (“n”), exponential likelihood (“e”), negative binomial likelihood or quasi-likelihood (“b”).
<code>v = positive_num</code> <code>(default = 1)</code>	Specify fixed value for QML parameter in normal and negative binomial quasi-likelihoods.
<code>q (default</code>	Use quadratic hill-climbing as the maximization algorithm.
<code>r</code>	Use Newton-Raphson as the maximization algorithm.
<code>b</code>	Use Berndt-Hall-Hausman as the maximization algorithm.
<code>h</code>	Quasi-maximum likelihood (QML) standard errors.
<code>g</code>	GLM standard errors.

---

<i>m = integer</i>	Set maximum number of iterations.
<i>c = scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<i>s</i>	Use the current coefficient values in C as starting values.
<i>s = number</i>	Specify a number between zero and one to determine starting values as a fraction of the EViews default values (out of range values are set to “s = 1”).

## Examples

The command:

```
count (d=n, v=2, g) y c x1 x2
```

estimates a normal QML count model of Y on a constant, X1, and X2, with fixed variance parameter 2, and GLM standard errors.

```
equation eq1.count arrest c job police
eq1.makeresid(g) res_g
```

estimates a Poisson count model of ARREST on a constant, JOB, and POLICE, and stores the generalized residuals in the series RES\_G.

```
equation eq1.count (d=p) y c x1
eq1.fit yhat
```

estimates a Poisson count model of Y on a constant and X1, and saves the fitted values (conditional mean) in the series YHAT.

```
equation eq1.count (d=p, h) y c x1
```

estimates the same model with QML standard errors and covariances.

## Cross-references

See “[Count Models](#)” on page 642 of the *User’s Guide* for additional discussion.

**COV**[Command](#) || [Group View](#) | [Matrix View](#) | [Sym View](#)

Compute covariance matrix.

### Syntax

Command:      `cov(options) arg1 [arg2 arg3 ...]`

Group View:    `group_name.cov(options)`

Matrix View:   `matrix_name.cov(options)`

In command form, EViews will create an untitled group from the listed series or groups, then will display the covariance matrix view for that group. When used as a matrix view, `cov` displays the covariance matrix computed from the columns of the matrix.

### Options

i                Compute covariances using pairwise samples (default is to use the common sample).

p                Print the correlation matrix.

### Examples

```
group grp1 height weight age  
grp1.cov
```

displays a  $3 \times 3$  covariance matrix for the three series in GRP1.

### Cross-references

See also [cor \(p. 246\)](#), [@cor \(p. 584\)](#), and [@cov \(p. 584\)](#).

**create**[Command](#)

Create workfile.

No longer supported. This command has been replaced by [wfcreate \(p. 503\)](#) and [pagecreate \(p. 369\)](#).

---

<b>cross</b>	<a href="#">Command</a>    <a href="#">Group View</a>
--------------	---

Displays cross correlations (correlograms) for a pair of series.

## Syntax

Command:      `cross(n,options) arg1 [arg2 arg3 ...]`

Group View:    `group_name.cross(n,options)`

You must specify the number of lags *n* to use in computing the cross correlations as the first option. In command form, EViews will create an untitled group from the specified series and groups, and will display the cross correlation view for the group. When used as a group view, cross correlations will be computed for the first two series in the group.

## Options

The following options may be specified inside the parentheses after the number of lags:

<code>p</code>	Print the cross correlogram.
----------------	------------------------------

## Examples

```
cross(36) log(m1) dlog(cpi)
```

displays the cross correlogram between the log of M1 and the first difference of the log of CPI, using up to 36 leads and lags.

```
equation eq1.arch sp500 c
eq1.makeresid(s) res_std
cross(24) res_std^2 res_std
```

The first line estimates a GARCH(1,1) model and the second line retrieves the standardized residuals. The third line plots the cross correlogram squared standardized residual and the standardized residual, up to 24 leads and lags . This correlogram provides a rough check of asymmetry in the ARCH effect.

## Cross-references

See “[Cross Correlations and Correlograms](#)” on page 375 of the *User’s Guide* for discussion.

<b>data</b>	<a href="#">Command</a>
-------------	-------------------------

Enter data from keyboard.

Opens an unnamed group window to edit one or more series.

### Syntax

Command:      **data** *arg1 [arg2 arg3 ...]*

Follow the **data** keyword by a list of series and group names. You can list existing names or new names. Unrecognized names will cause new series to be added to the workfile. These series will be initialized with the value “NA”.

### Examples

```
data group1 newx newy
```

opens a group window containing the series in group GROUP1, and the series NEWX and NEWY.

### Cross-references

See “[Entering Data](#)” on page 98 of the *User’s Guide* for a discussion of the process of entering data from the keyboard.

<b>datelabel</b>	<a href="#">Graph Proc</a>
------------------	----------------------------

Control labeling of the bottom date/time axis in time plots.

**dates** sets options that are specific to appearance of time/date labeling. Many of the options that also affect the appearance of the date axis are set by the [axis \(p. 213\)](#) command with the “bottom” option. These options include tick control, label and font options, and grid lines.

### Syntax

Graph Proc:      **graph\_name.datelabel** *option\_list*

## Options

format( <i>dateformat</i> [, <i>delimiter</i> ])	Specify date format: <i>dateformat</i> = “auto” or <i>dateformat</i> = “string”, where the string argument is one of the supported formats: “yy”, “yyyy:mm”, “yyyy:q”, “yy:q”, “mm:dd:yyyy”, “mm:dd:yy”. Each letter represents a single display digit representing year (“y”), quarter (“q”), month (“m”), and day (“d”). The optional delimiter argument allows you to control the date separator.
interval( <i>step_size</i> [, <i>steps</i> ] [, <i>align_date</i> ])	<p>where <i>step_size</i> takes one of the following values: “auto” (<i>steps</i> and <i>align_date</i> are ignored), “ends” (only label endpoints; <i>steps</i> and <i>align_date</i> are ignored), “all” (label every point; the <i>steps</i> and <i>align_date</i> options are ignored), “obs” (<i>steps</i> are one observation), “year” (<i>steps</i> are one year), “m” (<i>steps</i> are one month), “q” (<i>steps</i> are one quarter).</p> <p><i>steps</i> is a number (<i>default</i> = 1) indicating the number of steps between labels.</p> <p><i>align_date</i> is a date specified to receive a label.</p> <p>Note, the <i>align_date</i> should be in the units of the data being graphed, but may lie outside the current sample or workfile range.</p>
span / -span	[Allow/Do not allow] date labels to span an interval. Consider the case of a yearly label with monthly ticks. If <i>span</i> is on, the label is centered on the 12 monthly ticks. If the <i>span</i> option is off, year labels are put on the first quarter or month of the year.

## Examples

```
graph1.datelabel format(yyyy:mm)
```

will display dates using four-digit years followed by the default delimiter “:” and a two-digit month (e.g. – “1974:04”).

```
graph1.datelabel format(yy:mm, q)
```

will display a two-digit year followed by a “q” separator and then a two-digit month (e.g. – “74q04”)

```
graph1.datelabel interval(y, 2, 1951)
```

specifies labels every two years on odd numbered years.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graph options.

See also [axis \(p. 213\)](#), [scale \(p. 410\)](#), [options \(p. 358\)](#), and [setelem \(p. 426\)](#).

<b>dates</b>	<a href="#">Graph Proc</a>
--------------	----------------------------

See the replacement command [datelabel \(p. 252\)](#).

<b>db</b>	<a href="#">Command</a>
-----------	-------------------------

Open or create a database.

If the specified database does not exist, a new (empty) database will be created and opened. The opened database will become the default database.

### Syntax

Command:      **db**(*options*) [*path\*]*db\_name* [as *shorthand\_name*]

Follow the **db** command by the name of the database to be opened or to be created (if it does not already exist). You may include a path name to work with a database not in the default path.

You may use the “as” keyword to provide an optional *shorthand\_name* or short text label which may be used to refer to the database in commands and programs. If you leave this field blank, a default *shorthand\_name* will be assigned automatically. See [“Database Short-hands” on page 257](#) of the *User’s Guide* for additional discussion.

### Options

See [dbopen \(p. 257\)](#) for a list of available options for working with foreign format databases.

### Examples

```
db findat
```

opens the database FINDAT in the default path and makes it the default database from which to store and fetch objects. If the database FINDAT does not already exist, an empty database named FINDAT will be created and opened.

## Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases.

See also [dbcreate \(p. 255\)](#) and [dbopen \(p. 257\)](#).

<b>dbcopy</b>	<a href="#">Command</a>
---------------	-------------------------

Make a copy of an existing database.

## Syntax

Command:      **dbcopy** [path\]source\_name [path\]copy\_name

Follow the **dbcopy** command by the name of the existing database and a name for the copy. You should include a path name to copy from or to a database that is not in the default directory. All files associated with the database will be copied.

## Examples

```
dbcopy usdat c:\backup\usdat
```

makes a copy of all files associated with the database USDAT in the default path and stores it in the C:\BACKUP directory under the name “USDAT”.

## Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases.

See also [dbrename \(p. 259\)](#) and [dbdelete \(p. 256\)](#).

<b>dbcreate</b>	<a href="#">Command</a>
-----------------	-------------------------

Create a new database.

## Syntax

Command:      **dbcreate** [path\]db\_name {as shorthand\_name}

Follow the **dbcreate** keyword by a name for the new database. You may include a path name to create a database not in the default directory. The new database will become the default database.

You may use the “as” keyword to provide an optional *shorthand\_name* or a short text label which may be used to refer to the open database in commands and programs. If you leave this field blank, a default *shorthand\_name* will be assigned automatically. See “[Database Shorthands](#)” on page 257 of the *User’s Guide* for additional discussion.

## Examples

```
dbcreate macrodat
```

creates a new database named MACRODAT in the default path, and makes it the default database from which to store and fetch objects. This command will issue an error message if a database named MACRODAT already exists. To open an existing database, use [dbopen \(p. 257\)](#) or [db \(p. 254\)](#).

## Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases.

See also [db \(p. 254\)](#) and [dbopen \(p. 257\)](#).

<b>dbdelete</b>	<a href="#">Command</a>
-----------------	-------------------------

Delete an existing database (all files associated with the specified database).

## Syntax

Command:      **dbdelete** *[path\]db\_name*

Follow the dbdelete keyword by the name of the database to be deleted. You may include a path name to delete a database not in the default path.

## Examples

```
dbdelete c:\temp\testdat
```

Deletes all files associated with the TESTDAT database in the specified directory.

## Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases.

See also [dbcopy \(p. 255\)](#) and [dbdelete \(p. 256\)](#).

<b>dbopen</b>	<a href="#">Command</a>
---------------	-------------------------

Open an existing database.

### Syntax

Command:      **dbopen**(*options*) [*path*\]*db\_name* [as *shorthand\_name*]

Follow the **dbopen** keyword with the name of a database. You should include a path name to open a database not in the default path. The opened database will become the default database.

You may use the “as” keyword to provide an optional *shorthand\_name* or a short text label which is used to refer to the open database in commands and programs. If you leave this field blank, a default *shorthand\_name* will be assigned automatically. See “[Database Short-hands](#)” on page 257 of the *User’s Guide* for additional discussion.

By default, EViews will use the extension of the database file to determine type. For example, files with the extension “.EDB” will be opened as an EViews database, while files with the extension “.IN7” will be opened as a GiveWin database. You may use options to specify an explicit type.

### Options

<b>type = arg, t = arg</b>	Specify the database type: AREMOS-TSD (“a”, “aremos”, “tsd”), DRIBase (“b”, “dribase”), EViews (“e”, “evdb”), FAME (“f”, “fame”), GiveWin/PcGive (“g”, “give”), Haver Analytics (“h”, “haver”), Rats Portable/Troll (“l”, “trl”), RATS 4.x (“r”, “rats”), TSP portable (“t”, “tsp”).
----------------------------	--

The following options may be required when connecting to a remote server:

<b>s = server_id,</b>	Server name.
<b>server = server_id</b>	

<b>u = user,</b>	Username.
<b>username = user</b>	

<b>p = pswd,</b>	Password.
<b>password = pswd</b>	

## Examples

```
dbopen c:\data\us1
```

opens a database named US1 in the C:\DATA directory. The command:

```
dbopen us1
```

opens a database in the default path. If the specified database does not exist, EViews will issue an error message. You should use [db \(p. 254\)](#) or [dbccreate \(p. 255\)](#) to create a new database.

## Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases.

See also [db \(p. 254\)](#) and [dbccreate \(p. 255\)](#).

<b>dbpack</b>	<a href="#">Command</a>
---------------	-------------------------

Pack an existing database.

## Syntax

Command:      **dbpack** *[path\]db\_name*

Follow the dbpack keyword by a database name. You may include a path name to pack a database not in the default path.

## Examples

```
dbpack findat
```

packs the database named FINDAT in the default path.

## Cross-references

See [“Packing the Database” on page 280](#) of the *User’s Guide* for additional discussion.

See also [dbrebuild \(p. 258\)](#) and [dbrepair \(p. 260\)](#)

<b>dbrebuild</b>	<a href="#">Command</a>
------------------	-------------------------

Rebuild an existing database.

Rebuild a seriously damaged database into a new database file.

## Syntax

Command:      **dbrebuild** [path\[]source\_name [path\[]dest\_name

Follow the **dbrebuild** keyword by the name of the database to be rebuilt, and then a new database name.

## Examples

If you issue the command:

```
dbrebuild testdat fixed_testdat
```

EViews will attempt to rebuild the database TESTDAT into the database FIXED\_TESTDAT in the default directory.

## Cross-references

Note that **dbrepair** may be able to repair the existing database if the damage is not particularly serious. You should attempt to repair the database *before* rebuilding. See “[Maintaining the Database](#)” on page 279 of the *User’s Guide* for a discussion.

See also [dbpack](#) (p. 258) and [dbrepair](#) (p. 260).

<b>dbrename</b>	<a href="#">Command</a>
-----------------	-------------------------

Rename an existing database.

**dbrename** renames all files associated with the specified database.

## Syntax

Command:      **dbrename** [path\[]old\_name [path\[]new\_name

Follow the **dbrename** keyword with the current name of an existing database and the new name for the database.

## Examples

```
dbrename testdat mydat
```

Renames all files associated with the TESTDAT database in the specified directory to MYDAT in the default directory.

## Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases.

See also [db](#) (p. 254) and [dbccreate](#) (p. 255). See also [dbcopy](#) (p. 255) and [dbdelete](#) (p. 256).

<b>dbrepair</b>	<a href="#">Command</a>
-----------------	-------------------------

Repair an existing database.

This command is used to repair a damaged database.

### Syntax

Command:      **dbrepair** [*path\]db\_name*

Follow the **dbrepair** keyword by the name of the database to repair. You should include a path name to repair a database not in the default path.

### Examples

```
dbrepair testdat
```

EViews will attempt to repair the database TESTDAT in the default directory.

### Cross-references

If the database is so damaged that it cannot be repaired, you may have to rebuild it into a new database using the [dbrebuild](#) (p. 258) command. See “[Maintaining the Database](#)” on page 279 of the *User’s Guide* for a discussion of EViews database maintenance.

See also [dbpack](#) (p. 258) and [dbrebuild](#) (p. 258).

<b>decomp</b>	<a href="#">Var View</a>
---------------	--------------------------

Variance decomposition in VARs.

### Syntax

Var View:      var\_name.decomp(*n, options*) series\_list [@ @ ordering]

List the series names in the VAR whose variance decomposition you would like to compute. You may optionally specify the ordering for the factorization after two “@”-signs.

You must specify the number of periods *n* over which to compute the variance decompositions.

## Options

<code>g (default)</code>	Display combined graphs, with the decompositions for each variable shown in a graph.
<code>m</code>	Display multiple graphs, with each response-shock pair shown in a separate graph.
<code>t</code>	Show numerical results in table.
<code>imp = arg</code> ( <i>default</i> = “chol”)	Type of factorization for the decomposition: “chol” (Cholesky with d.f. correction), “mlechol” (Cholesky without d.f. correction), “struct” (structural).  The structural factorization is based on the estimated structural VAR. To use this option, you must first estimate the structural decomposition; see <a href="#">svar (p. 469)</a> .  The option “ <code>imp = mlechol</code> ” is provided for backward compatibility with EViews 3.x and earlier.
<code>se = mc</code>	Monte Carlo standard errors. You must specify the number of replications with the “ <code>rep = </code> ” option.  Currently available only when you have specified the Cholesky factorization (using the “ <code>imp = chol</code> ” option).
<code>rep = integer</code>	Number of Monte Carlo replications to be used in computing the standard errors. Must be used with the “ <code>se = mc</code> ” option.
<code>matbys = name</code>	Save responses by shocks (impulses) in named matrix. The first column is the response of the first variable to the first shock, the second column is the response of the second variable to the first shock, and so on.
<code>matbyr = name</code>	Save responses by response series in named matrix. The first column is the response of the first variable to the first shock, the second column is the response of the first variable to the second shock, and so on.
<code>p</code>	Print results.

If you use the “`matbys =` ” or “`matbyr =` ” options to store the results in a matrix, two matrices will be returned. The matrix with the specified name contains the variance decompositions, while the matrix with “`_FSE`” appended to the name contains the forecast standard errors for each response variable. If you have requested Monte Carlo standard errors, there will be a third matrix with “`_SE`” appended to the name which contains the variance decomposition standard errors.

## Examples

```
var var1.ls 1 4 m1 gdp cpi  
var1.decomp(10,t) gdp
```

The first line declares and estimates a VAR with three variables and lags from 1 to 4. The second line tabulates the variance decompositions of GDP up to 10 periods using the ordering as specified in VAR1.

```
var1.decomp(10,t) gdp @ @ cpi gdp m1
```

performs the same variance decomposition as above using a different ordering.

## Cross-references

See “[Variance Decomposition](#)” on page 715 of the *User’s Guide* for additional details.

See also [impulse \(p. 311\)](#).

<b>define</b>	<a href="#">Pool Proc</a>
---------------	---------------------------

Define cross section members (identifiers) in a pool.

## Syntax

Pool View:      `pool_name.define id1 [id2 id3 ...]`

List the cross section identifiers after the `define` keyword.

## Examples

```
pool spot uk jpn ger can  
spot.def uk ger ita fra
```

The first line declares a pool object named SPOT with cross section identifiers UK, JPN, GER, and CAN. The second line redefines the identifiers to be UK, GER, ITA, and FRA.

## Cross-references

See [Chapter 27, “Pooled Time Series, Cross-Section Data”, on page 809](#) of the *User’s Guide* for a discussion of cross-section identifiers.

See also [add \(p. 196\)](#), [drop \(p. 270\)](#) and [pool \(p. 386\)](#).

---

<b>delete</b>	<a href="#">Command</a>    <a href="#">Pool Proc</a>
---------------	--

Deletes objects from a workfile or a database, or removes identifiers from a pool.

### Syntax

Command:      `delete arg1 [arg2 arg3 ...]`

Pool Proc:      `pool_name.delete pool_ser1 [pool_ser2 pool_ser3 ...]`

Follow the keyword by a list of the names of any objects you wish to remove from the current workfile. Deleting does *not* remove objects that have been stored on disk in EViews database files.

You can delete an object from a database by prefixing the name with the database name and a double colon. You can use a pattern to delete all objects from a workfile or database with names that match the pattern. Use the “?” to match any one character and the “\*” to match zero or more characters.

*When used as a pool procedure*, `delete` allows you to delete series from the workfile using pool series names. You may use the cross-section identifier placeholder “?” in the series names.

If you use `delete` in a program file, EViews will delete the listed objects without prompting you to confirm each deletion.

### Examples

To delete all objects in the workfile with names beginning with “TEMP”, you may use the command:

```
delete temp*
```

To delete the objects CONS and INVEST from the database MACRO1, use:

```
delete macro1::cons macro1::invest
```

To delete all series in the workfile with names beginning with “CPI” that are followed by identifiers in the pool object MYPOOL.

```
mypool.delete cpi?
```

### Cross-references

See [Chapter 4, “Object Basics”, on page 65](#) of the *User’s Guide* for a discussion of working with objects, and [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases.

**derivs**[Equation View](#) | [System View](#)

Examine derivatives of the equation specification.

Display information about the derivatives of the equation specification in tabular, graphical, or summary form.

The (default) summary form shows information about how the derivative of the equation specification was computed, and will display the analytic expression for the derivative, or a note indicating that the derivative was computed numerically. The tabular form shows a spreadsheet view of the derivatives of the regression specification with respect to each coefficient (for each observation). The graphical form of the view shows this information in a multiple line graph.

### Syntax

Equation View:    `equation_name.derivs(options)`

### Options

`g`      Display multiple graph showing the derivatives of the equation specification with respect to the coefficients, evaluated at each observation.

`t`      Display spreadsheet view of the values of the derivatives with respect to the coefficients evaluated at each observation.

`p`      Print results.

Note that the “`g`” and “`t`” options may not be used at the same time.

### Examples

To show a table view of the derivatives:

```
eq1.derivs(t)
```

To display and print the summary view:

```
eq1.derivs(p)
```

### Cross-references

See “[Derivative Computation Options](#)” on page 930 of the *User’s Guide* for details on the computation of derivatives.

See also [makederivs](#) (p. 335) for additional routines for examining derivatives, and [grads](#) (p. 302), and [makegrads](#) (p. 337) for corresponding routines for gradients.

<b>describe</b>	<a href="#">Pool View</a>
-----------------	---------------------------

Computes and displays descriptive statistics for the pooled data.

### Syntax

Pool View: `pool_name.describe(options) pool_ser1 [pool_ser2 pool_ser3 ...]`

List the name of pool series for which you wish to compute descriptive statistics. You may use the cross-section identifier “?” in the series names.

By default, statistics are computed for each stacked pool series, using only common observations where *all* of the cross-sections for a given series have nonmissing data. A missing observation for a series in any one cross-section causes that observation to be dropped for all cross-sections for the corresponding series. You may change the default treatment of NAs using the “i” and “b” options.

EViews also allows you to compute statistics with the cross-section means removed, statistics for each cross-sectional series in a pool series, and statistics for each period, taken across all cross-section units.

### Options

- |   |  |
|---|--|
| m | Stack data and subtract cross-section specific means from each variable—this option provides the within estimators.  |
| c | Do not stack data—compute statistics individually for each cross-sectional unit.   |
| t | Time period specific—compute statistics for each period, taken over all cross-section identifiers.   |
| i | Individual sample—includes every valid observation for the series even if data are missing from other series in the list.  |
| b | Balanced sample—constrains each cross-section to have the <i>same observations</i> . If an observation is missing for any series, in any cross-section, it will be dropped for all cross-sections. |
| p | Print the descriptive statistics.  |

## Examples

```
pool1.describe(m) gdp? inv? cpi?
```

displays the “within” descriptive statistics of the three series GDP, INV, CPI for the POOL1 cross-section members.

```
pool1.describe(t) gdp?
```

computes the statistics for GDP for each period, taken across each of the cross-section identifiers.

## Cross-references

See [Chapter 27, “Pooled Time Series, Cross-Section Data”, on page 809](#) of the *User’s Guide* for a discussion of the computation of these statistics, and a description of individual and balanced samples.

displayname	Object Proc
-------------	-------------

Display names for objects.

Attaches a display name to an object which may be used in tables and graphs in place of the standard object name.

## Syntax

```
Object Proc:      object_name.displayname display_name
```

Display names are case-sensitive, and may contain a variety of characters, such as spaces, that are not allowed in object names.

## Examples

```
hrs.displayname Hours Worked
```

```
hrs.label
```

The first line attaches a display name “Hours Worked” to the object HRS, and the second line displays the label view of HRS, including its display name.

```
gdp.displayname US Gross Domestic Product
```

```
plot gdp
```

The first line attaches a display name “US Gross Domestic Product” to the series GDP. The line graph view of GDP from the second line will use the display name as the legend.

## Cross-references

See “[Labeling Objects](#)” on page 74 of the *User’s Guide* for a discussion of labels and display names.

See also [label](#) (p. 317) and [legend](#) (p. 319)

<b>do</b>	<a href="#">Command</a>
-----------	-------------------------

Execute without opening window.

### Syntax

Command:      **do** *procedure*

**do** is most useful in EViews programs where you wish to run a series of commands without opening windows in the workfile area.

### Examples

```
output(t) c:\result\junk1
do gdp.adf(c, 4, p)
```

The first line redirects table output to a file on disk. The second line carries out a unit root test of GDP without opening a window, and prints the results to the disk file.

## Cross-references

See also [show](#) (p. 445).

<b>draw</b>	<a href="#">Graph Proc</a>
-------------	----------------------------

Place horizontal or vertical lines and shaded areas on the graph.

### Syntax

Graph Proc:      *graph\_name.draw(draw\_type, axis\_id [,options]) position1 [position2]*

where *draw\_type* may be one of the following:

line / l	A solid line
dashline / d	A dashed line
shade	A shaded area

and where *axis\_id* may take the values:

left / l	Draw a horizontal line or shade using the left axis to define the drawing position
right / r	Draw a horizontal line or shade using the right axis to define the drawing position
bottom / b	Draw a vertical line or shade using the bottom axis to define the drawing position

If drawing a line, the drawing position is taken from *position1*. If drawing a shaded area, you must provide a *position1* and *position2* to define the boundaries of the shaded region.

### Line/Shade Options

The following options may be provided to change the characteristics of the specified line or shade:

color( <i>arg</i> ) / color( <i>n1,n2,n3</i> )	Specifies the color of the line or shade. the argument may be made up of <i>n1</i> , <i>n2</i> , and <i>n3</i> , a set of three integers from 0 to 255, representing the RGB values of the line or shade, or it may be one of the predefined color keywords (see <a href="#">setfillcolor (p. 429)</a> ). The default is black for lines and gray for shades. RGB values may be examined by calling up the color palette in the <b>Graph Options</b> dialog.
rgb( <i>n1,n2,n3</i> )	Specify the color of the line or shade using RGB values, where <i>n1</i> , <i>n2</i> , and <i>n3</i> , are integers from 0 to 255 representing the RGB values of the line or shade. Provided for backward compatibility; same as using the “color” option with three RGB values.
width( <i>n1</i> )	Specify the width, where <i>n1</i> is the line width in points (used only if object_type is “line” or “dashline”). The default is 0.5 points.

### Other Options

p	Print the graph object
---	------------------------

### Examples

```
graph1.draw(line, left, rgb(0,0,127)) 5.25
```

draws a horizontal blue line at the value “5.25” as measured on the left axis while:

---

```
graph1.draw(shade, right) 7.1 9.7
```

draws a shaded horizontal region bounded by the right axis values “7.1” and “9.7”. You may also draw vertical regions by using the “bottom” *axis\_id*:

```
graph1.draw(shade, bottom) 1980:1 1990:2
```

draws a shaded vertical region bounded by the dates “1980:1” and “1990:2”.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graph options.

See also [“Graph” \(p. 159\)](#) for a summary of the graph object command language.

driconvert	Command
------------	---------

Convert the entire DRI Basic Economics database into an EViews database.

You must create an EViews database to store the converted DRI data *before* you use this command. This command may be very time-consuming.

## Syntax

Command:      **driconvert** *db\_name*

Follow the command by listing the name of an existing EViews database into which you would like to copy the DRI data. You may include a path name to specify a database not in the default path.

## Examples

```
dbccreate dribleasic
driconvert dribleasic
driconvert c:\mydata\dribase
```

The first line creates a new (empty) database named DRIBASIC in the default directory. The second line copies all the data in the DRI Basic Economics database into in the DRIBASIC database. The last example copies the DRI data into the database DRIDBASE that is located in the C:\MYDATA directory.

## Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases.

See also [dbcreate \(p. 255\)](#) and [db \(p. 254\)](#).

<b>drop</b>	<a href="#">Group Proc</a>   <a href="#">Pool Proc</a>
-------------	--

Drops series from a group or drop cross-section members from a pool.

### Syntax

Group Proc:      `group_name.drop ser1 [ser2 ser3 ...]`

Pool Proc:      `pool_name.drop id1 [id2 id3 ...]`

List the series or cross-section members to be dropped from the group or pool.

### Examples

```
group gdplags gdp(-1 to -4)
```

```
gdplags.drop gdp(-4) gdp(-3)
```

drops the two series GDP(-4) and GDP(-3) from the group GDPLAGS.

```
crosssc.drop jpn kor hk
```

drops the cross-section members JPN, KOR, and HK from the pool CROSSSC.

### Cross-references

See “[Groups](#)” on page 85 of the *User’s Guide* for additional discussion of groups. “[Cross-section Identifiers](#)” on page 811 of the *User’s Guide* discusses pool identifiers.

See also [add \(p. 196\)](#).

<b>dtable</b>	<a href="#">Group View</a>
---------------	----------------------------

Dated data report table.

This group view is designed to make tables for reporting and presenting data, forecasts, and simulation results. You can display various transformations and various frequencies of the data in the same table.

The `dtable` view is currently available only for annual, semi-annual, quarterly, or monthly workfiles.

### Syntax

Group View:      `group_name.dtable(options)`

## Options

p	Print the report table.
---	-------------------------

## Examples

```
freeze(report) group1.dtable
```

freezes the dated table view of GROUP1 and saves it as a table object named REPORT.

## Cross-references

See “[Creating and Specifying a Dated Data Table](#)” on page 354 of the *User’s Guide* for a description of dated data tables and formatting options. Note that most of the options for formatting the table are only available interactively from the window.

ec	<a href="#">Var Method</a>
----	----------------------------

Estimate a vector error correction model (VEC).

## Syntax

Var Method: `var_name.ec(trend, n) lag_pairs endog_list {@ exog_list}`

Specify the order of the VEC by entering one or more pairs of lag intervals, then list the series or groups to be used as endogenous variables. *Note that the lag orders are those of the first differences, not the levels.* If you are comparing results to another software program, you should be certain that the specifications for the lag orders are comparable.

You may include exogenous variables, such as seasonal dummies, in the VEC by including an “@”-sign followed by the list of series or groups. *Do not include an intercept or trend* in the VEC specification, these terms should be specified using options, as described below.

You should specify the trend option and the number of cointegrating equations *n* to use in parentheses, separated by a comma (the default is *n* = 1). You must choose the trend from the following five alternatives:

- |             |  |
|-------------|--|
| a           | No deterministic trend in the data, and no intercept or trend in the cointegrating equation.     |
| b           | No deterministic trend in the data, and an intercept but no trend in the cointegrating equation. |
| c (default) | Linear trend in the data, and an intercept but no trend in the cointegrating equation.           |

d	Linear trend in the data, and both an intercept and a trend in the cointegrating equation.
e	Quadratic trend in the data, and both an intercept and a trend in the cointegrating equation.
restrict	Impose restrictions. See <a href="#">append (p. 202)</a> and <a href="#">coint (p. 240)</a> .
m = <i>integer</i>	Maximum number of iterations for restricted estimation (only valid if you choose the restrict option).
c = <i>scalar</i>	Convergence criterion for restricted estimation. (only valid if you choose the restrict option).

## Options

p	Print the results view.
---	-------------------------

## Examples

```
var macrol.ec 1 4 m1 gdp tb3
```

declares a var object MACRO1 and estimates a VEC with four lagged first differences, three endogenous variables and one cointegrating equation using the default trend option “c”.

```
var term.ec(b,2) 1 2 4 4 tb1 tb3 tb6 @ d2 d3 d4
```

declares a var object TERM and estimates a VEC with lagged first differences of order 1, 2, 4, three endogenous variables, three exogenous variables, and two cointegrating equations using trend option “b”.

## Cross-references

See “[Vector Error Correction \(VEC\) Models](#)” on page 733 of the *User’s Guide* for a discussion of VECs.

See also [var \(p. 500\)](#), [coint \(p. 240\)](#) and [append \(p. 202\)](#).

<b>edftest</b>	<a href="#">Series View</a>
----------------	-----------------------------

Computes goodness-of-fit tests based on the empirical distribution function.

## Syntax

Series View:      series\_name.edftest(*options*)

## Options

### General Options

<code>type = arg</code> <i>(default = "nomal")</i>	Distribution to test: “normal” (Normal distribution), “chisq” (Chi-square distribution), “exp” (Exponential distribution), “xmax” (Extreme Value - Type I maximum), “xmin” (Extreme Value Type I minimum), “gamma” (Gamma), “logit” (Logistic), “pareto” (Pareto), “uniform” (Uniform).
<code>p1 = number</code>	Specify the value of the first parameter of the distribution (as it appears in the dialog). If this option is not specified, the first parameter will be estimated.
<code>p2 = number</code>	Specify the value of the second parameter of the distribution (as it appears in the dialog). If this option is not specified, the second parameter will be estimated.
<code>p3 = number</code>	Specify the value of the third parameter of the distribution (as it appears in the dialog). If this option is not specified, the third parameter will be estimated.
<code>p</code>	Print test results.

### Estimation Options

The following options apply if iterative estimation of parameters is required:

<code>b</code>	Use Berndt-Hall-Hausman (BHHH) algorithm. The default is Marquardt.
<code>m = integer</code>	Maximum number of iterations.
<code>c = number</code>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<code>showopts / -showopts</code>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<code>s</code>	Take starting values from the C coefficient vector. By default, EViews uses distribution specific starting values that typically are based on the method of the moments.

## Examples

```
x.edftest
```

uses the default settings to test whether the series X comes from a normal distribution. Both the location and scale parameters are estimated from the data in X.

```
freeze(tab1) x.edftest(type=chisq, p1=5)
```

tests whether the series x comes from a  $\chi^2$  distribution with 5 degrees of freedom. The output is stored as a table object TAB1.

## Cross-references

See “[Empirical Distribution Tests](#)” on page 311 of the *User’s Guide* for a description of the goodness-of-fit tests.

See also [qqplot \(p. 389\)](#).

endog	<a href="#">Sspace View</a>   <a href="#">System View</a>   <a href="#">Var View</a>
-------	--

Displays a spreadsheet or graph view of the endogenous variables.

## Syntax

Object View:      object\_name.endog(*options*)

Note that `endog` and `makeendog` are no longer supported for model objects. See instead, [makegroup \(p. 339\)](#).

## Options

g	Multiple line graphs of the solved endogenous series.
---	---

p	Print the table of solved endogenous series.
---	--

## Examples

```
sys1.endog(g,p)
```

prints the graphs of the solved endogenous series.

## Cross-references

See also [makeendog \(p. 335\)](#), [system \(p. 472\)](#), [sspace \(p. 457\)](#) and [var \(p. 500\)](#).

**equation****Object Declaration**

Declare an equation object.

**Syntax**

Command: **equation eq\_name**

Command: **equation eq\_name.method(options) specification**

Follow the **equation** keyword with a name and an optional specification. If you wish to enter the specification, you should follow the new equation name with a period, an estimation method, and the equation specification. Valid estimation methods are [arch](#) (p. 203), [binary](#) (p. 217), [censored](#) (p. 233), [count](#) (p. 248), [gmm](#) (p. 297), [ls](#) (p. 329), [ordered](#) (p. 360), and [tsls](#) (p. 487).

**Examples**

```
equation cobdoug.ls log(y) c log(k) log(l)
```

declares and estimates an equation object named COBDOUG.

```
equation ces.ls log(y)=c(1)*log(k^c(2)+l^c(3))
```

declares an equation object named CES containing a nonlinear least squares specification.

```
equation demand.tsls q c p x @ x p(-1) gov
```

creates an equation object named DEMAND and estimates DEMAND using two-stage least squares with instruments X, lagged P, and GOV.

**Cross-references**

See the object reference entry for “[Equation](#)” (p. 155) for a summary of the equation object. [Chapter 15, “Basic Regression”, on page 427](#) of the *User’s Guide* provides basic information on estimation and equation objects.

**eqs****Model View**

View of model organized by equation.

Lists the equations in the model. This view also allows you to identify which equations are entered by text, or by link, and to access and modify the equation specifications.

**Syntax**

Model View: **model\_name.eq**s

## Cross-references

See “[Equation View](#)” on page 780 of the *User’s Guide* for details. See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a general discussion of models.

See also [block \(p. 219\)](#), [text \(p. 482\)](#), and [vars \(p. 501\)](#) for alternative representations of the model.

<b>errbar</b>	<a href="#">Command</a>    <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a>   <a href="#">Sym View</a>
---------------	---

Display error bar graph view of object, or change existing graph object type to error bar (if possible).

Sets the graph type to error bar or displays an error bar view of the group. If there are two series in the graph or group, the error bar will show the high and low values in the bar. The optional third series will be plotted as a symbol. When used as a matrix view, the columns of the matrix are used in place of series.

## Syntax

- |              |   |
|--------------|---|
| Command:     | <code>errbar(options) arg1 [arg2 arg3 ...]</code> |
| Graph Proc:  | <code>graph_name.errbar(options)</code>           |
| Object View: | <code>object_name.errbar(options)</code>          |

## Options

### *Template and printing options*

- |                             |  |
|-----------------------------|--|
| <code>o = graph_name</code> | Use appearance options from the specified graph.                           |
| <code>t = graph_name</code> | Use appearance options and copy text and shading from the specified graph. |
| <code>p</code>              | Print the error bar graph.   |

### *Panel options*

The following options apply when graphing panel structured data:

<b>panel = arg</b> (default taken from global set- tings)	Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “medmxmn” (plot median, max and min).
--	--

## Examples

The following commands:

```
group g1 x y
g1.errbar
```

display the error bar view of G1 using the X series as the high value of the bar and the Y series as the low value.

```
group g2 plus2se minus2se estimate
g2.errbar
```

display the error bar view of G2 with the PLUS2SE series as the high value of the bar, the MINUS2SE series as the low value, and ESTIMATE as a symbol.

```
group g1 x y
freeze(graph1) g1.line
graph1.errbar
```

first creates a graph object GRAPH1 containing a line graph of the series in G1, then changes the graph type to an error bar.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) for details on graph objects and types.

See also [graph \(p. 303\)](#) for graph declaration and other graph types.

exclude	<a href="#">Model Proc</a>
---------	----------------------------

Specifies (or merges) excluded endogenous variables in the active scenario.

### Syntax

Model Proc:      `model_name.exclude(options) ser1(smpl) ser2(smpl) ...`

Follow the `exclude` keyword with the argument list containing the endogenous variables you wish to exclude from the solution, along with an optional sample for exclusion. If a sample is not provided, the variable will be excluded for the entire solution sample.

### Options

m	Merge into instead of replace the existing exclude list.
---	--

### Examples

```
mod1.exclude fedfunds govexp("1990:01 1995:02")
```

will create an exclude list containing the variables FEDFUNDS and GOVEXP. FEDFUNDS will be excluded for the entire solution sample, while GOVEXP will only be excluded for the specified sample.

If you then issue the command:

```
mod1.exclude govexp
```

EViews will replace the original exclude list with one containing only GOVEXP. To add excludes to an existing list, use the “m” option:

```
mod1.exclude(m) fedfunds
```

The excluded list now contains both GOVEXP and FEDFUNDS.

### Cross-references

See the discussion in “[Specifying Scenarios](#)” on page 784 of the *User’s Guide*.

See also [model](#) (p. 353), [override](#) (p. 363) and [solveopt](#) (p. 452).

<b>exit</b>	<a href="#">Command</a>
-------------	-------------------------

Exit from EViews. Closes the EViews application.

You will be prompted to save objects and workfiles which have changed since the last time they were saved to disk. Be sure to save your workfiles, if desired, since all changes that you do not save to a disk file will be lost.

### Syntax

Command:      **exit**

### Cross-references

See also [close \(p. 238\)](#) and [save \(p. 407\)](#).

<b>expand</b>	<a href="#">Command</a>
---------------	-------------------------

Expand a workfile.

No longer supported. See the replacement command [pagestruct \(p. 378\)](#).

<b>fetch</b>	<a href="#">Command</a>    <a href="#">Pool Proc</a>
--------------	--

Fetch objects from databases or databank files into the workfile.

`fetch` reads one or more objects from EViews databases or databank files into the active workfile. The objects are loaded into the workfile using the object in the database or using the databank file name.

When used as a pool proc, EViews will first expand the list of series using the pool operator, and then perform the fetch.

If you fetch a series into a workfile with a different frequency, EViews will automatically apply the frequency conversion method attached to the series by `setconvert`. If the series does not have an attached conversion method, EViews will use the method set by **Options/Date-Frequency** in the main menu. You can override the conversion method by specifying an explicit conversion method option.

### Syntax

Command:      **fetch(options) object\_list**

Pool Proc:      `pool_name.fetch(options) pool_ser1 [pool_ser2 pool_ser3 ...]`

The `fetch` command keyword is followed by a list of object names separated by spaces. The default behavior is to fetch the objects from the default database (*this is a change from versions of EViews prior to EViews 3.x where the default was to fetch from individual databank files*).

You can precede the object name with a database name and the double colon “`::`” to indicate a specific database source. If you specify the database name as an option in parentheses (see below), all objects without an explicit database prefix will be fetched from the specified database. You may optionally fetch from individual databank files or search among registered databases.

You may use wild card characters, “`?`” (to match a single character) or “`*`” (to match zero or more characters), in the object name list. All objects with names matching the pattern will be fetched.

To fetch from individual databank files that are not in the default path, you should include an explicit path. If you have more than one object with the same file name (for example, an equation and a series named `CONS`), then you should supply the full object file name including identifying extensions.

## Options

`d = db_name` Fetch from specified database.

`d` Fetch all registered databases in registry order.

`i` Fetch from individual databank files.

The following options are available for fetch of group objects:

`g = arg` Group fetch options: “`b`” (fetch both group definition and series), “`d`” (fetch only the series in the group), “`l`” (fetch only the group definition).

The database specified by the double colon “`::`” takes precedence over the database specified by the “`d =`” option.

In addition, there are a number of options for controlling automatic frequency conversion when performing a fetch. The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *low* to *high* frequency:

`c = arg` Low to high conversion methods: “`r`” (constant match average), “`d`” (constant match sum), “`q`” (quadratic match average), “`t`” (quadratic match sum), “`i`” (linear match last), “`c`” (cubic match last).

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *high* to *low* frequency:

`c = arg`

*High to low conversion methods removing NAs:* “a” (average of the nonmissing observations), “s” (sum of the nonmissing observations), “f” (first nonmissing observation), “l” (last nonmissing observation), “x” (maximum nonmissing observation), “m” (minimum nonmissing observation).

*High to low conversion methods propagating NAs:* “an” or “na” (average, propagating missings), “sn” or “ns” (sum, propagating missings), “fn” or “nf” (first, propagating missings), “ln” or “nl” (last, propagating missings), “xn” or “nx” (maximum, propagating missings), “mn” or “nm” (minimum, propagating missings).

If no conversion method is specified, the series-specific or global default conversion method will be employed.

## Examples

To fetch M1, GDP, and UNEMP from the default database, use:

```
fetch m1 gdp unemp
```

To fetch M1 and GDP from the US1 database and UNEMP from the MACRO database, use the command:

```
fetch(d=us1) m1 gdp macro::unemp
```

You can fetch all objects with names starting with “SP” by searching all registered databases in the search order. The “c=f” option uses the first (nonmissing) observation to convert the frequency of any matching series with a higher frequency than the destination workfile frequency:

```
fetch(d,c=f) sp*
```

You can fetch M1 and UNEMP from individual databank files using:

```
fetch(i) m1 c:\data\unemp
```

To fetch all objects with names starting with “CONS” from the two databases USDAT and UKDAT, use the command:

```
fetch usdat::cons* ukdat::cons*
```

The command:

```
fetch a?income
```

will fetch all series beginning with the letter “A”, followed by any single character, and ending with the string “income”.

### Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of databases, databank files, and frequency conversion. [Appendix B, “Wildcards”, on page 921](#) of the *User’s Guide* describes the use of wildcard characters.

See also [setconvert \(p. 424\)](#), [store \(p. 465\)](#), and [copy \(p. 244\)](#).

fill	<a href="#">Coef Proc</a>   <a href="#">Matrix Proc</a>   <a href="#">Rowvector Proc</a>   <a href="#">Series Proc</a>   <a href="#">Sym Proc</a>   <a href="#">Vector Proc</a>
------	--

Fill an object with specified values.

`fill` can be used to set values of series, vectors and matrices in programs.

### Syntax

Object Proc:      `object_name.fill(options) n1[, n2, n3 ...]`

Follow the keyword with a list of values to place in the specified object. *Each value should be separated by a comma*. By default, series `fill` ignores the current sample and fills the series from the beginning of the workfile range. You may provide sample information using options.

Running out of values before the object is completely filled is not an error; the remaining cells or observations will be unaffected, unless the “I” option is specified. If, however, you list more values than the object can hold, EViews will not modify any observations and will return an error message.

### Options

#### *Options for series fill*

`1`                  Loop repeatedly over the list of values as many times as it takes to fill the series.

`o = [date, integer]` Set starting date or observation from which to start filling the series. Default is the beginning of the workfile range.

`s`                  Fill the series only for the current workfile sample. The “s” option overrides the “o” option.

---

`s = sample_name` Fill the series only for the specified subsample. The “`s`” option overrides the “`o`” option.

#### Options for `coef/vector/matrix fill`

<code>l</code>	Loop repeatedly over the list of values as many times as it takes to fill the object.
<code>o = integer</code> <i>(default = 1)</i>	Fill the object from the specified element. Default is the first element.
<code>b = arg</code> <i>(default = “c”)</i>	Matrix fill order: “ <code>c</code> ” (fill the matrix by column), “ <code>r</code> ” (fill the matrix by row).

#### Examples

To generate a series D70 that takes the value 1, 2, and 3 for all observations from 1970:1:

```
series d70=0
d70.fill(o=1970:1,1) 1,2,3
```

Note that the last argument in the `fill` command above is the *letter “l”*. The next three lines generate a dummy series D70S that takes the value one and two for observations from 1970:1 to 1979:4:

```
series d70s=0
smp1 1970:1 1979:4
d70s.fill(s,1) 1,2
smp1 @all
```

Assuming a quarterly workfile, the following generates a dummy variable for observations in either the third and fourth quarter:

```
series d34
d34.fill(1) 0, 0, 1, 1
```

Note that this series could more easily be generated using `@seas` or the special workfile functions (see “[Basic Date Functions](#)” on page 560).

The following example declares a four element coefficient vector MC, initially filled with zeros. The second line fills MC with the specified values and the third line replaces from row 3 to the last row with -1.

```
coef(4) mc
mc.fill 0.1, 0.2, 0.5, 0.5
mc.fill(o=3,1) -1
```

The commands,

```
matrix(2,2) m1
matrix(2,2) m2
m1.fill 1, 0, 1, 2
m2.fill(b=r) 1, 0, 1, 2
```

create the matrices:

$$m1 = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}, \quad m2 = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \quad (B.1)$$

## Cross-references

See [Chapter 3, “Matrix Language”, on page 23](#) for a detailed discussion of vector and matrix manipulation in EViews.

<b>fiml</b>	<a href="#">System Method</a>
-------------	-------------------------------

**Estimation by full information maximum likelihood.**

`fiml` estimates a system of equations by full information maximum likelihood (assuming a multivariate normal distribution).

## Syntax

System Method: `system_name.fiml(options)`

## Options

i	Iterate simultaneously over the covariance matrix and coefficient vector.
s ( <i>default</i> )	Iterate sequentially over the covariance matrix and coefficient vector.
<i>m</i> = <i>integer</i>	Maximum number of iterations.
<i>c</i> = <i>number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
b	Use Berndt-Hall-Hausman (BHHH) algorithm. Default method is Marquardt.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.

---

<code>deriv = keyword</code>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
<code>p</code>	Print estimation results.

## Examples

`sys1.fiml`

estimates SYS1 by FIML using the default settings. The command:

`sys1.fiml(d, s)`

sequentially iterates over the coefficients and the covariance matrix.

## Cross-references

See [Chapter 23, “System Estimation”, on page 679](#) of the *User’s Guide* for a discussion of systems in EViews.

<b>fit</b>	<a href="#">Command</a>    <a href="#">Equation Proc</a>
------------	--

Computes static forecasts or fitted values from an estimated equation.

When the regressor contains lagged dependent values or ARMA terms, `fit` uses the actual values of the dependent variable instead of the lagged fitted values. You may instruct `fit` to compare the forecasted data to actual data, and to compute forecast summary statistics.

Not available for equations estimated using ordered methods; use [makemodel \(p. 341\)](#) to create a model using the ordered equation results (see example below).

## Syntax

Command:      `fit(options) yhat [y_se]`

Equation Proc:    `eq_name.fit(options) yhat [y_se]`

ARCH Proc:      `eq_name.fit(options) yhat [y_se y_var]`

Following the `fit` keyword, you should type a name for the forecast series and, optionally, a name for the series containing the standard errors and, for ARCH specifications, a name for the conditional variance series.

Forecast standard errors are currently not available for binary, censored, and count models.

## Options

d	In models with implicit dependent variables, forecast the entire expression rather than the normalized variable.
u	Substitute expressions for all auto-updating series in the equation.
g	Graph the fitted values together with the $\pm 2$ standard error bands.
e	Produce the forecast evaluation table.
i	Compute the fitted values of the index. Only for binary, censored and count models.
s	Ignore ARMA terms and use only the structural part of the equation to compute the fitted values.
f = arg (default = "actual")	Out-of-fit-sample fill behavior: "actual" (fill observations outside the fit sample with actual values for the fitted variable), "na" (fill observations outside the fit sample with missing values).

## Examples

```
equation eq1.ls cons c cons(-1) inc inc(-1)
eq1.fit c_hat c_se
genr c_up=c_hat+2*c_se
genr c_low=c_hat-2*c_se
line cons c_up c_low
```

The first line estimates a linear regression of CONS on a constant, CONS lagged once, INC, and INC lagged once. The second line stores the static forecasts and their standard errors as C\_HAT and C\_SE. The third and fourth lines compute the  $+/- 2$  standard error bounds. The fifth line plots the actual series together with the error bounds.

```
equation eq2.binary(d=1) y c wage edu
eq2.fit yf
eq2.fit(i) xbeta
genr yhat = 1-@clogit(-xbeta)
```

The first line estimates a logit specification for Y with a conditional mean that depends on a constant, WAGE, and EDU. The second line computes the fitted probabilities, and the

third line computes the fitted values of the index. The fourth line computes the probabilities from the fitted index using the cumulative distribution function of the logistic distribution. Note that YF and YHAT should be identical.

Note that you cannot fit values from an ordered model. You must instead solve the values from a model. The following lines generate fitted probabilities from an ordered model:

```
equation eq3.ordered y c x z
eq3.makemodel(oprob1)
solve oprob1
```

The first line estimates an ordered probit of Y on a constant, X, and Z. The second line makes a model from the estimated equation with a name OPROB1. The third line solves the model and computes the fitted probabilities that each observation falls in each category.

## Cross-references

To perform dynamic forecasting, use [forecast \(p. 287\)](#). See [makemodel \(p. 341\)](#) and [solve \(p. 451\)](#) for forecasting from systems of equations or ordered equations.

See [Chapter 18, “Forecasting from an Equation”, on page 527](#) of the *User’s Guide* for a discussion of forecasting in EViews and [Chapter 21, “Discrete and Limited Dependent Variable Models”, on page 605](#) of the *User’s Guide* for forecasting from binary, censored, truncated, and count models. See [“Forecasting” on page 740](#) of the *User’s Guide* for a discussion of forecasting from sspace models.

<b>forecast</b>	<a href="#">Command</a>    <a href="#">Equation Proc</a>   <a href="#">Sspace Proc</a>
-----------------	--

Computes ( $n$ -period ahead) dynamic forecasts of an estimated equation or forecasts of the signals and states for an estimated state space.

`forecast` computes the forecast for all observations in a specified sample. In some settings, you may instruct `forecast` to compare the forecasted data to actual data, and to compute summary statistics.

## Syntax

- |                |  |
|----------------|--|
| Command:       | <code>forecast(options) yhat [y_se]</code>   |
| Equation Proc: | <code>eq_name.forecast(options) yhat [y_se]</code>   |
| ARCH Proc:     | <code>eq_name.forecast(options) yhat [y_se y_var]</code>                                       |
| Sspace Proc:   | <code>ss_name.forecast(options) keyword1 names1 [keyword2 names2] [keyword3 names3] ...</code> |

When used with an equation, you should enter a name for the forecast series and, optionally, a name for the series containing the standard errors and, for ARCH specifications, a name for the conditional variance series. Forecast standard errors are currently not available for binary or censored models. `forecast` is not available for models estimated using ordered methods.

When used with a `sspace`, you should enter a *type*-keyword followed by a list of names for the target series or a wildcard expression, and if desired, additional *type*-keyword and target pairs. The following are valid keywords: “@STATE”, “@STATESE”, “@SIGNAL”, “@SIGNALSE”. The first two keywords instruct EViews to forecast the state series and the values of the state standard error series. The latter two keywords instruct EViews to forecast the signal series and the values of the signal standard error series.

If a list is used to identify the targets in `sspace` forecasting, the number of target series must match the number of names implied by the keyword. Note that wildcard expressions may not be used for forecasting signal variables that contain expressions. In addition, the “\*” wildcard expression may not be used for forecasting signal variables since this would overwrite the original data.

## Options

### *Options for Equation forecasting*

d	In models with implicit dependent variables, forecast the entire expression rather than the normalized variable.
u	Substitute expressions for all auto-updating series in the equation.
g	Graph the forecasts together with the $\pm 2$ standard error bands.
e	Produce the forecast evaluation table.
i	Compute the forecasts of the index. Only for binary, censored and count models.
s	Ignore ARMA terms and use only the structural part of the equation to compute the forecasts.
f = arg (default = “actual”)	Out-of-forecast-sample fill behavior: “actual” (fill observations outside the forecast sample with actual values for the fitted variable), “na” (fill observations outside the forecast sample with missing values).

### Options for Sspace forecasting

<code>i = arg (default = "o")</code>	State initialization options: “o” (one-step), “e” (diffuse), “u” (user-specified), “s” (smoothed).
<code>m = arg (default = "d")</code>	Basic forecasting method: “n” ( $n$ -step ahead forecasting), “s” (smoothed forecasting), “d” (dynamic forecasting).
<code>mprior = vector_name</code>	Name of state initialization (use if option “i = u” is specified).
<code>n = arg (default = 1)</code>	Number of $n$ -step forecast periods (only relevant if $n$ -step forecasting is specified using the <i>method</i> option).
<code>vprior = sym_name</code>	Name of state covariance initialization (use if option “i = u” is specified).

### Examples

The following lines:

```
smp1 1970q1 1990q4
equation eq1.ls con c con(-1) inc
smp1 1991q1 1995q4
eq1.fit con_s
eq1.forecast con_d
plot con_s con_d
```

estimate a linear regression over the period 1970Q1–1990Q4, compute static and dynamic forecasts for the period 1991Q1–1995Q4, and plot the two forecasts in a single graph.

```
equation eq1.ls m1 gdp ar(1) ma(1)
eq1.forecast m1_bj bj_se
eq1.forecast(s) m1_s s_se
plot bj_se s_se
```

estimates an ARMA(1,1) model, computes the forecasts and standard errors with and without the ARMA terms, and plots the two forecast standard errors.

The following command performs  $n$ -step forecasting of the signals and states from a sspace object:

```
ss1.forecast(method=n, n=4) @state * @signal y1f y2f
```

Here, we save the state forecasts in the names specified in the sspace object, and we save the two signal forecasts in the series Y1F and Y2F.

## Cross-references

To perform static forecasting with equation objects see [fit \(p. 285\)](#). For multiple equation forecasting, see [makemodel \(p. 341\)](#), and [solve \(p. 451\)](#).

For more information on equation forecasting in EViews, see [Chapter 18, “Forecasting from an Equation”, on page 527](#) of the *User’s Guide*. State space forecasting is described in [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide*. For additional discussion of wildcards, see [Appendix B, “Wildcards”, on page 921](#) of the *User’s Guide*.

freeze	Command
--------	---------

Creates graph, table, or text objects from a view.

### Syntax

Command:      `freeze(name) object_name.view_command`

If you follow the keyword `freeze` with an object name but no view of the object, `freeze` will use the default view for the object. You may provide a destination name for the object containing the frozen view in parentheses.

### Examples

```
freeze gdp.uroot(4,t)
```

creates an untitled table that contains the results of the unit root test of the series GDP.

```
group rates tb1 tb3 tb6
freeze(gral) rates.line(m)
show gral.align(2,1,1)
```

freezes a graph named GRA1 that contains multiple line graphs of the series in the group RATES, realigns the individual graphs, and displays the resulting graph.

```
freeze(mygra) gral gra3 gra4
show mygra.align(2,1,1)
```

creates a graph object named MYGRA that combines three graph objects GRA1, GRA2, and GRA3, and displays MYGRA in two columns.

## Cross-references

See [“Object Commands” on page 6](#). See also [Chapter 4, “Object Basics”, on page 65](#) of the *User’s Guide* for further discussion of objects and views of objects. Freezing graph views is described in [“Creating Graphs” on page 403](#) of the *User’s Guide*.

## freq

[Alpha View](#) | [Group View](#) | [Series View](#)

Compute frequency tables.

When used as a series view (or for a group containing a single series), `freq` performs a one-way frequency tabulation. The options allow you to control binning (grouping) of observations.

When used with a group containing multiple series, `freq` produces an  $N$ -way frequency tabulation for all of the series in the group.

### Syntax

Object View:      `object_name.freq(options)`

### Options

*Options common to both one-way and N-way frequency tables*

<code>dropna (default)</code>	[Drop/Keep] NA as a category. <code>/ keepna</code>
<code>v = integer (default = 100)</code>	Make bins if the number of distinct values or categories exceeds the specified number.
<code>nov</code>	Do not make bins on the basis of number of distinct values; ignored if you set “ <code>v = integer</code> .”
<code>a = number (default = 2)</code>	Make bins if average count per distinct value is less than the specified number.
<code>noa</code>	Do not make bins on the basis of average count; ignored if you set “ <code>a = number</code> .”
<code>b = integer (default = 5)</code>	Maximum number of categories to bin into.
<code>n, obs, count (default)</code>	Display frequency counts.
<code>nocount</code>	Do not display frequency counts.
<code>p</code>	Print the table.

*Options for one-way tables*

<code>total (default) /</code>	[Display / Do not display] totals. <code>nototal</code>
--------------------------------	--

pct ( <i>default</i> ) /	[Display / Do not display] percent frequencies.
nopct	
cum ( <i>default</i> ) /	(Display/Do not) display cumulative frequency counts/
nocum	percentages.

*Options for N-way tables*

table ( <i>default</i> )	Display in table mode.
list	Display in list mode.
rowm ( <i>default</i> ) /	[Display / Do not display] row marginals.
norowm	
colm ( <i>default</i> ) /	[Display / Do not display] column marginals.
nocolm	
tabm ( <i>default</i> ) /	[Display / Do not display] table marginals—only for
notabm	more than two series.
subm ( <i>default</i> ) /	[Display / Do not display] sub marginals—only for “l”
nosubm	option with more than two series.
full ( <i>default</i> ) /	(Full/Sparse) tabulation in list display.
sparse	
totpct / nototpct	[Display / Do not display] percentages of total observa-
( <i>default</i> )	tions.
tabpct / notab-	[Display / Do not display] percentages of table observa-
pct ( <i>default</i> )	tions—only for more than two series.
rowpct / norow-	[Display / Do not display] percentages of row total.
pct ( <i>default</i> )	
colpct / nocolpct	[Display / Do not display] percentages of column total.
( <i>default</i> )	
exp / noexp	[Display / Do not display] expected counts under full
( <i>default</i> )	independence.
tabexp / nota-	[Display / Do not display] expected counts under table
bexp ( <i>default</i> )	independence—only for more than two series.
test ( <i>default</i> ) /	[Display / Do not display] tests of independence.
notest	

## Examples

```
hrs.freq(nov,noa)
```

tabulates each value (no binning) of HRS in ascending order with counts, percentages, and cumulatives.

```
inc.freq(v=20,b=10,noa)
```

tabulates INC excluding NAs. The observations will be binned if INC has more than 20 distinct values; EViews will create at most 10 equal width bins. The number of bins may be smaller than specified.

```
group labor lwage gender race
labor.freq(v=10,norowm,nocolm)
```

displays tables of LWAGE against GENDER for each bin/value of RACE.

## Cross-references

See “[One-Way Tabulation](#)” on page 313 and “[N-Way Tabulation](#)” on page 369 of the *User’s Guide* for a discussion of frequency tables.

<b>frml</b>	<a href="#">Command</a>    <a href="#">Object Declaration (Alpha)</a>   <a href="#">Object Declaration (Series)</a>
-------------	---

Declare a series object with a formula for auto-updating, or specify a formula for an existing series.

## Syntax

Command:      `frml series_name = series_expression`

Command:      `frml series_name = @clear`

Follow the `frml` keyword with a name for the object, and an assignment statement. The special keyword “`@CLEAR`” is used to return the auto-updating series to an ordinary numeric or alpha series.

## Examples

To define an auto-updating numeric or alpha series, you must use the `frml` keyword prior to entering an assignment statement. The following example creates a series named `LOW` that uses a formula to compute its values.:

```
frml low = inc<=5000 or edu<13
```

The auto-updating series takes the value 1 if either INC is less than or equal to 5000 or EDU is less than 13, and 0 otherwise, and will be re-evaluated whenever INC or EDU change.

If FIRST\_NAME and LAST\_NAME are alpha series, then the formula declaration:

```
frml full_name = first_name + " " + last_name
```

creates an auto-updating alpha series FULL\_NAME.

You may apply a `frml` to an existing series. The commands:

```
series z = 3
```

```
frml z = (x+y) / 2
```

makes the previously created series Z an auto-updating series containing the average of series X and Y. Note that once a series is defined to be auto-updating, it may not be modified directly. Here, you may not edit Z, nor may you generate values into the series.

Note that the commands :

```
series z = 3
```

```
z = (x+y) / 2
```

while similar, produce quite different results, since the absence of the `frml` keyword in the second example means that EViews will generate fixed values in the series instead of defining a formula to compute the series values. In this latter case, the values in the series Z are fixed, and may be modified.

One particularly useful feature of auto-updating series is the ability to reference series in databases. The command:

```
frml gdp = usdata::gdp
```

creates a series called GDP that obtains its values from the series GDP in the database USDATA. Similarly:

```
frml lgdp = log(usdata::gdp)
```

creates an auto-updating series that is the log of the values of GDP in the database USDATA.

To turn off auto-updating for a series or alpha, you should use the special expression “@CLEAR” in your `frml` assignment. The command:

```
frml z = @clear
```

sets the series to numeric or alpha value format, freezing the contents of the series at the current values.

## Cross-references

See “[Auto-Updating Series](#)” on page 141 of the *User’s Guide*.

See also [link \(p. 323\)](#).

<b>garch</b>	<a href="#">Equation View</a>
--------------	-------------------------------

Conditional standard deviation graph of (G)ARCH equation.

Displays the conditional standard deviation graph of an equation estimated by ARCH.

## Syntax

Equation View:    `eq_name.garch(options)`

## Options

p	Print the graph
---	-----------------

## Examples

```
equation eq1.arch sp500 c
eq1.garch
```

estimates a GARCH(1,1) model and displays the estimated conditional standard deviation graph.

## Cross-references

ARCH estimation is described in [Chapter 20, “ARCH and GARCH Estimation”, on page 585](#) of the *User’s Guide*.

See also [arch \(p. 203\)](#) and [makegarch \(p. 336\)](#).

<b>genr</b>	<a href="#">Command</a>    <a href="#">Object Declaration (Alpha)</a>   <a href="#">Object Declaration (Series)</a>   <a href="#">Pool Proc</a>
-------------	--

Generate series.

Generate series or alphas. This procedure also allows you to generate multiple series using the cross-section identifiers in a pool.

## Syntax

Command:      `genr ser_name = expression`  
Pool Proc:      `pool_name.genr ser_name = expression`

For pool series generation, you may use the cross section identifier “?” in the series name and/or in the expression on the right-hand side.

## Examples

```
genr y = 3 + x
```

generates a numeric series that takes the values from the series X and adds 3.

```
genr full_name = first_name + last_name
```

creates an alpha series formed by concatenating the alpha series FIRST\_NAME and LAST\_NAME.

The commands,

```
pool pool1  
pool1.add 1 2 3  
pool1.genr y? = x? - @mean(x?)
```

are equivalent to generating separate series for each cross-section:

```
genr y1 = x1 - @mean(x1)  
genr y2 = x2 - @mean(x2)  
genr y3 = x3 - @mean(x3)
```

Similarly:

```
pool pool2  
pool2.add us uk can  
pool2.genr y_? = log(x_?) - log(x_us)
```

generates three series Y\_US, Y\_UK, Y\_CAN that are the log differences from X\_US. Note that Y\_US = 0.

It is worth noting that the pool `genr` command simply loops across the cross-section identifiers, performing the evaluations using the appropriate substitution. Thus, the command,

```
pool2.genr z = y_?
```

is equivalent to entering:

```
genr z = y_us  
genr z = y_uk
```

---

```
genr z = y_can
```

so that upon completion, the ordinary series Z will contain Y\_CAN.

### Cross-references

See [Chapter 27, “Pooled Time Series, Cross-Section Data”, on page 809](#) of the *User’s Guide* for a discussion of the computation of pools, and a description of individual and balanced samples.

See [series \(p. 418\)](#) and [alpha \(p. 201\)](#) for a discussion of the expressions allowed in genr.

gmm	<a href="#">Command</a>    <a href="#">Equation Method</a>   <a href="#">System Method</a>
-----	--

Estimation by generalized method of moments (GMM).

The equation or system object must be specified with a list of instruments.

### Syntax

Command:      *gmm(options) y x1 [x2 x3 ...] @ z1 [z2 z3 ...]*  
*gmm(options) specification @ z1 [z2 z3 ...]*

Equation Method: *eq\_name.gmm(options) y x1 [x2 x3 ...] @ z1 [z2 z3 ...]*  
*eq\_name.gmm(options) specification @ z1 [z2 z3 ...]*

System Method:    *system\_name.gmm(options)*

To use `gmm` as a command or equation method, follow the name of the dependent variable by a list of regressors, followed by the “@” symbol, and a list of instrumental variables which are orthogonal to the residuals. Alternatively, you can specify an expression using coefficients, an “@” symbol, and a list of instrumental variables. There must be at least as many instrumental variables as there are coefficients to be estimated.

In panel settings, you may specify dynamic instruments corresponding to predetermined variables. To specify a dynamic instrument, you should tag the instrument using “@DYN”, as in “@DYN(X)”. By default, EViews will use a set of period-specific instruments corresponding to lags from -2 to “-infinity”. You may also specify a restricted lag range using arguments in the “@DYN” tag. For example, to use lags from -5 to “-infinity” you may enter “@DYN(X, -5)”; to specify lags from -2 to -6, use “@DYN(X, -2, -6)” or “@DYN(X, -6, -2)”.

Note that dynamic instrument specifications may easily generate excessively large numbers of instruments.

## Options

### *General Options*

$m = \text{integer}$	Maximum number of iterations.
$c = \text{number}$	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
$l = \text{number}$	Set maximum number of iterations on the first-stage iteration to get the one-step weighting matrix.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
deriv = <i>keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
p	Print results.

### *Additional Options for Non-Panel Equation and System estimation*

w	Use White’s diagonal weighting matrix (for cross section data).
b = <i>arg</i> ( <i>default</i> = “nw”)	Specify the bandwidth: “nw” (Newey-West fixed bandwidth based on the number of observations), “ <i>number</i> ” (user specified bandwidth), “v” (Newey-West automatic variable bandwidth selection), “a” (Andrews automatic selection).
q	Use the quadratic kernel. Default is to use the Bartlett kernel.
n	Prewhiten by a first order VAR before estimation.
i	Iterate simultaneously over the weighting matrix and the coefficient vector.
s	Iterate sequentially over the weighting matrix and coefficient vector.
o ( <i>default</i> )	Iterate only on the coefficient vector with one step of the weighting matrix.

c One step (iteration) of the coefficient vector following one step of the weighting matrix.

e TSLS estimates with GMM standard errors.

*Additional Options for Panel Equation estimation*

cx = *arg* Cross-section effects method: (default) none, fixed effects estimation (“cx = f”), first-difference estimation (“cx = fd”), orthogonal deviation estimation (“cx = od”)

per = *arg* Period effects method: (default) none, fixed effects estimation (“per = f”).

levelper Period dummies always specified in levels (even if one of the transformation methods is used, “cx = fd” or “cx = od”).

wgt = *arg* GLS weighting: (default) none, cross-section system weights (“wgt = cxsur”), period system weights (“wgt = persur”), cross-section diagonal weights (“wgt = cxdiag”), period diagonal weights (“wgt = perdiag”).

gmm = *arg* GMM weighting: Identity (“gmm = ident”), White period system covariances (Arellano-Bond 2-step/n-step) (“gmm = perwhite”), White cross-section system (“gmm = cxwhite”), White diagonal (“gmm = stackedwhite”), Period system (“gmm = persur”), Cross-section system (“gmm = cxsur”), Period heteroskedastic (“cov = perdiag”), Cross-section heteroskedastic (“gmm = cxdiag”).

By default, uses the identity matrix unless estimated with first difference transformation (“cx = fd”), in which case, uses (Arellano-Bond 1-step) difference weighting matrix. In this latter case, you should specify identity weights (“gmm = ident”) for Anderson-Hsiao estimation.

cov = <i>arg</i>	Coefficient covariance method: (default) ordinary, White cross-section system robust (“cov = cxwhite”), White period system robust (“cov = perwhite”), White heteroskedasticity robust (“cov = stackedwhite”), Cross-section system robust/PCSE (“cov = cxsur”), Period system robust/PCSE (“cov = persur”), Cross-section heteroskedasticity robust/PCSE (“cov = cxdiag”), Period heteroskedasticity robust (“cov = perdiag”).
keepwgts	Keep full set of GLS/GMM weights used in estimation with object, if applicable (by default, only weights which take up little memory are saved).
nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
coef = <i>arg</i>	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the “C” coefficient vector.
iter = <i>arg</i> ( <i>default</i> = “onec”)	Iteration control for GLS and GMM weighting specifications: perform one weight iteration, then iterate coefficients to convergence (“iter = onec”), iterate weights and coefficients simultaneously to convergence (“iter = sim”), iterate weights and coefficients sequentially to convergence (“iter = seq”), perform one weight iteration, then one coefficient step (“iter = oneb”).

Note that some options are only available for a subset of specifications.

## Examples

In a non-panel workfile, we may estimate equations using the standard GMM options. The specification:

```
eq.gmm(w) y c f k @ c z1 z2 z3
```

estimates the linear specification using a White diagonal weighting matrix (one-step, with no iteration). The command:

```
eq.gmm(e, b=v) c(1) + c(2)*(m^c(1) + k^(1-c(1))) @ c z1 z2 z3
```

estimates the nonlinear model using two-stage least squares (instrumental variables) with GMM standard errors computed using Newey-West automatic bandwidth selected weights.

For system estimation, the command:

---

```
sys1.gmm (b=a, q, i)
```

estimates the system SYS1 by GMM with a quadratic kernel, Andrews automatic bandwidth selection, and iterates until convergence.

When working with a workfile that has a panel structure, you may use the panel equation estimation options. The command

```
eq.gmm(cx=fd, per=f) dj dj(-1) @ @dyn(dj)
```

estimates an Arellano-Bond “1-step” estimator with differencing of the dependent variable DJ, period fixed effects, and dynamic instruments constructed using DJ with observation specific lags from period  $t - 2$  to 1.

To perform the “2-step” version of this estimator, you may use:

```
eq.gmm(cx=fd, per=f, gmm=perwhite, iter=oneb) dj dj(-1) @
@dyn(dj)
```

where the combination of the options “gmm = perwhite” and (the default) “iter = oneb” instructs EViews to estimate the model with the difference weights, to use the estimates to form period covariance GMM weights, and then re-estimate the model.

You may iterate the GMM weights to convergence using:

```
eq.gmm(cx=fd, per=f, gmm=perwhite, iter=seq) dj dj(-1) @
@dyn(dj)
```

Alternately:

```
eq.gmm(cx=od, gmm=perwhite, iter=oneb) dj dj(-1) x @
@dyn(dj,-2,-6) x(-1) y(-1)
```

estimates an Arellano-Bond “2-step” equation using orthogonal deviations of the dependent variable, dynamic instruments constructed from DJ from period  $t - 6$  to  $t - 2$ , and ordinary instruments X(-1) and Y(-1).

## Cross-references

See [Chapter 16, “Additional Regression Methods”, on page 445](#), [Chapter 23, “System Estimation”, on page 679](#), and [Chapter 29, “Panel Estimation”, on page 885](#) of the *User’s Guide* for discussion of the various GMM estimation techniques.

**grads**[Equation View](#) | [Logl View](#) | [Sspace View](#) | [System View](#)

Gradients of the objective function.

Displays the gradients of the objective function (where available) for objects containing an estimated equation or equations.

The (default) summary form shows the value of the gradient vector at the estimated parameter values (if valid estimates exist) or at the current coefficient values. Evaluating the gradients at current coefficient values allows you to examine the behavior of the objective function at starting values. The tabular form shows a spreadsheet view of the gradients for each observation. The graphical form shows this information in a multiple line graph.

### Syntax

Object View:      `object_name.grads(options)`

### Options

`g`      Display multiple graph showing the gradients of the objective function with respect to the coefficients evaluated at each observation (not available for systems).

`t (default)`      Display spreadsheet view of the values of the gradients of the objective function with respect to the coefficients evaluated at each observation (not available for systems).

`p`      Print results.

### Examples

To show a summary view of the gradients:

```
eq1.grads
```

To display and print the table view:

```
eq1.grads(t, p)
```

### Cross-references

See also [derivs](#) (p. 264), [makederivs](#) (p. 335), and [makegrads](#) (p. 337).

**graph****Object Declaration**

Create named graph object containing the results of a graph command, or created when merging multiple graphs into a single graph.

**Syntax**

Command:      `graph graph_name.graph_command(options) arg1 [arg2 arg3 ...]`

Command:      `graph graph_name.merge graph1 graph2 [graph3 ...]`

Follow the keyword with a name for the graph, a period, and then a statement used to create a graph. There are two distinct forms of the command.

In the first form of the command, you create a graph using one of the graph commands, and then name the object using the specified name. The portion of the command given by,

`graph_command(options) arg1 [arg2 arg3 ...]`

should follow the form of one of the standard EViews graph commands:

area	Area graph ( <a href="#">area (p. 207)</a> ).
bar	Bar graph ( <a href="#">bar (p. 215)</a> ).
errbar	Error bar graph ( <a href="#">errbar (p. 276)</a> ).
hilo	High-low(-open-close) graph ( <a href="#">hilo (p. 307)</a> ).
line	Line graph ( <a href="#">line (p. 320)</a> ).
pie	Pie graph ( <a href="#">pie (p. 384)</a> ).
scat	Scatterplot—same as XY, but lines are initially turned off, symbols turned on, and a $3 \times 3$ frame is used ( <a href="#">scat (p. 412)</a> ).
spike	Spike graph ( <a href="#">spike (p. 455)</a> ).
xy	XY line-symbol graph with one X plotted against one or more Y's using existing line-symbol settings ( <a href="#">xy (p. 528)</a> ).
xyline	Same as XY, but symbols are initially turned off, lines turned on, and a $4 \times 3$ frame is used ( <a href="#">xyline (p. 530)</a> ).
xypair	Same as XY but sets XY settings to display pairs of X and Y plotted against each other ( <a href="#">xypair (p. 532)</a> ).

In the second form of the command, you instruct EViews to merge the listed graphs into a single graph, and then name the graph object using the specified name.

## Options

p	Print the graph (for use when specified with a graph command).
---	--

Additional options will depend on the type of graph chosen. See the entry for each graph type for a list of the available options (for example, see [bar \(p. 215\)](#) for details on bar graphs).

## Examples

```
graph gra1.line(s, p) gdp m1 inf
```

creates and prints a stacked line graph object named GRA1. This command is equivalent to running the command:

```
line(s, p) gdp m1 inf
```

freezing the view, and naming the graph GRA1.

```
graph mygra.merge gr_line gr_scat gr_pie
```

creates a multiple graph object named MYGRA that merges three graph objects named GR\_LINE, GR\_SCAT, and GR\_PIE.

## Cross-references

The graph object is described in greater detail in “[Graph \(p. 159\)](#)”. See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a general discussion of graphs.

See also [freeze \(p. 290\)](#) and [merge \(p. 351\)](#).

group	<a href="#">Object Declaration</a>
-------	------------------------------------

Declare a group object containing a group of series.

## Syntax

Command:      **group** **group\_name** *ser1 ser2 [ser3 ...]*

Follow the group name with a list of series to be included in the group.

## Examples

```
group g1 gdp cpi inv
group g1 tb3 m1 gov
g1.add gdp cpi
```

The first line creates a group named G1 that contains three series GDP, CPI, and INV. The second line redeclares group G1 to contain the three series TB3, M1, and GOV. The third line adds two series GDP and CPI to group G1 to make a total of five series. See [add \(p. 196\)](#).

```
group rhs d1 d2 d3 d4 gdp(0 to -4)
ls cons rhs
ls cons c rhs(6)
```

The first line creates a group named RHS that contains nine series. The second line runs a linear regression of CONS on the nine series in RHS. The third line runs a linear regression of CONS on C and only the sixth series GDP(-1) of RHS.

## Cross-references

See “[Group](#)” on page 161 for a complete description of the group object. See also [Chapter 12, “Groups”, on page 354](#) of the *User’s Guide* for additional discussion.

See also [add \(p. 196\)](#) and [drop \(p. 270\)](#).

<b>hconvert</b>	<a href="#">Command</a>
-----------------	-------------------------

Convert an entire Haver Analytics Database into an EViews database.

## Syntax

Command:      **hconvert haver\_path db\_name**

*You must have a Haver Analytics database installed on your computer to use this feature.* You must also create an EViews database to store the converted Haver data *before* you use this command.

Be aware that this command may be very time-consuming.

Follow the command by a *full path name* to the Haver database and the name of an existing EViews database to store the Haver database. You can include a path name to the EViews database not in the default path.

## Examples

```
dbccreate hdata
```

```
hconvert d:\haver\haver hdata
```

The first line creates a new (empty) database named HDATA in the default directory. The second line converts all the data in the Haver database and stores it in the HDATA database.

### Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews and Haver databases.

See also [dbcreate \(p. 255\)](#), [db \(p. 254\)](#), [hfetch \(p. 306\)](#) and [hlabel \(p. 309\)](#).

<b>hfetch</b>	<a href="#">Command</a>
---------------	-------------------------

Fetch a series from a Haver Analytics database into a workfile.

`hfetch` reads one or more series from a Haver Analytics Database into the active workfile. *You must have a Haver Analytics database installed on your computer to use this feature.*

### Syntax

Command:      `hfetch(database_name) series_name`

`hfetch`, if issued alone on a command line, will bring up a Haver dialog box which has fields for entering both the database name and the series names to be fetched. If you provide the database name (including the full path) in parentheses after the `hfetch` command, EViews will read the database and copy the series requested into the current workfile. It will also display information about the series on the status line. The database name is optional if a default database has been specified.

`hfetch` can read multiple series with a single command. List the series names, each separated by a space.

### Examples

```
hfetch(c:\data\us1) pop gdp xyz
```

reads the series POP, GDP, and XYZ from the US1 database into the current active workfile, performing frequency conversions if necessary.

### Cross-references

See also [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews and Haver databases. Additional information on EViews frequency conversion is provided in [“Frequency Conversion” on page 107](#) of the *User’s Guide*.

See also [dbcreate \(p. 255\)](#), [db \(p. 254\)](#), [hconvert \(p. 305\)](#) and [hlabel \(p. 309\)](#).

<b>hilo</b>	<a href="#">Command</a>    <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Sym View</a>
-------------	--

Display high-low[-open-close] graph view of object, or change existing graph object type to high-low[-open-close] (if possible).

## Syntax

Command:	<code>hilo(options) high_ser low_ser [close_ser]</code>
	<code>hilo(options) high_ser low_ser open_ser close_ser</code>
	<code>hilo(options) arg</code>
Graph Proc:	<code>graph_name.hilo(options)</code>
Object View:	<code>object_name.hilo(options)</code>

For a high-low, or a high-low-close graph, follow the command name with the name of the high series, followed by the low series, and an optional close series. If four series names are provided, EViews will use them in the following order: high-low-open-close.

When used with a matrix or group or an existing graph, EViews will use the first series as the high series, the second series as the low series, and if present, the third series as the close. If there are four or more series, EViews will use them in the following order: high-low-open-close. When used as a matrix view, the columns of the matrix are used in place of the series.

Note that if you wish to display a high-low-open graph, you should use an “NA”-series for the close values.

## Options

### *Template and printing options*

<code>o = graph_name</code>	Use appearance options from the specified graph.
<code>t = graph_name</code>	Use appearance options and copy text and shading from the specified graph.
<code>p</code>	Print the bar graph.

### *Panel options*

The following options apply when graphing panel structured data.

panel = *arg*  
(default taken  
from global set-  
tings) Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “medmxmn” (plot median, max and min).

## Examples

```
hilo mshigh mslow
```

displays a high-low graph using the series MSHIGH and MSLOW.

```
hilo(p) mshigh mslow msopen msclose
```

plots and prints the high-low-open-close graph of the four series.

```
group stockprice mshigh mslow msclose
```

```
stockprice.hilo
```

displays the high-low-close view of the group STOCKPRICE.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for additional details on using graphs in EViews.

See also [graph \(p. 303\)](#) for graph declaration and other graph types.

hist	<a href="#">Command</a>    <a href="#">Equation View</a>   <a href="#">Series View</a>
------	--

Histogram and descriptive statistics of a series.

When used as a command or a series view, `hist` computes descriptive statistics and displays a histogram for the series. When used as an equation view, `hist` displays a histogram and descriptive statistics of the residual series.

## Syntax

Command:      **hist(*options*) *series\_name***  
 Object View:    **object\_name.hist(*options*)**

## Options

p	Print the histogram.
---	----------------------

## Examples

`lwage.hist`

Displays the histogram and descriptive statistics of LWAGE.

## Cross-references

See “[Histogram and Stats](#)” on page 298 of the *User’s Guide* for a discussion of the descriptive statistics reported in the histogram view.

hlabel	Command
--------	---------

Display a Haver Analytics database series description.

`hlabel` reads the description of a series from a Haver Analytics Database and displays it on the status line at the bottom of the EViews window. Use this command to verify the contents of a Haver database series name.

*You must have a Haver Analytics database installed on your computer to use this feature.*

## Syntax

Command:      **hlabel(*database\_name*) *series\_name***

`hlabel`, if issued alone on a command line, will bring up a Haver dialog box which has fields for entering both the database name and the series names to be examined. If you provide the database name in parentheses after the `hlabel` command, EViews will read the database and find the key information about the series in question, such as the start date, end date, frequency, and description. This information is displayed in the status line at the bottom of the EViews window. Note that the *database\_name* should refer to the full path to the Haver database but need not be specified if a default database has been specified in HAVERDIR.INI.

If several series names are placed on the line, `hlabel` will gather the information about each of them, but the information display may scroll by so fast that only the last will be visible.

## Examples

```
hlabel(c:\data\us1) pop
```

displays the description of the series POP in the US1 database.

## Cross-references

See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews and Haver databases.

See also [hfetch \(p. 306\)](#) and [hconvert \(p. 305\)](#).

hpf	<a href="#">Command</a>    <a href="#">Series Proc</a>
-----	--

Smooth a series using the Hodrick-Prescott filter.

## Syntax

Command:        `hpf(options) series_name filtered_name`

Series Proc:     `series_name.hpf(options) filtered_name`

When used as a series procedure, `hpf` smooths the series without displaying the graph of the smoothed series.

## Smoothing Options

The degree of smoothing may be specified as an option. You may specify the smoothing as a value, or using a power rule:

<code>lambda = arg</code>	Set smoothing parameter value to <i>arg</i> ; a larger number results in greater smoothing.
---------------------------	---

<code>power = arg (default = 2)</code>	Set smoothing parameter value using the frequency power rule of Ravn and Uhlig (2002) (the number of periods per year divided by 4, raised to the power <i>arg</i> , and multiplied by 1600).
--	---

Hodrick and Prescott recommend the value 2; Ravn and Uhlig recommend the value 4.

If no smoothing option is specified, EViews will use the power rule with a value of 2.

## Other Options

<code>p</code>	Print the graph of the smoothed series and the original series.
----------------	---

## Examples

```
gdp.hpf(lambda=1000) gdp_hp
```

smooths the GDP series with a smoothing parameter “1000” and saves the smoothed series as GDP\_HP.

## Cross-references

See “[Hodrick-Prescott Filter](#)” on page 344 of the *User’s Guide* for details.

impulse	<a href="#">Var View</a>
---------	--------------------------

Display impulse response functions of var object with an estimated VAR or VEC.

## Syntax

Var View:	<code>var_name.impulse(<i>n, options</i>) ser1 [ser2 ser3 ...] [@ shock_series [@ ordering_series]]</code>
-----------	--

You must specify the number of periods *n* for which you wish to compute the impulse responses.

List the series names in the var whose responses you would like to compute . You may optionally specify the sources of shocks. To specify the shocks, list the series after an “@”. By default, EViews computes the responses to all possible sources of shocks using the ordering in the Var.

If you are using impulses from the Cholesky factor, you may change the Cholesky ordering by listing the order of the series after a second “@”.

## Options

g ( <i>default</i> )	Display combined graphs, with impulse responses of one variable to all shocks shown in one graph. If you choose this option, standard error bands will not be displayed.
m	Display multiple graphs, with impulse response to each shock shown in separate graphs.
t	Tabulate the impulse responses.
a	Accumulate the impulse responses.

<code>imp = arg</code>	Type of factorization for the decomposition: unit
( <i>default</i> = “chol”)	impulses, ignoring correlations among the residuals (“ <code>imp = unit</code> ”), non-orthogonal, ignoring correlations among the residuals (“ <code>imp = nonort</code> ”), Cholesky with d.f. correction (“ <code>imp = chol</code> ”), Cholesky without d.f. correction (“ <code>imp = mlechol</code> ”), Generalized (“ <code>imp = gen</code> ”), structural (“ <code>imp = struct</code> ”), or user specified (“ <code>imp = user</code> ”).
	The structural factorization is based on the estimated structural VAR. To use this option, you must first estimate the structural decomposition; see <a href="#">svar (p. 469)</a> .
	For user-specified impulses, you must specify the name of the vector/matrix containing the impulses using the “ <code>fname = </code> ” option.
	The option “ <code>imp = mlechol</code> ” is provided for backward compatibility with EViews 3.x and earlier.
<code>fname = name</code>	Specify name of vector/matrix containing the impulses. The vector/matrix must have $k$ rows and 1 or $k$ columns, where $k$ is the number of endogenous variables.
<code>se = arg</code>	Standard error calculations: “ <code>se = a</code> ” (analytic), “ <code>se = mc</code> ” (Monte Carlo).  If selecting Monte Carlo, you must specify the number of replications with the “ <code>rep = </code> ” option.  Note the following: <ol style="list-style-type: none"><li>(1) Analytic standard errors are currently not available for (a) VECs and (b) structural decompositions identified by long-run restrictions. The “<code>se = a</code>” option will be ignored for these cases.</li><li>(2) Monte Carlo standard errors are currently not available for (a) VECs and (b) structural decompositions. The “<code>se = mc</code>” option will be ignored for these cases.</li></ol>
<code>rep = integer</code>	Number of Monte Carlo replications to be used in computing the standard errors. Must be used with the “ <code>se = mc</code> ” option.
<code>matbys = name</code>	Save responses by shocks (impulses) in named matrix. The first column is the response of the first variable to the first shock, the second column is the response of the second variable to the first shock, and so on.

matbyr = <i>name</i>	Save responses by response series in named matrix. The first column is the response of the first variable to the first shock, the second column is the response of the first variable to the second shock, and so on.
p	Print the results.

## Examples

```
var var1.ls 1 4 m1 gdp cpi
var1.impulse(10,m) gdp @ m1 gdp cpi
```

The first line declares and estimates a VAR with three variables. The second line displays multiple graphs of the impulse responses of GDP to shocks to the three series in the VAR using the ordering as specified in VAR1.

```
var1.impulse(10,m) gdp @ m1 @ cpi gdp m1
```

displays the impulse response of GDP to a one standard deviation shock in M1 using a different ordering.

## Cross-references

See [Chapter 24, “Vector Autoregression and Error Correction Models”, on page 705](#) of the *User’s Guide* for a discussion of variance decompositions in VARs.

See also [decomp \(p. 260\)](#).

jbera	<a href="#">Var View</a>
-------	--------------------------

Multivariate residual normality test.

## Syntax

Var View:            `var_name.jbera(options)`

You must specify a factorization method using the “factor =” option.

## Options

factor = chol	Factorization by the inverse of the Cholesky factor of the residual covariance matrix.
factor = cor	Factorization by the inverse square root of the residual correlation matrix (Doornik and Hansen, 1994).

factor = cov	Factorization by the inverse square root of the residual covariance matrix (Urzua, 1997).
factor = svar	Factorization matrix from structural VAR. You must first estimate the structural factorization to use this option; see <a href="#">svar (p. 469)</a> .
name = <i>arg</i>	Save the test statistics in a named matrix object. See below for a description of the statistics contained in the stored matrix.
p	Print the test results.

The “name = ” option stores the following matrix. Let the VAR have  $k$  endogenous variables. Then the stored matrix will have dimension  $(k + 1) \times 4$ . The first  $k$  rows contain statistics for each orthogonal component, where the first column contains the third moments, the second column contains the  $\chi_1^2$  statistics for the third moments, the third column contains the fourth moments, and the fourth column holds the  $\chi_1^2$  statistics for the fourth moments. The sum of the second and fourth columns are the Jarque-Bera statistics reported in the last output table.

The last row contains statistics for the joint test. The second and fourth column of the  $(k + 1)$  row is simply the sum of all the rows above in the corresponding column and are the  $\chi_k^2$  statistics for the joint skewness and kurtosis tests, respectively. These joint skewness and kurtosis statistics add up to the joint Jarque-Bera statistic reported in the output table, except for the “factor = cov” option. When this option is set, the joint Jarque-Bera statistic includes all cross moments (in addition to the pure third and fourth moments). The overall Jarque-Bera statistic for this statistic is stored in the first column of the  $(k + 1)$  row (which will be a missing value for all other options).

## Examples

```
var var1.ls 1 6 lgdp lm1 lcpi  
show var1.jbera(factor=cor, name=jb)
```

The first line declares and estimates a VAR. The second line carries out the residual multivariate normality test using the inverse square root of the residual correlation matrix as the factorization matrix and stores the results in a matrix named JB.

## Cross-references

See [Chapter 24, “Vector Autoregression and Error Correction Models”, on page 705](#) of the *User’s Guide* for a discussion of the test and other VAR diagnostics.

---

<b>kdensity</b>	<a href="#">Series View</a>
-----------------	-----------------------------

Kernel density plots. Displays nonparametric kernel density estimates of the specified series.

## Syntax

Series View:      `series_name.kdensity(options)`

## Options

<code>k = arg</code> <i>(default = "e")</i>	Kernel type: "e" (Epanechnikov), "r" (Triangular), "u" (Uniform), "n" (Normal-Gaussian), "b" (Biweight-Quartic), "t" (Triweight), "c" (Cosinus).
<code>s</code> ( <i>default</i> )	Automatic bandwidth (Silverman).
<code>b = number</code>	Specify a number for the bandwidth.
<code>b</code>	Bracket bandwidth.
<code>integer</code> <i>(default = 100)</i>	Number of points to evaluate.
<code>x</code>	Exact evaluation of kernel density.
<code>o = arg</code>	Name of matrix to hold results of kernel density computation. The first column of the matrix contains the evaluation points and the remaining columns contain the kernel estimates.
<code>p</code>	Print the kernel density plot.

## Examples

```
lwage.kdensity(k=n)
```

plots the kernel density estimate of LWAGE using a normal kernel and the automatic bandwidth selection.

## Cross-references

See “[Kernel Density](#)” on page 384 of the *User’s Guide* for a discussion of kernel density estimation.

**kerfit**[Group View](#)

Fits a kernel regression of the second series in the group (vertical axis) against the first series in the group (horizontal axis).

**Syntax**

Group View:      `group_name.kerfit(options)`

**Options**

`k = arg`      Kernel type: “e” (Epanechnikov), “r” (Triangular), “u” (Uniform), “n” (Normal–Gaussian), “b” (Biweight–Quartic), “t” (Triweight), “c” (Cosinus).  
*(default = “e”)*

`b = number`      Specify a number for the bandwidth.

`b`      Bracket bandwidth.

`integer`      Number of grid points to evaluate.  
*(default = 100)*

`x`      Exact evaluation of the polynomial fit.

`d = integer`      Degree of polynomial to fit. Set “d = 0” for Nadaraya–Watson regression.  
*(default = 1)*

`s = name`      Save fitted series.

`p`      Print the kernel fit plot.

**Examples**

```
group gg weight height  
gg.kerfit(s=w_fit, 200)
```

Fits a kernel regression of HEIGHT on WEIGHT using 200 points and saves the fitted series as W\_FIT.

**Cross-references**

See “[Scatter with Kernel Fit](#)” on page 393 of the *User’s Guide* for a discussion of kernel regression.

See also [linefit \(p. 322\)](#), [nnfit \(p. 355\)](#).

---

<b>label</b>	Object View   Object Proc
--------------	---------------------------

Display or change the label view of the object, including the last modified date and display name (if any).

As a procedure, `label` changes the fields in the object label.

### Syntax

Object View:	<code>object_name.label</code>
Object Proc:	<code>object_name.label(options) text</code>

### Options

To modify the label, you should specify one of the following options along with optional text. If there is no text provided, the specified field will be cleared:

c	Clears all text fields in the label.
d	Sets the description field to <i>text</i> .
s	Sets the source field to <i>text</i> .
u	Sets the units field to <i>text</i> .
r	Appends <i>text</i> to the remarks field as an additional line.
p	Print the label view.

### Examples

The following lines replace the remarks field of LWAGE with “Data from CPS 1988 March File”:

```
lwage.label(r)
lwage.label(r) Data from CPS 1988 March File
```

To append additional remarks to LWAGE, and then to print the label view:

```
lwage.label(r) Log of hourly wage
lwage.label(p)
```

To clear and then set the units field, use:

```
lwage.label(u) Millions of bushels
```

### Cross-references

See “[Labeling Objects](#)” on page 74 of the *User’s Guide* for a discussion of labels.

See also [displayname](#) (p. 266).

laglen	<a href="#">Var View</a>
--------	--------------------------

VAR lag order selection criteria.

### Syntax

Var View:      `var_name.laglen(m, options)`

You must specify the maximum lag order *m* for which you wish to test.

### Options

`vname = arg`      Save selected lag orders in named vector. See below for a description of the stored vector.

`mname = arg`      Save lag order criteria in named matrix. See below for a description of the stored matrix.

`p`      Print table of lag order criteria.

The “`vname =`” option stores a vector with 5 rows containing the selected lags from the following criteria: sequential modified LR test (row 1), final prediction error (row 2), Akaike information criterion (row 3), Schwarz information criterion (row 4), Hannan-Quinn information criterion (row 5).

The “`mname =`” option stores a  $q \times 6$  matrix, where  $q = m + 1$  if there are no exogenous variables in the VAR; otherwise  $q = m + 2$ . The first  $(q - 1)$  rows contain the information displayed in the table view, following the same order. The saved matrix has an additional row which contains the lag order selected from each column criterion. The first column (corresponding to the log likelihood values) of the last row is always an NA.

### Examples

```
var var1.ls 1 6 lgdp lm1 lcpi  
show var1.laglen(12,vname=v1)
```

The first line declares and estimates a VAR. The second line computes the lag length criteria up to a maximum of 12 lags and stores the selected lag orders in a vector named V1.

### Cross-references

See [Chapter 24, “Vector Autoregression and Error Correction Models”, on page 705](#) of the *User’s Guide* for a discussion of the various criteria and other VAR diagnostics.

See also [testlags](#) (p. 480).

legend	<a href="#">Graph Proc</a>
--------	----------------------------

Set legend appearance and placement in graphs.

When `legend` is used with a multiple graph, the legend settings apply to all graphs. See [setelem \(p. 426\)](#) for setting legends for individual graphs in a multiple graph.

## Syntax

Graph Proc:      `graph_name.legend option_list`

Note: the syntax of the `legend` proc has changed considerably from version 3.1 of EViews. While not documented here, the EViews 3 options are still (for the most part) supported. However, we do not recommend using the old options as future support is not guaranteed.

## Options

<code>columns(arg)</code>	<i>Columns for legend:</i> “auto” (automatically choose number of columns), <i>int</i> (put legend in specified number of columns).
<code>display/-display</code>	Display/do not display the legend.
<code>font(n1)</code>	Size of font for legend.
<code>inbox/-inbox</code>	Put legend in box/remove box around legend.
<code>position(arg)</code>	Position for legend: “left” or “l” (place legend on left side of graph), “right” or “r” (place legend on right side of graph), “botleft” or “bl” (place left-justified legend below graph), “botcenter” or “bc” (place centered legend below graph), “botright” or “br” (place right-justified legend below graph), “(h, v)” (the first number <i>h</i> specifies the number of virtual inches to offset to the right from the origin. The second number <i>v</i> specifies the virtual inch offset below the origin. The origin is the upper left hand corner of the graph).
<code>p</code>	Print the graph.

The default settings are taken from the global defaults.

## Examples

```
mygra1.legend display position(l) inbox
```

places the legend of MYGRA1 in a box to the left of the graph.

```
mygra1.legend position(.2,.2) -inbox
```

places the legend of MYGRA1 within the graph, indented slightly from the upper left corner with no box surrounding the legend text.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graph objects in EViews.

See also [setelem \(p. 426\)](#) for changing legend text and other graph options.

line	<a href="#">Command</a>    <a href="#">Coef View</a>   <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a>   <a href="#">Series View</a>   <a href="#">Sym View</a>   <a href="#">Vector View</a>
------	---

Display a line graph of object, or change existing graph object type to line plot.

The line graph view of a group plots all series in the group in a graph. The line graph view of a matrix plots each column in the matrix in a graph.

## Syntax

- Command:      `line(options) arg1 [arg2 arg3 ...]`  
Object View:    `object_name.line(options)`  
Graph Proc:    `graph_name.line(options)`

## Options

### *Template and printing options*

- |                             |  |
|-----------------------------|--|
| <code>o = graph_name</code> | Use appearance options from the specified graph.                           |
| <code>t = graph_name</code> | Use appearance options and copy text and shading from the specified graph. |
| <code>p</code>              | Print the line graph.  |

### *Scale options (for multiple line views of group and matrix objects)*

- |                          |   |
|--------------------------|---|
| <code>a (default)</code> | Automatic single scale.   |
| <code>d</code>           | Dual scaling with no crossing. The first series is scaled on the left and all other series are scaled on the right. |
| <code>x</code>           | Dual scaling with possible crossing. See the “d” option.  |
| <code>n</code>           | Normalized scale (zero mean and unit standard deviation). May not be used with the “s” option.                      |

---

s	Stacked line graph. Each area represents the cumulative total of the series listed. The difference between areas corresponds to the value of a series. May not be used with the “l” option.
m	Plot lines in multiple graphs (will override the “s” or “l” options). Not for use with an existing graph object.

### Panel options

The following options apply when graphing panel structured data:

panel = <i>arg</i> (default taken from global set- tings)	Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “medmxmn” (plot median, max and min).
--	--

### Examples

```
group g1 gdp cons m1
g1.line(d)
```

plots line graphs of the three series in group G1 with dual scaling (no crossing). The latter two series will share the same scale.

```
matrix1.line(t=mygra)
```

displays line graphs of the columns of MATRIX1 using the graph object MYGRA as a template.

```
line(m) gdp cons m1
```

creates an untitled graph object that contains three line graphs of GDP, CONS, and M1, with each plotted separately.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a detailed discussion of graphs in EViews.

See also [graph \(p. 303\)](#) for graph declaration and other graph types.

<b>linefit</b>	<a href="#">Group View</a>
----------------	----------------------------

Scatter plot with bivariate fit.

Displays the scatter plot of the second series (horizontal axis) and the first series (vertical axis) with a regression fit line. You can specify various transformation methods and weighting for the bivariate fit.

### Syntax

Group View:      `group_name.linefit(options)`

### Options

<code>yl</code>	Take the natural log of first series, $y$ .
<code>yi</code>	Take the inverse of $y$ .
<code>yp = number</code>	Take $y$ to the power of the specified number.
<code>yb = number</code>	Take the Box-Cox transformation of $y$ with the specified parameter.
<code>xl</code>	Take the natural log of $x$ .
<code>xi</code>	Take the inverse of $x$ .
<code>xp = number</code>	Take $x$ to the power of the specified number.
<code>xb = number</code>	Take the Box-Cox transformation of $x$ with the specified parameter.
<code>xd = integer</code>	Fit a polynomial of $x$ up to the specified power.
<code>m = integer</code>	Set number of robustness iterations.
<code>s = name</code>	Save the fitted $y$ series.
<code>p</code>	Print the scatter plot.

If the polynomial degree of  $x$  leads to singularities in the regression, EViews will automatically drop the high order terms to avoid collinearity.

## Examples

```
group g1 inf unemp
g1.linefit(y1,x1,s=yfit)
```

displays a scatter plot of log UNEMP against log INF together with the fitted values from a regression of log UNEMP on the log INF. The fitted values are saved in a series named YFIT. Note that the saved fitted values are for the original UNEMP, not the log transform.

```
g1.linefit(yb=0.5,m=10)
```

The Box-Cox transformation of UNEMP with parameter 0.5 is regressed on INF with 10 iterations of bisquare weights.

## Cross-references

See “[Scatter with Regression](#)” on page 389 of the *User’s Guide* for a discussion of scatter plot with regression fit.

See also [nnfit \(p. 355\)](#) and [kerfit \(p. 316\)](#).

<a href="#">link</a>	<a href="#">Object Declaration</a>
----------------------	------------------------------------

Create a series link object.

Declares a link object which may be used to refer to data in a series contained in a different workfile page. Links are used to create automatically updating match merges using identifier series or using dates (frequency conversion).

## Syntax

Command:        **link** link\_name

Command:        **link** link\_name.linkto(*options*) *link specification*

Follow the **link** keyword with the name to be given to the link object. If desired, you may combine the declaration with the [linkto \(p. 324\)](#) proc in order to provide a full link specification.

## Examples

```
link mylink
```

creates the link MYLINK with no link specification, while,

```
link l1.linkto(c=obs,nacat) indiv:::x @src ind1 ind2 @dest ind1
ind2
```

combines the link declaration with the link specification step.

## Cross-references

For a discussion of linking, see [Chapter 8, “Series Links”, on page 169](#) of the *User’s Guide*.

See also [linkto \(p. 324\)](#) and [unlink \(p. 491\)](#).

<b>linkto</b>	<a href="#">Link Proc</a>
---------------	---------------------------

Define the specification of a series link.

Specify the method by which the object uses data in an existing series. Links are used to perform cross-page match merging or frequency conversion.

### Syntax

Link Proc:      `link_name.linkto(options) source_page::series_name [src_id  
dest_id]`

Link Proc:      `link_name.linkto(options) source_page::series_name [@src  
src_ids @dest dest_ids]`

The most common use of `linkto` will be to define a link that employs general match merging. You should use the keyword `linkto` followed by any desired options, and then provide the name of the source series followed by the names of the source and destination IDs. If more than one identifier series is used, you must separate the source and destination IDs using the “@SRC” and “@DEST” keywords.

In the special case where you wish to link your data using date matching, you must use the special keyword “@DATE” as an ID series for a regular frequency page. If “@DATE” is not specified as either a source or destination ID, EViews will perform an exact match merge using the specified identifiers.

The other use of `linkto` will be to define a frequency conversion link between two regular frequency pages. To specify a frequency conversion link, you should use the `linkto` keyword followed by any desired options and then the name of a *numeric* source series. You must not specify ID series since a frequency conversion link uses the implicit dates associated with the regular frequency pages—if ID series are specified, the link will instead employ general match merging. Note also that if ID series are not specified, but a general match merge specific conversion option is provided (e.g., “c = med”), “@DATE @DATE” will be appended to the list of IDs and a general match merge employed.

It is worth mentioning that a frequency conversion link that uses an alpha source series will generate an evaluation error.

Note that linking by frequency conversion is the same as linking by general match merge using the source and destination IDs “@DATE @DATE” with the following exceptions:

- General match merge linking offers contraction methods not available with frequency conversion (e.g., median, variance, skewness).
- General match merge linking allows you to use samples to restrict the source observations used in evaluating the link.
- General match merge linking allows you to treat NA values in the ID series as a category to be used in matching.
- Frequency conversion linking offers expansion methods other than repeat.
- Frequency conversion linking provides options for the handling of NA values.

## Options

<code>smp1 = "sample_spec"</code>	Sample to be used when computing contractions in a link by match merge. By default, EViews will use the entire workfile sample “@ALL”.
<code>c = arg</code>	<p>Set the match merge contraction or the frequency conversion method.</p> <p>If you are linking a numeric source series by general match merge, the argument can be one of: “mean”, “med” (median), “max”, “min”, “sum”, “sumsq” (sum-of-squares), “var” (variance), “sd” (standard deviation), “skew” (skewness), “kurt” (kurtosis), “quant” (quantile, used with “quant = ” option), “obs” (number of observations), “nas” (number of NA values), “first” (first observation in group), “last” (last observation in group), “unique” (single unique group value, if present), “none” (disallow contractions).</p> <p>If linking an alpha series, only the non-summary methods “max”, “min”, “obs”, “nas”, “first”, “last”, “unique” and “none” are supported. For numeric links, the default contraction method is “c = mean”; for alpha links, the default is “c = unique”.</p> <p>If you are linking by frequency conversion, you may use this argument to specify the up- or down-conversion method using the options found in <a href="#">fetch (p. 279)</a>. The default frequency conversion methods are taken from the series defaults.</p>
<code>quant = number</code>	Quantile value to be used when contracting using the “c = quant” option (e.g, “quant = .3”).

nacat              Treat “NA” values as a category when performing link by general match merge operations.

Most of the conversion options should be self-explanatory. As for the others: “first” and “last” give the first and last non-missing observed for a given group ID; “obs” provides the number of non-missing values for a given group; “nas” reports the number of NAs in the group; “unique” will provide the value in the source series if it is the identical for all observations in the group, and will return NA otherwise; “none” will cause the link to fail if there are multiple observations in any group—this setting may be used if you wish to prohibit all contractions.

On a match merge expansion, linking by ID will repeat the values of the source for every matching value of the destination. If both the source and destination have multiple values for a given ID, EViews will first perform a contraction in the source (if not ruled out by “c = none”), and then perform the expansion by replicating the contracted value in the destination.

## Examples

### *General Match Merge Linking*

Let us start with a concrete example. Suppose our active workfile page contains observations on the 50 states of the US, and contains a series called STATE containing the unique state identifiers. We also have a workfile page called INDIV that contains data on individuals from all over the country, their incomes (INCOME), and their state of birth (BIRTHSTATE).

Now suppose that we wish to find the median income of males in our data for each possible state of birth, and then to match merge that value into our 50 observation state page.

The following commands:

```
link male_income  
male_income.linkto(c=med, smpl="if male=1") indiv::income  
                  birthstate state
```

create the series link MALE\_INCOME. MALE\_INCOME contains links to the individual INCOME data, telling EViews to subsample only observations where MALE = 1, to compute median values for individuals in each BIRTHSTATE, and to match observations by comparing the values of BIRTHSTATE to STATE in the current page.

In this next example, we link to the series X in the INDIV page, matching values of the IND1 and the IND2 series in the two workfile pages. The link will compute the number of valid observations in the X series for each index group, with NA values in the ID series treated as a valid identifier value.

---

```
link 11.linkto(c=obs,nacat) indiv::x @src ind1 ind2 @dest ind1
ind2
```

You may wish to use the “@DATE” keyword as an explicit identifier, in order to gain access to our expanded date matching feature. In our annual workfile, the command:

```
link gdp.linkto(c=sd) monthly::gdp @date @date
```

will create link that computes the standard deviation of the values of GDP for each year and then match merges these values to the years in the current page. Note that this command is equivalent to:

```
link gdp.linkto(c=sd) quarterly::gdp
```

since the presence of the match merge option “c = sd” and the absence of indices instructs EViews to perform the link by ID matching using the defaults “@DATE” and “@DATE”.

### *Frequency Conversion Linking*

Suppose that we are in an annual workfile page and wish to link data from a quarterly page. Then the commands:

```
link gdp
gdp.linkto quarterly::gdp
```

creates a series link GDP in the current page containing a link by date to the GDP series in the QUARTERLY workfile page. When evaluating the link, EViews will automatically frequency convert the quarterly GDP to the annual frequency of the current page, using the series default conversion options. If we wish to control the conversion method, we can specify the conversion method as an option:

```
gdp.linkto(c=s) quarterly::gdp
```

links to GDP in the QUARTERLY page, and will frequency convert by summing the non-missing observations.

### Cross-references

For a detailed discussion of linking, see [Chapter 8, “Series Links”, on page 169](#) of the *User’s Guide*.

See also [link \(p. 323\)](#) and [unlink \(p. 491\)](#).

load	<a href="#">Command</a>
------	-------------------------

Load a workfile.

Provided for backward compatibility. Same as [wfopen \(p. 504\)](#).

logit	Command
-------	---------

Estimate binary models with logistic errors.

Provide for backward compatibility. Equivalent to issuing the command, `binary` with the option “(d = l)”.

See [binary \(p. 217\)](#).

logl	Object Declaration
------	--------------------

Declare likelihood object.

### Syntax

Command:      `logl logl_name`

### Examples

```
logl ll1
```

declares a likelihood object named LL1.

```
ll1.append @logl logl1  
ll1.append res1 = y-c(1)-c(2)*x  
ll1.append logl1 = log(@dnorm(res1/@sqrt(c(3))))-log(c(3))/2
```

specifies the likelihood function for LL1 and estimates the parameters by maximum likelihood.

### Cross-references

See [Chapter 22, “The Log Likelihood \(LogL\) Object”, on page 655](#) of the *User’s Guide* for further examples of the use of the likelihood object.

See also [append \(p. 202\)](#) for adding specification lines to an existing likelihood object, and [m1 \(p. 352\)](#) for estimation.

ls	<a href="#">Command</a>    <a href="#">Equation Method</a>   <a href="#">Pool Method</a>   <a href="#">System Method</a>   <a href="#">Var Method</a>
----	---

Estimation by linear or nonlinear least squares regression.

When used as a pool proc or when the current workfile has a panel structure, ls also estimates cross-section weighed least squares, feasible GLS, and fixed and random effects models.

### Syntax

Command:      `ls(options) y x1 [x2 x3 ...]`

`ls(options) specification`

Equation Method: `eq_name.ls(options) y x1 [x2 x3 ...]`

`eq_name.ls(options) specification`

`paneleq_name.ls(options) y x1 [x2 x3 ...] [@cxreg z1 z2 ...]`

`[@perreg z3 z4 ...]`

`paneleq_name.ls(options) specification`

Pool Method:    `pool_name.ls(options) y x1 [x2 x3 ...] [@cxreg z1 z2 ...] [@perreg z3 z4 ...]`

System Method:   `system_name.ls(options)`

VAR Method:     `var_name.ls(options) lag_pairs endog_list [@ exog_list]`

Additional issues associated with each usage of the keyword are described in the following sections.

### Equations

For linear specifications, list the dependent variable first, followed by a list of the independent variables. Use a “C” if you wish to include a constant or intercept term; unlike some programs, EViews does not automatically include a constant in the regression. You may add AR, MA, SAR, and SMA error specifications, and PDL specifications for polynomial distributed lags. If you include lagged variables, EViews will adjust the sample automatically, if necessary.

Both dependent and independent variables may be created from existing series using standard EViews functions and transformations. EViews treats the equation as linear in each of the variables and assigns coefficients C(1), C(2), and so forth to each variable in the list.

Linear or nonlinear single equations may also be specified by explicit equation. You should specify the equation as a formula. The parameters to be estimated should be included explicitly: “C(1)”, “C(2)”, and so forth (assuming that you wish to use the default coeffi-

cient vector “C”). You may also declare an alternative coefficient vector using `coef` and use these coefficients in your expressions.

### *Pools*

When used as a pool method, `ls` carries out pooled data estimation. Type the name of the dependent variable followed by one or more lists of regressors. The first list should contain ordinary and pool series that are restricted to have the same coefficient across all members of the pool. The second list, if provided, should contain pool variables that have different coefficients for each cross-section member of the pool. If there is a cross-section specific regressor list, the two lists must be separated by “@CXREG”. The third list, if provided, should contain pool variables that have different coefficients for each period. The list should be separated from the previous lists by “@PERREG”.

For pool estimation, you may include AR terms as regressors in either the common or cross-section specific lists. AR terms are, however, not allowed for some estimation methods. MA terms are not supported.

### *VARs*

`ls` estimates an unrestricted VAR using equation-by-equation OLS. You must specify the order of the VAR (using one or more pairs of lag intervals), and then provide a list of series or groups to be used as endogenous variables. You may include exogenous variables such as trends and seasonal dummies in the VAR by including an “@-sign” followed by a list of series or groups. A constant is automatically added to the list of exogenous variables; to estimate a specification without a constant, you should use the option “noconst”.

### Options

#### *General options*

<code>m = integer</code>	Set maximum number of iterations.
<code>c = scalar</code>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<code>s</code>	Use the current coefficient values in “C” as starting values for equations with AR or MA terms (see also <a href="#">param (p. 383)</a> ).

<i>s = number</i>	Determine starting values for equations specified by list with AR or MA terms. Specify a number between zero and one representing the fraction of preliminary least squares estimates computed without AR or MA terms to be used. Note that out of range values are set to “ <i>s = 1</i> ”. Specifying “ <i>s = 0</i> ” initializes coefficients to zero. By default EViews uses “ <i>s = 1</i> ”.
<i>showopts / -showopts</i>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<i>deriv = keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one or two letter string. The first letter should either be “ <i>f</i> ” or “ <i>a</i> ” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “ <i>n</i> ” (always use numeric) or “ <i>a</i> ” (use analytic if possible). If omitted, EViews will use the global defaults.
<i>p</i>	Print basic estimation results.

#### *Additional Options for Equation, System, and Var estimation*

<i>w = series_name</i>	Weighted Least Squares. Each observation will be weighted by multiplying by the specified series.
<i>h</i>	White’s heteroskedasticity consistent standard errors.
<i>n</i>	Newey-West heteroskedasticity and autocorrelation consistent (HAC) standard errors.
<i>z</i>	Turn off backcasting in ARMA models.
<i>noconst</i>	Do not include a constant in exogenous regressors list for VARs.

*Note:* not all options are available for all equation methods. See the *User’s Guide* for details on each estimation method.

#### *Additional Options for Pool and Panel Equation estimation*

<i>cx = arg</i>	Cross-section effects: (default) none, fixed effects (“ <i>cx = f</i> ”), random effects (“ <i>cx = r</i> ”).
<i>per = arg</i>	Period effects: (default) none, fixed effects (“ <i>per = f</i> ”), random effects (“ <i>per = r</i> ”).

wgt = <i>arg</i>	GLS weighting: (default) none, cross-section system weights (“wgt = cxsur”), period system weights (“wgt = persur”), cross-section diagonal weights (“wgt = cxdiag”), period diagonal weights (“wgt = perdiag”).
cov = <i>arg</i>	Coefficient covariance method: (default) ordinary, White cross-section system robust (“cov = cxwhite”), White period system robust (“cov = perwhite”), White heteroskedasticity robust (“cov = stackedwhite”), Cross-section system robust/PCSE (“cov = cxsur”), Period system robust/PCSE (“cov = persur”), Cross-section heteroskedasticity robust/PCSE (“cov = cxdiag”), Period heteroskedasticity robust/PCSE (“cov = perdiag”).
keepwgts	Keep full set of GLS weights used in estimation with object, if applicable (by default, only small memory weights are saved).
rancalc = <i>arg</i> ( <i>default</i> = “sa”)	Random component method: Swamy-Arora (“rancalc = sa”), Wansbeek-Kapteyn (“rancalc = wk”), Wallace-Hussain (“rancalc = wh”).
nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
b	Estimate using a balanced sample (pool estimation only).
coef = <i>arg</i>	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the “C” coefficient vector.
iter = <i>arg</i> ( <i>default</i> = “onec”)	Iteration control for GLS specifications: perform one weight iteration, then iterate coefficients to convergence (“iter = onec”), iterate weights and coefficients simultaneously to convergence (“iter = sim”), iterate weights and coefficients sequentially to convergence (“iter = seq”), perform one weight iteration, then one coefficient step (“iter = oneb”).  Note that random effects models currently do not permit weight iteration to convergence.

## Examples

```
equation eq1.ls m1 c uemp inf(0 to -4) @trend(1960:1)
```

estimates a linear regression of M1 on a constant, UEMP, INF (from current up to four lags), and a linear trend.

```
equation eq2.ls(z) d(tbill) c inf @seas(1) @seas(1)*inf ma(2)
```

regresses the first difference of TBILL on a constant, INF, a seasonal dummy, and an interaction of the dummy and INF, with an MA(2) error. The “z” option turns off backcasting.

```
coef(2) beta
```

```
param beta(1) .2 beta(2) .5 c(1) 0.1
```

```
equation eq3.ls(h) q = beta(1)+beta(2)*(1^c(1) + k^(1-c(1)))
```

estimates the nonlinear regression starting from the specified initial values. The “h” option reports heteroskedasticity consistent standard errors.

```
equation eq4.ls r = c(1)+c(2)*r(-1)+div(-1)^c(3)
```

```
sym betacov = eq4.@cov
```

declares and estimates a nonlinear equation and stores the coefficient covariance matrix in a symmetric matrix called BETACOV.

```
pool1.ls dy? inv? edu? year
```

estimates pooled OLS of DY? on a constant, INV?, EDU? and YEAR.

```
pool1.ls(cx=f) dy? @cxreg inv? edu? year ar(1)
```

estimates a fixed effects model without restricting any of the coefficients to be the same across pool members.

```
group rhs c dum1 dum2 dum3 dum4
```

```
ls cons rhs ar(1)
```

uses the group definition for RHS to regress CONS on C, DUM1, DUM2, DUM3, and DUM4, with an AR(1) correction.

## Cross-references

[Chapter 15, “Basic Regression”, on page 427](#) and [Chapter 16, “Additional Regression Methods”, on page 445](#) of the *User’s Guide* discuss the various regression methods in greater depth.

See [Chapter 27, “Pooled Time Series, Cross-Section Data”, on page 809](#) of the *User’s Guide* for a discussion of pool estimation, and [Chapter 29, “Panel Estimation”, on page 885](#) of the *User’s Guide* for a discussion of panel equation estimation.

See “Equation” (p. 155), “Pool” (p. 169), “System” (p. 182), and “Var” (p. 187) for a complete description of these objects, including a list of saved results. See also [Appendix C, “Special Expression Reference”, on page 535](#), for special terms that may be used in `ls` specifications.

<b>makecoint</b>	<a href="#">Var Proc</a>
------------------	--------------------------

Create group containing the estimated cointegrating relations from a VEC.

### Syntax

Var Proc:            `var_name.makecoint [group_name]`

The series contained in the group are given names of the form “COINTEQ##”, where ## are numbers such that “COINTEQ##” is the next available unused name.

If you provide a name for the group in parentheses after the keyword, EViews will quietly create the named group in the workfile. If you do not provide a name, EViews will open an untitled group window if the command is executed from the command line, otherwise no group will be created.

This proc will return an error message unless you have estimated an error correction model using the var object.

### Examples

```
var vec1.ec(b,2) 1 4 y1 y2 y3  
vec1.makecoint gcoint
```

The first line estimates a VEC with 2 cointegrating relations. The second line creates a group named GCOINT which contains the two estimated cointegrating relations. The two cointegrating relations will be stored as series named COINTEQ01 and COINTEQ02 (if these names have not yet been used in the workfile).

### Cross-references

See [Chapter 24, “Vector Autoregression and Error Correction Models”, on page 705](#) of the *User’s Guide* for details.

See also [coint \(p. 240\)](#).

**makederivs**[Equation Proc](#)

Make a group containing individual series which hold the derivatives of the equation specification.

**Syntax**

Equation Proc:     `equation_name.makederivs(options) [ser1 ser2 ...]`

If desired, enclose the name of a new group object to hold the series in parentheses following the command name.

The argument specifying the names of the series is also optional. If not provided, EViews will name the series “DERIV##” where ## is a number such that “DERIV##” is the next available unused name. If the names are provided, the number of names must match the number of target series.

**Cross-references**

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for details on state space estimation.

See also [derivs \(p. 264\)](#), [grads \(p. 302\)](#), [makegrads \(p. 337\)](#).

**makeendog**[Sspace Proc](#) | [System Proc](#) | [Var Proc](#)

Make a group out of the endogenous series.

**Syntax**

Object Proc:     `object_name.makeendog name`

Following the keyword `makeendog`, you should provide a name for the group to hold the endogenous series. If you do not provide a name, EViews will create an untitled group.

Note that in `endog` and `makeendog` are no longer supported for model objects. See instead, [makegroup \(p. 339\)](#).

**Examples**

```
var1.makeendog grp_v1
```

creates a group named GRP\_V1 that contains the endogenous series in VAR1.

## Cross-references

See also [endog \(p. 274\)](#) and [makegroup \(p. 339\)](#).

<b>makefilter</b>	<a href="#">SSpace Proc</a>
-------------------	-----------------------------

Create a “Kalman filter” sspace object.

Creates a new sspace object with all estimated parameter values substituted out of the specification. This procedure allows you to use the structure of the sspace without reference to estimated coefficients or the estimation sample.

## Syntax

SSpace Proc:      `sspace_name.makefilter [filter_name]`

If you provide a name for the sspace object in parentheses after the keyword, EViews will quietly create the named object in the workfile. If you do not provide a name, EViews will open an untitled sspace window if the command is executed from the command line.

## Examples

```
ss1.makefilter kfilter
```

creates a new sspace object named KFILTER, containing the specification in SS1 with estimated parameter values substituted for coefficient elements.

## Cross-references

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for details on state space models.

See also [makesignals \(p. 344\)](#) and [makestates \(p. 345\)](#).

<b>makegarch</b>	<a href="#">Equation Proc</a>
------------------	-------------------------------

Generate conditional variance series.

Saves the estimated conditional variance (from an equation estimated using ARCH) as a named series.

## Syntax

Equation Proc:      `eq_name.makegarch series_name`

You should provide a name for the saved series following the `makegarch` keyword. If you do not provide a name, EViews will name the series using the next available name of the form “GARCH##” (if GARCH01 already exists, it will be named GARCH02, and so on).

## Examples

```
equation eq1.arch sp c
eq1.makegarch cvar
plot cvar^.5
```

estimates a GARCH(1,1) model, saves the conditional variance as a series named CVAR, and plots the conditional standard deviation. If you merely wish to view a plot of the conditional standard deviation without saving the series, use the [garch \(p. 295\)](#) view.

## Cross-references

See [Chapter 20, “ARCH and GARCH Estimation”, on page 585](#) of the *User’s Guide* for a discussion of GARCH models.

See also [arch \(p. 203\)](#) [archtest \(p. 206\)](#) and [garch \(p. 295\)](#).

## makegrads

[Equation Proc](#) | [Logl Proc](#) | [Sspace Proc](#)

Make a group containing individual series which hold the gradients of the objective function.

## Syntax

Object Proc:      `object_name.makegrads(options) [ser1 ser2 ...]`

The argument specifying the names of the series is also optional. If the argument is not provided, EViews will name the series “GRAD##” where ## is a number such that “GRAD##” is the next available unused name. If the names are provided, the number of names must match the number of target series.

## Options

<code>n = arg</code>	Name of group object to contain the series.
----------------------	---

## Examples

```
eq1.grads (n=out)
```

creates a group named OUT containing series named GRAD01, GRAD02, and GRAD03.

```
eq1.grads(n=out) g1 g2 g3
```

creates the same group, but names the series G1, G2 and G3.

### Cross-references

See also [derivs \(p. 264\)](#), [makederivs \(p. 335\)](#), [grads \(p. 302\)](#).

makegraph	<a href="#">Model Proc</a>
-----------	----------------------------

Make graph object showing model series.

### Syntax

Model Proc:      `model_name.makegraph(options) graph_name model_vars`

where *graph\_name* is the name of the resulting graph object, and *models\_vars* are the names of the series. The list of *model\_vars* may include the following special keywords:

@all	All model variables.
@endog	All endogenous model variables.
@exog	All exogenous model variables.
@addfactor	All add factor variables in the model.

### Options

a	Include actuals.
c	Include comparison scenarios.
d	Include deviations.
n	Do not include active scenario (by default the active scenario is included).
t = <i>trans_type</i> (default = level)	Transformation type: “level” (display levels in graph), “pch” (display percent change in graph), “pcha” (display percent change - annual rates - in graph), “pchy” (display 1-year percent change in graph), “dif” (display 1-period differences in graph), “dify” (display 1-year differences in graph).
s = <i>sol_type</i> (default = “d”)	Solution type: “d” (deterministic), “m” (mean of stochastic), “s” (mean and $\pm 2$ std. dev. of stochastic), “b” (mean and confidence bounds of stochastic).

`g = grouping`      Grouping setting for graphs: “v” (group series in graph by model variable), “s” (group series in graph by scenario), “u” (ungrouped - each series in its own graph).  
`(default = “v”)`

## Examples

```
mod1.makegraph(a) gr1 y1 y2 y3
```

creates a graph containing the model series Y1, Y2, and Y3 in the active scenario and the actual Y1, Y2, and Y3.

```
mod1.makegraph(a, t=pchy) gr1 y1 y2 y3
```

plots the same graph, but with data displayed as 1-year percent changes.

## Cross-references

See “[Displaying Data](#)” on page 802 of the *User’s Guide* for details. See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a general discussion of models.

See [makegroup \(p. 339\)](#).

<b>makegroup</b>	<a href="#">Model Proc</a>   <a href="#">Pool Proc</a>
------------------	--

Make a group out of pool and ordinary series using a pool object, or make a group out of model series and display dated data table.

## Syntax

Pool Proc:      `pool_name.makegroup(group_name) pool_series1 [pool_series2  
pool_series3 ...]`

Model Proc:      `model_name.makegroup(options) grp_name model_vars`

When used as a pool proc, you should provide a name for the new group in parentheses, then list the ordinary and pool series to be placed in the group.

When used as a model proc, the `makegroup` keyword should be followed by options, the name of the destination group, and the list of model variables to be created. The options control the choice of model series, and transformation and grouping features of the resulting dated data table view. The list of `model_vars` may include the following special keywords:

`@all`      All model variables.

`@endog`      All endogenous model variables.

`@exog`      All exogenous model variables.

@addfactor All add factor variables in the model.

## Options

### *For Model Proc*

a	Include actuals.
c	Include comparison scenarios.
d	Include deviations.
n	Do not include active scenario (by default the active scenario is included).
t = <i>arg</i> ( <i>default</i> = level)	Transformation type: “level” (display levels), “pch” (percent change), “pcha” (display percent change - annual rates), “pchy” (display 1-year percent change), “dif” (display 1-period differences), “dify” (display 1-year differences).
s = <i>arg</i> ( <i>default</i> = “d”)	Solution type: “d” (deterministic), “m” (mean of stochastic), “s” (mean and $\pm 2$ std. dev. of stochastic), “b” (mean and confidence bounds of stochastic).
g = <i>arg</i> ( <i>default</i> = “v”)	Grouping setting for graphs: “v” (group series in graph by model variable), “s” (group series in graph by scenario).

## Examples

```
pool1.makegroup(g1) x? z y?
```

places the ordinary series Z, and all of the series represented by the pool series X? and Y?, in the group G1.

```
model1.makegroup(a, n) group1 @endog
```

places all of the actual endogenous series in the group GROUP1.

## Cross-references

See “[Displaying Data](#)” on page 802 of the *User’s Guide* for details. See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a general discussion of models.

See also [makegraph \(p. 338\)](#).

**makelimits**[Equation Proc](#)

Create vector of limit points from ordered models.

`makelimits` creates a vector of the estimated limit points from equations estimated by [ordered \(p. 360\)](#).

### Syntax

Equation Proc:     `eq_name.makelimits [vector_name]`

Provide a name for the vector after the `makelimits` keyword. If you do not provide a name, EViews will name the vector with the next available name of the form LIMITS## (if LIMITS01 already exists, it will be named LIMITS02, and so on).

### Examples

```
equation eq1.ordered edu c age race gender
eq1.makelimit cutoff
```

Estimates an ordered probit and saves the estimated limit points in a vector named CUT-OFF.

### Cross-references

See “[Ordered Dependent Variable Models](#)” on page 622 of the *User’s Guide* for a discussion of ordered models.

**makemodel**

[Equation Proc](#) | [Logl Proc](#) | [Pool Proc](#) | [Sspace Proc](#) | [System Proc](#) | [Var Proc](#)

Make a model from an estimation object.

### Syntax

Object Proc:     `object_name.makemodel(name) assign_statement`

If you provide a name for the model in parentheses after the keyword, EViews will create the named model in the workfile. If you do not provide a name, EViews will open an untitled model window if the command is executed from the command line.

### Examples

```
var var3.ls 1 4 m1 gdp tb3
var3.makemodel(varmod) @prefix s_
```

estimates a VAR and makes a model named VARMOD from the estimated var object. VARMOD includes an assignment statement “`ASSIGN @PREFIX S_`”. Use the command “`show varmod`” or “`varmod.spec`” to open the VARMOD window.

### Cross-references

See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews.

See also [append \(p. 202\)](#), [merge \(p. 351\)](#) and [solve \(p. 451\)](#). For sspace objects, see also [makefilter \(p. 336\)](#).

<b>makeregs</b>	<a href="#">Equation Proc</a>
-----------------	-------------------------------

Make regressor group.

Creates a group containing the dependent and independent variables from an equation specification.

### Syntax

Equation Proc:     `equation_name.makeregs grp_name`

Follow the keyword `makeregs` with the name of the group.

### Examples

```
equation eq1.ls y c x1 x2 x3 z  
eq1.makeregs reggroup
```

creates a group REGGROUP containing the series Y X1 X2 X3 and Z.

### Cross-references

See also [group \(p. 304\)](#) and [equation \(p. 275\)](#).

<b>makeresids</b>	<a href="#">Equation Proc</a>   <a href="#">Pool Proc</a>   <a href="#">System Proc</a>   <a href="#">Var Proc</a>
-------------------	--

Create residual series.

Creates and saves residuals in the workfile from an object with an estimated equation or equations.

## Syntax

Equation Proc:    `equation_name.makeresids(options) [res1]`

Object Proc:    `object_name.makeresids[res1 res2 res3]`

Pool Proc:    `pool_name.makeresids [poolser]`

Follow the object name with a period and the `makeresids` keyword, then provide a list of names to be given to the stored residuals. For pool residuals, you may use a cross section identifier “?” to specify a set of names.

For multiple equation objects, you should provide as many names as there are equations. If there are fewer names than equations, EViews creates the extra residual series with names RESID01, RESID02, and so on. If you do not provide any names, EViews will also name the residuals RESID01, RESID02, and so on.

[NOTE: `makeresids` is no longer supported for the `sspace` object— see [makesignals \(p. 344\)](#) for replacement routines].

## Options

The following options are available only for the residuals from single equation objects:

`o (default)`    Ordinary residuals.

`s`    Standardized residuals (available only after weighted estimation and GARCH, binary, ordered, censored, and count models).

`g (default for ordered models)`    Generalized residuals (available only for binary, ordered, censored, and count models).

`n = arg`    Create group object to hold the residual series.

## Examples

```
equation eq1.ls y c m1 inf unemp
eq1.makeresids res_eq1
```

estimates a linear regression of Y on a constant, M1, INF, UNEMP, and saves the residuals as a series named RES\_EQ1.

```
var macro_var.ls 1 4 y m1 r
macro_var.makeresids resay res_m1 riser
```

estimates an unrestricted VAR with four lags and endogenous variables Y, M1, and R, and stores the residuals as RES\_Y, RES\_M1, RES\_R.

```
equation probit1.binary y c x z
```

```
probit1.makeresids(g) res_g
series score1 = res_g
series score2 = res_g*x
series score3 = res_g*z
```

estimates a probit model of Y on a constant, X, Z and stores the generalized residuals as RES\_G. Then the vector of scores are computed as SCORE1, SCORE2, SCORE3.

```
pool1.makeresids res1_?
```

The residuals of each pool member will have a name starting with “RES1\_” and the cross-section identifier substituted for the “?”.

### Cross-references

See “[Weighted Least Squares](#)” on page 453 of the *User’s Guide* for a discussion of standardized residuals after weighted least squares and [Chapter 21, “Discrete and Limited Dependent Variable Models”, on page 605](#) of the *User’s Guide* for a discussion of standardized and generalized residuals in binary, ordered, censored, and count models.

For state space models, see [makesignals \(p. 344\)](#).

<b>makesignals</b>	<a href="#">Sspace Proc</a>
--------------------	-----------------------------

Generate signal series or signal standard error series from an estimated sspace object.

Options allow you to choose to generate one-step ahead and smoothed values for the signals and the signal standard errors.

### Syntax

Sspace Proc:      name.makesignals(options) [name\_spec]

Follow the object name with a period and the makesignal keyword, options to determine the output type, and a list of names or wildcard expression identifying the series to hold the output. If a list is used to identify the targets, the number of target series must match the number of states implied in the sspace. If any signal variable contain expressions, you may not use wildcard expressions in the destination names.

## Options

**t = *output\_type*** (*default* = pred) Defines output type: one-step ahead signal predictions (“pred”), RMSE of the one-step ahead signal predictions (“predse”, “residse”), error in one-step ahead signal predictions (“resid”), standardized one-step ahead prediction residual (“stdresid”), smoothed signals (“smooth”), RMSE of the smoothed signals (“smoothse”), estimate of the disturbances (“disturb”), RMSE of the estimate of the disturbances (“disturbse”), standardized estimate of the disturbances (“stddisturb”).

**n = *group\_name*** Name of group to hold newly created series.

## Examples

```
ss1.makesignals(t=smooth) sm*
```

produces smoothed signals in the series with names beginning with “sm”, and ending with the name of the signal dependent variable.

```
ss2.makesignals(t=pred, n=pred_sigs) sig1 sig2 sig3
```

creates a group named PRED\_SIGS which contains the one-step ahead signal predictions in the series SIG1, SIG2, and SIG3.

## Cross-references

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for details on state space models. For additional discussion of wildcards, see [Appendix B, “Wildcards”, on page 921](#) of the *User’s Guide*.

See also [forecast \(p. 287\)](#), [makefilter \(p. 336\)](#) and [makestates \(p. 345\)](#).

<b>makestates</b>	<a href="#">Sspace Proc</a>
-------------------	-----------------------------

Generate state series or state standard error series from an estimated sspace object.

Options allow you to generate one-step ahead, filtered, or smoothed values for the states and the state standard errors.

## Syntax

Sspace Proc:      name.**makestates**(*options*) [*name\_spec*]

Follow the object name with a period and the `makestate` keyword, options to determine the output type, and a list of names or a wildcard expression identifying the series to hold the output. If a list is used to identify the targets, the number of target series must match the number of names implied by the keyword.

## Options

<code>t = output_type</code> <i>(default = “pred”)</i>	Defines output type: one-step ahead state predictions (“pred”), RMSE of the one-step ahead state predictions (“predse”), error in one-step ahead state predictions (“resid”), RMSE of the one-step ahead state prediction (“residse”), filtered states (“filt”), RMSE of the filtered states (“filtse”), standardized one-step ahead prediction residual (“stdresid”), smoothed states (“smooth”), RMSE of the smoothed states (“smoothse”), estimate of the disturbances (“disturb”), RMSE of the estimate of the disturbances (“disturbse”), standardized estimate of the disturbances (“stddisturb”).
<code>n = group_name</code>	Name of group to hold newly created series.

## Examples

```
ss1.makestates(t=smooth) sm*
```

produces smoothed states in the series with names beginning with “sm”, and ending with the name of the state dependent variable.

```
ss2.makestates(t=pred, n=pred_states) sig1 sig2 sig3
```

creates a group named PRED\_STATES which contains the one-step ahead state predictions in series SIG1, SIG2, and SIG3.

## Cross-references

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for details on state space models. For additional discussion of wildcards, see [Appendix B, “Wildcards”, on page 921](#) of the *User’s Guide*.

See also [forecast \(p. 287\)](#), [makefilter \(p. 336\)](#) and [makesignals \(p. 344\)](#).

**makestats****Pool Proc**

Create and save series of descriptive statistics computed from a pool object.

### Syntax

Pool Proc:        `pool_name.makestats(options) pool_series1 [pool_series2 ...] @  
stat_list`

You should provide options, a list of series names, an “@” separator, and a list of command names for the statistics you wish to compute. The series will have a name with the cross-section identifier “?” replaced by the statistic command.

### Options

*Options in parentheses specify the sample to use to compute the statistics*

- i                  Use individual sample.
- c (default)      Use common sample.
- b                  Use balanced sample.

*Command names for the statistics to be computed*

- obs               Number of observations.
- mean              Mean.
- med               Median.
- var               Variance.
- sd                Standard deviation.
- skew              Skewness.
- kurt              Kurtosis.
- jarq              Jarque-Bera test statistic.
- min               Minimum value.
- max               Maximum value.

## Examples

```
pool1.makestats gdp_? edu_? @ mean sd
```

computes the mean and standard deviation of the GDP\_? and EDU\_? series in each period (across the cross-section members) using the default common sample. The mean and standard deviation series will be named GDP\_MEAN, EDU\_MEAN, GDP\_SD, and EDU\_SD.

```
pool1.makestats(b) gdp_? @ max min
```

Computes the maximum and minimum values of the GDP\_? series in each period using the balanced sample. The max and min series will be named GDP\_MAX and GDP\_MIN.

## Cross-references

See [Chapter 27, “Pooled Time Series, Cross-Section Data”, on page 809](#) of the *User’s Guide* for details on the computation of these statistics and a discussion of the use of individual, common, and balanced samples in pool.

See also [describe \(p. 265\)](#).

<b>makesystem</b>	<a href="#">Pool Proc</a>   <a href="#">Var Proc</a>
-------------------	--

Create system from a pool object or var.

## Syntax

Pool Proc:            `pool_name.makesystem(options) pool_spec`  
Var Proc:            `var_name.makesystem(options)`

The first usage creates a system out of the pool equation specification. See [ls \(p. 329\)](#) for details on the syntax of *pool\_spec*. Note that period specific coefficients and effects are not available in this routine.

The second form is used to create a system out of the current var specification. You may order the equations by series (*default*) or by lags.

## Options

### *Pool Options*

*name*              Specify name for the object.

### *VAR Options*

*bylag*              Specify system by lags (default is to order by variables).

*n = name*           Specify name for the object.

## Examples

```
pool1.makesystem(sys1) inv? cap? @ val?
```

creates a system named SYS1 with INV? as the dependent variable and a common intercept for each cross-section member. The regressor CAP? is restricted to have the same coefficient in each equation, while the VAL? regressor has a different coefficient in each equation.

```
pool1.makesystem(sys2,cx=f) inv? @cxreg cap? @cxinst @trend  
inv?(-1)
```

This command creates a system named SYS2 with INV? as the dependent variable and a different intercept for each cross-section member equation. The regressor CAP? enters each equation with a different coefficient and each equation has two instrumental variables @TREND and INV? lagged.

## Cross-references

See [Chapter 23, “System Estimation”, on page 679](#) of the *User’s Guide* for a discussion of system objects in EViews.

map	<a href="#">Alpha Proc</a>   <a href="#">Series Proc</a>
-----	--

Assign or remove value map setting.

## Syntax

Object Proc:      series\_name.map [*valmap\_name*]

If the optional valmap name is provided, the procedure will assign the specified value map to the series or alpha. If no name is provided, EViews will remove an existing valmap assignment.

## Examples

```
alpha1.map mymap
```

assigns the valmap object MYMAP to the alpha series ALPHA1.

```
series1.map
```

removes an existing valmap assignment from SERIES1.

## Cross-references

See [“Value Maps” on page 155](#) of the *User’s Guide* for a discussion of valmap objects in EViews.

**matrix**[Object Declaration](#)

Declare and optionally initializes a matrix object.

### Syntax

Command:      `matrix(r, c) matrix_name[= assignment]`

The `matrix` keyword is followed by the name you wish to give the matrix. `matrix` also takes an optional argument specifying the row *r* and column *c* dimension of the matrix. Once declared, matrices may be resized by repeating the `matrix` command using the original name.

You may combine matrix declaration and assignment. If there is no assignment statement, the matrix will initially be filled with zeros.

You should use `sym` for symmetric matrices.

### Examples

```
matrix mom
```

declares a matrix named MOM with one element, initialized to zero.

```
matrix(3,6) coefs
```

declares a 3 by 6 matrix named COEFS, filled with zeros.

### Cross-references

See [Chapter 3, “Matrix Language”, beginning on page 23](#) of the *Command and Programming Reference* for further discussion.

See [“Matrix” \(p. 166\)](#), [“Rowvector” \(p. 172\)](#) and [“Vector” \(p. 190\)](#) and [“Sym” \(p. 180\)](#) for full descriptions of the various matrix objects.

**means**[Equation View](#)

Descriptive statistics by category of dependent variable.

Computes and displays descriptive statistics of the explanatory variables (regressors) of an equation categorized by values of the dependent variable. `means` is currently available only for equations estimated by `binary`.

## Syntax

Equation View: `eq_name.means(options)`

## Options

<code>p</code>	Print the descriptive statistics table.
----------------	---

## Examples

```
equation eq1.binary(d=1) work c edu faminc
eq1.means
```

estimates a logit and displays the descriptive statistics of the regressors C, EDU, FAMINC for WORK = 0 and WORK = 1.

## Cross-references

See [Chapter 21, “Discrete and Limited Dependent Variable Models”, on page 605](#) of the *User’s Guide* for a discussion of binary dependent variable models.

<code>merge</code>	<a href="#">Graph Proc</a>   <a href="#">Model Proc</a>
--------------------	---

Merge objects.

When used as a model procedure, merges equations from an estimated equation, model, pool, system, or var object. When used as a graph procedure, merges graph objects.

If you supply only the object’s name, EViews first searches the current workfile for the object containing the equation. If the object is not found, EViews looks in the default directory for an equation or pool file (.DBE). If you want to merge the equations from a system file (.DBS), a var file (.DBV), or a model file (.DBL), include the extension in the command and an optional path when merging files. You must merge objects to a model one at a time; `merge` appends the object to the equations already existing in the model.

When used as a graph procedure, `merge` combines graph objects into a single graph object. The graph objects to merge must exist in the current workfile.

## Syntax

Model Proc: `model_name.merge(options) object_name`

Graph Proc: `graph name.merge graph1 graph2 [graph3 ...]`

For use as a model procedure, follow the keyword with a name of an object containing estimated equation(s) to merge.

For use as a graph procedure, follow the keyword with a list of existing graph object names to merge.

## Options

t	Merge an ASCII text file (only for model merge).
---	--

## Examples

```
eq1.makemodel(mod1)
mod1.merge eq2
mod1.merge(t) c:\data\test.txt
```

The first line makes a model named MOD1 from EQ1. The second line merges (appends) EQ2 to MOD1 and the third line further merges (appends) the text file TEST from the specified directory.

```
graph mygra.merge gra1 gra2 gra3 gra4
show mygra.align(4,1,1)
```

The first line merges the four graphs GRA1, GRA2, GRA3, GRA4 into a graph named MYGRA. The second line displays the four graphs in MYGRA in a single row.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graphs.

<b>metafile</b>	<a href="#">Graph Proc</a>
-----------------	----------------------------

Save graph to disk as an enhanced or ordinary Windows metafile.

Provided for backward compatibility, `metafile` has been replaced by the more general graph proc [save \(p. 407\)](#), which allows for saving graphs in metafile or postscript files, with additional options for controlling the output.

<b>ml</b>	<a href="#">Logl Method</a>   <a href="#">Sspace Method</a>
-----------	---

Maximum likelihood estimation of logl and state space models.

## Syntax

Object Method:    `object_name.ml(options)`

## Options

b	Use Berndt-Hall-Hausman (BHHH) algorithm (default is Marquardt).
m = <i>integer</i>	Set maximum number of iterations.
c = <i>scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
p	Print basic estimation results.

## Examples

bvar.ml

estimates the sspace object BVAR by maximum likelihood.

## Cross-references

See [Chapter 22, “The Log Likelihood \(LogL\) Object”, on page 655](#) and [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for a discussion of user specified likelihood and state space models.

model	<a href="#">Object Declaration</a>
-------	------------------------------------

Declare a model object.

## Syntax

Command:      **model** *model\_name*

The keyword **model** should be followed by a name for the model. To fill the model, you may use [append \(p. 202\)](#) or [merge \(p. 351\)](#).

## Examples

```
model macro
macro.append cs = 10+0.8*y(-1)
macro.append i = 0.7*(y(-1)-y(-2))
macro.append y = cs+i+g
```

declares an empty model named MACRO and adds three lines to MACRO.

## Cross-references

See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews.

See also [append \(p. 202\)](#), [merge \(p. 351\)](#) and [solve \(p. 451\)](#).

<b>msg</b>	<a href="#">Model View</a>
------------	----------------------------

Display model solution messages.

Show view containing messages generated by the most recent model solution.

### Syntax

Model View:      `model_name.msg(options)`

### Options

<code>p</code>	Print the model solution messages.
----------------	------------------------------------

## Cross-references

See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews.

See also [solve \(p. 451\)](#) and [solveopt \(p. 452\)](#).

<b>name</b>	<a href="#">Graph Proc</a>
-------------	----------------------------

Change the names used for legends or axis labels in XY graphs.

Allows you to provide an alternative to the names used for legends or for axis labels in XY graphs. The name command is available only for single graphs and will be ignored in multiple graphs.

### Syntax

Graph Proc:      `graph_name.name(n) legend_text`

Provide a series number in parentheses and *legend\_text* for the legend (or axis label) after the keyword. If you do not provide text, the current legend will be removed from the legend/axis label.

## Examples

```
graph g1.line(d) unemp gdp
g1.name(1) Civilian unemployment rate
g1.name(2) Gross National Product
```

The first line creates a line graph named G1 with dual scale, no crossing. The second line replaces the legend of the first series UNEMP, and the third line replaces the legend of the second series GDP.

```
graph g2.scat id w h
g2.name(1)
g2.name(2) weight
g2.name(3) height
g2.legend(1)
```

The first line creates a scatter diagram named G2. The second line removes the legend of the horizontal axis, and the third and fourth lines replace the legends of the variables on the vertical axis. The last line moves the legend to the left side of the graph.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of working with graphs.

See also [displayname \(p. 266\)](#).

<b>nnfit</b>	<a href="#">Group View</a>
--------------	----------------------------

Nearest neighbor fit.

Displays the fit of the second series (vertical axis) on the first series (horizontal axis) in the group.

## Syntax

Group View:      `group_name.nnfit(options)`

## Options

<code>b = fraction</code> <i>(default = 0.3)</i>	Bandwidth as a fraction of the total sample. The larger the fraction, the smoother the fit.
<code>d = integer</code> <i>(default = 1)</i>	Degree of polynomial to fit.

b	Bracket bandwidth span.
x	Exact (full) sampling. Default is Cleveland subsampling.
integer (default = 100)	Approximate number of data points at which to compute the fit (if performing Cleveland subsampling).
u	No local weighting. Default is local weighting using tricube weights.
m = <i>integer</i>	Set number of robustness iterations.
s	Symmetric neighbors. Default is nearest neighbors.
s = <i>name</i>	Save fitted series.
p	Print the view.

## Examples

```
group gr1 gdp90 gdp50
gr1.nnfit(x,m=3)
```

displays the nearest neighbor fit of GDP50 on GDP90 with exact (full) sampling and 3 robustness iterations. Each local regression fits the default linear regression, with tricube weighting and a bandwidth of span 0.3.

```
group gro1 weight height
gro1.nnfit(50,d=2,m=3)
```

displays the nearest neighbor fit of HEIGHT on WEIGHT by fitting approximately 50 data points. Each local regression fits a quadratic, using tricube robustness weights with 3 robustness iterations.

## Cross-references

See “[Scatter with Nearest Neighbor Fit](#)” on page 390 of the *User’s Guide* for a discussion of nearest neighbor regressions.

See also [linefit \(p. 322\)](#) and [kerfit \(p. 316\)](#).

open	<a href="#">Command</a>
------	-------------------------

Opens a program file, or text (ASCII) file.

This command should be used to open program files or text (ASCII) files for editing.

You may also use the command to open workfiles or databases. This use of the `open` command for this purposes is provided for backward compatibility. We recommend instead that you use the new commands [wfopen \(p. 504\)](#) and [pageload \(p. 371\)](#) to open a workfile, and [dbopen \(p. 257\)](#) to open databases.

## Syntax

Command:      `open(options) [path\]file_name`

You should provide the name of the file to be opened including the extension (and optionally a path), or the file name without an extension but with an option identifying its type. Specified types always take precedence over automatic type identification. If a path is not provided, EViews will look for the file in the default directory.

Files with the “.PRG” extension will be opened as program files, unless otherwise specified. Files with the “.TXT” extension will be opened as text files, unless otherwise specified.

For backward compatibility, files with extensions that are recognized as database files are opened as EViews databases, unless an explicit type is specified. Similarly, files with the extensions “.WF” and “.WF1”, and foreign files with recognized extensions will be opened as workfiles, unless otherwise specified.

All other files will be read as text files.

## Options

<code>p</code>	Open file as program file.
<code>t</code>	Open file as text file.
<code>type = arg</code> (“prg” or “txt”)	Specify text or program file type using keywords.

## Examples

```
open finfile.txt
```

opens a text file named “FINFILE.TXT” in the default directory.

```
open "c:\program files\my files\test1.prg"
```

opens a program file named “TEST1.PRG” from the specified directory.

```
open a:\mymemo.tex
```

opens a text file named “MYMEMO.TEX” from the A: drive.

## Cross-references

See [wfopen \(p. 504\)](#) and [pageload \(p. 371\)](#) for opening files as workfiles or workfile pages, and [dbopen \(p. 257\)](#) for opening database files.

options	<a href="#">Graph Proc</a>
---------	----------------------------

**Set options for a graph object.**

Allows you to change the option settings of an existing graph object. When `options` is used with a multiple graph, the options are applied to all graphs.

### Syntax

Graph Proc:      `graph_name.options option_list`

Note: the syntax of the `options` proc has changed considerably from version 3.1 of EViews. While not documented here, the EViews 3 options are still (for the most part) supported. However, we do not recommend using the old options, as future support is not guaranteed.

### Options

<code>size(w, h)</code>	Specifies the size of the plotting frame in virtual inches ( $w$ = width, $h$ = height).
<code>inbox / -inbox</code>	[Enclose / Do not enclose] the plotting frame in a box and [include / remove] the yellow background.
<code>indent / -indent</code>	[Indent / Do not indent] lines/bars/areas/spikes in the plotting frame.
<code>lineauto</code>	Use solid lines when drawing in color and use patterns and grayscale when drawing in black and white.
<code>linesolid</code>	Always use solid lines.
<code>linepat</code>	Always use line patterns.
<code>color / -color</code>	Specifies that lines/filled areas [use / do not use] color. Note that if the “ <code>lineauto</code> ” option option is specified, this choice will also influence the type of line or filled area drawn on screen: if color is specified, solid colored lines and filled areas will be drawn; if color is turned off, lines will be drawn using black and white line patterns, and gray scales will be used for filled areas.

---

barlabelabove / -barlabelabove	[Place / Do not place] text value of data above bar in bar graph.
barlabelinside / -barlabelinside	[Place / Do not place] text value of data inside bar in bar graph.
outlinebars / -outlinebars	[Outline / Do not outline] bars in a bar graph.
outlinearea / -outlinearea	[Outline / Do not outline] areas in an area graph.
barspace / -barspace	[Put / Do not put] space between bars in bar graph.
pielabel / -pielabel	[Place / Do not place] text value of data in pie chart.

The options which support the “-” may be proceeded by a “+” or “-” indicating whether to turn on or off the option. The “+” is optional.

Data labels in bar and pie graphs will only be visible when there is sufficient space in the graph.

## Examples

```
graph1.option size(4,4) +inbox color
```

sets GRAPH1 to use a  $4 \times 4$  frame enclosed in a box. The graph will use color.

```
graph1.option linepat -color size(2,8) -inbox
```

sets GRAPH1 to use a  $2 \times 8$  frame with no box. The graph does not use color, with the lines instead being displayed using patterns.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graph options in EViews.

See also [axis \(p. 213\)](#), [datelabel \(p. 252\)](#), [scale \(p. 410\)](#) and [setelem \(p. 426\)](#).

**ordered**[Command](#) || [Equation Method](#)

Estimation of ordered dependent variable models.

### Syntax

Command:      **ordered**(*options*) *y x1 [x2 x3 ...]*

Equation Method: *equation name.ordered*(*options*) *y x1 [x2 x3 ...]*

When used as an equation procedure, **ordered** estimates the model and saves the results as an equation object with the given name.

### Options

<b>d = arg</b> ( <i>default</i> = "n")	Specify likelihood: normal likelihood function, ordered probit ("n"), logistic likelihood function, ordered logit ("l"), Type I extreme value likelihood function, ordered Gompit ("x").
<b>q</b> ( <i>default</i> )	Use quadratic hill climbing as the maximization algorithm.
<b>r</b>	Use Newton-Raphson as the maximization algorithm.
<b>b</b>	Use Berndt-Hall-Hausman as maximization algorithm.
<b>h</b>	Quasi-maximum likelihood (QML) standard errors.
<b>g</b>	GLM standard errors.
<b>m = integer</b>	Set maximum number of iterations.
<b>c = scalar</b>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<b>s</b>	Use the current coefficient values in C as starting values.
<b>s = number</b>	Specify a number between zero and one to determine starting values as a fraction of preliminary EViews default values (out of range values are set to "s = 1").
<b>showopts / -showopts</b>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.

`deriv = keyword` Set derivative methods. The argument *keyword* should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.

p	Print results.
---	----------------

If you choose to employ user specified starting values, the parameters corresponding to the limit points must be in ascending order.

## Examples

```
ordered(d=1,h) y c wage edu kids
```

estimates an ordered logit model of Y on a constant, WAGE, EDU, and KIDS with QML standard errors. This command uses the default quadratic hill climbing algorithm.

```
param c(1) .1 c(2) .2 c(3) .3 c(4) .4 c(5) .5
equation eq1.binary(s) y c x z
coef betahat = eq1.@coefs
eq1.makelimit gamma
```

estimates an ordered probit model of Y on a constant, X, and Z from the specified starting values. The estimated coefficients are then stored in the coefficient vector BETAHAT, and the estimated limit points are stored in the vector GAMMA.

## Cross-references

See “[Ordered Dependent Variable Models](#)” on page 622 of the *User’s Guide* for additional discussion.

See [binary \(p. 217\)](#) for the estimation of binary dependent variable models. See also [makelimits \(p. 341\)](#).

<b>output</b>	<a href="#">Command</a>    <a href="#">Equation View</a>   <a href="#">Logl View</a>   <a href="#">Pool View</a>   <a href="#">Sspace View</a>   <a href="#">System View</a>   <a href="#">Var View</a>
---------------	--

Redirect printer output or display estimation output.

When used as a command, `output` redirects printer output. You may specify that any procedure that would normally send output to the printer puts output in a text file, in a Rich Text Format (RTF) file, or into frozen table or graph objects in the current workfile.

When used as a view of an estimation object, `output` changes the default object view to display the estimation output (equivalent to using [results \(p. 400\)](#)).

## Syntax

Command:	<code>output[(f)] base_name</code>
Command:	<code>output(options) [path\]file_name</code>
Command:	<code>output off</code>
Object View:	<code>object_name.output</code>

By default, the `output` command redirects the output into frozen objects. You should supply a base name after the `output` keyword. Each subsequent print command will create a new table or graph object in the current workfile, using the base name and an identifying number. For example, if you supply the base name of “OUT”, the first print command will generate a table or graph named OUT01, the second print command will generate OUT02, and so on.

You can also use the optional settings, described below, to redirect table and text output to a text file or all output to an RTF file. If you elect to redirect output to a file, you must specify a filename.

When followed by the optional keyword `off`, the `output` command turns off output redirection. Subsequent print commands will be directed to the printer.

## Options

### *Options for output command*

f	Redirect all output to frozen objects in the default workfile, using <i>base_name</i> .
t	Redirect table and text output to a text file. Graphic output will still be sent to the printer.
r	Redirect all output to an Rich Text Format (RTF) file.
o	Overwrite file if necessary. If the specified filename for text or RTF output exists, overwrite the file. The default is to append to the file. Only applicable for RTF and text file output (specified using options “t” or “r”).
c	Command logging. Output both the output, and the command used to generate the output. Only applicable for RTF and text file output (specified using options “t” or “r”).

### Options for output view

p	Print estimation output for estimation object
---	---

### Examples

```
output print_
```

causes the first print command to generate a table or graph object named PRINT\_01, the second print command to generate an object named PRINT\_02, and so on.

```
output(t) c:\data\results
equation eq1.ls(p) log(gdp) c log(k) log(l)
eq1.resids(g,p)
output off
```

The second line redirects printing to the RESULTS.TXT file, while the print option of the third line sends the graph output to the printer. The last line turns output redirection off and restores normal printer use.

If instead, the first line read:

```
output(r) c:\data\results
```

all subsequent output would be sent to the RTF file RESULTS.RTF.

The `output` keyword may also be used to change the default view of an estimation object. Entering the command:

```
eq1.output
```

displays the estimation output for equation EQ1.

### Cross-references

See “[Print Setup](#)” beginning on page 919 of the *User’s Guide* for further discussion.

See also [pon](#) (p. 608), [poff](#) (p. 608).

<b>override</b>	<a href="#">Model Proc</a>
-----------------	----------------------------

Specifies (or merges) overridden exogenous variables and add factors in the active scenario.

### Syntax

Model Proc:      `model_name.override(options) ser1 [ser2 ser3 ...]`

Follow the keyword with the argument list containing the exogenous variables or add factors you wish to override.

## Options

m	Merge into (instead of replace) the existing override list.
---	---

## Examples

```
mod1.override fed1 add1
```

creates an override list containing the variables FED1 and ADD1.

If you then issue the command:

```
mod1.override fed1
```

EViews will replace the original exclude list with one containing only FED1. To add overrides to an existing list, use the “m” option:

```
mod1.override (m) add1
```

The override list now contains both series.

## Cross-references

See the discussion in “[Specifying Scenarios](#)” on page 784 of the *User’s Guide*. See also [Chapter 26, “Models”,](#) on page 761 of the *User’s Guide* for a general discussion of models.

See also [model \(p. 353\)](#), [exclude \(p. 278\)](#) and [solveopt \(p. 452\)](#).

pageappend	Command
------------	---------

Append contents of the specified workfile page to the active workfile page.

## Syntax

Command:      **pageappend**(*options*) *wfname*\pgname [object\_list]

where *wfname* is the name of a workfile that is currently in memory. You may optionally provide the name of a page in *wfname* that you wish to use as a source, and the list of objects to be read from the source workfile page. If no *wfname* is provided, EViews will use the default page in the source workfile.

The command appends the contents of the source page to the active page in the default workfile. The target page is first unstructured (if necessary) and its range is expanded to encompass the combined range of the sample from the source workfile page, and the destination page.

The default behavior is to append all series and alpha objects (but not other types) from the source page, but the optional *object\_list* may be provided to specify specific series, or to specify objects other than series or alpha objects to be included. Command options may also be used to modify the list of objects to be included.

Note that since this operation is performed in place, the original workfile page cannot be recovered. We recommend that you consider backing up your original page using [page-copy \(p. 367\)](#).

## Options

<code>smpl = smpl_spec</code>	Specifies an optional sample identifying which observations from the source page are to be appended. The default is “@all”.
<code>allobj</code>	Specifies that all objects (including non-series and non-alpha objects) should be appended. For objects other than series and alphas, appending involves simply copying the objects from the source page to the destination page. This option may not be used with an explicit <i>object_list</i> specification.
<code>match</code>	Specifies that only series and alphas in the append page that match series and alphas of the same name in the active page should be appended. This option may not be used with “allobj” or with an explicit <i>object_list</i> specification.
<code>sufix = suffix_arg (default = “_a”)</code>	Specifies a string to be added to the end of the source object name, if necessary, to avoid name collision when creating a new object in the target page.
<code>obsid = arg</code>	Provides the name of a series used to hold the date or observation ID of each observation in the destination workfile.
<code>wfid = arg</code>	Provides the name of a (boolean) series to hold an indicator of the source for each observation in the destination workfile (0, if from the destination; 1, if from the source).

## Examples

```
pageappend updates
```

appends, to the default workfile page, all of observations in all of the series in the active page of the workfile UPDATES.

```
pageappend(match, smpl="1999 2003") updates
```

restricts the series to those which match (by name) those in the default workfile page, and restricts the observations to merge to those between 1999 and 2003.

```
pageappend newdat\page1 income cons
```

takes only the INCOME and CONS series from the PAGE1 of the NEWDATA workfile, and appends them to the current workfile page.

```
pageappend(alltypes, suffix="_1") mydata
```

appends all objects from MYDATA, merging series with matching names, and renaming other matching objects by appending “\_1” to the end of the name.

### Cross-references

See “[Appending to a Workfile](#)” on page 226 of the *User’s Guide* for discussion.

See also [pagecopy \(p. 367\)](#).

pagecontract	Command
--------------	---------

Contract the active workfile page according to the specified sample.

### Syntax

Command:      **pagecontract** *smpl\_spec*

where *smpl\_spec* is a sample specification. Contraction removes observations not in specified sample from the active workfile page. Note that since this operation is performed in place, you may wish to backup your original page (see [pagecopy \(p. 367\)](#)) prior to contracting.

### Examples

```
pagecontract if income<50000 and race=2
```

removes all observations with INCOME values less than or equal to 50000 and RACE not equal to 2.

```
pagecontract 1920 1940 1945 2000
```

removes observations for the years 1941 to 1944.

### Cross-references

See “[Contracting a Workfile](#)” on page 229 of the *User’s Guide* for discussion.

See also [pagecopy \(p. 367\)](#).

<b>pagecopy</b>	<a href="#">Command</a>
-----------------	-------------------------

Copies all or part of the active workfile page to a new workfile, or to a new page within the default workfile.

### Syntax

Command: **pagecopy**(*options*) [*object\_list*]

where the optional *object\_list* specifies the workfile objects to be copied. If *object\_list* is not provided, all objects will be copied subject to the option restrictions discussed below.

If copying objects to a new page within the default workfile, you may choose to copy series objects (series, alphas, and links) by value or by link (by employing the “bylink” option). If you elect to copy by value, links in the source page will converted to ordinary series and alpha objects when they are copied. If you copy by link, series and alpha objects in the source page are converted to links when copied. The default is to copy by value.

If you copy objects to a new workfile, data objects must be copied by value.

### Options

<b>bylink</b>	Specifies that series and alpha objects be copied as links to the source page. This option is not available if you use the “wf = ” option, since linking requires that the destination page be in the same workfile as the source page. Automatically sets the “dataonly” option so that only series, alphas, links, and valmaps will be copied.
<b>smpl = <i>smpl_spec</i></b>	Specifies an optional sample identifying which observations from the source page are to be copied. The default is “@all”.
<b>rndobs = <i>integer</i></b>	Copy only a random subsample of <i>integer</i> observations from the specified sample. Not available with “bylink” or “rndpct”.
<b>rnpct = <i>arg</i></b>	Copy only a random subsample of <i>arg</i> (a number between 0 and 1) of the specified sample. Not available with “bylink” or “rndobs”.

dataonly	Only series, alphas, links, and valmaps should be copied. The default is to copy all objects (unless the “bylink” option is specified, in which case only series objects are copied).
nolinks	Do not copy links from the source page.
wf = <i>wf_name</i>	Optional name for the destination workfile. If not provided, EViews will create a new untitled workfile. Not available if copying using the “bylink” option.
page = <i>page_name</i>	Optional name for the newly created page. If not provided, EViews will use the next available name of the form given by the page structure (e.g., “Undated”, “Annual”, “Daily5”).

## Examples

```
pagecopy (page=allvalue, wf=newwf)
```

will first create a new workfile named NEWWF, with a page ALLVALUE that has the same structure as the current page. Next, all of the objects in the active workfile page will be copied into the new page, with the series objects copied by value. In contrast,

```
pagecopy (bylink, page=alllink)
```

will instead create a page ALLLINK in the existing workfile, and will copy all series objects by creating links in the new page.

```
pagecopy (page=partcopy, bylink, smpl="1950 2000 if gender="male"" a* *z
```

will create a new page named PARTCOPY in the existing workfile containing the specified subsample of observations, and will copy all series objects in the current page beginning with the letter “A” or ending with the letter “Z”. The objects will be copied by creating links in the new page.

```
pagecopy (page=rndcopy, smpl="1950 2000 if gender="male"",
rndobs=200, dataonly, nolinks)
```

creates a new workfile and page RNDCOPY containing a 200 observation random sample from the specified subsample. Series and alpha objects only will be copied by value from the source page.

## Cross-references

See “[Copying from a Workfile](#)” on page 230 of the *User’s Guide* for discussion.

See also [pageappend \(p. 364\)](#).

<b>pagecreate</b>	<a href="#">Command</a>
-------------------	-------------------------

Create a new page in the default workfile. The new page becomes the active page.

### Syntax

- Command: `pagecreate(options) freq start_date end_date [num_cross_sections]`
- Command: `pagecreate(options) u num_observations`
- Command: `pagecreate(byid[, options]) identifier_list`

The first two forms of this command are the same as [wfcreate \(p. 503\)](#) except that instead of creating a new workfile, we create a new page in the default workfile.

The first form of the command may be used to create a regular frequency page with the specified frequency, and start and end date. If you include the optional argument *num\_cross\_sections*, EViews will create a balanced panel page using integer identifiers for each of the cross-sections. Note that more complex panel structures may be created using [pagestruct \(p. 378\)](#).

The second form of the command creates an unstructured workfile with the specified number of observations.

The last form of the command, which uses IDs, creates a new page from the active workfile page using the unique values of the specified identifier series in the specified sample. The *identifier\_list* should include all desired ID series.

If you wish to use ID series as date IDs to create a dated page, you should enclose the series (or the date ID component series) inside parentheses with a special “@DATE” keyword identifier. If you wish to use the existing date ID of the source workfile page, you may use “@DATE” without arguments.

### Options

<code>byid</code>	Create new page using the identifier list and optional sample in the default workfile page.
<code>smpl = smpl_spec</code>	Specifies an optional sample identifying which observations to use when creating a page using the “byid” option. The default is “@all”.

`page = page_name` Optional name for the newly created page. If not provided, EViews will use the next available name of the form given by the page structure (e.g., “Undated”, “Annual”, “Daily5”).

## Examples

The two commands:

```
pagecreate(page="annual") a 1950 2005  
pagecreate(page="unstruct") u 1000
```

create new pages in the existing workfile. The first page is an annual page named ANNUAL, containing data from 1950 to 2005; the second is a 1000 observation unstructured page named UNSTRUCT.

```
pagecreate(page="mypanel") a 1935 1954 10
```

creates a new workfile page named MYPANEL, containing a 10 cross-section annual panel for the years 1935 to 1954.

```
pagecreate(byid, page="stateind") state industry
```

creates a new page named STATEIND, using the distinct STATE/INDUSTRY values in the active page.

```
pagecreate(byid, page="stateyear") state @date(year)  
pagecreate(byid, page="statemonth") @date(year, month)
```

use the YEAR, and the YEAR and MONTH series, respectively, to form date identifiers that will be used in creating the new dated workfile pages.

```
pagecreate(byid, smpl="if sex=1) crossid @date
```

creates a new page using the CROSSID and existing date ID values of the active workfile page. Note that only observations in the subsample defined by “@all if sex = 1” are used to determine the unique values.

## Cross-references

See “[Creating a Workfile Page](#)” on page 58 of the *User’s Guide* for discussion.

See also [wfcreate \(p. 503\)](#) and [pagedelete \(p. 371\)](#).

---

<b>pagedelete</b>	<a href="#">Command</a>
-------------------	-------------------------

Delete the named page from the default workfile.

### Syntax

Command:      **pagedelete** *pgname*

where *pgname* is the name of a page in the default workfile.

### Examples

```
pagedelete page1
```

### Cross-references

See also [pagecreate](#) (p. 369).

---

<b>pageload</b>	<a href="#">Command</a>
-----------------	-------------------------

Load one or more new pages in the default workfile.

### Syntax

Command:      **pageload** [*path\|workfile\_name*]

Command:      **pageload**(*options*) *source\_description* [**@keep** *keep\_list*] [**@drop** *drop\_list*] [**@keepmap** *keepmap\_list*] [**@dropmap** *dropmap\_list*] [**@selectif** *condition*]

Command:      **pageload**(*options*)*source\_description* *table\_description* [**@keep** *keep\_list*] [**@drop** *drop\_list*] [**@keepmap** *keepmap\_list*] [**@dropmap** *dropmap\_list*] [**@selectif** *condition*]

The basic syntax for pageload follows that of [wfopen](#) (p. 504). The difference between the two commands is that pageload creates a new page in the default workfile, rather than opening or creating a new workfile. If a page is loaded with a name that matches an existing page, EViews will rename the new page to the next available name (e.g., “INDIVID” will be renamed “INDIVID1”).

## Options

`type = arg / t = arg` Optional type specification: Access database file (“access”), Aremon-TSD file (“a”, “aremon”, “tsd”), Binary file (“binary”), Excel file (“excel”), Gauss dataset file (“gauss”), GiveWin/PcGive file (“g”, “give”), HTML file/page (“html”), ODBC database (“odbc”), ODBC Dsn file (“dsn”), ODBC query file (“msquery”), MicroTSP workfile (“dos” “microtsp”), MicroTSP Macintosh workfile (“mac”), Rats file (“r”, “rats”), Rats portable/Troll file (“l”, “trl”), SAS program file (“sasprog”), SAS transport file (“sasxport”), SPSS file (“spss”), SPSS portable file (“spssport”), Stata file (“stata”), Text file (“text”), TSP portable file (“t”, “tsp”).

## Examples

```
pageload "c:\my documents\data\panel1"
```

loads the workfile PANEL1.WF1 from the specified directory. All of the pages in the workfile will be loaded as new pages into the current workfile.

```
pageload f.wf1
```

loads all of the pages in the workfile F.WF1 located in the default directory.

See the extensive set of examples in [wfopen \(p. 504\)](#).

## Cross-references

See “[Creating a Page by Loading a Workfile or Data Source](#)” on page 60 of the *User’s Guide* for discussion.

See also [wfopen \(p. 504\)](#) and [pagecreate \(p. 369\)](#).

<b>pagerename</b>	<b>Command</b>
-------------------	----------------

Load the specified workfile as one or more new pages in the default workfile.

## Syntax

Command:      **pagerename** *old\_name new\_name*

renames the *old\_name* page in the default workfile to *new\_name*. Page names are case-insensitive for purposes of comparison, even though they are displayed using the input case.

## Examples

```
pagerename Page1 StateByYear
```

## Cross-references

See also [pagecreate \(p. 369\)](#).

pagesave	Command
----------	---------

Save the active page in the default workfile as an EViews workfile (.wf1 file) or as a foreign data source.

## Syntax

Command:	<code>pagesave(options) [path\]filename</code>
Command:	<code>pagesave(options) source_description [@keep keep_list] [@drop drop_list] [@keepmap keepmap_list] [@dropmap dropmap_list] [@smpl smpl_spec]</code>
Command:	<code>pagesave(options) source_description table_description [@keep keep_list] [@drop drop_list] [@keepmap keepmap_list] [@dropmap dropmap_list] [@smpl smpl_spec]</code>

The command saves the active page in the specified directory using *filename*. By default, the page is saved as an EViews workfile, but options may be used to save all or part of the page in a foreign file or data source. See [wfopen \(p. 504\)](#) for details on the syntax of *source\_descriptions* and *table\_descriptions*.

## Options

type = <i>arg</i> , t = <i>arg</i>	Optional type specification. Access database file (“access”), Aremon-TSD file (“a”, “aremon”, “tsd”), Binary file (“binary”), EViews database file (“e”, “evdb”), Excel file (“excel”), Gauss dataset file (“gauss”), GiveWin/PcGive file (“g”, “give”), HTML file/page (“html”), ODBC database (“odbc”), ODBC Dsn file (“dsn”), ODBC query file (“msquery”), MicroTSP workfile (“dos”, “microtsp”), MicroTSP Macintosh workfile (“mac”), Rats file (“r”, “rats”), Rats portable/Troll file (“l”, “trl”), SAS transport file (“sasxport”), SPSS file (“spss”), SPSS portable file (“spss-port”), Stata file (“stata”), Text file (“text”), TSP portable file (“t”, “tsp”).
mode = create	Create new file only; error on attempt to overwrite.
maptype = <i>arg</i>	Write selected maps as: numeric (“n”), character (“c”), both numeric and character (“b”).
mapval	Write mapped values for series with attached value labels.

## Examples

```
pagesave new_wf
```

saves the EViews workfile NEW\_WF.WF1 in the default directory.

```
pagesave "c:\documents and settings\my data\consump"
```

saves the workfile CONSUMP.WF1 in the specified path.

## Cross-references

See also [wfopen \(p. 504\)](#) and [wfsave \(p. 512\)](#).

pageselect	<a href="#">Command</a>
------------	-------------------------

Make the specified page in the default workfile the active page.

## Syntax

Command:      **pageselect** *pgname*

where *pgname* is the name of a page in the default workfile.

## Examples

```
pageselect page2
```

changes the active page to PAGE2.

## Cross-references

See also [wfselect \(p. 514\)](#).

pagestack	Command
-----------	---------

Create a panel structured workfile page using series, alphas, or links from the default workfile page (convert repeated series to repeated observations).

Series in the new panel workfile may be created by stacking series, alphas, and links whose names contain a pattern (series with names that differ only by a “stack identifier”), or by repeating a single series, alpha, or link, for each value in a set of stack identifiers.

## Syntax

Command:        `pagestack(options) stack_id_list [@ series_to_stack]`

Command:        `pagestack(options) pool_name [@ series_to_stack]`

Command:        `pagestack(options) series_name_pattern [@ series_to_stack]`

The resulting panel workfile will use the identifiers specified in one of the three forms of the command:

- *stack\_id\_list* includes a list of the ID values (e.g., “US UK JPN”).
- *pool\_name* is the name of a pool object that contains the ID values.
- *series\_name\_pattern* contains an expression from which the ID values may be determined. The pattern should include the “?” character as a stand in for the parts of the series names containing the stack identifiers. For example, if “CONS?” is the *series\_name\_pattern*, EViews will find all series with names beginning with “CONS” and will extract the IDs from the trailing parts of the observed names.

The *series\_to\_stack* list may contain two types of entries: stacked series (corresponding to sets of series, alphas, and links whose names contain the stack IDs) and simple series (other series, alphas, and links).

To stack a set of series whose names differ only by the stack IDs, you should enter an expression that includes the “?” character in place of the IDs. You may list the names of a single stacked series (e.g., “GDP?” or “?CONS”), or you may use expressions containing the wildcard character “\*” (e.g., “\*?” and “?C\*”) to specify multiple series.

By default, the stacked series will be named in the new workfile using the base portion of the series name (if you specify “?CONS” the stacked series will be named “CONS”), and will contain the values of the individual series stacked one on top of another. If one of the individual series associated with a particular stack ID does not exist, the corresponding stacked values will be assigned the value NA.

Individual (simple) series may also be stacked. You may list the names of individual simple series (e.g., “POP INC”), or you can specify your series using expressions containing the wildcard character “\*” (e.g., “\*”, “\*C”, and “F\*”). A simple series will be stacked on top of itself, once for each ID value. If the target workfile page is in the same workfile, EViews will create a link in the new page; otherwise, the stacked series will contain (repeated) copies of the original values.

When evaluating wildcard expressions, stacked series take precedence over simple series. This means that simple series wildcards will be processed using the list of series not already included as a stacked series.

If the *series\_to\_stack* list is not specified, the expression “\*? \*”, is assumed.

## Options

? = <i>name_patt</i> ,	Specifies the characters to use instead of the identifier, “?”, in naming the stacked series.
<i>name_patt</i>	By default, the <i>name_patt</i> is blank, indicating, for example, that the stacked series corresponding to the pattern “GDP?” will be named “GDP” in the stacked workfile page. If pattern is set to “STK”, the stacked series will be named GDPSTK.
interleave	Interleave the observations in the destination stacked workfile (stack by using all of the series values for the first source observation, followed by the values for the second observation, and so on). The default is to stack observations by identifier (stack the series one on top of each other).
wf = <i>wf_name</i>	Optional name for the new workfile. If not provided, EViews will create a new page in the default workfile.
page = <i>page_name</i>	Optional name for the page in the destination workfile. If not provided, EViews will use the next available name of the form “Untitled”.

## Examples

Consider a workfile that contains the seven series: GDPUS, GDPUK, GDPJPN, CONSUS, CONSUK, CONSJPN, CONSFR, and WORLDGDP.

```
pagestack us uk jpn @ *?
```

creates a new, panel structured workfile page with the series GDP and CONS, containing the stacked GDP? series (GDPUS, GDPUK, and GDPJPN) and stacked CONS? series (CONSUS, CONSUK, and CONSJPN). Note that CONSFR and WORLDGDP will not be copied or stacked.

We may specify the stacked series list explicitly. For example:

```
pagestack(page=stackctry) gdp? @ gdp? cons?
```

first determines the stack IDs from the names of series beginning with “GDP”, the stacks the GDP? and CONS? series. Note that this latter example also names the new workfile page STACKCTRY.

If we have a pool object, we may instruct EViews to use it to specify the IDs:

```
pagestack(wf=newwf, page=stackctry) countrypool @ gdp? cons?
```

Here, the panel structured page STACKCTRY will be created in the workfile NEWWF.

Simple series may be specified by adding them to the stack list, either directly, or using wildcard expressions. Both commands:

```
pagestack us uk jpn @ gdp? cons? worldgdp consfr  
pagestack(wf=altwf) us uk jpn @ gdp? cons? *
```

stack the various GDP? and CONS? series on top of each other, and stack the simple series GDPFR and WORLDGDP on top of themselves.

In the first case, we create a new panel structured page in the same workfile containing the stacked series GDP and CONS and link objects CONSFR and WORLDGDP, which repeat the values of the series. In the second case, the new panel page in the workfile ALTWF will contain the stacked GDP and CONS, and series named CONSFR and WORLDGDP containing repeated copies of the values of the series.

The following two commands are equivalent:

```
pagestack(wf=newwf) us uk jpn @ *? *  
pagestack(wf=newwf) us uk jpn
```

Here, every series, alpha, and link in the source workfile is stacked and copied to the destination workfile, either by stacking different series containing the *stack\_id* or by stacking simple series on top of themselves.

The “?= ” option may be used to prevent name collision.

```
pagestack(?>"stk") us uk jpn @ gdp? gdp
```

stacks GDPUS, GDPUK and GDPJPN into a series called GDPSTK and repeats the values of the simple series GDP in the destination series GDP.

## Cross-references

For additional discussion, see [“Stacking a Workfile” on page 243](#) in the *User’s Guide*. See also [pageunstack \(p. 381\)](#).

pagestruct	Command
------------	---------

Assign a structure to the active workfile page.

### Syntax

Command:      `pagestruct(options) [id_list]`

Command:      `pagestruct(options) *`

where *id\_list* is an (optional) list of ID series. The “\*” may be used as shorthand for the indices currently in place.

If an *id\_list* is provided, EViews will attempt to auto determine the workfile structure. Auto-determination may be overridden through the use of options.

If you do not provide an *id\_list*, the workfile will be restructured as a regular frequency workfile. In this case, either the “none” or the “freq = ” and “start = ” options described below must be provided.

### Options

none	Remove the existing workfile structure.
freq = <i>arg</i>	Specifies a regular frequency; if not provided EViews will auto-determine the frequency.
start = <i>arg</i>	Start date of the regular frequency structure; if not specified, defaults to “@FIRST”. Legal values for <i>arg</i> are described below.
end = <i>arg</i>	End date of the regular frequency structure; if not specified, defaults to “@LAST”. Legal values for <i>arg</i> are described below.

regular, reg	When used with a date ID, this option informs EViews to insert observations (if necessary) to remove gaps in the date series so that it follows the regular frequency calendar. The option has no effect unless a date index is specified.
create	<p>Allow creation of a new series to serve as an additional ID series when duplicate ID values are found in any group. EViews will use this new series as the observation ID.</p> <p>The default is to prompt in interactive mode and to fail in programs.</p>
balance = <i>arg</i> , bal = <i>arg</i>	<p>Balance option (for panel data) describing how EViews should handle data that are unbalanced across ID (cross-section) groups.</p> <p>The <i>arg</i> should be formed using a combination of starts (“s”), ends (“e”), and middles (“m”), as in “balance = se” or “balance = m”.</p> <p>If balancing starts (<i>arg</i> contains “s”), EViews will (if necessary) add observations to your workfile so that each cross-section begins at the same observation (the earliest date or observation observed).</p> <p>If balancing ends (<i>arg</i> contains “e”), EViews will add any observations required so that each cross-section ends at the same point (the last date or observation observed).</p> <p>If balancing middles (<i>arg</i> contains “m”) EViews will add observations to ensure that each cross-section has consecutive observations from the earliest date or observation for the cross-section to the last date or observation for the cross-section.</p> <p>Note that “balance = m” is implied by the “regular” option.</p>
dropna	Specifies that observations which contain missing values (NAs or blank strings) in any ID series (including the date or observation ID) be removed from the workfile. If “dropna” is not specified and NAs are found, EViews will prompt in interactive mode and fail in programs.

**dropbad**

Specifies that observations for which any of the date index series contain values that do not represent dates be removed from the workfile. If “dropbad” is not provided and bad dates are present, EViews will prompt in interactive mode and fail in programs.

The values for start and end dates should contain date literals (actual dates or periods), e.g., “1950q1” and “2/10/1951”, “@FIRST”, or “@LAST” (first and last observed dates in the date ID series). Date literals must be used for the “start =” option when restructuring to a regular frequency.

In addition, offsets from these date specifications can be specified with a “+” or “-” followed by an integer: “@FIRST-5”, “@LAST + 2”, “1950m12 + 6”. Offsets are most often used when resizing the workfile to add or remove observations from the endpoints.

## Examples

```
pagestruct state industry
```

structures the workfile using the IDs in the STATE and INDUSTRY series.

A date ID series (or a series used to form a date ID) should be tagged using the “@DATE” keyword. For example:

```
pagestruct state @date(year)  
pagestruct(regular) @date(year, month)
```

A “\*” may be used to indicate the indices defined in the current workfile structure.

```
pagestruct(end=@last+5) *
```

adds 5 observations to the end of the current workfile.

When you omit the *id\_list*, EViews will attempt to restructure the workfile to a regular frequency. In this case you must either provide the “freq =” and “start =” options to identify the regular frequency structure, or you must specify “none” to remove the existing structure:

```
pagestruct(freq=a, start=1950)  
pagestruct(none)
```

## Cross-references

For extensive discussion, see “[Structuring a Workfile](#)” beginning on page 199 in the *User’s Guide*.

pageunstack	<a href="#">Command</a>
-------------	-------------------------

Unstack workfile page (convert repeated observations to repeated series).

Create a new workfile page by taking series objects (series, alphas, or links) in the default workfile page and breaking them into multiple series (or alphas), one for each distinct value found in a user supplied list of series objects. Typically used on a page with a panel structure.

## Syntax

Command: `pageunstack(options) stack_id obs_id [@ series_to_unstack]`

where *stack\_id* is a single series containing the unstacking ID values used to identify the individual unstacked series, *obs\_id* is a series containing the observation IDs, and *series\_to\_unstack* is an optional list of series objects to be copied to the new workfile.

## Options

<code>namepat = name_pattern</code>	Specifies the pattern from which unstacked series names are constructed, where “*” indicates the original series name and “?” indicates the stack ID.  By default the <i>name_pattern</i> is “*?”, indicating, for example, that if we have the IDs “US”, “UK”, “JPN”, the unstacked series corresponding to the series GDP should be named “GDPUS”, “GDPUK”, “GDPJPN” in the unstacked workfile page.
<code>wf = wf_name</code>	Optional name for the new workfile. If not provided, EViews will create a new page in the default workfile.
<code>page = page_name</code>	Optional name for the page in the destination workfile. If not provided, EViews will use the next available name of the form “Untitled”.

## Examples

Consider a workfile that contains the series GDP and CONS which contain the values of Gross Domestic Product and consumption for three countries stacked on top of each other. Suppose further there is an alpha object called COUNTRY containing the observations “US”, “UK”, and “JPN”, which identify which country each observation of GDP and CONS comes. Finally, suppose there is a date series DATEID which identifies the date for each observation. The command:

```
pageunstack country dateid @ gdp cons
```

creates a new workfile page using the workfile frequency and dates found in DATEID. The page will contain the 6 series GDPUS, GDPUK, GDPJPN, CONSUS, CONSUK, and CON-SJPN corresponding to the unstacked GDP and CONS.

Typically the source workfile described above would be structured as a dated panel with the cross-section ID series COUNTRY and the date ID series DATEID. Since the panel has built-in date information, we may use the “@DATE” keyword as the DATEID. The command:

```
pageunstack country @date @ gdp cons
```

uses the date portion of the current workfile structure to identify the dates for the unstacked page..

The *stack\_id* must be an ordinary, or an alpha series that uniquely identifies the groupings to use in unstacking the series. *obs\_id* may be one or more ordinary series or alpha series, the combination of which uniquely identify each observation in the new workfile.

You may provide an explicit list of series to unstack following an “@” immediately after the *obs\_id*. Wildcards may be used in this list. For example:

```
pageunstack country dateid @ g* c*
```

unstacks all series and alphas that have names that begin with “G” or “C”.

If no *series\_to\_unstack* list is provided, all series in the source workfile will be unstacked. Thus, the two commands:

```
pageunstack country dateid @ *
pageunstack country dateid
```

are equivalent.

By default, series are named in the destination workfile page by appending the *stack\_id* values to the original series name. Letting “\*” stand for the original series name and “?” for the *stack\_id*, names are constructed as “\*?”. This default may be changed using the “namepat = ” option. For example:

```
pageunstack(namepat="?_*") country dateid @ gdp cons
```

creates the series US\_GDP, UK\_GDP, JPN\_GDP, etc.

## Cross-references

For additional discussion and examples, see “[Unstacking a Workfile](#)” on page 236 of the *User’s Guide*. See also [pagestack \(p. 375\)](#).

<b>param</b>	<a href="#">Command</a>
--------------	-------------------------

**Set parameter values.**

Allows you to set the current values of coefficient vectors. The command may be used to provide starting values for the parameters in nonlinear least squares, nonlinear system estimation, and (optionally) ARMA estimation.

### Syntax

Command:      **param** *coef\_name1 number1* [*coef\_name2 number2* *coef\_name3 number3...*]

List, in pairs, the names of the coefficient vector and its element number followed by the corresponding starting values for any of the parameters in your equation.

### Examples

```
param c(1) .2 c(2) .1 c(3) .5
```

resets the first three values of the coefficient vector C.

```
coef(3) beta
param beta(2) .1 beta(3) .5
```

The first line declares a coefficient vector BETA of length 3 that is initialized with zeros. The second line sets the second and third elements of BETA to 0.1 and 0.5, respectively.

### Cross-references

See “[Starting Values](#)” on page 471 of the *User’s Guide* for a discussion of setting initial values in nonlinear estimation.

<b>pcomp</b>	<a href="#">Group View</a>
--------------	----------------------------

**Principal components analysis.**

### Syntax

Group View:      **group\_name.pcomp**(*options*) [*ser1 ser2 ...*]

Enter the name of the group followed by a period, the keyword and optionally, a list of  $k$  names to store the first  $k$  principal components. Separate each name in the list with a space and do not list more names than the number of series in the group.

## Options

cor ( <i>default</i> )	Use sample correlation matrix.
cov	Use sample covariance matrix.
dof	Degrees of freedom adjustment if “cov” option used. Default is no adjustment (compute sample covariance dividing by $n$ rather than $n - 1$ ).
eigval = <i>vec_name</i>	Specify name of vector to hold the saved the eigenvalues in workfile.
eigvec = <i>mat_name</i>	Specify name of matrix to hold the save the eigenvectors in workfile.
p	Print results.

## Examples

```
group g1 x1 x2 x3 x4  
freeze(tab1) g1.pcomp(cor, eigval=v1, eigvec=m1) pc1 pc2
```

The first line creates a group named G1 containing the four series X1, X2, X3, X4. The second line stores the first two principal components of the sample correlation matrix in series named PC1 and PC2. The output view is stored in a table named TAB1, the eigenvalues in a vector named V1, and the eigenvectors in a matrix named M1.

## Cross-references

See “[Principal Components](#)” on page 373 of the *User’s Guide* for further discussion.

pie	<a href="#">Command</a>    <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a>   <a href="#">Sym View</a>
-----	---

[Command](#) || [Graph Command](#) | [Group View](#) | [Matrix View](#) | [Rowvector View](#) | [Sym View](#)

Display pie graph view of data in object, or change existing graph object type to pie chart.

Display pie charts for any number of series or data in a matrix object. There will be one pie for each date or observation number, or each row of a matrix, provided the values are positive. Each series or column is shown as a wedge in a different color/pattern, where the width of the wedge equals the percentage contribution of the series/column to the total of all listed series.

## Syntax

Command: `pie(options) arg1 [arg2 arg3 ...]`

Object View: `object_name.pie(options)`

Graph Proc: `graph_name.pie(options)`

To use `pie` as a command, simply list the name of one or more series or groups, or a matrix object to include in the pie chart. You may also change the exiting graph type by using `pie` as a proc. Simply list the graph name, followed by a period, and the `pie` keyword.

## Options

`o = graph_name` Use appearance options from the specified graph.

`t = graph_name` Use appearance options and copy text and shading from the specified graph.

`p` Print the pie chart.

### Panel options

The following options apply when graphing panel structured data.

`panel = arg`  
 (default taken  
 from global set-  
 tings) Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “medmxmn” (plot median, max and min).

## Examples

```
smp1 1990 1995
```

```
pie cons inv gov
```

shows six pie charts, each divided into CONS, INV, and GOV.

```
graph gr1.line cons inv gov
```

gr1.pie

creates a line graph GR1 and then changes the graph to a pie chart.

### Cross-references

See [Chapter 14](#) of the *User’s Guide* for a discussion of graphs and templates.

See also [graph \(p. 303\)](#) for graph declaration and other graph types.

plot	<a href="#">Command</a>
------	-------------------------

Line graph.

Provided for backward compatibility. See [line \(p. 320\)](#).

pool	<a href="#">Object Declaration</a>
------	------------------------------------

Declare pool object.

### Syntax

Command:      **pool** *name* [*id1 id2 id3 ...*]

Follow the **pool** keyword with a *name* for the pool object. You may optionally provide the identifiers for the cross-section members of the pool object. Pool identifiers may be added or removed at any time using [add \(p. 196\)](#) and [drop \(p. 270\)](#).

### Examples

pool zool dog cat pig owl ant

Declares a pool object named ZOO1 with the listed cross-section identifiers.

### Cross-references

“Pool” on page 169 contains a complete description of the pool object. See [Chapter 27, “Pooled Time Series, Cross-Section Data”, on page 809](#) of the *User’s Guide* for a discussion of working with pools in EViews.

See [add \(p. 196\)](#) and [drop \(p. 270\)](#). See also [ls \(p. 329\)](#) for details on estimation using a pool object.

**predict**[Equation View](#)

Prediction table for binary and ordered dependent variable models.

The prediction table displays the actual and estimated frequencies of each distinct value of the discrete dependent variable.

### Syntax

Equation Proc:    `eq_name.predict(options)`

For binary models, you may optionally specify how large the estimated probability must be to be considered a success ( $y = 1$ ). Specify the cutoff level as an option in parentheses after the keyword `predict`.

### Options

`n (default = .5)`      Cutoff probability for success prediction in binary models (between 0 and 1).

`p`                  Print the prediction table.

### Examples

```
equation eq1.binary(d=1) work c edu age race
eq1.predict(0.7)
```

Estimates a logit and displays the expectation-prediction table using a cutoff probability of 0.7.

### Cross-references

See “[Binary Dependent Variable Models](#)” on page [605](#) of the *User’s Guide* for a discussion of binary models, and “[Expectation-Prediction \(Classification\) Table](#)” on page [613](#) of the *User’s Guide* for examples of prediction tables.

**print**[Command](#)

Sends views of objects to the default printer.

### Syntax

Command:    `print(options) object1 [object2 object3 ...]`

Command:    `print(options) object_name.view_command`

`print` should be followed by a list of object names or a view of an object to be printed. The list of names must be of the same object type. If you do not specify the view of an object, `print` will print the default view for the object.

## Options

p	Print in portrait orientation.
l	Print in landscape orientation.

The default orientation is set by clicking on **Print Setup**.

## Examples

```
print gdp log(gdp) d(gdp) @pch(gdp)
```

sends a table of GDP, log of GDP, first difference of GDP, and the percentage change of GDP to the printer.

```
print graph1 graph2 graph3
```

prints three graphs on a single page.

To merge the three graphs, realign them in one row, and print in landscape orientation, you may use the commands:

```
graph mygra.merge graph1 graph2 graph3  
mygra.align(3,1,1)  
print(l) mygra
```

To estimate the equation EQ1 and send the output view to the printer.

```
print eq1.ls gdp c gdp(-1)
```

## Cross-references

See “[Print Setup](#)” beginning on page 919 of the *User’s Guide* for a discussion of print options and the **Print Setup** dialog.

See [output \(p. 361\)](#) for print redirection.

<b>probit</b>	<a href="#">Command</a>
---------------	-------------------------

Estimation of binary dependent variable models with normal errors.

Equivalent to “`binary (d=n)`”.

See [binary \(p. 217\)](#).

<b>program</b>	<a href="#">Command</a>
----------------	-------------------------

Declare a program.

### Syntax

Command:      **program** [*path\*]*prog\_name*

Enter a name for the program after the `program` keyword. If you do not provide a name, EViews will open an untitled program window. Programs are text files, not objects.

### Examples

```
program runreg
```

opens a program window named RUNREG which is ready for program editing.

### Cross-references

See [Chapter 6, “EViews Programming”, on page 83](#) of the *Command and Programming Reference* for further details, and examples of writing EViews programs.

See also [open \(p. 356\)](#).

<b>qqplot</b>	<a href="#">Group View</a>   <a href="#">Series View</a>
---------------	--

Quantile-quantile plots.

Plots the (empirical) quantiles of a series against the quantiles of a theoretical distribution or the empirical quantiles of another series. You may specify the theoretical distribution and/or the method used to compute the empirical quantiles as options.

### Syntax

Object View:      `object_name.qqplot(options)`

### Options

n	Plot against the quantiles of a normal distribution.
u	Plot against the quantiles of a uniform distribution.
e	Plot against the quantiles of an exponential distribution.
l	Plot against the quantiles of a logistic distribution.

x	Plot against the quantiles of an extreme value distribution.
s = <i>series_name</i>	Plot against the (empirical) quantiles of the specified series.
q = <i>arg</i> (default = "r")	Compute quantiles using the definition: "b" (Blom), "r" (Rankit-Cleveland), "o" (simple fraction), "t" (Tukey), "v" (van der Waerden).
p	Print the QQ-plot.

## Examples

```
equation eq1.binary(d=1) work c edu age race  
eq1.makeresid(o) res1  
res1.qqplot(1)
```

estimates a logit, retrieves the residuals, and plots the quantiles of the residuals against the quantiles from the logistic distribution. If the error distribution is correctly specified, the QQ-plot should lie on a straight line.

## Cross-references

See “Quantile-Quantile” on page 381 of the *User’s Guide* for a discussion of QQ-plots.

See also [cdfplot \(p. 230\)](#).

<a href="#">qstats</a>	<a href="#">Var View</a>
------------------------	--------------------------

Multivariate residual autocorrelation Portmanteau tests.

## Syntax

Var View: `var_name.qstats(h, options)`

You must specify the highest order of lag *h* to test for serial correlation. *h must be larger than the VAR lag order.*

## Options

name = <i>arg</i>	Save <i>Q</i> -statistics in the named matrix object. The matrix has two columns: the first column contains the unmodified <i>Q</i> -statistic; the second column contains the modified <i>Q</i> -statistics.
p	Print the Portmanteau test results.

## Examples

```
var var1.ls 1 6 lgdp lm1 lcpi
show var1.qstats(12, name=q)
```

The first line declares and estimates a VAR. The second line displays the portmanteau tests for lags up to 12, and stores the  $Q$ -statistics in a matrix named Q.

## Cross-references

See “[Diagnostic Views](#)” on page 708 of the *User’s Guide* for a discussion of the Portmanteau tests and other VAR diagnostics.

See [arlm](#) (p. 209) for a related multivariate residual serial correlation LM test.

<b>range</b>	<a href="#">Command</a>
--------------	-------------------------

Reset the workfile range for a regular frequency workfile.

No longer supported. See the replacement command [pagestruct](#) (p. 378).

<b>read</b>	<a href="#">Command</a>    <a href="#">Coef Proc</a>   <a href="#">Matrix Proc</a>   <a href="#">Pool Proc</a>   <a href="#">Rowvector Proc</a>   <a href="#">Sym Proc</a>   <a href="#">Vector Proc</a>
-------------	--

Import data from a foreign disk file into an existing EViews workfile.

May be used to import data into an existing workfile from a text, Excel, or Lotus file on disk. Used as a command, it imports data into series in the workfile. When used as a procedure, `read` imports data directly into pool and matrix objects.

## Syntax

Command:	<code>read(options) [path\]file_name name1 [name2 name3 ...]</code>
Command:	<code>read(options) [path\]file_name n</code>
Coef Proc:	<code>coef_name.read(options) [path\]file_name</code>
Pool Proc:	<code>pool_name.read(options) [path\]file_name pool_ser1 [pool_ser2 pool_ser3 ...]</code>
Matrix Proc:	<code>matrix_name.read(options) [path\]file_name</code>

You must supply the name of the source file. If you do not include the optional path specification, EViews will look for the file in the default directory. Path specifications may point to local or network drives. If the path specification contains a space, you may enclose the entire expression in double quotation marks.

The input specification follows the source file name. In the command form of `read`, there are two ways to specify the input series. First, you may list the names of the series in the order they appear in the file. Second, if the data file contains a header line for the series names, you may specify the number,  $n$ , of series in the file instead of a list of names. EViews will name the  $n$  series using the names given in the header line. If you specify a number and the data file does not contain a header line, EViews will name the series as SER01, SER02, SER03, and so on.

For the pool proc form of `read`, you must provide a list of ordinary or pool series.

## Options

### *File type options*

`t = dat, txt`      ASCII (plain text) files.

`t = wk1, wk3`      Lotus spreadsheet files.

`t = xls`      Excel spreadsheet files.

If you do not specify the “`t`” option, EViews uses the file name extension to determine the file type. If you specify the “`t`” option, the file name extension will not be used to determine the file type.

### *Options for ASCII text files*

`t`      Read data organized by series (or transpose the data for matrix objects). Default is to read by observation with series in columns.

`na = text`      Specify text for NAs. Default is “NA”.

`d = t`      Treat tab as delimiter (note: you may specify multiple delimiter options). The *default* is “`d = c`” only.

`d = c`      Treat comma as delimiter.

`d = s`      Treat space as delimiter.

`d = a`      Treat alpha numeric characters as delimiter.

`custom = symbol`      Specify symbol/character to treat as delimiter.

`mult`      Treat multiple delimiters as one.

`name`      Series names provided in file.

`label = integer`      Number of lines between the header line and the data.  
Must be used with the “`name`” option.

rect ( <i>default</i> ) / norect	[Treat / Do not treat] file layout as rectangular.
skipcol = <i>integer</i>	Number of columns to skip. Must be used with the “rect” option.
skiprow = <i>integer</i>	Number of rows to skip. Must be used with the “rect” option.
comment = <i>symbol</i>	Specify character/symbol to treat as comment sign. Everything to the right of the comment sign is ignored. Must be used with the “rect” option.
singlequote	Strings are in single quotes, not double quotes.
dropstrings	Do not treat strings as NA; simply drop them.
negparen	Treat numbers in parentheses as negative numbers.
allowcomma	Allow commas in numbers (note that using commas as a delimiter takes precedence over this option).
currency = <i>symbol</i>	Specify symbol/character for currency data.

#### Options for spreadsheet (Lotus, Excel) files

t	Read data organized by series (or transpose the data for matrix objects). Default is to read by observation with series in columns.
letter_number ( <i>default</i> = “b2”)	Coordinate of the upper-left cell containing data.
s = <i>sheet_name</i>	Sheet name for Excel 5–8 Workbooks.

#### Options for pool reading

bycross ( <i>default</i> ) / byper	Stack pool data by [cross-section / date or period] (only for pool write).
---------------------------------------	--

#### Examples

```
read(t=dat,na=.) a:\mydat.raw id lwage hrs
```

reads data from an ASCII file MYDAT.RAW in the A: drive. The data in the file are listed by observation, the missing value NA is coded as a “.” (dot or period), and there are three series, which are to be named ID, LWAGE, HRS (from left to right).

```
read(a2, s=sheet3) cps88.xls 10
```

reads data from an Excel file CPS88 in the default directory. The data are organized by observation, the upper left data cell is A2, and 10 series are read from a sheet named SHEET3 using names provided in the file.

```
read(a2, s=sheet2) "\\network\dr 1\cps91.xls" 10
```

reads the Excel file CPS91 from the network drive specified in the path.

### Cross-references

See “[Importing Data](#)” on page 97 of the *User’s Guide* for a discussion and examples of importing data from external files.

For powerful, easy-to-use tools for reading data into a new workfile, see “[Creating a Workfile by Reading from a Foreign Data Source](#)” on page 47 of the *User’s Guide*, [pageload](#) (p. 371), and [wfopen](#) (p. 504).

See also [write](#) (p. 517).

rename	<a href="#">Command</a>
--------	-------------------------

Rename an object in the active workfile or database.

### Syntax

Command:      **rename** *old\_name new\_name*

After the **rename** keyword, list the old object name followed by the new name. Note that the name specifications may include matching wildcard patterns.

### Examples

```
rename temp_u u2
```

renames an object named TEMP\_U as U2.

```
rename aa::temp_u aa::u2
```

renames the object TEMP\_U to U2 in database AA.

```
rename a* b*
```

renames all objects beginning with the letter “A” to begin with the letter “B”.

## Cross-references

See [Chapter 4, “Object Basics”, on page 65](#) of the *User’s Guide* for a discussion of working with objects in EViews.

<b>representations</b>	<a href="#">Equation View</a>   <a href="#">Pool View</a>   <a href="#">Var View</a>
------------------------	--

Display text of specification for equation, pool, and var objects.

### Syntax

Object View:      object\_name.representation(*options*)

### Options

p	Print the representation text.
---	--------------------------------

### Examples

`pool1.representations`

displays the specifications of the estimation object POOL1.

## Cross-references

See also [spec \(p. 454\)](#).

<b>resample</b>	<a href="#">Group Proc</a>   <a href="#">Series Proc</a>
-----------------	--

Resample from observations in a series or group.

### Syntax

Object Proc:      object\_name.resample(*options*) [*output\_spec*]

You should follow the `resample` keyword and options with a list of names or a wildcard expression identifying the series to hold the output. If a list is used to identify the targets, the number of target series must match the number of names implied by the keyword.

### Options

<code>outsmpl = "smpl"</code>	Sample to fill the new series. Either provide the sample range in double quotes or specify a named sample object. The default is the current workfile sample.
-------------------------------	---

name =	Name of group to hold created series.
<i>group_name</i>	
permute	Draw from rows without replacement. Default is to draw with replacement.
weight =	
<i>series_name</i>	Name of series to be used as weights. The weight series must be non-missing and non-negative in the current workfile sample. The default is equal weights.
block = <i>integer</i>	Block length for each draw. Must be a positive integer. The default block length is 1.
withna ( <i>default</i> )	[Draw / Do not draw] from all rows in the current sample, including those with NAs.
dropna	Do not draw from rows that contain missing values in the current workfile sample.
fixna	Excludes NAs from draws but copies rows containing missing values to the output series.

- Since we append a suffix for the new name, you may not use groups that contain auto-series. For example, resampling from a group containing the series X(-1) or LOG(X) will error. You will have to generate new series, say by setting XLAG = X(-1) or LOGX = LOG(X). Then create a new group consisting of XLAG and LOGX and call the bootstrap procedure on this new group.
- If the group name you provide already exists and is a group object, the group object will be overwritten. If the object already exists but is not a group object, EViews will error.
- Block bootstrap (block length larger than 1) requires a continuous output sample. Therefore a block length larger than 1 cannot be used together with the “fixna” option, and the “outsmpl” should not contain any gaps.
- The “fixna” option will have an effect only if there are missing values in the overlapping sample of the input sample (current workfile sample) and the output sample specified by “outsmpl”.
- If you specify “fixna”, we first copy any missing values in the overlapping sample to the output series. Then the input sample is adjusted to drop rows containing missing values and the output sample is adjusted so as not to overwrite the copied values.
- If you choose “dropna” and the block length is larger than 1, the input sample may shrink in order to ensure that there are no missing values in any of the drawn blocks.

- If you choose “permute”, the block option will be reset to 1, the “dropna” and “fixna” options will be ignored (reset to the default “withna” option), and the “weight” option will be ignored (reset to default equal weights).

## Examples

```
group g1 x y
g1.resample
```

creates new series X\_B and Y\_B by drawing with replacement from the rows of X and Y in the current workfile sample. If X\_B or Y\_B already exist in the workfile, they will be overwritten if they are series objects, otherwise EViews will error. Note that only values of X\_B and Y\_B in the output sample (in this case the current workfile sample) will be overwritten.

```
g1.resample(weight=wt,suffix=_2) g2
```

will append “\_2” to the names for the new series, and will create a group object named G2 containing these series. The rows in the sample will be drawn with probabilities proportional to the corresponding values in the series WT. WT must have non-missing non-negative values in the current workfile sample.

## Cross-references

See “[Resample](#)” on page 322 of the *User’s Guide* for a discussion of the resampling procedure. For additional discussion of wildcards, see [Appendix B, “Wildcards”, on page 921](#) of the *User’s Guide*.

See also [@resample \(p. 595\)](#) and [@permute \(p. 594\)](#) for sampling from matrices.

<b>reset</b>	<a href="#">Command</a>    <a href="#">Equation View</a>
--------------	--

Compute Ramsey’s regression specification error test.

## Syntax

Command:	<code>reset(<i>n, options</i>)</code>
Equation View:	<code>eq_name.reset(<i>n, options</i>)</code>

You must provide the number of powers of fitted terms *n* to include in the test regression.

## Options

p	Print the test result.
---	------------------------

## Examples

```
equation eq1.ls lwage c edu race gender  
eq1.reset(2)
```

carries out the RESET test by including the square and the cube of the fitted values in the test equation.

## Cross-references

See “[Ramsey's RESET Test](#)” on page 570 of the *User's Guide* for a discussion of the RESET test.

residcor	<a href="#">Pool View</a>   <a href="#">Sspace View</a>   <a href="#">System View</a>   <a href="#">Var View</a>
----------	--

Residual correlation matrix.

Displays the correlations of the residuals from each equation in the system, sspace, or var object, or the residuals from each pool cross-section equation. The sspace object residuals used in the calculation are the standardized, one-step ahead signal forecast errors.

## Syntax

Object View:      object\_name.residcor(*options*)

## Options

p	Print the correlation matrix.
---	-------------------------------

## Examples

```
sys1.residcor
```

displays the residual correlation matrix of SYS1.

## Cross-references

See also [residcov](#) (p. 398) and [makeresids](#) (p. 342).

residcov	<a href="#">Pool View</a>   <a href="#">Sspace View</a>   <a href="#">System View</a>   <a href="#">Var View</a>
----------	--

Residual covariance matrix.

Displays the covariances of the residuals from each equation in the system, sspace, or var object, or the residuals from each pool cross-section equation. The sspace object residuals used in the calculation are the standardized, one-step ahead signal forecast errors.

## Syntax

Object View: `object_name.residcov(options)`

## Options

<code>p</code>	Print the covariance matrix.
----------------	------------------------------

## Examples

```
var1.residcov
```

displays the residual covariance matrix of VAR1.

## Cross-references

See also [residcor \(p. 398\)](#) and [makeresids \(p. 342\)](#).

<b>resids</b>	<a href="#">Equation View</a>   <a href="#">Pool View</a>   <a href="#">Sspace View</a>   <a href="#">System View</a>   <a href="#">Var View</a>
---------------	--

**Display residuals.**

For equation and pool objects, `resids` allows you to display the actual, fitted values and residuals in either tabular or graphical form.

For sspace objects, `resids` allows you to display the actual-fitted-residual graph.

For system, var or pool objects, `resids` displays multiple graphs of the residuals. Each graph will contain the residuals for each equation in the system or VAR, or for each cross-section in the pool.

## Syntax

Object View: `object_name.resids(options)`

## Options

<code>g (default)</code>	Display graph(s) of residuals.
--------------------------	--------------------------------

<code>t</code>	Display table(s) of residuals (not available for system, pool, sspace or var objects).
----------------	--

<code>p</code>	Print the table/graph.
----------------	------------------------

## Examples

```
equation eq1.ls m1 c inc tb3 ar(1)
eq1.resids
```

regresses M1 on a constant, INC, and TB3, correcting for first order serial correlation, and displays a table of actual, fitted, and residual series.

```
eq1.resids(g)
```

displays a graph of the actual, fitted, and residual series.

### Cross-references

See also [makeresids \(p. 342\)](#).

<b>results</b>	<a href="#">Equation View</a>   <a href="#">Log View</a>   <a href="#">Pool View</a>   <a href="#">SSpace View</a>   <a href="#">System View</a>   <a href="#">Var View</a>
----------------	---

Displays the results view of objects containing estimation output.

### Syntax

Object View:      object\_name.results(*options*)

### Options

p                  Print the view.

### Examples

```
equation eq1.ls m1 c inc tb3 ar(1)  
eq1.results(p)
```

estimates an equation using least squares, and displays and prints the results.

```
var mvar.ls 1 4 8 8 m1 gdp tb3 @ @trend(70.4)  
mvar.results(p)
```

prints the estimation results from the estimated VAR.

<b>rls</b>	<a href="#">Equation View</a>
------------	-------------------------------

### Recursive least squares regression.

The rls view of an equation displays the results of recursive least squares (rolling) regression. This view is only available for (non-panel) equations estimated by ordinary least squares without ARMA terms.

You may plot various statistics from rls by choosing an option.

## Syntax

Equation View: `eq_name.rls(options) c(1) c(2) ...`

## Options

<code>r</code>	Plot the recursive residuals about the zero line with plus and minus two standard errors.
<code>r,s</code>	Plot the recursive residuals and save the residual series and their standard errors as series named R_RES and R_RESSE, respectively.
<code>c</code>	Plot the recursive coefficient estimates with two standard error bands.
<code>c,s</code>	Plot the listed recursive coefficients and save all coefficients and their standard errors as series named R_C1, R_C1SE, R_C2, R_C2SE, and so on.
<code>o</code>	Plot the <i>p</i> -values of recursive one-step Chow forecast tests.
<code>n</code>	Plot the <i>p</i> -values of recursive <i>n</i> -step Chow forecast tests.
<code>q</code>	Plot the CUSUM (standardized cumulative recursive residual) and 5 percent critical lines.
<code>v</code>	Plot the CUSUMSQ (CUSUM of squares) statistic and 5 percent critical lines.
<code>p</code>	Print the view.

## Examples

```
equation eq1.ls m1 c tb3 gdp
eq1.rls(r,s)
eq1.rls(c) c(2) c(3)
```

plots and saves the recursive residual series from EQ1 and their standard errors as R\_RES and R\_RESSE. The third line plots the recursive slope coefficients of EQ1.

```
equation eq2.ls m1 c pdl(tb3,12,3) pdl(gdp,12,3)
eq2.rls(c) c(3)
eq2.rls(q)
```

The second command plots the recursive coefficient estimates of PDL02, the linear term in the polynomial of TB3 coefficients. The third line plots the CUSUM test statistic and the 5% critical lines.

### Cross-references

See “[Recursive Least Squares](#)” on page 572 of the *User’s Guide*.

<b>rndint</b>	Command
---------------	---------

Generate uniform random integers.

The `rndint` command fills series, vector, and matrix objects with (pseudo) random integers drawn uniformly from zero to a user specified maximum. The `rndint` command ignores the current sample and fills the entire object with random integers.

### Syntax

Command:      `rndint(object_name, n)`

Type the name of the series, vector, or matrix object to fill, followed by an integer value representing the maximum value *n* of the random integers. *n* should a positive integer.

### Examples

```
series index  
rndint(index, 10)
```

fills the entire series INDEX with integers drawn randomly from 0 to 10. Note that unlike standard series assignment using `genr`, `rndint` ignores the current sample and fills the series for the entire workfile range.

```
sym(3) var3  
rndint(var3, 5)
```

fills the entire symmetric matrix VAR3 with random integers ranging from 0 to 5.

### Cross-references

See the list of available random number generators in [Appendix D, “Operator and Function Reference”, beginning on page 543](#).

See also [nrnd](#) (p. 539), [rnd](#) (p. 541) and [rndseed](#) (p. 403).

<b>rndseed</b>	<a href="#">Command</a>
----------------	-------------------------

Seed the random number generator.

Use `rndseed` when you wish to generate a repeatable sequence of random numbers, or to select the generator to be used.

Note that EViews 5 has updated the seeding routines of two of our pseudo-random number generators (backward compatible options are provided). It is strongly recommended that you use new generators.

## Syntax

Command:      `rndseed(options) integer`

Follow the `rndseed` keyword with the optional generator type and an integer for the seed.

## Options

<code>type = arg</code> <i>(default = “kn”)</i>	Type of random number generator: improved Knuth generator (“kn”), improved Mersenne Twister (“mt”), Knuth’s (1997) lagged Fibonacci generator used in EViews 4 (“kn4”), L’Ecuyer’s (1999) combined multiple recursive generator (“le”), Matsumoto and Nishimura’s (1998) Mersenne Twister used in EViews 4 (“mt4”).
--	---

When EViews starts up, the default generator type is set to the improved Knuth lagged Fibonacci generator. Unless changed using `rndseed`, Knuth’s generator will be used for subsequent pseudo-random number generation.

	Knuth (“kn4”)	L’Ecuyer (“le”)	Mersenne Twister (“mt4”)
Period	$2^{129}$	$2^{319}$	$2^{19937}$
Time (for $10^7$ draws)	27.3 secs	15.7 secs	1.76 secs
Cases failed Diehard test	0	0	0

## Examples

```
rndseed 123456
genr t3=@qtdist(rnd,3)
rndseed 123456
```

```
genr t30=@qtdist(rnd,30)
```

generates random draws from a *t*-distribution with 3 and 30 degrees of freedom using the same seed.

### Cross-references

See the list of available random number generators in [Appendix D, “Operator and Function Reference”, beginning on page 543](#).

At press time, further information on the improved seeds may be found on the web at the following addresses:

Knuth generator: <http://sunburn.stanford.edu/~knuth/news02.html#rng>

Mersenne twister: <http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html>

See also [nrnd \(p. 539\)](#), [rnd \(p. 541\)](#) and [rndint \(p. 402\)](#).

<b>rowvector</b>	<a href="#">Object Declaration</a>
------------------	------------------------------------

Declare a rowvector object.

The `rowvector` command declares and optionally initializes a (row) vector object.

### Syntax

Command:        `rowvector(n1) vector_name`

Command:        `rowvector vector_name = assignment`

You may optionally specify the size (number of columns) of the row vector in parentheses after the `rowvector` keyword. If you do not specify the size, EViews creates a rowvector of size 1, unless the declaration is combined with an assignment.

By default, all elements of the vector are set to 0, unless an assignment statement is provided. EViews will automatically resize new rowvectors, if appropriate.

### Examples

```
rowvector rvec1  
rowvector(20) coefvec = 2  
rowvector newcoef = coefvec
```

RVEC1 is a row vector of size one with value 0. COEFVEC is a row vector of size 20 with all elements equal to 2. NEWCOEF is also a row vector of size 20 with all elements equal to the same values as COEFVEC.

## Cross-references

See “[Rowvector](#)” on page 172 for a complete description of the rowvector object.

See also [coef](#) (p. 238) and [vector](#) (p. 501).

<b>run</b>	<a href="#">Command</a>
------------	-------------------------

**Run a program.**

The `run` command executes a program. The program may be located in memory or stored in a program file on disk.

### Syntax

Command:      `run(options) [path\]prog_name [%0 %1 ...]`

If the program has arguments, you should list them after the filename. EViews first checks to see if the specified program is in memory. If not, it looks for the program on disk in the current working directory, or in the specified path. EViews expects that program files will have a “.PRG” extension.

### Options

<code>integer (default = 1)</code>	Set maximum errors allowed before halting the program.
<code>c</code>	Run program file without opening a window for display of the program file.
<code>verbose / quiet</code>	Verbose mode in which messages will be sent to the status line at the bottom of the EViews window (slower execution), or quiet mode which suppresses workfile display updates (faster execution).
<code>v / q</code>	Same as [verbose / quiet].
<code>ver4 / ver5</code>	Execute program in [version 4 / version 5] compatibility mode.

### Examples

```
run(q) simul x xhat
```

quietly runs a program named SIMUL from the default directory using arguments X and XHAT.

Since `run` is a command, it may also be placed in a program file. You should note that if you put the `run` command in a program file and then execute the program, EViews will stop after executing the program referred to by the `run` command. For example, if you have a program containing:

```
run simul  
print x
```

the `print` statement will not be executed since execution will stop after executing the commands in `SIMUL.PRG`. If this behavior is not intended, you should consider using the [include \(p. 607\)](#) statement.

## Cross-references

See “[Executing a Program](#)” on page 84 for further details.

See also [include \(p. 607\)](#).

<b>sample</b>	<a href="#">Object Declaration</a>
---------------	------------------------------------

Declare a sample object.

The `sample` statement declares, and optionally defines, a sample object.

### Syntax

Command:      **sample** *smp\_name* [*smp\_statement*]

Follow the `sample` keyword with a name for the sample object and a sample statement. If no sample statement is provided, the sample object will be set to the current workfile sample.

To reset the sample dates in a sample object, you must use the [set \(p. 420\)](#) procedure.

### Examples

```
sample ss
```

declares a sample object named SS and sets it to the current workfile sample.

```
sample s2 1974q1 1995q4
```

declares a sample object named S2 and sets it from 1974Q1 to 1995Q4.

```
sample fe_bh @all if gender=1 and race=3  
smp fe_bh
```

The first line declares a sample FE\_BL that includes observations where GENDER = 1 and RACE = 3. The second line sets the current sample to FE\_BL.

```
sample sf @last-10 @last
```

declares a sample object named SF and sets it to the last 10 observations of the current workfile range.

```
sample s1 @first 1973q1
s1.set 1973q2 @last
```

The first line declares a sample object named S1 and sets it from the beginning of the workfile range to 1973Q1. The second line resets S1 from 1973Q2 to the end of the workfile range.

## Cross-references

See “[Samples](#)” on page 87 of the *User’s Guide*, and “[Dates](#)” on page 127 for a discussion of using dates and samples in EViews.

See also [set](#) (p. 420) and [smp1](#) (p. 449).

<b>save</b>	<a href="#">Command</a>   <a href="#">Graph Proc</a>   <a href="#">Table Proc</a>
-------------	---

Save graph to disk as a Windows metafile or PostScript file. Save table to disk as a CSV, tab-delimited ASCII text, RTF, or HTML file.

`save` may also be used as a command to save the current workfile to disk. This latter usage is provided only for backward compatibility, as it has been replaced with the equivalent [wfsave](#) (p. 512) command.

## Syntax

Object Proc:	<code>object_name.save(options) [path\]file_name</code>
--------------	---

Follow the keyword with a name for the file. *file\_name* may include the file type extension, or the file type may be specified using the “t =” option. The graph may be saved with an .EMF, .WMF, or .EPS extension.

If an explicit path is not specified, the file will be stored in the default directory, as set in the **File Locations** global options.

## General Graph Options

<code>t = file_type</code>	Specifies the file type, where <i>file_type</i> may be one of: Enhanced Windows metafile (“emf”, “meta”, or “metafile”), ordinary Windows metafile (“wmf”), or Encapsulated PostScript (“eps”, “ps” or “postscript”). Files will be saved with the “.emf”, “.wmf”, and “.eps” extensions, respectively.
<code>u = units</code>	Specify units of measurement, where <i>units</i> is one of: “in” (inches), “cm” (centimeters), “pt” (points), “pica” (picas).
<code>w = width</code>	Set width of the graphic in the selected units.
<code>h = height</code>	Set height of the graphic in the selected units.
<code>c / -c</code>	[Save / Do not save] the graph in color.

Note that if only a *width* or a *height* option is specified, EViews will calculate the other dimension holding the aspect ratio of the graph constant. If both *width* and *height* are provided, the aspect ratio will no longer be locked. EViews will default to the current graph dimensions if size is unspecified.

All defaults are taken from the global graph export settings (**Options/Graphics Defaults.../Exporting**).

## Postscript specific Graph Options

<code>box / -box</code>	[Save / Do not save] the graph with a bounding box. The bounding box is an invisible rectangle placed around the graphic to indicate its boundaries. The default is taken from the global graph export settings.
<code>land</code>	Save the graph in landscape orientation. The default uses portrait mode.

## Table Options

<code>t = file_type (default = “csv”)</code>	Specifies the file type, where <i>file_type</i> may be one of: “csv”(CSV - comma-separated), “rtf” (Rich-text format), “txt” (tab-delimited text), or “html” (HTML - Hypertext Markup Language). Files will be saved with the “.csv”, “.rtf”, “.txt”, and “.htm” extensions, respectively.
--	--

---

<code>s = arg</code>	Scale size, where <i>arg</i> is from 0.05 to 2, representing the fraction of the original table size (only valid for HTML or RTF files).
<code>r = cell_range</code>	Range of table cells to be saved. See <a href="#">setfillcolor (p. 429)</a> for the <i>cell_range</i> syntax. If a range is not provided, the entire table will be saved.
<code>n = string</code>	Replace all cells that contain NA values with the specified string. “NA” is the default.
<code>f / -f</code>	[Use full precision values/ Do not use full precision] when saving values to the table (only applicable to numeric cells). By default, the values will be saved as they appear in the currently formatted table.

## Examples

```
graph1.save(t=ps, -box, land) c:\data\MyGra1
```

saves GRAPH1 as a PostScript file MYGRA1.EPS. The graph is saved in landscape orientation without a bounding box.

```
graph2.save(u=pts, w=300, h=300) MyGra2
```

saves GRAPH2 in the default directory as an Enhanced Windows metafile MYGRA2.EMF. The image will be scaled to  $300 \times 300$  points.

The command:

```
tab1.save mytable
```

saves TAB1 to a CSV file named “MYTABLE.CSV” in the default directory.

```
tab1.save(t=csv, n="NAN") mytable
```

saves TAB1 to a CSV (comma separated value) file named MYTABLE.CSV and writes all NA values as “NAN”.

```
tab1.save(r=B2:C10, t=html, s=.5) mytable
```

saves from data from the second row, second column, to the tenth row, third column of TAB1 to a HTML file named MYTABLE.HTM at half of the original size.

```
tab1.save(f, n=".") mytable
```

saves the data in the second column in full precision to a CSV file named “MYTABLE.CSV”, and writes all NA values as “.”.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, beginning on page 403](#) of the *User’s Guide* for a discussion of graphs and tables.

To save a workfile, see [wfsave \(p. 512\)](#). To redirect output, see [output \(p. 361\)](#).

<b>scalar</b>	<a href="#">Object Declaration</a>
---------------	------------------------------------

Declare a scalar object.

The `scalar` command declares a scalar object and optionally assigns a value.

### Syntax

Command:      `scalar scalar_name[=assignment]`

The `scalar` keyword should be followed by a valid name, and optionally, by an assignment. If there is no explicit assignment, the scalar will be initialized with a value of zero.

### Examples

```
scalar alpha
```

declares a scalar object named ALPHA with value zero.

```
equation eq1.ls res c res(-1 to -4) x1 x2  
scalar lm = eq1.@regobs*eq1.@r2  
show lm
```

runs a regression, saves the  $nR^2$  as a scalar named LM, and displays its value in the status line at the bottom of the EViews window.

## Cross-references

See [“Scalar” on page 174](#) for a summary of the features of scalar objects in EViews.

<b>scale</b>	<a href="#">Graph Proc</a>
--------------	----------------------------

Sets axis and data scaling characteristics for the graph.

By default, EViews optimally chooses the axes to fit the graph data.

### Syntax

Graph Proc:      `graph_name.scale(axis_id) options_list`

The *axis\_id* parameter identifies which of the axes the proc modifies. If no option is specified, the proc will modify all of the axes. *axis\_id* may take on one of the following values:

left / l	Left vertical axis.
right / r	Right vertical axis.
bottom / b	Bottom axis for XY and scatter graphs ( <a href="#">scat (p. 412)</a> , <a href="#">xy (p. 528)</a> , <a href="#">xyligne (p. 530)</a> , <a href="#">xypair (p. 532)</a> ).
top / t	Top axis for XY and scatter graphs ( <a href="#">scat (p. 412)</a> , <a href="#">xy (p. 528)</a> , <a href="#">xyligne (p. 530)</a> , <a href="#">xypair (p. 532)</a> ).
all / a	All axes.

Note: the syntax of `scale` has changed considerably from version 4.1 of EViews (see, in particular [axis \(p. 213\)](#)). While not documented here, the previous options are still (for the most part) supported. However, we do not recommend using the old options as future support is not guaranteed.

## Options

The options list may include any of the following options:

### *Data scaling options*

linear	Linear data scaling ( <i>default</i> ).
linearzero	Linear data scaling (include zero when auto range selection is employed).
log	Logarithmic scaling (does not apply to boxplots).
norm	Norm (standardize) the data prior to plotting (does not apply to boxplots).

### *Axes scaling options*

range(arg)	Specifies the endpoints for the scale: automatic choice (“auto”), use the maximum and minimum values of the data (“minmax”), set minimum to <i>n1</i> and maximum to <i>n2</i> (“ <i>n1, n2</i> ”), e.g. “range(3, 9)”.
overlap / -overlap	[Overlap / Do not overlap] scales on dual scale graphs.
lap	
invert / -invert	[Invert / do not invert] scale.

Note that the default settings are taken from the global defaults.

## Examples

To set the right scale to logarithmic with manual range, you can enter:

```
graph1.scale(right) log range(10, 30)
```

Alternatively,

```
graph1.scale -invert range(minmax)
```

draws duplicate inverted scales on the left and right axes which are defined to match the data range.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graph options.

See also [axis \(p. 213\)](#), [datelabel \(p. 252\)](#), [options \(p. 358\)](#) and [setelem \(p. 426\)](#).

scat	<a href="#">Command</a>    <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a>   <a href="#">Sym View</a>
------	---

Display scatterplot graph of object, or change existing graph object type to scatterplot (if possible).

By default, the first series or column of data will be located along the horizontal axis, and the remaining data on the vertical axis. You may optionally choose to plot the data in pairs, where the first two series or columns are plotted against each other, the second two series or columns are plotted against each other, and so forth.

## Syntax

Command:        `scat(options) arg1 [arg2 arg3 ...]`

Object View:     `group_name.scat(options)`

Graph Proc:     `graph_name.scat(options)`

If used as a command, follow the keyword by a list of series and group objects, or by a matrix object. There must be at least two series or columns in the data to be graphed.

Scatterplots are simply XY plots (see [xy \(p. 528\)](#)) with symbols turned on, and lines turned off (see [setelem \(p. 426\)](#)).

## Options

*Template and printing options*

`o = graph_name`   Use appearance options from the specified graph object.

**t = graph\_name** Use appearance options and copy text and shading from the specified graph.

**p** Print the scatter plot.

Note that the use of a template option will override the symbol setting.

#### Scale options

<b>a (default)</b>	Automatic single scale.
<b>b</b>	Plot series or columns in pairs (the first two against each other, the second two against each other, and so forth).
<b>n</b>	Normalized scale (zero mean and unit standard deviation). May not be used with the “s” option.
<b>d</b>	Dual scaling with no crossing.
<b>x</b>	Dual scaling with possible crossing.
<b>m</b>	Display XY plots in multiple graphs (will override the “s” option). Not for use with an existing graph object.

#### Panel options

The following options apply when graphing panel structured data.

**panel = arg**  
 (default taken  
 from global set-  
 tings) Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “medmxmn” (plot median, max and min).

## Examples

```
scat unemp inf want
```

produces an UNTITLED graph object containing a scatter plot, with UNEMP on the horizontal and INF and WANT on the vertical axis.

```
group med age height weight  
med.scat(t=scat2)
```

produces a scatter plot view of the group object MED, using the graph object SCAT2 as a template.

```
group pairs age height weight length  
pairs.scat(b)
```

produces a scatter plot view with AGE plotted against HEIGHT, and WEIGHT plotted against LENGTH.

If there is an existing graph GRAPH01, the expression:

```
graph01.scat(b)
```

changes its type to a scatterplot with data plotted in pairs (if possible), with the remaining XY graph settings at their default values.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graphs and templates.

See [xy \(p. 528\)](#), and [graph \(p. 303\)](#) for graph declaration and modification, and additional graph types. See also [xyline \(p. 530\)](#) for XY line graphs.

<b>scatmat</b>	<a href="#">Group View</a>
----------------	----------------------------

**Matrix of scatter plots.**

The scatmat view displays a matrix of scatter plots for all pairs of series in a group.

## Syntax

Group View:      `group_name.scatmat(options)`

## Options

p

Print the scatter plot matrix.

## Examples

```
group g1 weight height age
g1.scatmat
```

displays a  $3 \times 3$  matrix of scatter plots for all pairs of the three series in group G1.

## Cross-references

See “[Scatter](#)” on page [364](#) of the *User’s Guide* for a discussion of scatter plot matrices.

scenario	Model Proc
----------	------------

Manage the model scenarios.

The scenario procedure is used to set the active and comparison scenarios for a model, to create new scenarios, to initialize one scenario with settings from another scenario, to delete scenarios, and to change the variable aliasing associated with a scenario.

## Syntax

Model Proc:      `model_name.scenario(options) "name"`

performs scenario options on a scenario given by the specified name (entered in double quotes). By default the scenario procedure also sets the active scenario to the specified name.

## Options

<code>c</code>	Set the comparison scenario to the named scenario.
<code>n</code>	Create a new scenario with the specified name.
<code>i = "name"</code>	Copy the Excludes and Overrides from the named scenario.
<code>d</code>	Delete the named scenario.
<code>a = string</code>	Set the scenario alias string to be used when creating aliased variables ( <i>string</i> is a 1 to 3 alphanumeric string to be used in creating aliased variables). If an underscore is not specified, one will be added to the beginning of the string. Examples: “_5”, “_T”, “S2”. The string “A” may not be used since it may conflict with add factor specifications.

## Examples

The command string,

```
mod1.scenario "baseline"
```

sets the active scenario to the baseline, while:

```
mod1.scenario(c) "actuals"
```

sets the comparison scenario to the actuals (warning: this action will overwrite any historical data in the solution period).

A newly created scenario will become the active scenario. Thus:

```
mod1(n) "Peace Scenario"
```

creates a scenario called "Peace Scenario" and makes it the active scenario. The scenario will automatically be assigned a unique numeric alias. To change the alias, simply use the "a = " option:

```
mod1(a=_ps) "Peace Scenario"
```

changes the alias for "Peace Scenario" to "\_PS" and makes this scenario the active scenario.

The command:

```
mod1.scenario(n, a=w, i="Peace Scenario", c) "War Scenario"
```

creates a scenario called "War Scenario", initializes it with the Excludes and Overrides contained in "Peace Scenario", associates it with the alias "\_W", and makes this scenario the comparison scenario.

```
mod1.scenario(i="Scenario 1") "Scenario 2"
```

copies the Excludes and Overrides in "Scenario 1" to "Scenario 2" and makes "Scenario 2" the active scenario.

### *Compatibility Notes*

For backward compatibility with EViews 4, the single character option "a" may be used to set the comparison scenario, but future support for this option is not guaranteed.

In all of the arguments above the quotation marks around scenario name are currently optional. Support for the non-quoted names is provided for backward compatibility, but may be dropped in the future, thus

```
mod1.scenario Scenario 1
```

is currently valid, but may not be in future versions of EViews.

## Cross-references

Scenarios are described in detail beginning on [page 776](#) of the *User's Guide*. [Chapter 26, "Models", on page 761](#) of the *User's Guide* documents EViews models in great depth.

See also [solve \(p. 451\)](#).

<b>seas</b>	<a href="#">Command</a>    <a href="#">Series Proc</a>
-------------	--

Seasonal adjustment.

The `seas` command carries out seasonal adjustment using either the ratio to moving average, or the difference from moving average technique.

EViews also performs Census X11 and X12 seasonal adjustment. For details, see [x11 \(p. 520\)](#) and [x12 \(p. 522\)](#).

## Syntax

Command:      `seas(options) series_name name_adjust [name_fac]`

Series Proc:    `series_name.seas(options) name_adjust [name_fac]`

To use `seas` as a command, list the name of the original series and the name to be given to the seasonally adjusted series. You may optionally include an additional name for the seasonal factors. `seas` will display the seasonal factors using the convention of the Census X11 program.

`seas` used as a series procedure applies seasonal adjustment to a series.

## Options

m	Multiplicative (ratio to moving average) method.
---	--

a	Additive (difference from moving average) method.
---	---

## Examples

```
seas(a) pass pass_adj pass_fac
```

seasonally adjusts the series PASS using the additive method, and saves the adjusted series as PASS\_ADJ and the seasonal factors as PASS\_FAC.

```
sales.seas(m) adj_sales
```

seasonally adjusts the series SALES using the multiplicative method and saves the adjusted series as ADJ\_SALES.

## Cross-references

See “[Seasonal Adjustment](#)” on page 324 of the *User’s Guide* for a discussion of seasonal adjustment methods.

See also [seasplot](#) (p. 418), [x11](#) (p. 520) and [x12](#) (p. 522).

<b>seasplot</b>	<a href="#">Series View</a>
-----------------	-----------------------------

Seasonal line graph.

`seasplot` displays a line graph view of a series ordered by season. Available only for quarterly and monthly frequencies.

## Syntax

Series View:      `series_name.seasplot(options)`

## Options

**m**                  Plot series split by season. Default is to plot series stacked by season.

## Examples

```
freeze(gra_ip) ipnsa.seasplot
```

creates a graph object named GAR\_IP that contains the stacked seasonal line graph view of the series IPNSA.

## Cross-references

See “[Spreadsheet and Graph Views](#)” on page 297 of the *User’s Guide* for a brief discussion of seasonal line graphs.

See also [seas](#) (p. 417), [x11](#) (p. 520) and [x12](#) (p. 522).

<b>series</b>	<a href="#">Object Declaration</a>
---------------	------------------------------------

Declare a series object.

The `series` command creates and optionally initializes a series, or modifies an existing series.

## Syntax

Command:      **series** *ser\_name[ = formula ]*

The **series** command should be followed by either the name of a new series, or an explicit or implicit expression for generating a series. If you create a series and do not initialize it, the series will be filled with NAs. Rules for composing a formula are given in “[Numeric Expressions](#)” on page 121 of the *User’s Guide*.

## Examples

```
series x
```

creates a series named X filled with NAs.

Once a series is declared, you do not need to include the **series** keyword prior to entering the formula. The following example generates a series named LOW that takes value 1 if either INC is less than or equal to 5000 or EDU is less than 13, and 0 otherwise.

```
series low
low = inc<=5000 or edu<13
```

This example solves for the implicit relation and generates a series named Z which is the double log of Y so that  $Z = \log(\log(Y))$ .

```
series exp(exp(z)) = y
```

The command:

```
series z = (x+y)/2
```

creates a series named Z which is the average of series X and Y.

```
series cwage = wage*(hrs>5)
```

generates a series named CWAGE which is equal to WAGE if HRS exceeds 5, and zero otherwise.

```
series 10^z = y
```

generates a series named Z which is the base 10 log of Y.

The commands:

```
series y_t = y
smp1 if y<0
y_t = na
smp1 @all
```

generate a series named Y\_T which replaces negative values of Y with NAs.

```
series z = @movav(x(+2),5)
```

creates a series named Z which is the *centered* moving average of the series X with two leads and two lags.

```
series z = (.5*x(6)+@movsum(x(5),11)+.5*x(-6))/12
```

generates a series named Z which is the *centered* moving average of the series X over twelve periods.

```
genr y = 2+(5-2)*rnd
```

creates a series named Y which is a random draw from a uniform distribution between 2 and 5.

```
series y = 3+@sqr(5)*nrnd
```

generates a series named Y which is a random draw from a normal distribution with mean 3 and variance 5.

## Cross-references

There is an extensive set of functions that you may use with series:

- A list of mathematical functions is presented in [Appendix D, “Operator and Function Reference”, on page 543](#).
- Workfile functions that provide information about observations identifiers or allow you to construct time trends are described in [Appendix E, “Workfile Functions”, on page 559](#).
- Functions for working with strings and dates are documented in [“String Function Summary” on page 127](#) and [“Date Function Summary” on page 150](#).

See [“Numeric Expressions” on page 121](#) of the *User’s Guide* for a discussion of rules for forming EViews expressions.

<b>set</b>	<a href="#">Sample Proc</a>
------------	-----------------------------

Set the sample in a sample object.

The **set** procedure resets the sample of an existing sample object.

## Syntax

Sample Proc:      *sample\_name.set sample\_description*

Follow the **set** command with a sample description. See **sample** for instructions on describing a sample.

## Examples

```
sample s1 @first 1973
s1.set 1974 @last
```

The first line declares and defines a sample object named S1 from the beginning of the workfile range to 1973. The second line resets S1 from 1974 to the end of the workfile range.

## Cross-references

See “[Samples](#)” on page 87 of the *User’s Guide* for a discussion of samples in EViews.

See also [sample \(p. 406\)](#) and [smp1 \(p. 449\)](#).

<b>setbpelem</b>	<a href="#">Graph Proc</a>
------------------	----------------------------

Enable/disable individual boxplot elements.

## Syntax

Graph Proc:      graph\_name.setbpelem *element\_list*

The *element\_list* may contain one or more of the following:

median, med / -median, -med	[Show / Do not show] the medians.
mean / -mean	[Show / Do not show] the means.
whiskers, w / -whiskers, -w	[Show / Do not show] the whiskers (lines from the box to the staples).
staples, s / -staples, -s	[Show / Do not show] the staples (lines drawn at the last data point within the inner fences).
near / -near	[Show / Do not show] the near outliers (values between the inner and outer fences).
far / -far	[Show / Do not show] the far outliers (values beyond the outer fences).
width( <i>arg</i> ) ( <i>default</i> = “fixed”)	Set the width settings for the boxplots, where <i>arg</i> is one of: “fixed” (uniform width), “n” (proportional to sample size), “rootn” (proportional to the square root of sample size).

`ci = arg` Set the display method for the confidence intervals,  
`(default =` where *arg* is one of: “none” (do not display), “shade”  
`“shade”)` (shaded intervals), “notch” (notched intervals).  
`ci = arg (default`  
`= “shade”)`

## Examples

```
graph01.setbpelem -far width(n) ci(notch)
```

hides the far outliers, sets the box widths proportional to the number of observations, and enables notching of the confidence intervals.

## Cross-references

See “[Boxplots](#)” on page 397 of the *User’s Guide* for a description of boxplots.

See [setelem](#) (p. 426) to modify line and symbol attributes. See also [bplabel](#) (p. 226), [options](#) (p. 358), [axis](#) (p. 213), and [scale](#) (p. 410).

setcell	Command
---------	---------

Insert contents into cell of a table.

The `setcell` command puts a string or number into a cell of a table.

## Syntax

Command: `setcell(table_name, r, c, content[, "options"])`

## Options

Provide the following information in parentheses in the following order: the name of the table object, the row number, *r*, of the cell, the column number, *c*, of the cell, a number or string to put in the cell, and optionally, a justification and/or numerical format code. A string of text must be enclosed in double quotes.

The justification options are:

`c` Center the text/number in the cell.

`r` Right-justify the text/number in cell.

`l` Left-justify the text/number in cell.

The numerical format code determines the format with which a number in a cell is displayed; cells containing strings will be unaffected. The format code can either be a positive

integer, in which case it specifies the number of decimal places to be displayed after the decimal point, or a negative integer, in which case it specifies the total number of characters to be used to display the number. These two cases correspond to the **Fixed decimal** and **Fixed character** fields in the number format dialog.

Note that when using a negative format code, one character is always reserved at the start of a number to indicate its sign, and that if the number contains a decimal point, that will also be counted as a character. The remaining characters will be used to display digits. If the number is too large or too small to display in the available space, EViews will attempt to use scientific notation. If there is insufficient space for scientific notation (six characters or less), the cell will contain asterisks to indicate an error.

## Examples

```
setcell(tab1, 2, 1, "Subtotal")
```

puts the string "Subtotal" in row 2, column 1 of the table object named TAB1.

```
setcell(tab1, 1, 1, "Price and cost", "r")
```

puts the a right-justify string "Price and cost" in row 1, column 1 of the table object named TAB1.

## Cross-references

[Chapter 4](#) describes table formatting using commands. See [Chapter 14](#) of the *User's Guide* for a discussion and examples of table formatting in EViews.

See also [setwidth \(p. 444\)](#).

<b>setcolwidth</b>	<a href="#">Command</a>
--------------------	-------------------------

Set width of a column of a table.

Provided for backward compatibility. See [setwidth \(p. 444\)](#) for the new method of setting the width of table and spreadsheet columns.

## Syntax

Command:      **setcolwidth**(*table\_name, c, width*)

## Options

To change the width of a column, provide the following information in parentheses, in the following order: the name of the table, the column number *c*, and the number of characters *width* for the new width. EViews measures units in terms of the width of a numeric character. Because different characters have different widths, the actual number of characters

that will fit may differ slightly from the number you specify. By default, each column is approximately 10 characters wide.

## Examples

```
setcolwidth (mytab, 2, 20)
```

sets the second column of table MYTAB to fit approximately 20 characters.

## Cross-references

[Chapter 4, “Working with Tables”, on page 47](#) describes table formatting using commands. See also [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion and examples of table formatting in EViews.

See also [setwidht \(p. 444\)](#) and [setheight \(p. 436\)](#).

<b>setconvert</b>	<a href="#">Series Proc</a>
-------------------	-----------------------------

**Set frequency conversion method.**

Determines the default frequency conversion method for a series when copied or linked between different frequency workfiles.

You may override this default conversion method by specifying a frequency conversion method as an option in the specific command (using [copy \(p. 244\)](#), [fetch \(p. 279\)](#), or [linkto \(p. 324\)](#)).

If you do not set a conversion method and if you do not specify a conversion method as an option in the command, EViews will use the conversion method set in the global option.

## Syntax

Series Proc:        ser\_name.setconvert [*up\_method down\_method*]

Follow the series name with a period, the keyword, and option letters to specify the frequency conversion method. If either the up-conversion or down-conversion method is omitted, EViews will set the corresponding method to **Use EViews default**.

## Options

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *low* to *high* frequency:

Low to high conversion methods	“r” (constant match average), “d” (constant match sum), “q” (quadratic match average), “t” (quadratic match sum), “i” (linear match last), “c” (cubic match last).
--------------------------------	--

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *high* to *low* frequency:

High to low conversion methods	<p><i>High to low conversion methods removing NAs</i>: “a” (average of the nonmissing observations), “s” (sum of the nonmissing observations), “f” (first nonmissing observation), “l” (last nonmissing observation), “x” (maximum nonmissing observation), “m” (minimum nonmissing observation).</p> <p><i>High to low conversion methods propagating NAs</i>: “an” or “na” (average, propagating missings), “sn” or “ns” (sum, propagating missings), “fn” or “nf” (first, propagating missings), “ln” or “nl” (last, propagating missings), “xn” or “nx” (maximum, propagating missings), “mn” or “nm” (minimum, propagating missings).</p>
--------------------------------	--

## Examples

```
unemp.setconvert a
```

sets the default down-conversion method of the series UNEMP to take the average of non-missing observations, and resets the up-conversion method to use the global default.

```
ibm_hi.setconvert xn d
```

sets the default down-conversion method for IBM\_HI to take the largest observation of the higher frequency observations, propagating missing values, and the default up-conversion method to constant, match sum.

```
consump.setconvert
```

resets both methods to the global default.

## Cross-references

See “[Frequency Conversion](#)” on page 107 of the *User’s Guide* for a discussion of frequency conversion and the treatment of missing values.

See also [copy](#) (p. 244), [fetch](#) (p. 279) and [linkto](#) (p. 324).

**setelem****Graph Proc**

Set individual line, bar and legend options for each series in the graph.

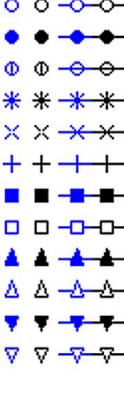
**Syntax**

Graph Proc:      `graph_name.setelem(graph_elem) argument_list`

where *graph\_elem* is the identifier for the graph element whose options you wish to modify:

<i>integer</i>	Index for graph element (for non-boxplot graphs). For example, if you provide the integer “2”, EViews will modify the second line in the graph.
<i>box_elem</i>	Boxplot element to be modified: box (“b”), median (“med”), mean (“mean”), near outliers (“near” or “no”), far outliers (“far” or “fo”), whiskers (“w”), staples (“s”). For boxplot graphs only.

The *argument list* for `setelem` may contain one or more of the following:

<code>symbol(arg)</code>	Sets the drawing symbol: <i>arg</i> can be an integer from 1–13, or one of the matching keywords. Selecting a symbol automatically turns on symbol use. The “none” option turns off symbol use.	(1) circle (2) filledcircle (3) transcircle (4) star (5) diagcross (6) cross (7) fillsquare (8) square (9) filledtriup (10) triup (11) filledtridown (12) tridown (13) none	
--------------------------	---	---	---

<code>linecolor(args), lcolor(args)</code>	Sets the line and symbol color. The <i>args</i> value may set by using one of the color keywords (e.g., “blue”), or by using the RGB values (e.g., “@RGB(255, 255, 0)”). For a description of the available color keywords, see <a href="#">setfillcolor (p. 429)</a> .
--	---

<code>linewidth(<i>n1</i>), lwidth(<i>n1</i>)</code>	Sets the line and symbol width: <i>n1</i> should be a number between “.25” and “5”, indicating the width in points.																														
<code>linepattern(<i>arg</i>), lpat(<i>arg</i>)</code>	<p>Sets the line pattern to the type specified by <i>arg</i>. <i>arg</i> can be an integer from 1–11 or one of the matching keywords.</p> <p>Note that the option interacts with the graph options for “color”, “lineauto”, “linesolid”, “linepat” (see <a href="#">options (p. 358)</a>, for details). You may need to set the graph option for “linepat” to enable the display of line patterns. See <a href="#">options (p. 358)</a>.</p> <p>The “none” option turns off lines and uses only symbols.</p>																														
<code>fillcolor(<i>arg</i>), fcolor(<i>arg</i>)</code>	Sets the fill color for symbols, bars, and pies. The <i>args</i> value may set by using of the color keywords (e.g., “blue”), or by using the RGB values (e.g., “@RGB(255, 255, 0)”). For a description of the available color keywords, see <a href="#">setfillcolor (p. 429)</a> .																														
<code>fillgray(<i>n1</i>), gray(<i>n1</i>)</code>	Sets the gray scale for bars and pies: <i>n1</i> should be an integer from 1–15 corresponding to one of the predefined gray scale settings (from lightest to darkest).																														
	<table> <tbody> <tr><td>1</td><td></td></tr> <tr><td>2</td><td></td></tr> <tr><td>3</td><td></td></tr> <tr><td>4</td><td></td></tr> <tr><td>5</td><td></td></tr> <tr><td>6</td><td></td></tr> <tr><td>7</td><td></td></tr> <tr><td>8</td><td></td></tr> <tr><td>9</td><td></td></tr> <tr><td>10</td><td></td></tr> <tr><td>11</td><td></td></tr> <tr><td>12</td><td></td></tr> <tr><td>13</td><td></td></tr> <tr><td>14</td><td></td></tr> <tr><td>15</td><td></td></tr> </tbody> </table>	1		2		3		4		5		6		7		8		9		10		11		12		13		14		15	
1																															
2																															
3																															
4																															
5																															
6																															
7																															
8																															
9																															
10																															
11																															
12																															
13																															
14																															
15																															

<code>fillhatch(arg), hatch(arg)</code>	Sets the hatch characteristics for bars and pies: <i>arg</i> can be an integer from 1–7, or one of the matching keywords.	<table border="1"> <tbody> <tr><td>(1) none</td><td></td></tr> <tr><td>(2) diagcross</td><td></td></tr> <tr><td>(3) horizontal</td><td></td></tr> <tr><td>(4) fdiagonal</td><td></td></tr> <tr><td>(5) vertical</td><td></td></tr> <tr><td>(6) cross</td><td></td></tr> <tr><td>(7) bdiagonal</td><td></td></tr> </tbody> </table>	(1) none		(2) diagcross		(3) horizontal		(4) fdiagonal		(5) vertical		(6) cross		(7) bdiagonal	
(1) none																
(2) diagcross																
(3) horizontal																
(4) fdiagonal																
(5) vertical																
(6) cross																
(7) bdiagonal																
<code>preset(<i>n1</i>)</code>	Sets line and fill characteristics to the specified EViews preset values, where <i>n1</i> is an integer from 1–30. Simultaneously sets “linecolor”, “linepattern”, “linewidth”, “symbol”, “fillcolor”, “fillgray”, and “fillhatch” to the EViews predefined definitions for graph element <i>n1</i> .  When applied to boxplots, the line color of the specified element will be applied to the box, whiskers, and staples.															
<code>default(<i>n1</i>)</code>	Sets line and fill characteristics to the specified user-defined default settings where <i>n1</i> is an integer from 1–30. Simultaneously sets “linecolor”, “linepattern”, “linewidth”, “symbol”, “fillcolor”, “fillgray”, and “fillhatch” to the values in the user-defined global defaults for graph element <i>n1</i> .  When applied to boxplots, the line color of the specified settings will be applied to the box, whiskers, and staples.															
<code>axis(<i>arg</i>), axisscale(<i>arg</i>)</code>	Assigns the element to an axis: left (“l”), right (“r”), bottom (“b”), top (“t”). The latter two options are only applicable for XY and scatter graphs ( <a href="#">scat (p. 412)</a> , <a href="#">xy (p. 528)</a> , <a href="#">xyline (p. 530)</a> , <a href="#">xypair (p. 532)</a> ).															
<code>legend(<i>str</i>)</code>	Assigns legend text for the element. <i>str</i> will be used in the legend to label the element.															

## Examples

```
graph1.setelem(2) lcolor(blue) linewidth(2) symbol(circle)
sets the second line of GRAPH1 to be a blue line of width 2 with circle symbols.
```

---

```
graph1.setelem(1) lcolor(blue)
graph1.setelem(1) linecolor(0, 0, 255)
```

are equivalent methods of setting the linecolor to blue.

```
graph1.setelem(1) fillgray(6)
```

sets the gray-scale color for the first graph element.

The lines:

```
graph1.setelem(1) scale(1)
graph1.setelem(2) scale(1)
graph1.setelem(3) scale(r)
```

create a dual scale graph where the first two series are scaled together and labeled on the left axis, and the third series is scaled and labeled on the right axis.

```
graph1.setelem(2) legend("gross domestic product")
```

sets the legend for the second graph element.

## Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of graph options in EViews.

See also [axis \(p. 213\)](#), [datelabel \(p. 252\)](#), [scale \(p. 410\)](#) and [options \(p. 358\)](#).

<b>setfillcolor</b>	<a href="#">Table Proc</a>
---------------------	----------------------------

Set the fill (background) color of the specified table cells.

## Syntax

Table Proc:      `table_name.setfillcolor(cell_range) color_arg`

where *cell\_range* can take one of the following forms:

<code>@all</code>	Apply to all cells in the table.
<code>cell</code>	Cell identifier. You can identify cells using either the column letter and row number (e.g., “A1”), or by using “R” followed by the row number followed by “C” and the column <i>number</i> (e.g., “R1C2”). Apply to cell.
<code>row[,] col</code>	Row number, followed by column letter or number (e.g., “2,C”, or “2,3”), separated by “,”. Apply to cell.

<i>row</i>	Row number (e.g., “2”). Apply to all cells in the row.
<i>col</i>	Column <i>letter</i> (e.g., “B”). Apply to all cells in the column.
<i>first_cell[:]:last_cell</i> , <i>first_cell[,]last_cell</i>	Top left cell of the selection range (specified in “ <i>cell</i> ” format), followed by bottom right cell of the selection range (specified in “ <i>cell</i> ” format), separated by a “:” or “,” (e.g., “A2:C10”, “A2,C10”, or “R2C1:R10C3”, “R2C1,R10C3”). Apply to all cells in the rectangular region defined by the first cell and last cell.
<i>first_cell_row[,]</i> <i>first_cell_col[,]</i> <i>last_cell_row[,]</i> <i>last_cell_col</i>	Top left cell of the selection range (specified in “ <i>row[,] col</i> ” format), followed by bottom right cell of the selection range (specified in “ <i>row[,] col</i> ” format), separated by a “,” (e.g., “2,A,10,C” or “2,1,10,3”). Apply to all cells in the rectangular region defined by the first cell and last cell.

The *color\_arg* specifies the color to be applied to the text in the cells. The color may be specified using predefined color names, or by specifying the individual red-green-blue (RGB) components using the special “@RGB” function. The latter method is obviously more difficult, but allows you to use custom colors.

The predefined colors are given by the keywords (with their RGB equivalents):

blue	@rgb(0, 0, 255)
red	@rgb(255, 0, 0)
green	@rgb(0, 128, 0)
black	@rgb(0, 0, 0)
white	@rgb(255, 255, 255)
purple	@rgb(128, 0, 128)
orange	@rgb(255, 128, 0)
yellow	@rgb(255, 255, 0)
gray	@rgb(128, 128, 128)
ltgray	@rgb(192, 192, 192)

## Examples

To set a purple background color for the cell in the second row and third column of TAB1, you may use any of the following:

```
tab1.setfillcolor(C2) @rgb(128, 0, 128)
```

---

```
tab1.setfillcolor(2,C) @RGB(128, 0, 128)
tab1.setfillcolor(2,3) purple
tab1.setfillcolor(r2c3) purple
```

You may also specify a yellow color for the background of an entire column, or an entire row,

```
tab1.setfillcolor(C) @RGB(255, 255, 0)
tab1.setfillcolor(2) yellow
```

or for the background of the cells in a rectangular region:

```
tab1.setfillcolor(R2C3:R3C6) ltgray
tab1.setfillcolor(r2c3,r3c6) ltgray
tab1.setfillcolor(2,C,3,F) @rgb(192, 192, 192)
tab1.setfillcolor(2,3,3,6) @rgb(192, 192, 192)
```

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”, on page 47](#).

See [settextcolor \(p. 443\)](#) and [setfont \(p. 431\)](#) for details on changing text color and font, and [setlines \(p. 440\)](#) for drawing lines between around and through cells. See [setformat \(p. 432\)](#) for setting cell formats.

<b>setfont</b>	<a href="#">Table Proc</a>
----------------	----------------------------

Set the font for text in the specified table cells.

### Syntax

Table Proc:      `table_name.setfont(cell_range) font_args`

where *cell\_range* describes the table cells to be modified, and *font\_args* is a set of arguments used to modify the existing font settings. See [setfillcolor \(p. 429\)](#) for the syntax for *cell\_range*.

The *font\_args* may include one or more of the following:

<i>face_name</i>	A string that specifies the typeface name of the font, enclosed in double quotes.
------------------	---

<i>integer[pt]</i>	Integer font size, in points, followed by the “pt” identifier (e.g., “12pt”).
--------------------	---

<i>+ b / -b</i>	[Add / remove] boldface.
-----------------	--------------------------

+ i / -i	[Add / remove] italics.
+ s / -s	[Add / remove] strikethrough.
+ u / -u	[Add / remove] underscore.

## Examples

```
tab1.setfont(B3:D10) "Times New Roman" +i
```

sets the font to Times New Roman Italic for the cells defined by the rectangle from B3 (row 3, column 2) to D10 (row 10, column 4).

```
tab1.setfont(3,B,10,D) 8pt
```

changes all of text in the region to 8 point.

```
tab1.setfont(4,B) +b -i
```

removes the italic, and adds boldface to the B4 cell (row 4, column 2).

The commands:

```
tab1.setfont(b) -s +u 14pt  
tab1.setfont(2) "Batang" 14pt +u
```

modify the fonts for the column B, and row 2, respectively. The first command changes the point size to 14, removes strikethrough and adds underscoring. The second changes the typeface to Batang, and adds underscoring,

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”](#).

See [settextcolor \(p. 443\)](#) and [setfillcolor \(p. 429\)](#) for details on changing the text and cell background colors. See [setformat \(p. 432\)](#) for setting the cell formats.

## setformat

[Alpha Proc](#) | [Coef Proc](#) | [Group Proc](#) | [Matrix Proc](#) | [Series Proc](#) |  
[Rowvector Proc](#) | [Sym Proc](#) | [Table Proc](#) | [Vector Proc](#)

Set the display format for cells in object spreadsheet and table views.

## Syntax

Group Proc:      `group_name.setformat(col_range) format_arg`

Table Proc:      `table_name.setformat(cell_range) format_arg`

Object Proc:      `object_name.setformat format_arg`

where *format\_arg* is a set of arguments used to specify format settings. If necessary, you should enclose the *format\_arg* in double quotes.

The *cell\_range* option is used to describe the columns to be updated in groups and the cells to be modified in tables. For all other objects, `setformat` operates on all of the cells in the object.

For groups, the *col\_range* may take one of the following forms:

<code>@all</code>	Apply to all series in the group.
<code>col</code>	Column number or letter (e.g., “2”, “B”). Apply to the series corresponding to the column.
<code>first_col[:last_col</code>	Colon delimited range of columns (from low to high, e.g., “3:5”). Apply to all series corresponding to the column range.

For tables, the *cell\_range* may take one of the following forms:

<code>@all</code>	Apply to all cells in the table.
<code>cell</code>	Cell identifier. You can identify cells using either the column letter and row number (e.g., “A1”), or by using “R” followed by the row number followed by “C” and the column number (e.g., “R1C2”). Apply to cell.
<code>row[,] col</code>	Row number, followed by column letter or number (e.g., “2,C”, or “2,3”), separated by “,”. Apply to cell.
<code>row</code>	Row number (e.g., “2”). Apply to all cells in the row.
<code>col</code>	Column letter (e.g., “B”). Apply to all cells in the column.
<code>first_cell[:last_cell,</code> <code>first_cell[,]last_cell</code>	Top left cell of the selection range (specified in “cell” format), followed by bottom right cell of the selection range (specified in “cell” format), separated by a “:” or “,” (e.g., “A2:C10”, “A2,C10”, or “R2C1:R10C3”, “R2C1,R10C3”). Apply to all cells in the rectangular region defined by the first cell and last cell.
<code>first_cell_row[,]</code> <code>first_cell_col[,]</code> <code>last_cell_row[,]</code> <code>last_cell_col</code>	Top left cell of the selection range (specified in “row[,] col” format), followed by bottom right cell of the selection range (specified in “row[,] col” format), separated by a “,” (e.g., “2,A,10,C” or “2,1,10,3”). Apply to all cells in the rectangular region defined by the first cell and last cell.

To format numeric values, you should use one of the following format specifications:

<code>g[.precision]</code>	significant digits
<code>f[.precision]</code>	fixed decimal places
<code>c[.precision]</code>	fixed characters
<code>e[.precision]</code>	scientific/float
<code>p[.precision]</code>	percentage
<code>r[.precision]</code>	fraction

In order to set a format that groups digits into thousands, place a “t” after a specified format. For example to obtain a fixed number of decimal places, with commas used to separate thousands use “`ft[.precision]`”. To obtain a fixed number of characters, with a period used to separate thousands use “`ct[.precision]`”.

If you wish to display negative numbers surrounded by parentheses (*i.e.*, display the number -37.2 as “(37.2)”), then you should enclose the format string in “()” (*e.g.*, “`f(.8)`”).

To format numeric values using date and time formats, you may use a subset of the possible date format strings (see “[Date Formats](#) on page 130”). The possible format arguments, along with an example of the date number 730856.944793113 (January 7, 2002 10:40:30.125 p.m) formatted using the argument are given by:

<code>WF</code>	(uses current EViews workfile date display format)
<code>YYYY</code>	“2002”
<code>YYYY-Mon</code>	“2002-Jan”
<code>YYYYMon</code>	“2002 Jan”
<code>YYYY[M]MM</code>	“2002[M]01”
<code>YYYY:MM</code>	“2002:01”
<code>YYYY[Q]Q</code>	“2002[Q]1”
<code>YYYY:Q</code>	“2002:Q”
<code>YYYY[S]S</code>	“2002[S]1” (semi-annual)
<code>YYYY:S</code>	“2002:1”
<code>YYYY-MM-DD</code>	“2002-01-07”
<code>YYYY Mon dd</code>	“2002 Jan 7”
<code>YYYY Month dd</code>	“2002 January 7”
<code>YYYY-MM-DD HH:MI</code>	“2002-01-07 22:40”
<code>YYYY-MM-DD HH:MI:SS</code>	“2002-01-07 22:40:30”

YYYY-MM-DD HH:MI:SS.SSS	“2002-01-07 22:40:30.125”
Mon-YYYY	“Jan-2002”
Mon dd YYYY	“Jan 7 2002”
Mon dd, YYYY	“Jan 7, 2002”
Month dd YYYY	“January 7 2002”
Month dd, YYYY	“January 7, 2002”
MM/DD/YYYY	“01/07/2002”
mm/DD/YYYY	“1/07/2002”
mm/DD/YYYY HH:MI	“1/07/2002 22:40”
mm/DD/YYYY HH:MI:SS	“1/07/2002 22:40:30”
mm/DD/YYYY HH:MI:SS.SSS	“1/07/2002 22:40:30.125”
mm/dd/YYYY	“1/7/2002”
mm/dd/YYYY HH:MI	“1/7/2002 22:40”
mm/dd/YYYY JJ:MI:SS	“1/7/2002 22:40:30”
mm/dd/YYYY hh:MI:SS.SSS	“1/7/2002 22:40:30.125”
dd/MM/YYYY	“7/01/2002”
dd/MM/YYYY HH:MI	“7/01/2002 22:40”
dd/MM/YYYY HH:MI:SS	“7/01/2002 22:40:30”
dd/MM/YYYY HH:MI:SS.SSS	“7/01/2002 22:40:30.125”
hm:MI am	“10:40 pm”
hm:MI:SS am	“10:40:30 pm”
hm:MI:SS.SSS am	“10:40:30.125 pm”
HH:MI	“22:40”
HH:MI:SS	“22:40:30”
HH:MI:SS.SSS	“22:40:30.125”

## Examples

To set the format for a series or a matrix object to fixed 5-digit precision, simply provide the format specification:

```
ser1.setformat f.5
matrix1.setformat f.5
```

For groups, provide the column identifier and format. The commands:

```
group1.setformat(2) f(.7)
group1.setformat(c) e.5
```

set the formats for the second and third series in the group.

You may use any of the date formats given above:

```
ser2.setformat YYYYMon  
group1(d).setformat "YYYY-MM-DD HH:MI:SS.SSS"
```

to set the series display characteristics.

For tables, you must provide a valid cell specification:

```
tab1.setformat(B2) hh:MI:SS.SSS  
tab1.setformat(2,B,10,D) g(.3)  
tab1.setformat(R2C2:R4C4) "dd/MM/YY HH:MI:SS.SSS"
```

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”](#).

See also [setfont \(p. 431\)](#), [settextcolor \(p. 443\)](#) and [setfillcolor \(p. 429\)](#) for details on changing the text font and text and cell background colors.

## setheight

### [Table Proc](#)

Set the row height of rows in a table.

#### Syntax

Table Proc:      `table_name.setheight(row_range) height_arg`

where *row\_range* is either a single row number (e.g., “5”), a colon delimited range of rows (from low to high, e.g., “3:5”), or the “@ALL” keyword, and *height\_arg* specifies the height unit value, where height units are specified in character heights. The character height is given by the font-specific sum of the units above and below the baseline and the leading, where the font is given by the default font for the current table (the EViews table default font at the time the table was created). *height\_arg* values may be non-integer values with resolution up to 1/10 of a height unit.

#### Examples

```
tab1.setheight(2) 1
```

sets the height of row 2 to match the table font character height, while:

```
tab1.setheight(2) 1.5
```

increases the row height to 1-1/2 character heights.

Similarly, the command:

```
tab1.setheight(2:4) 1
```

sets the heights for rows 2 through 4.

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”, beginning on page 47](#).

For details on setting the column widths in a table, see [setwidth \(p. 444\)](#).

<b>setindent</b>	<a href="#">Alpha Proc</a>   <a href="#">Coef Proc</a>   <a href="#">Group Proc</a>   <a href="#">Matrix Proc</a>   <a href="#">Series Proc</a>   <a href="#">Rowvector Proc</a>   <a href="#">Sym Proc</a>   <a href="#">Table Proc</a>   <a href="#">Vector Proc</a>
------------------	---

Set the display indentation for cells in object spreadsheet and table views.

## Syntax

Group Proc: `group_name.setindent(col_range) indent_arg`

Table Proc: `table_name.setindent(cell_range) indent_arg`

Object Proc: `object_name.setindent indent_arg`

where *indent\_arg* is an indent value specified in 1/5 of a width unit. The width unit is computed from representative characters in the default font for the current table (the EViews table default font at the time the table was created), and corresponds roughly to a single character. Indentation is only relevant for non-center justified cells.

The default value is taken from the Global Defaults at the time the view or table is created.

The *col\_range* option is used to describe the columns to be updated in groups and the *cell\_range* defines the cells to be modified in tables. For all other objects, *setformat* operates on all of the cells in the object. See [setformat \(p. 432\)](#) for the syntax for group *col\_range* and table *cell\_range* specifications.

## Examples

To set the justification for a series or a matrix object:

```
ser1.setindent 2
matrix1.setindent top 4
```

For groups, you should provide the column identifier, and the format. The commands,

```
group1.setindent(2) 3
group1.setindent(c) 2
```

set the formats for the second and third series in the group, while:

```
group2.setindent(@all) 3
```

sets formats for all of the series.

For tables, you must provide a valid cell specification:

```
tab1.setindent(@all) 2  
tab1.setindent(2,B,10,D) 4  
tab1.setindent(R2C2:R4C4) 2
```

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”, on page 47](#).

See [setWidth \(p. 444\)](#) and [setjust \(p. 438\)](#) for details on setting spreadsheet and table widths and justification.

<b>setjust</b>	<a href="#">Alpha Proc</a>   <a href="#">Coef Proc</a>   <a href="#">Group Proc</a>   <a href="#">Matrix Proc</a>   <a href="#">Series Proc</a>   <a href="#">Rowvector Proc</a>   <a href="#">Sym Proc</a>   <a href="#">Table Proc</a>   <a href="#">Vector Proc</a>
----------------	---

Set the display justification for cells in object spreadsheet and table views.

## Syntax

Group Proc:      `group_name.setjust(col_range) format_arg`

Table Proc:      `table_name.setjust(cell_range) format_arg`

Object Proc:      `object_name.setjust format_arg`

where *format\_arg* is a set of arguments used to specify format settings. You should enclose the *format\_arg* in double quotes if it contains any spaces or delimiters.

The *col\_range* option is used to describe the columns to be updated in groups and the *cell\_range* defines the cells to be modified in tables. For all other objects, setformat operates on all of the cells in the object. See [setformat \(p. 432\)](#) for the syntax for group *col\_range* and table *cell\_range* specifications.

The *format\_arg* may be formed using the following:

`top / middle / bottom` ] Vertical justification setting.

`auto / left / center / right` Horizontal justification setting. “Auto” uses left justification for strings, and right for numbers.

You may enter one or both of the justification settings. The default settings are object specific: series and matrix object defaults are taken from the Global Defaults for spreadsheet views; table defaults are taken from the original view when created by freezing a view, or as “middle bottom” for newly created tables.

## Examples

To set the justification for a series or a matrix object:

```
ser1.setjust middle
matrix1.setjust top left
```

For groups, you should provide the column identifier, and the format. The commands,

```
group1.setjust(2) bottom center
group1.setjust(c) center middle
```

set the formats for the second and third series in the group, while:

```
group2.setjust(@all) right
```

sets all of the series formats.

For tables, you must provide a valid cell specification:

```
tab1.setjust(@all) top
tab1.setjust(2,B,10,D) left bottom
tab1.setjust(R2C2:R4C4) right top
```

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”, on page 47](#).

See [setWidth \(p. 444\)](#) and [setindent \(p. 437\)](#) for details on setting spreadsheet and table widths and indentation.

<b>setline</b>	<a href="#">Command</a>
----------------	-------------------------

Place a double horizontal line in a table.

Provided for backward compatibility. For a more general method of setting the line characteristics and borders for a set of table cells, see the table proc [setlines \(p. 440\)](#).

## Syntax

Command: `setline(table_name, r)`

## Options

Specify the name of the table and the row number  $r$  in which to place the horizontal line.

## Examples

```
setline(tab3,8)
```

places a (double) horizontal line in the eighth row of the table object TAB3.

## Cross-references

[Chapter 4, “Working with Tables”, on page 47](#) describes table formatting using commands. See also [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion and examples of table formatting in EViews.

See [setlines \(p. 440\)](#) for more flexible line drawing tools.

<b>setlines</b>	<a href="#">Table Proc</a>
-----------------	----------------------------

Sets line characteristics and borders for a set of table cells.

## Syntax

Table Proc:      `table_name.setlines(cell_range) line_args`

where *cell\_range* describes the table cells to be modified, and *line\_args* is a set of arguments used to modify the existing line settings. See [setfillcolor \(p. 429\)](#) for the syntax for *cell\_range*.

The *line\_args* may contain one or more of the following:

+ t / -t	Top border [on/off].
+ b / -b	Bottom border [on/off].
+ l / -l	Left border [on/off].
+ r / -r	Right border [on/off].
+ i / -i	Inner borders [on/off].
+ o / -o	Outer borders [on/off].
+ v / -v	Vertical inner borders [on/off].
+ h / -h	Horizontal inner borders [on/off].
+ a / -a	All borders [on/off].
+ d / -d	Double middle lines [on/of]f.

## Examples

```
tab1.setlines(b2:d6) +o
```

draws borders around the outside of the rectangle defined by B2 and D6. Note that this command is equivalent to:

```
tab1.setlines(b2:d6) +a -h -v
```

which adds borders to all of the cells in the rectangle defined by B2 and D6, then removes the inner horizontal and vertical borders.

```
tab1.setlines(2,b) +o
```

puts a border around all four sides of the B2 cell.

```
tab1.setlines(2,b) -l -r +i
```

then removes both the left and the right border from the cell. In this example, “+i” option has no effect; since the specification involves a single cell, there are no inner borders.

```
tab1.setlines(@all) -a
```

removes all borders from the table.

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”, on page 47](#).

See also [settextcolor \(p. 443\)](#) for details on setting cell colors. See [setmerge \(p. 441\)](#) for cell merging, and [comment \(p. 242\)](#) for adding comments to cells.

<b>setmerge</b>	<a href="#">Table Proc</a>
-----------------	----------------------------

Merges/unmerges one or more table cells.

## Syntax

Table Proc:      `table_name.setmerge(cell_range)`

where *cell\_range* describes the table cells (in a single row) to be merged. The *cell\_range* specifications are given by:

<i>first_cell[:last_cell]</i>	Left (right) cell of the selection range (specified in “cell” format), followed by right (left) cell of the selection range (specified in “cell” format), separated by a “:” or “,” (e.g., “A2:C2”, “A2,C2”, or “R2C1:R2C3”, “R2C1,R2C3”). Merge all cells in the region defined by the first column and last column for the specified row.
<i>cell_row[,]</i>	Left (right) cell of the selection range (specified in “row[,] col” format), followed by right (left) cell of the selection range (specified in “row[,] col” format, with a fixed row), separated by a “,” (e.g., “2,A,2,C” or “2,1,2,3”). Merge all cells in the row defined by the first column and last column identifier.
<i>first_cell_col[,]</i>	
<i>cell_row[,]</i>	
<i>last_cell_col</i>	

If the first specified column is less than the last specified column (left specified before right), the cells in the row will be merged left to right, otherwise, the cells will be merged from right to left. The contents of the merged cell will be taken from the first non-empty cell in the merged region. If merging from left to right, the leftmost cell contents will be used; if merging from right to left, the rightmost cell contents will be displayed.

If you specify a merge involving previously merged cells, EViews will unmerge all cells within the specified range.

## Examples

```
tab1.setmerge(a2:d2)  
tab1.setmerge(2,1,2,4)
```

merges the cells in row 2, columns 1 to 4, from left to right.

```
tab2.setmerge(r2c5:r2c2)
```

merges the cells in row 2, columns 2 to 5, from right to left. We may then unmerge cells by issuing the command using any of the previously merged cells:

```
tab2.setmerge(r2c4:r2c4)
```

unmerges the previously merged cells.

Note that in all cases, the `setmerge` command must be specified using cells in a single row. The command:

```
tab3.setmerge(r2c1:r3c5)
```

generates an error since the cell specification involves rows 2 and 3.

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”, on page 47](#).

See also [setlines \(p. 440\)](#).

<b>settextcolor</b>	<a href="#">Table Proc</a>
---------------------	----------------------------

Changes the text color of the specified table cells.

## Syntax

Table Proc:      `table_name.settextcolor(cell_range) color_arg`

where *cell\_range* describes the table cells to be modified, and *color\_arg* specifies the color to be applied to the text in the cells. See [setfillcolor \(p. 429\)](#) for the syntax for *cell\_range* and *color\_arg*.

## Examples

To set an orange text color for the cell in the second row and sixth column of TAB1, you may use:

```
tab1.settextcolor(f2) @rgb(255, 128, 0)
tab1.settextcolor(2,f) @RGB(255, 128, 0)
tab1.settextcolor(2,6) orange
tab1.settextcolor(r2c6) orange
```

You may also specify a blue color for the text in an entire column, or an entire row,

```
tab1.settextcolor(C) @RGB(0, 0, 255)
tab1.settextcolor(2) blue
```

or a green color for the text in cells in a rectangular region:

```
tab1.settextcolor(R2C3:R3C6) green
tab1.settextcolor(r2c3,r3c6) green
tab1.settextcolor(2,C,3,F) @rgb(0, 255, 0)
tab1.settextcolor(2,3,3,6) @rgb(0, 255, 0)
```

## Cross-references

For additional discussion of tables see [Chapter 4, “Working with Tables”, on page 47](#).

See [setfont \(p. 431\)](#) and [setfillcolor \(p. 429\)](#) for details on changing the text font and cell background color.

**setwidth**

[Alpha Proc](#) | [Coef Proc](#) | [Group Proc](#) | [Matrix Proc](#) | [Series Proc](#) |  
[Rowvector Proc](#) | [Sym Proc](#) | [Table Proc](#) | [Vector Proc](#)

Set the column width for selected columns in an object spreadsheet.

### Syntax

Object Proc:      `object_name.setwidth width_arg`

Object Proc:      `object_name.setwidth(col_range) width_arg`

where *col\_range* is either a single column number or letter (e.g., “5”, “E”), a colon delimited range of columns (from low to high, e.g., “3:5”, “C:E”), or the keyword “@ALL”, and *width\_arg* specifies the width unit value. The width unit is computed from representative characters in the default font for the current table (the EViews table default font at the time the table was created), and corresponds roughly to a single character. *width\_arg* values may be non-integer values with resolution up to 1/10 of a width unit.

### Examples

```
tab1.setWidth(2) 12
```

sets the width of column 2 to 12 width units.

```
tab1.setWidth(2:10) 20
```

sets the widths for columns 2 through 10 to 20 width units.

### Cross-references

See [setindent \(p. 437\)](#) and [setjust \(p. 438\)](#) for details on setting spreadsheet and table indentation and justification. For details on setting the row heights in a table, see [setheight \(p. 436\)](#).

For additional discussion of tables see [Chapter 4, “Working with Tables”, on page 47](#). See also [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion and examples of table formatting in EViews.

**sheet**

[Alpha View](#) | [Coef View](#) | [Group View](#) | [Matrix View](#) | [Pool View](#) |  
[Rowvector View](#) | [Series View](#) | [Table View](#) | [Sym View](#) | [Vector View](#)

Spreadsheet view of object.

The `sheet` view displays the spreadsheet view of the named object. For table objects, `sheet` simply displays the table.

## Syntax

Object View:      `object_name.sheet(options)`

Pool View:      `pool_name.sheet(options) pool_ser1 [pool_ser2 pool_ser3 ...]`

The `sheet` view of pool objects displays the spreadsheet view of the series in the pool. Follow the word `sheet` by a list of series to display; you may use the cross section identifier “?” in the series name.

## Options

p	Print the spreadsheet view.
---	-----------------------------

## Examples

`tab1.sheet`

displays the spreadsheet view of TAB1.

`pool1.sheet x? y? z?`

displays the pool spreadsheet view of the series X?, Y?, and Z?.

## Cross-references

See [Chapter 5, “Basic Data Handling”, on page 79](#) of the *User’s Guide* for a discussion of the spreadsheet view of series and groups, and [Chapter 27, “Pooled Time Series, Cross-Section Data”, on page 809](#) of the *User’s Guide* for a discussion of pools.

<b>show</b>	<a href="#">Command</a>
-------------	-------------------------

Display objects.

The `show` command displays series or other objects on your screen. A scalar object is displayed in the status line at the bottom of the EViews window.

## Syntax

Command:      `show object_name.view_command`

Command:      `show object1 [object2 object3 ...]`

The command `show` should be followed by the name of an object, or an object name with an attached view.

For series and graph objects, `show` can operate on a list of names. The list of names must be of the same type. `show` creates and displays an untitled group or multiple graph object.

## Examples

```
genr x=nrnd  
show x.hist  
close x
```

generates a series X of random draws from a standard normal distribution, displays the histogram view of X, and closes the series window.

```
show wage log(wage)
```

opens an untitled group window with the spreadsheet view of the two series.

```
freeze(gra1) wage.hist  
genr lwage=log(wage)  
freeze(gra2) lwage.hist  
show gra1 gra2
```

opens an untitled graph object with two histograms.

## Cross-references

See “Object Commands” on page 6 for discussion, and Appendix A, “Object, View and Procedure Reference”, on page 151 for a complete listing of the views of the various objects.

See also [close \(p. 238\)](#).

<b>signalgraphs</b>	<a href="#">Sspace View</a>
---------------------	-----------------------------

Graph signal series.

Display graphs of a set of signal series computed using the Kalman filter.

## Syntax

Sspace View:      `object_name.signalgraphs(options)`

## Options

<code>t = output_type (default = "pred")</code>	Defines output type: “pred” (one-step ahead signal predictions), “predse” (RMSE of the one-step ahead signal predictions), “resid” (error in one-step ahead signal predictions), “residse” (RMSE of the one-step ahead signal prediction; same as “predse”), “stdresid” (standardized one-step ahead prediction residual), “smooth” (smoothed signals), “smoothse” (RMSE of the smoothed signals), “disturb” (estimate of the disturbances), “disturbse” (RMSE of the estimate of the disturbances), “stddisturb” (standardized estimate of the disturbances).
---	--

## Examples

```
ss1.signalgraphs(t=smooth)
ss1.signalgraphs(t=smoothse)
```

displays a graph view containing the smoothed signal values, and then displays a graph view containing the root MSE of the smoothed states.

## Cross-references

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for a discussion of state space models.

See also [stategraphs \(p. 459\)](#), [makesignals \(p. 344\)](#) and [makestates \(p. 345\)](#).

<b>smooth</b>	<a href="#">Command</a>    <a href="#">Series Proc</a>
---------------	--

### Exponential smoothing.

Forecasts a series using one of a number of exponential smoothing techniques. By default, `smooth` estimates the damping parameters of the smoothing model to minimize the sum of squared forecast errors, but you may specify your own values for the damping parameters.

`smooth` automatically calculates in-sample forecast errors and puts them into the series `RESID`.

## Syntax

Command:      `smooth(method) series_name smooth_name [freq]`

Series Proc:    `series_name.smooth(method) smooth_name [freq]`

You should follow the `smooth` keyword with the name of the series and a name for the smoothed series. You must also specify the smoothing method in parentheses. The optional `freq` may be used to override the default for the number of periods in the seasonal cycle. By default, this value is set to the workfile frequency (e.g. — 4 for quarterly data). For undated data, the default is 5.

## Options

### *Smoothing method options*

<code>s[,x]</code>	Single exponential smoothing for series with no trend. You may optionally specify a number <i>x</i> between zero and one for the mean parameter.
<code>d[,x]</code>	Double exponential smoothing for series with a trend. You may optionally specify a number <i>x</i> between zero and one for the mean parameter.
<code>n[,x,y]</code>	Holt-Winters without seasonal component. You may optionally specify numbers <i>x</i> and <i>y</i> between zero and one for the mean and trend parameters, respectively.
<code>a[,x,y,z]</code>	Holt-Winters with additive seasonal component. You may optionally specify numbers <i>x</i> , <i>y</i> , and <i>z</i> , between zero and one for the mean, trend, and seasonal parameters, respectively.
<code>m[,x,y,z]</code>	Holt-Winters with multiplicative seasonal component. You may optionally specify numbers <i>x</i> , <i>y</i> , and <i>z</i> , between zero and one for the mean, trend, and seasonal parameters, respectively.

### *Other Options:*

<code>p</code>	Print a table of forecast statistics.
----------------	---------------------------------------

If you wish to set only some of the damping parameters and let EViews estimate the other parameters, enter the letter “e” where you wish the parameter to be estimated.

If the number of seasons is different from the frequency of the workfile (an unusual case that arises primarily if you are using an undated workfile for data that are not monthly or quarterly), you should enter the number of seasons after the smoothed series name. This optional input will have no effect on forecasts without seasonal components.

## Examples

```
smooth(s) sales sales_f
```

smooths the SALES series by a single exponential smoothing method and saves the smoothed series as SALES\_F. EViews estimates the damping (smoothing) parameter and displays it with other forecast statistics in the SALES series window.

```
smooth(n,e,.3) tb3 tb3_hw
```

smooths the TB3 series by a Holt-Winters no seasonal method and saves the smoothed series as TB3\_HW. The mean damping parameter is estimated while the trend damping parameter is set to 0.3.

```
smpl @first @last-10
smooth(m,.1,.1,.1) order order_hw
smpl @all
graph gral.line order order_hw
show gral
```

smooths the ORDER series by a Holt-Winters multiplicative seasonal method leaving the last 10 observations. The damping parameters are all set to 0.1. The last three lines plot and display the actual and smoothed series over the full sample.

## Cross-references

See “[Exponential Smoothing](#)” on page 339 of the *User’s Guide* for a discussion of exponential smoothing methods.

smpl	<a href="#">Command</a>
------	-------------------------

Set sample range.

The `smpl` command sets the workfile sample to use for statistical operations and series assignment expressions.

## Syntax

Command:	<code>smpl smpl_spec</code>
Command:	<code>smpl sample_name</code>

List the date or number of the first observation and the date or number of the last observation for the sample. Rules for specifying dates are given in “[Dates](#)” on page 127 of the *User’s Guide*. The sample spec may contain more than one pair of beginning and ending observations.

The `smp1` command also allows you to select observations on the basis of conditions specified in an `if` statement. This enables you to use logical operators to specify what observations to include in EViews' procedures. Put the `if` statement after the pairs of dates.

You can also use `smp1` to set the current observations to the contents of a named sample object; put the name of the sample object after the keyword.

### Special keywords for `smp1`

The following “@-keywords” can be used in a `smp1` command:

<code>@all</code>	The entire workfile range.
<code>@first</code>	The first observation in the workfile.
<code>@last</code>	The last observation in the workfile.

In panel settings, you may use the additional keywords:

<code>@firstmin</code>	The earliest of the first observations (computed across cross-sections).
<code>@firstmax</code>	The latest of the first observations.
<code>@lastmin</code>	The earliest of the last observations.
<code>@lastmax</code>	The latest of the last observations.

### Examples

```
smp1 1955m1 1972m12
```

sets the workfile sample from 1955M1 to 1972M12.

```
smp1 @first 1940 1946 1972 1975 @last
```

excludes observations (or years) 1941–1945 and 1973–1974 from the workfile sample.

```
smp1 if union=1 and edu<=15
```

sets the sample to those observations where UNION takes the value 1 and EDU is less than or equal to 15.

```
sample half @first @first+@obs(x)/2
```

```
smp1 half
```

```
smp1 if x>0
```

```
smp1 @all if x>0
```

The first line declares a sample object named HALF which includes the first half of the series X. The second line sets the sample to HALF and the third line sets the sample to

those observations in HALF where X is positive. The last line sets the sample to those observations where X is positive over the full sample.

## Cross-references

See “[Samples](#)” on page [87](#) of the *User’s Guide* for a discussion of samples in EViews.

See also [set](#) (p. 420) and [sample](#) (p. 406).

<b>solve</b>	<a href="#">Command</a>    <a href="#">Model Proc</a>
--------------	---

**Solve the model.**

`solve` finds the solution to a simultaneous equation model for the set of observations specified in the current workfile sample.

## Syntax

Command:      `solve(options) model_name`

Model Proc:    `model_name.solve(options)`

Note: when `solve` is used in a program (batch mode) models are always solved over the workfile sample. If the model contains a solution sample, it will be ignored in favor of the workfile sample.

You should follow the name of the model after the `solve` command or use `solve` as a procedure of a named model object. The default solution method is dynamic simulation. You may modify the solution method as an option.

`solve` first looks for the specified model in the current workfile. If it is not present, `solve` attempts to `fetch` a model file (.DBL) from the default directory or, if provided, the path specified with the model name.

## Options

`solve` can take any of the options available in [solveopt](#) (p. 452).

## Examples

```
solve mod1
```

solves the model MOD1 using the default solution method.

```
nonlin2.solve(m=500,e)
```

solves the model NONLIN2 with an extended search of up to 500 iterations.

## Cross-references

See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a discussion of models.

See also [model \(p. 353\)](#), [msg \(p. 354\)](#) and [solveopt \(p. 452\)](#).

<b>solveopt</b>	<a href="#">Model Proc</a>
-----------------	----------------------------

Solve options for models.

`solveopt` sets options for model solution but does not solve the model. The same options can be set directly in a `solve` procedure.

### Syntax

Model Proc:      `model_name.solveopt(options)`

### Options

<code>s = arg</code> <i>(default = “d”)</i>	Solution type: “d” (deterministic), “m” (stochastic – collect means only), “s” (stochastic – collect means and s.d.), “b” (stochastic – collect means and confidence bounds), “a” (stochastic – collect all; means, s.d. and confidence bounds).
<code>d = arg</code> <i>(default = “d”)</i>	Model solution dynamics: “d” (dynamic solution), “s” (static solution), “f” (fitted values – single equation solution).
<code>m = integer</code> <i>(default = 5000)</i>	Maximum number of iterations for solution (maximum 100,000).
<code>c = number</code> <i>(default = 1e-8)</i>	Convergence criterion. Based upon the maximum change in any of the endogenous variables in the model. You may set a number between 1e-15 and 0.01.
<code>r = integer</code> <i>(default = 1000)</i>	Number of stochastic repetitions (used with stochastic “s = ” options).
<code>b = number</code> <i>(default = .95)</i>	Size of stochastic confidence intervals (used with stochastic “s = ” options).
<code>a = arg</code> <i>(default = “f”)</i>	Alternate scenario solution: “t” (true - solve both active and alternate scenario and collect deviations for stochastic), “f” (false - solve only the active scenario).

<code>o = arg</code> ( <i>default</i> = “g”)	Solution method: “g” (Gauss-Seidel), “e” (Gauss-Seidel with extended search/reduced step size), “n” (Newton), “m” (Newton with extended search/reduced step size).
<code>i = arg</code> ( <i>default</i> = “a”)	Set initial (starting) solution values: “a” (actuals), “p” (values in period prior to start of solution period).
<code>n = arg</code> ( <i>default</i> = “t”)	NA behavior: “t” (true - stop on “NA” values), “f” (false - do not stop when encountering “NA” values). Only applies to deterministic solution; EViews will always stop on “NA” values in stochastic solution.
<code>e = arg</code> ( <i>default</i> = “t”)	Excluded variables initialized from actuals: “t” (true), “f” (false).
<code>cu = arg</code> ( <i>default</i> = “f”)	Include coefficient uncertainty: “t” (true), “f” (false).
<code>t = arg</code> ( <i>default</i> = “u”)	Terminal condition for forward solution: “u” (user supplied - actuals), “l” (constant level), “d” (constant difference), “g” (constant growth rate).
<code>g = arg</code> ( <i>default</i> = 7)	Number of digits to round solution: an integer value (number of digits), “n” (do not roundoff).
<code>z = arg</code> ( <i>default</i> = 1e-7)	Zero value: a positive number below which the solution (absolute value) is set to zero, “n” (do not set to zero).

## Cross-references

See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a discussion of models.

See also [model \(p. 353\)](#), [msg \(p. 354\)](#) and [solve \(p. 451\)](#).

<code>sort</code>	<a href="#">Command</a>
-------------------	-------------------------

### Sort the workfile.

The `sort` command sorts *all* series in the workfile on the basis of the values of one or more of the series. For purposes of sorting, NAs are considered to be smaller than any other value. By default, EViews will sort the series in ascending order. You may use options to override the sort order.

EViews will first remove any workfile structures and then will sort the workfile using the specified settings.

## Syntax

Command:      **sort**(*options*) *arg1* [*arg2 arg3...*]

List the name of the series or groups by which you wish to sort the workfile. If you list two or more series, **sort** uses the values of the second series to resolve ties from the first series, and values of the third series to resolve ties from the second, and so on.

## Options

d                  sort in descending order.

## Examples

`sort(d) inc`

sorts all series in the workfile in order of the INC series with the highest value of INC first. NAs in INC (if any) will be placed at the bottom.

`sort gender race wage`

sorts all series in the workfile in order of the values of GENDER from low to high, with ties resolved by ordering on the basis of RACE, with further ties resolved by ordering on the basis of WAGE.

## Cross-references

See “[Sorting a Workfile](#)” on page [251](#) of the *User’s Guide*.

<b>spec</b>	<a href="#">Logl View</a>   <a href="#">Model View</a>   <a href="#">Sspace View</a>   <a href="#">System View</a>
-------------	--

Display the text specification view for logl, model, sspace, system objects.

## Syntax

Object View:      object\_name.spec(*options*)

## Options

p                  Print the specification text.

## Examples

`model1.spec`

displays the specification of the object MODEL1.

## Cross-references

See also [append \(p. 202\)](#), [merge \(p. 351\)](#), [text \(p. 482\)](#).

spike	<a href="#">Command</a>    <a href="#">Coef View</a>   <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a>   <a href="#">Series View</a>   <a href="#">Sym View</a>   <a href="#">Vector View</a>
-------	--

Display spike graph view of data, or change existing graph object type to spike.

The spike graph view of a series or group creates spike graphs for all specified series or matrix columns.

## Syntax

- |              |   |
|--------------|---|
| Command:     | <code>spike arg1 [arg2 arg3 ...]</code> |
| Object View: | <code>object_name.spike(options)</code> |
| Graph Proc:  | <code>graph_name.spike(options)</code>  |

## Options

### *Template and printing options*

- |                             |  |
|-----------------------------|--|
| <code>o = graph_name</code> | Use appearance options from the specified graph.                           |
| <code>t = graph_name</code> | Use appearance options and copy text and shading from the specified graph. |
| <code>p</code>              | Print the spike graph.   |

### *Scale options*

- |                          |   |
|--------------------------|---|
| <code>a (default)</code> | Automatic single scale.   |
| <code>d</code>           | Dual scaling with no crossing. The first series is scaled on the left and all other series are scaled on the right.           |
| <code>x</code>           | Dual scaling with possible crossing. See the “d” option.  |
| <code>n</code>           | Normalized scale (zero mean and unit standard deviation). May not be used with the “s” option.                                |
| <code>s</code>           | Stacked spike graph. Each segment represents the cumulative total of the series listed (may not be used with the “l” option). |
| <code>l</code>           | Spike graph for the first series listed and a line graph for all subsequent series (may not be used with the “s” option).     |

**m**

Plot spikes in multiple graphs (will override the “l” option). Not for use with an existing graph object.

### *Panel options*

The following options apply when graphing panel structured data.

**panel = arg**  
(default taken  
from global set-  
tings)

Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “medmxmn” (plot median, max and min).

### Examples

```
group g1 gdp cons m1  
g1.spike(d)
```

plots line graphs of the three series in group G1 using dual scaling.

```
matrix1.spike(t=mygra)
```

displays spike graphs of the columns of MATRIX1 using the graph object MYGRA as a template.

```
graph1.spike(d)
```

changes GRAPH1 so that it contains spike graphs of each of the series in the original graph, using dual scaling.

### Cross-references

See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a detailed discussion of graphs in EViews.

See also [graph \(p. 303\)](#) for graph declaration and additional graph types.

**sspace****Object Declaration**

Declare state space object.

**Syntax**

Command:      **sspace sspace\_name**

Follow the **sspace** keyword with a name to be given the **sspace** object.

**Examples**

```
sspace stsp1
```

declares a **sspace** object named STSP1.

```
sspace tvp
tvp.append cs = c(1) + sv1*inc
tvp.append @state sv1 = sv1(-1) + [var=c(2)]
tvp.ml
```

declares a **sspace** object named TVP, specifies a time varying coefficient model, and estimates the model by maximum likelihood.

**Cross-references**

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for a discussion of state space models. The **sspace** object is documented in greater detail beginning on [page 177](#).

[append \(p. 202\)](#) may be used to add lines to an existing **sspace** object. See also [ml \(p. 352\)](#) for estimation of state space models.

**statby****Series View**

Basic statistics by classification.

The **statby** view displays descriptive statistics for the elements of a series classified into categories by one or more series.

**Syntax**

Series View:      **series\_name.statby(options) classifier\_names**

Follow the series name with a period, the `statby` keyword, and a name (or a list of names) for the series or group by which to classify. The options control which statistics to display and in what form. By default, `statby` displays the means, standard deviations, and counts for the series.

## Options

### *Options to control statistics to be displayed*

sum	Display sums.
med	Display medians.
max	Display maxima.
min	Display minima.
quant = <i>arg</i> ( <i>default</i> = .5)	Display quantile with value given by the argument.
q = <i>arg</i> ( <i>default</i> = "r")	Compute quantiles using the specified definition: "b" (Blom), "r" (Rankit-Cleveland), "o" (simple fraction), "t" (Tukey), "v" (van der Waerden).
skew	Display skewness.
kurt	Display kurtosis.
na	Display counts of NAs.
nomean	Do not display means.
nostd	Do not display standard deviations.
nocount	Do not display counts.

### *Options to control layout*

l	Display in list mode (for more than one classifier).
nor	Do not display row margin statistics.
noc	Do not display column margin statistics.
nom	Do not display table margin statistics (unconditional tables); for more than two classifier series.
nos	Do not display sub-margin totals in list mode; only used with "l" option and more than two classifier series.
sp	Display sparse labels; only with list mode option, "l".

### Options to control binning

dropna ( <i>default</i> ), keepna	[Drop/Keep] NA as a category.
v = <i>integer</i> ( <i>default</i> = 100)	Bin categories if classification series take on more than the specified number of distinct values.
nov	Do not bin based on the number of values of the classification series.
a = <i>number</i> ( <i>default</i> = 2)	Bin categories if average cell count is less than the specified number.
noa	Do not bin based on the average cell count.
b = <i>integer</i> ( <i>default</i> = 5)	Set maximum number of binned categories.

### Other options

p	Print the descriptive statistics table.
---	---

### Examples

```
wage.statby(max,min) sex race
```

displays the mean, standard deviation, max, and min of the series WAGE by (possibly binned) values of SEX and RACE.

### Cross-references

See “[By-Group Statistics](#)” on page 549 for functions to compute by-group statistics. See also “[Stats by Classification](#)” on page 300, and “[Descriptive Statistics](#)” on page 367 of the *User’s Guide*.

See also [hist](#) (p. 308), [boxplotby](#) (p. 221) and [linkto](#) (p. 324).

stategraphs	<a href="#">SSpace View</a>
-------------	-----------------------------

Display graphs of a set of state series computed using the Kalman filter.

### Syntax

SSpace View:      `sspace_name.stategraph(options)`

## Options

`t = output_type`      Defines output type: “pred” (one-step ahead signal predictions), “predse” (RMSE of the one-step ahead signal predictions), “resid” (error in one-step ahead signal predictions), “residse” (RMSE of the one-step ahead signal prediction; same as “predse”), “stdresid” (standardized one-step ahead prediction residual), “smooth” (smoothed signals), “smoothse” (RMSE of the smoothed signals), “disturb” (estimate of the disturbances), “disturbse” (RMSE of the estimate of the disturbances), “stddisturb” (standardized estimate of the disturbances).  
*(default = “pred”)*

### Other options

<code>p</code>	Print the view.
----------------	-----------------

## Examples

```
ss1.stategraphs (t=filt)
```

displays a graph view containing the filtered state values.

## Cross-references

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for a discussion of state space models.

See also [signalgraphs \(p. 446\)](#), [makesignals \(p. 344\)](#) and [makestates \(p. 345\)](#).

<b>statefinal</b>	<a href="#">Sspace View</a>
-------------------	-----------------------------

Display final state values.

Show the one-step ahead state predictions or the state prediction covariance matrix at the final values ( $T + 1 | T$ ), where  $T$  is the last observation in the estimation sample. By default, EViews shows the state predictions.

## Syntax

Sspace View:      `sspace_name.statefinal(options)`

## Options

c	Display the state prediction covariance matrix.
p	Print the view.

## Examples

```
ss1.statefinal(c)
```

displays a view containing the final state covariances (the one-step ahead covariances for the first out-of-(estimation) sample period).

## Cross-references

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for a discussion of state space models.

See also [stateinit \(p. 461\)](#).

<a href="#">stateinit</a>	<a href="#">SSpace View</a>
---------------------------	-----------------------------

Display initial state values.

Show the state initial values or the state covariance initial values used to initialize the Kalman Filter. By default, EViews shows the state values.

## Syntax

Sspace View:      `sspace_name.stateinit(options)`

## Options

c	Display the covariance matrix.
p	Print the view.

## Examples

```
ss1.stateinit
```

displays a view containing the initial state values (the one-step ahead predictions for the first period).

## Cross-references

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for a discussion of state space models.

See also [statefinal](#) (p. 460).

<b>stats</b>	<a href="#">Command</a>    <a href="#">Coef View</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a> <a href="#">Sym View</a>   <a href="#">Valmap View</a>   <a href="#">Vector View</a>
--------------	--

**Descriptive statistics.**

Computes and displays a table of means, medians, maximum and minimum values, standard deviations, and other descriptive statistics of one or more series or a group of series. Alternately, displays summary statistics for the definitions of a valmap.

When used as a command, **stats** creates an untitled group containing all of the specified series, and opens a statistics view of the group. By default, if more than one series is given, the statistics are calculated for the common sample.

When used for coef or matrix objects, **stats** computes the statistics for each column of data.

### Syntax

Command:      **stats**(*options*) *ser1 [ser2 ser3 ...]*

Object View:    *object\_name.stats*(*options*)

### Options

i                Individual sample for each series (for group only) By default, EViews computes the statistics using a common sample.

p                Print the stats table.

### Examples

```
stats height weight age
```

opens an untitled group window displaying the histogram and descriptive statistics for the common sample of the three series.

```
group group1 wage hrs edu  
group1.stats(i)
```

displays the descriptive statistics view of GROUP1 for the individual samples.

```
map1.stats
```

displays the summary descriptive view of the definitions in the valmap MAP1.

## Cross-references

See “[Descriptive Statistics](#)” on page 298 and [page 367](#) of the *User’s Guide* for a discussion of the descriptive statistics views of series and groups.

See also [boxplot \(p. 219\)](#) and [hist \(p. 308\)](#).

<b>statusline</b>	<a href="#">Command</a>
-------------------	-------------------------

Send text to the status line.

Displays a message in the status line at the bottom of the EViews main window. The message may include text, control variables, and string variables.

## Syntax

Command:      **statusline** *message\_string*

## Examples

```
statusline Iteration Number: !t
```

Displays the message “Iteration Number: !t” in the status line replacing “!t” with the current value of the control variable in the program.

## Cross-references

See [Chapter 6, “EViews Programming”, on page 83](#) for a discussion and examples of programs, control variables and string variables.

<b>stom</b>	<a href="#">Group Proc</a>   <a href="#">Series Proc</a>
-------------	--

Convert a series or group to a vector or matrix.

Fills a vector or matrix with the data from a series or group.

## Syntax

Series Proc:      **stom**(*series\_name*, *vector[, sample]*)

Group Proc:      **stom**(*group\_name*, *matrix[, sample]*)

Include the series or group name in parentheses followed by a comma and the vector or matrix name. By default, the series values in the current workfile sample are used to fill the vector or matrix; you may optionally provide an alternative sample.

There are two important features of **stom** that you should keep in mind:

- If any of the series contain NAs, those observations will be dropped from the vector/matrix (for alternative behavior, see [stomna \(p. 464\)](#)).
- If the vector or matrix already exists in the workfile, EViews automatically resizes the vector/matrix to fit the series/group.

## Examples

```
series lwage=log(wage)
stom(lwage, vec1)
```

converts the series LWAGE into a vector named VEC1 using the current workfile sample. Any NAs in LWAGE will be dropped from VEC1.

```
group rhs x1 x2 x3
sample s1 1951 1990
stom(rhs, x, s1)
```

converts a group of three series named X1, X2 and X3 to a matrix named X using sample S1. The matrix X will have 40 rows and 3 columns (provided there are no NAs).

## Cross-references

See [Chapter 3, “Matrix Language”, on page 23](#) of the *Command and Programming Reference* for further discussion and examples of matrices.

See also [stomna \(p. 464\)](#) and [mtos \(p. 592\)](#).

<b>stomna</b>	<a href="#">Group Proc</a>   <a href="#">Series Proc</a>
---------------	--

Convert a series or group to a vector or matrix without dropping NAs.

Fills a vector or matrix with the data from a series or group without dropping observations with missing values.

Works in identical fashion to [stom \(p. 463\)](#), but does not drop observations containing NAs.

## Syntax

Series Proc:      `stomna(series, vector[, sample])`  
Group Proc:      `stomna(group, matrix[, sample])`

Include the series or group name in parentheses, followed by a comma and the vector or matrix name. By default, the series values in the current workfile sample are used to fill the vector or matrix. You may optionally provide an alternative sample.

## Examples

```
series lwage=log(wage)
stomna(lwage,vec1)
```

converts the series LWAGE into a vector named VEC1 using the current workfile sample. Any NAs in LWAGE will be placed in VEC1.

```
group rhs x1 x2 x3
sample s1 1951 1990
stom(rhs,x,s1)
```

converts a group of three series RHS to a matrix named X using sample S1. The matrix X will always have 40 rows and 3 columns.

## Cross-references

See [Chapter 3, “Matrix Language”, on page 23](#) of the *Command and Programming Reference* for further discussion and examples of matrices.

See also [stom \(p. 463\)](#) and [mtos \(p. 592\)](#).

<b>store</b>	<a href="#">Command</a>    <a href="#">Pool Proc</a>
--------------	--

**Store objects in databases and databank files.**

Stores one or more objects in the current workfile in EViews databases or individual databank files on disk. The objects are stored under the name that appears in the workfile.

When used as a pool proc, EViews will first expand the list of series using the pool operator, and then perform the operation.

## Syntax

Command:      *store(options) object\_list*

Pool Proc:      *pool\_name.store(options) pool\_ser1 [pool\_ser2 pool\_ser3 ...]*

Follow the **store** command keyword with a list of object names (each separated by a space) that you wish to store. The default is to store the objects in the default database. (*This behavior is a change from EViews 2 and earlier where the default was to store objects in individual databank files*).

You may precede the object name with a database name and the double colon “::” to indicate a specific database. You can also specify the database name as an option in parentheses, in which case all objects without an explicit database name will be stored in the specified database.

When used as a command, you may use wild card characters “?” (to match any single character) or “\*” (to match zero or more characters) in the object name list. All objects with names matching the pattern will be stored. You may not use “?” as a wildcard character if `store` is being used as a pool proc, since this conflicts with the pool identifier.

You can optionally choose to store the listed objects in individual databank files. To store in files other than the default path, you should include a path designation before the object name.

## Options

<code>d = db_name</code>	Store to the specified database.
<code>i</code>	Store to individual databank files.
<code>1 / 2</code>	Store series in [single / double] precision to save space.
<code>o</code>	Overwrite object in database (default is to merge data, where possible).
<code>g = arg</code>	Group store from workfile to database: “s” (copy group definition and series as separate objects), “t” (copy group definition and series as one object), “d” (copy series only as separate objects), “l” (copy group definition only).

If you do not specify the precision option (1 or 2), the global option setting will be used. See “[Database Registry / Database Storage Defaults](#)” on page 916 of the *User’s Guide*.

## Examples

```
store m1 gdp unemp
```

stores the three objects M1, GDP, UNEMP in the default database.

```
store(d=us1) m1 gdp macro::unemp
```

stores M1 and GDP in the US1 database and UNEMP in the MACRO database.

```
store usdat::gdp macro::gdp
```

stores the same object GDP in two different databases USDAT and MACRO.

```
store(1) cons*
```

stores all objects with names starting with CONS in the default database. The “1” option uses single precision to save space.

```
store(i) m1 c:\data\unemp
stores M1 and UNEMP in individual databank files.
```

### Cross-references

[“Basic Data Handling” on page 79](#) of the *User’s Guide* discusses exporting data in other file formats. See [Chapter 10, “EViews Databases”, on page 253](#) of the *User’s Guide* for a discussion of EViews databases and databank files.

For additional discussion of wildcards, see [Appendix B, “Wildcards”, on page 921](#) of the *User’s Guide*.

See also [fetch \(p. 279\)](#) and [copy \(p. 244\)](#).

<b>structure</b>	<a href="#">Sspace View</a>
------------------	-----------------------------

Display summary of sspace specification.

Show view which summarizes the system transition matrices or the covariance structure of the state space specification. EViews can display either the formulae (default) or the values of the system transition matrices or covariance.

### Syntax

Sspace View:      `sspace_name.structure(options) [argument]`

If you choose to display the values for a time-varying system using the “v” option, you should use the optional *[argument]* to specify a single date at which to evaluate the matrices. If none is provided, EViews will use the first date in the current sample.

### Options

v	Display the values of the system transition or covariance matrices.
c	Display the system covariance matrix.
p	Print the view.

### Examples

```
ss1.structure
```

displays a system transition matrices.

```
ss1.structure 1993q4
```

displays the transition matrices evaluated at 1993Q4.

### Cross-references

See [Chapter 25, “State Space Models and the Kalman Filter”, on page 737](#) of the *User’s Guide* for a discussion of state space models.

<b>sur</b>	<a href="#">System Method</a>
------------	-------------------------------

Estimate a system object using seemingly unrelated regression (SUR).

Note that the EViews procedure is more general than textbook versions of SUR since the system of equations may contain cross-equation restrictions on parameters.

### Syntax

System Method:    `system_name.sur(options)`

### Options

i                Iterate on the weighting matrix and coefficient vector simultaneously.

s                Iterate on the weighting matrix and coefficient vector sequentially.

o (*default*)    Iterate only on the coefficient vector with one step of the weighting matrix.

c                One step iteration on the coefficient vector after one step of the weighting matrix.

`m = integer`   Maximum number of iterations.

`c = number`   Set convergence criterion.

`l = number`   Set maximum number of iterations on the first-stage iteration to get one-step weighting matrix.

`showopts / -showopts`   [Do / do not] display the starting coefficient values and estimation options in the estimation output.

`deriv = keyword` Set derivative methods. The argument *keyword* should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.

`p` Print estimation results.

## Examples

```
sys1.sur(i)
```

estimates SYS1 by SUR, iterating simultaneously on the weighting matrix and coefficient vector.

```
nlsys.sur(showopts,m=500)
```

estimates NLSYS by SUR with up to 500 iterations. The “showopts” option displays the starting values.

## Cross-references

See [Chapter 23, “System Estimation”, on page 679](#) of the *User’s Guide* for a discussion of system estimation.

<b>svar</b>	<a href="#">Var Proc</a>
-------------	--------------------------

Estimate factorization matrix for structural innovations.

## Syntax

Var Proc:	<code>var_name.svar(options)</code>
-----------	-------------------------------------

The var object must previously have been estimated in unrestricted form.

You must specify the identifying restrictions either in text form by the `append` proc or by a pattern matrix option. See [“Specifying the Identifying Restrictions” on page 717](#) of the *User’s Guide* for details on specifying restrictions.

## Options

You must specify one of the following restriction type:

<code>rtype = text</code>	Text form restrictions. The restrictions must be specified by the <code>append</code> command to use this option.
<code>rtype = patsr</code>	Short-run pattern restrictions. You must provide the names of the patterned matrices by the “namea = ” and “nameb = ” options as described below.
<code>rtype = patlr</code>	Long-run pattern restrictions. You must provide the name of the patterned matrix by the “namelr = ” option as described below.

#### Other Options:

<code>namea = arg</code> , <code>nameb = arg</code>	Names of the pattern matrices for A and B matrices. Must be used with “ <code>rtype = patsr</code> ”.
<code>namelr = arg</code>	Name of the pattern matrix for long-run impulse responses. Must be used with “ <code>rtype = patlr</code> ”.
<code>fsign</code>	Do not apply the sign normalization rule. Default is to apply the sign normalization rule whenever applicable. See “ <a href="#">Sign Indeterminacy</a> ” on page 722 of the <i>User’s Guide</i> for a discussion of the sign normalization rule.
<code>f0 = arg</code> ( <i>default</i> = 0.1)	Starting values for the free parameters: “ <i>scalar</i> ” (specify fixed value for starting values), “ <i>s</i> ” (user specified starting values are taken from the C coefficient vector), “ <i>u</i> ” (draw starting values for free parameters from a uniform distribution on [0,1]), “ <i>n</i> ” (draw starting values for free parameters from standard normal).
<code>maxiter = integer</code>	Maximum number of iterations. Default is taken from global option setting.
<code>conv = number</code>	Convergence criterion. Default is taken from global option setting.
<code>trace = integer</code>	Trace iterations process every <i>integer</i> iterations (displays an untitled text object containing summary information).
<code>nostop</code>	Suppress “Near Singular Matrix” error message even if Hessian is singular at final parameter estimates.

#### Examples

```
var var1.ls 1 4 m1 gdp cpi
matrix(3,3) pata
```

```
'fill matrix in row major order
pata.fill(by=r) 1,0,0, na,1,0, na,na,1
matrix(3,3) patb
pata.fill(by=r) na,0,0, 0,na,0, 0,0,na
var1.svar(rtype=patsr,namea=pata,nameb=patb)
```

The first line declares and estimates a VAR with three variables. Then we create the short-run pattern matrices and estimate the factorization matrix.

```
var var1.ls 1 8 dy u @
var1.append(svar) @lr1(@u1)=0
freeze(out1) var1.svar(rtype=text)
```

The first line declares and estimates a VAR with two variables without a constant. The next two lines specify a long-run restriction in text form and stores the estimation output in a table object named OUT1.

## Cross-references

See “[Structural \(Identified\) VARs](#)” on page 717 of the *User’s Guide* for a discussion of structural VARs.

sym	<a href="#">Object Declaration</a>
-----	------------------------------------

### Declare a symmetric matrix object.

The `sym` command declares and optionally initializes a matrix object.

#### Syntax

Command:      `sym(n) sym_name[ = assignment ]`

`sym` takes an optional argument *n* specifying the row and column dimension of the matrix and is followed by the name you wish to give the matrix.

You may also include an assignment in the `sym` command. The `sym` will be resized, if necessary. Once declared, symmetric matrices may be resized by repeating the `sym` command for a given matrix name.

#### Examples

```
sym mom
```

declares a symmetric matrix named MOM with one zero element.

```
sym y=@inner(x)
```

declares a symmetric matrix Y and assigns to it the inner product of the matrix X.

### Cross-references

See “[Matrix Language](#)” on page 23 for a discussion of matrix objects in EViews.

See also [matrix](#) (p. 350).

<b>system</b>	<a href="#">Object Declaration</a>
---------------	------------------------------------

Declare system of equations.

### Syntax

Command:      **system** *system\_name*

Follow the **system** keyword by a name for the system. If you do not provide a name, EViews will open an untitled system object (if in interactive mode).

### Examples

```
system mysys
```

creates a system named MYSYS.

### Cross-references

[Chapter 23, “System Estimation”, on page 679](#) of the *User’s Guide* provides a full discussion of system objects.

See [append](#) (p. 202) for adding specification lines to an existing system. The system object is described in greater detail in “[System](#)” (p. 182).

<b>table</b>	<a href="#">Object Declaration</a>
--------------	------------------------------------

Declare a table object.

The **table** command declares and optionally sizes a table object.

### Syntax

Command:      **table**(*rows,cols*) *table\_name*

The `table` command takes two optional arguments specifying the row and column dimension of the table, and is followed by the name you wish to give the matrix. If no sizing information is provided, the table will contain a single cell.

You may also include an assignment in the `sym` command. The symmetric matrix will be resized, if necessary. Once declared, symmetric matrices may be resized by repeating the `sym` command with new dimensions.

## Examples

```
table onelement
```

declares a one element table

```
table(10,5) outtable
```

creates a table OUTTABLE with 10 rows and 5 columns.

## Cross-references

[Chapter 4, “Working with Tables”, on page 47](#) describes table formatting using commands. See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a general discussion and examples of table formatting in EViews.

See also [freeze \(p. 290\)](#).

template	<a href="#">Graph Proc</a>
----------	----------------------------

Applies templates to graph objects.

If you apply template to a multiple graph object, the template options will be applied to each graph in the multiple graph. If the template graph is a multiple graph, the options of the first graph will be used.

## Syntax

Graph Proc:      `graph_name.template(options) graph_name`

Follow the name of the graph to which you want to apply the template options with a period, the keyword `template`, and the name of a graph object to use as a template.

## Options

t	Copy any text labels and shading in the template graph in addition to its option settings.
---	--

## Examples

```
gra_cs.template gra_gdp
```

applies the option settings in the graph object GRA\_GDP to the graph GRA\_CS. Text and shadings in GRA\_GDP will not be applied to GRA\_CS.

```
g1.template(t) mygraph1
```

applies the option settings of MYGRAPH1, and all text and shadings in the graph to the graph G1. Text and shading objects include those added with the [addtext \(p. 199\)](#) or [draw \(p. 267\)](#) commands. Note that the “t” option overwrites any existing text and shading objects in the target graph.

If you are using a boxplot as a template for another graph type, or vice versa, note that the graph types and boxplot specific attributes will not be changed. In addition, when the “t” option is used, vertical lines or shaded areas will not be copied between the graphs, since the horizontal scales differ.

## Cross-references

See “[Graph Templates](#)” on page 410 of the *User’s Guide* for additional discussion.

<b>testadd</b>	<a href="#">Command</a>    <a href="#">Equation View</a>
----------------	--

Test whether to add regressors to an estimated equation.

Tests the hypothesis that the listed variables were incorrectly omitted from an estimated equation (only available for equations estimated by list). The test displays some combination of Wald and LR test statistics, as well as the auxiliary regression.

## Syntax

Command:      **testadd** *arg1 [arg2 arg3 ...]*

Equation View:    *eq\_name.testadd arg1 [arg2 arg3 ...]*

List the names of the series or groups of series to test for omission after the keyword. The command form applies the test to the default equation.

## Options

<b>p</b>	Print output from the test.
----------	-----------------------------

## Examples

```
ls sales c adver lsales ar(1)
```

```
testadd gdp gdp(-1)
```

tests whether GDP and GDP(-1) belong in the specification for SALES. The commands:

```
equation oldeq.ls sales c adver lsales ar(1)
oldeq.testadd gdp gdp(-1)
```

perform the same test using a named equation object.

### Cross-references

See “[Coefficient Tests](#)” on page 554 of the *User’s Guide* for further discussion.

See also [testdrop \(p. 477\)](#) and [wald \(p. 502\)](#).

<b>testbtw</b>	<a href="#">Group View</a>
----------------	----------------------------

Test equality of the mean, median or variance between (among) series in a group.

### Syntax

Group View:      `group_name.testbtw(options)`

Specify the type of test as an option.

### Options

mean ( <i>default</i> )	Test equality of mean.
med	Test equality of median.
var	Test equality of variance.
c	Use common sample.
i ( <i>default</i> )	Use individual sample.
p	Print the test results.

### Examples

```
group g1 wage_m wage_f
g1.testbtw
g1.testbtw(var, c)
```

tests the equality of means between the two series WAGE\_M and WAGE\_F.

### Cross-references

See “[Tests of Equality](#)” on page 368 of the *User’s Guide* for further discussion of these tests.

See also [testby](#) (p. 476), [teststat](#) (p. 481).

<b>testby</b>	<a href="#">Series View</a>
---------------	-----------------------------

Test equality of the mean, median, or variance of a series across categories classified by a list of series or a group.

### Syntax

Series View:      `series_name.testby(options) arg1 [arg2 arg2 ...]`

Follow the `testby` keyword by a list of the names of the series or groups to use as classifiers. Specify the type of test as an option.

### Options

<code>mean</code> ( <i>default</i> )	Test equality of mean.
<code>med</code>	Test equality of median.
<code>var</code>	Test equality of variance.
<code>dropna</code> ( <i>default</i> ), <code>keepna</code>	[Drop /Keep] NAs as a classification category.
<code>v = integer</code> ( <i>default</i> = 100)	Bin categories if classification series take more than the specified number of distinct values.
<code>nov</code>	Do not bin based on the number of values of the classification series.
<code>a = number</code> ( <i>default</i> = 2)	Bin categories if average cell count is less than the specified number.
<code>noa</code>	Do not bin on the basis of average cell count.
<code>b = integer</code> ( <i>default</i> = 5)	Set maximum number of binned categories.
<code>p</code>	Print the test results.

### Examples

```
wage.testby(med) race
```

Tests equality of medians of WAGE across groups classified by RACE.

## Cross-references

See “[Equality Tests by Classification](#)” on page 306 of the *User’s Guide* for a discussion of equality tests.

See also [testbtw](#) (p. 475), [teststat](#) (p. 481).

<b>testdrop</b>	<a href="#">Command</a>    <a href="#">Equation View</a>
-----------------	--

Test whether to drop regressors from a regression.

Tests the hypothesis that the listed variables were incorrectly included in the estimated equation (only available for equations estimated by list). The test displays some combination of *F* and LR test statistics, as well as the test regression.

## Syntax

Command:      **testdrop** *arg1 [arg2 arg3 ...]*

Equation View:    *eq\_name.testdrop arg1 [arg2 arg3 ...]*

List the names of the series or groups of series to test for omission after the keyword. The command form applies the test to the default equation.

## Options

p	Print output from the test.
---	-----------------------------

## Examples

```
ls sales c adver lsales ar(1)
testdrop adver
```

tests whether ADVER should be excluded from the specification for SALES. The commands:

```
equation oldeq.ls sales c adver lsales ar(1)
oldeq.testdrop adver
```

perform the same test using a named equation object.

## Cross-references

See “[Coefficient Tests](#)” on page 554 of the *User’s Guide* for further discussion of testing coefficients.

See also [testadd](#) (p. 474) and [wald](#) (p. 502).

**testexog**[Var View](#)

Perform exogeneity (Granger causality) tests on a VAR object.

**Syntax**

Var View:      `var_name.testexog(options)`

**Options**

`name = arg`      Save the Wald test statistics in named matrix object. See below for a description of the statistics stored in the matrix.

`p`      Print output from the test.

The `name=` option stores the results in a  $(k + 1) \times k$  matrix, where  $k$  is the number of endogenous variables in the VAR. In the first  $k$  rows, the  $i$ -th row,  $j$ -th column contains the Wald statistic for the joint significance of lags of the  $i$ -th endogenous variable in the  $j$ -th equation (note that the entries in the main diagonal are not reported in the table view). The degrees of freedom of the Wald statistics is the number of lags included in the VAR.

In the last row, the  $j$ -th column contains the Wald statistic for the joint significance of all lagged endogenous variables (excluding lags of the dependent variable) in the  $j$ -th equation. The degrees of freedom of the Wald statistics in the last row is  $(k - 1)$  times the number of lags included in the VAR.

**Examples**

```
var var1.ls 1 6 lgdp lm1 lcpi  
freeze(tab1) var1.testexog(name=exog)
```

The first line declares and estimates a VAR. The second line stores the exclusion test results in a named table TAB1, and stores the Wald statistics in a matrix named EXOG.

**Cross-references**

See “[Diagnostic Views](#)” on page 708 of the *User’s Guide* for a discussion of other VAR diagnostics.

See also [testlags](#) (p. 480).

**testfit**[Equation View](#)

Carry out the Hosmer-Lemeshow and/or Andrews goodness-of-fit tests for estimated binary models.

**Syntax**

Equation View:    `binary_equation.testfit(options)`

**Options**

`h`              Group by the predicted values of the estimated equation.

`s = series_name`    Group by the specified series.

`integer  
(default = 10)`    Specify the number of quantile groups in which to classify observations.

`u`              Unbalanced grouping. Default is to randomize ties to balance the number of observations in each group.

`v`              Group according to the values of the reference series.

`l = integer  
(default = 100)`    Limit the number of values to use for grouping. Should be used with the “v” option.

`p`              Print the result of the test.

**Examples**

```
equation eq1.binary work c age edu
eq1.testfit(h,5,u)
```

estimates a probit specification, and tests goodness-of-fit by comparing five unbalanced groups of actual data to those estimated by the model.

**Cross-references**

See “[Goodness-of-Fit Tests](#)” on page 615 of the *User’s Guide* for a discussion of the Andrews and Hosmer-Lemeshow tests.

testlags	<a href="#">Var View</a>
----------	--------------------------

Perform lag exclusion (Wald) tests on a VAR.

### Syntax

Var View:      `var_name.testlags(options)`

### Options

`name = arg`      Save the Wald test statistics in named matrix object. See below for a description of the statistics contained in the stored matrix.

`p`      Print the result of the test.

The “`name =` ” option stores results in an  $m \times (k + 1)$  matrix, where  $m$  is the number of lagged terms and  $k$  is the number of endogenous variables in the VAR. In the first  $k$  columns, the  $i$ -th row,  $j$ -th column entry is the Wald statistic for the joint significance of all  $i$ -th lagged endogenous variables in the  $j$ -th equation. These Wald statistics have a  $\chi^2$  distribution with  $k$  degrees of freedom under the exclusion null.

In the last column, the  $i$ -th row contains the system Wald statistic for testing the joint significance of all  $i$ -th lagged endogenous variables in the VAR system. The system Wald statistics has a chi-square distribution with  $k^2$  degrees of freedom under the exclusion null.

### Examples

```
var var1.ls 1 6 lgdp lm1 lcpi  
freeze(tab1) var1.testlags(name=lags)
```

The first line declares and estimates a VAR. The second line stores the lag exclusion test results in a table named TAB1, and stores the Wald statistics in a matrix named LAGS.

### Cross-references

See “[Diagnostic Views](#)” on page 708 of the *User’s Guide* for a discussion other VAR diagnostics.

See also [laglen](#) (p. 318) and [testexog](#) (p. 478).

**teststat**[Series View](#)

Test simple hypotheses of whether the mean, median, or variance of a series is equal to a specified value.

**Syntax**

Series View:      `series_name.teststat(options)`

Specify the type of test and the value under the null hypothesis as an option.

**Options**

`mean = number` Test the null hypothesis that the mean equals the specified number.

`med = number` Test the null hypothesis that the median equals the specified number.

`var = number` Test the null hypothesis that the variance equals the specified number. The number must be positive.

`std = number` Test equality of mean conditional on the specified standard deviation. The standard deviation must be positive.

`p` Print the test results.

**Examples**

```
smp1 if race=1
lwage.teststat(var=4)
```

tests the null hypothesis that the variance of LWAGE is equal to 4 for the subsample with RACE = 1.

**Cross-references**

See “[Tests for Descriptive Stats](#)” on page 303 of the *User’s Guide* for a discussion of simple hypothesis tests.

See also [testbtw](#) (p. 475), [testby](#) (p. 476).

**text**[Object Declaration](#) || [Model View](#) | [Text View](#)

Declare a text object when used as a command, or display text representation of the model specification or text object.

### Syntax

Command:      `text object_name`

Model View:     `model_name.text(options)`

Text View:      `text_name.text(options)`

Follow the keyword with a name of the text object. When used as a model view, `text` is equivalent to [spec \(p. 454\)](#).

### Options

**p**                Print the model text specification.

### Examples

```
text notes1
```

declares a text object named NOTES1.

### Cross-references

See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for further details on models. See [Chapter 14, “Graphs, Tables, and Text Objects”, on page 403](#) of the *User’s Guide* for a discussion of text objects in EViews.

See also [spec \(p. 454\)](#).

**tic**[Command](#)

Reset the timer.

### Syntax

Command:      `tic`

### Examples

The sequence of commands:

```
tic
```

*[some commands]*

toc

resets the timer, executes commands, and then displays the elapsed time in the status line.  
Alternatively:

tic

*[some commands]*

`!elapsed = @toc`

resets the time, executes commands, and saves the elapsed time in the control variable  
`!ELAPSED`.

## Cross-references

See also [toc \(p. 483\)](#) and [@toc \(p. 613\)](#).

<b>toc</b>	<a href="#">Command</a>
------------	-------------------------

Display elapsed time (since timer reset) in seconds.

## Syntax

Command:      toc

## Examples

The sequence of commands:

```
tic
[some commands]
toc
```

resets the timer, executes commands, and then displays the elapsed time in the status line.

The set of commands:

```
tic
[some commands]
!elapsed = @toc
[more commands]
toc
```

resets the time, executes commands, saves the elapsed time in the control variable  
`!ELAPSED`, executes additional commands, and displays the total elapsed time in the status line.

## Cross-references

See also [tic \(p. 482\)](#) and [@toc \(p. 613\)](#).

trace	<a href="#">Model View</a>
-------	----------------------------

Display trace view of a model showing iteration history for selected solved variables.

### Syntax

Model View:      `model_name.trace(options)`

### Options

p                  Print the block structure view.

## Cross-references

See “[Diagnostics](#)” on page 797 of the *User’s Guide* for further details on tracing model solutions.

See also [msg \(p. 354\)](#), [solve \(p. 451\)](#) and [solveopt \(p. 452\)](#).

tramoseats	<a href="#">Series Proc</a>
------------	-----------------------------

Run the external seasonal adjustment program Tramo/Seats using the data in the series.

`tramoseats` is available for annual, semi-annual, quarterly, and monthly series. The procedure requires at least n observations and can adjust up to 600 observations where:

$$n = \begin{cases} 36 & \text{for monthly data} \\ \max\{12, 4s\} & \text{for other seasonal data} \end{cases} \quad (\text{B.2})$$

### Syntax

Series Proc:      `series_name.tramoseats(options) [base_name]`

Enter the name of the original series followed by a dot, the keyword, and optionally provide a base name (no more than 20 characters long) to name the saved series. The default base name is the original series name. The saved series will have postfixes appended to the base name.

## Options

<code>runtype = arg</code> <i>(default = "ts")</i>	Tramo/Seats Run Specification: “ts” (run Tramo followed by Seats; the “opt = ” options are passed to Tramo, and Seats is run with the input file returned from Tramo), “t” (run only Tramo), “s” (run only Seats).
<code>save = arg</code>	<p>Specify series to save in workfile: you must use one or more from the following key word list: “hat” (forecasts of original series), “lin” (linearized series from Tramo), “pol” (interpolated series from Tramo), “sa” (seasonally adjusted series from Seats), “trd” (final trend component from Seats), “ir” (final irregular component from Seats), “sf” (final seasonal factor from Seats), “cyc” (final cyclical component from Seats).</p> <p>To save more than one series, separate the list of key words with a space. <i>Do not use commas</i> within the list of save series.</p> <p>The special key word “<code>save = *</code>” will save all series in the key word list. The five key words “sa”, “trd”, “ir”, “sf”, “cyc” will be ignored if “<code>runtype = t</code>”.</p>
<code>opt = arg</code>	A space delimited list of input namelist. <i>Do not use commas within the list</i> . The syntax for the input namelist is explained in the .PDF documentation file. See also “ <a href="#">Notes</a> ” on page 485 below.
<code>reg = arg</code>	A space delimited list for one line of reg namelist. <i>Do not use commas within the list</i> . This option must be used in pairs, either with another “ <code>reg = </code> ” option or “ <code>regname = </code> ” option. The reg namelist is available only for Tramo and its syntax is explained in the .PDF documentation file. See also “ <a href="#">Notes</a> ” on page 485 below.
<code>regname = arg</code>	Name of a series or group in the current workfile that contains the exogenous regressors specified in the previous “ <code>reg = </code> ” option. See “ <a href="#">Notes</a> ” on page 485 below.
<code>p</code>	Print the results of the Tramo/Seats procedure.

## Notes

The command line interface to Tramo/Seats does very little error checking of the command syntax. EViews simply passes on the options you provide “as is” to Tramo/Seats. If the

syntax contains an error, you will most likely to see the EViews error message “output file not found”. If you see this error message, check the input files produced by EViews for possible syntax errors as described in [“Trouble Shooting” on page 337](#) of the *User’s Guide*.

Additionally, here are some of the more commonly encountered syntax errors.

- To replicate the dialog options from the command line, use the following input options in the “opt =” list. See the .PDF documentation file for a description of each option.
  1. data frequency: “mq =”.
  2. forecast horizon: “npred =” for Tramo and “fh =” for Seats.
  3. transformation: “lam =”.
  4. ARIMA order search: “inic =” and “idif =”.
  5. Easter adjustment: “ieast =”.
  6. trading day adjustment: “itrad =”.
  7. outlier detection: “iatip =” and “aio =”.
- The command option input string list must be space delimited. *Do not use commas*. Options containing an equals sign should not contain any spaces around the equals; the space will be interpreted as a delimiter by Tramo/Seats.
- If you set “rtype = ts”, you are responsible for supplying either “seats = 1” or “seats = 2” in the “opt =” option list. EViews will issue the error message “Seats.itr not found” if the option is omitted. Note that the dialog option **Run Seats after Tramo** sets “seats = 2”.
- Each “reg =” or “regname =” option is passed to the input file as a separate line in the order that they appear in the option argument list. Note that these options must come in pairs. A “reg =” option must be followed by another “reg =” option that specifies the outlier or intervention series or by a “regname =” option that provides the name for an exogenous series or group in the current workfile. See the sample programs in the “./Example Files” directory.
- If you specify exogenous regressors with the “reg =” option, you must set the appropriate “ireg =” option (for the total number of exogenous series) in the “opt =” list.
- To use the “regname =” option, the preceding “reg =” list must contain the “user = -1” option and the appropriate “ilong =” option. *Do not use “user = 1” since EViews will always write data in a separate external file.* The “ilong =” option must be at least the number of observations in the current workfile sample *plus* the number of forecasts. The exogenous series should not contain any missing values in this range. *Note that Tramo may increase the forecast horizon, in which case the exogenous series is extended by appending zeros at the end.*

## Examples

```
freeze(tab1) show x.tramoseats(runtype=t, opt="lam=-1 iatip=1
    aio=2 va=3.3 noadmiss=1 seats=2", save=*) x
```

replicates the example file EXAMPLE.1 in Tramo. The output file from Tramo is stored in a text object named tab1. This command returns three series named X\_HAT, X\_LIN, X\_POL.

```
show x.TramoSeats(runtype=t, opt="NPRED=36 LAM=1 IREG=3
    INTERP=2 IMEAN=0 P=1 Q=0 D=0", reg="ISEQ=1 DELTA=1.0",
    reg="61 1", reg="ISEQ=8 DELTAS=1.0", reg="138 5 150 5 162 5
    174 5 186 5 198 5 210 5 222 5", reg="ISEQ=8 DELTAS=1.0",
    reg="143 7 155 7 167 7 179 7 191 7 203 7 215 7 227 7") x
```

replicates the example file EXAMPLE.2 in Tramo. This command produces an input file containing the lines:

```
$INPUT NPRED=36 LAM=1 IREG=3 INTERP=2 IMEAN=0 P=1 Q=0 D=0, $
$REG ISEQ=1 DELTA=1.0$
61 1
$REG ISEQ=8 DELTAS=1.0$
138 5 150 5 162 5 174 5 186 5 198 5 210 5 222 5
$REG ISEQ=8 DELTAS=1.0$
143 7 155 7 167 7 179 7 191 7 203 7 215 7 227 7
```

Additional examples replicating many of the example files provided by Tramo/Seats can be found in the “./Example Files” directory. You will also find files that compare seasonal adjustments from Census X12 and Tramo/Seats.

## Cross-references

See “[Tramo/Seats](#)” on page 333 of the *User’s Guide* for discussion. See also the Tramo/Seats documentation that accompanied your EViews distribution.

See [seas \(p. 417\)](#) and [x12 \(p. 522\)](#).

tsls	<a href="#">Command</a>    <a href="#">Equation Method</a>   <a href="#">Pool Method</a>   <a href="#">System Method</a>
------	--

**Two-stage least squares.**

Carries out estimation for equations, pools, or systems using two-stage least squares.

## Syntax

Command:	<code>tsls(options) y x1 [x2 x3 ...] @ z1 [z2 z3 ...]</code> <code>tsls(options) specification @ z1 [z2 z3 ...]</code>
Equation Method:	<code>eq_name.tsls(options) y x1 [x2 x3 ...] @ z1 [z2 z3 ...]</code> <code>eq_name.tsls(options) specification @ z1 [z2 z3 ...]</code>
Pool Method:	<code>pool_name.tsls(options) y x1 [x2 x3 ...] [@cxreg w1 w2 ...] [@per-reg w3 w4 ...] [@inst z1 z2 ...] [@cxinst z3 z4 ...] [@perinst z5 z6 ...]</code>
System Method:	<code>system_name.tsls(options)</code>

To use `tsls` as a command or equation method, list the dependent variable first, followed by the regressors, then any AR or MA error specifications, then an “@”-sign, and finally, a list of exogenous instruments. You may estimate nonlinear equations or equations specified with formulas by first providing a specification, then listing the instrumental variables after an “@”-sign.

There must be at least as many instrumental variables as there are independent variables. All exogenous variables included in the regressor list should also be included in the instrument list. A constant is included in the list of instrumental variables even if not explicitly specified.

## Options

### *General options*

<code>m = integer</code>	Set maximum number of iterations.
<code>c = number</code>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<code>deriv = keyword</code>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
<code>showopts / -showopts</code>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<code>p</code>	Print estimation results.

### *Additional Options for Non-Panel Equation estimation*

w = <i>series_name</i>	Weighted TSLS. Each observation will be weighted by multiplying by the specified series.
h	White's heteroskedasticity consistent standard errors.
n	Newey-West heteroskedasticity and autocorrelation consistent (HAC) standard errors.
s	Use the current coefficient values in "C" as starting values for equations with AR or MA terms (see also <a href="#">param (p. 383)</a> ).
s = <i>number</i>	Determine starting values for equations specified by list with AR or MA terms. Specify a number between zero and one representing the fraction of preliminary least squares estimates computed without AR or MA terms. Note that out of range values are set to "s = 1". Specifying "s = 0" initializes coefficients to zero. By default, EViews uses "s = 1".
z	Turn off backcasting in ARMA models.

### *Additional Options for Pool and Panel Equation estimation*

cx = <i>arg</i>	Cross-section effects. For fixed effects estimation, use "cx = f"; for random effects estimation, use "cx = r".
per = <i>arg</i>	Period effects. For fixed effects estimation, use "cx = f"; for random effects estimation, use "cx = r".
wgt = <i>arg</i>	GLS weighting: (default) none, cross-section system weights ("wgt = cxsur"), period system weights ("wgt = persur"), cross-section diagonal weights ("wgt = cxdiag"), period diagonal weights ("wgt = perdiag").
cov = <i>arg</i>	Coefficient covariance method: (default) ordinary, White cross-section system robust ("cov = cxwhite"), White period system robust ("cov = perwhite"), White heteroskedasticity robust ("cov = stackedwhite"), Cross-section system robust/PCSE ("cov = cxsur"), Period system robust/PCSE ("cov = persur"), Cross-section heteroskedasticity robust/PCSE ("cov = cxdiag"), Period heteroskedasticity robust ("cov = perdiag").

keepwgts	Keep full set of GLS weights used in estimation with object, if applicable (by default, only small memory weights are saved).
rancalc = <i>arg</i> ( <i>default</i> = “sa”)	Random component method: Swamy-Arora (“rancalc = sa”), Wansbeek-Kapteyn (“rancalc = wk”), Wallace-Hussain (“rancalc = wh”).
nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
coef = <i>arg</i>	Specify the name of the coefficient vector (if specified by list); the default is to use the “C” coefficient vector.
iter = <i>arg</i> ( <i>default</i> = “onec”)	Iteration control for GLS specifications: perform one weight iteration, then iterate coefficients to convergence (“iter = onec”), iterate weights and coefficients simultaneously to convergence (“iter = sim”), iterate weights and coefficients sequentially to convergence (“iter = seq”), perform one weight iteration, then one coefficient step (“iter = oneb”).  Note that random effects models currently do not permit weight iteration to convergence.
s	Use the current coefficient values in “C” as starting values for equations with AR or MA terms (see also <a href="#">param (p. 383)</a> ).
s = <i>number</i>	Determine starting values for equations specified by list with AR terms. Specify a number between zero and one representing the fraction of preliminary least squares estimates computed without AR terms. Note that out of range values are set to “s = 1”. Specifying “s = 0” initializes coefficients to zero. By default, EViews uses “s = 1”.

#### *Additional options for systems*

i	Iterate on the weighting matrix and coefficient vector simultaneously.
s	Iterate on the weighting matrix and coefficient vector sequentially.
o ( <i>default</i> )	Iterate only on the coefficient vector with one step of the weighting matrix.

c One step iteration of the coefficient vector after one step of the weighting matrix.

*l=number* Set maximum number of iterations on the first-stage iteration to get one-step weighting matrix.

## Examples

```
eq1.tsls y_d c cpi inc ar(1) @ lw(-1 to -3)
```

estimates EQ1 using TSLS regression of Y\_D on a constant, CPI, INC with AR(1) using a constant, LW(-1), LW(-2), and LW(-3) as instruments.

```
param c(1) .1 c(2) .1
eq1.tsls(s,m=500) y_d=c(1)+inc^c(2) @ cpi
```

estimates a nonlinear TSLS model using a constant and CPI as instruments. The first line sets the starting values for the nonlinear iteration algorithm.

```
sys1.tsls
```

estimates the system object using TSLS.

## Cross-references

See “[Two-stage Least Squares](#)” on page 457 and “[Two-Stage Least Squares](#)” on page 681 of the *User’s Guide* for details on two-stage least squares estimation in single equations and systems, respectively. “[Instrumental Variables](#)” on page 851 discusses estimation using pool objects, while “[Instrumental Variables Estimation](#)” on page 888 discusses estimation in panel structured workfiles.

See also [ls \(p. 329\)](#). For estimation of weighted TSLS in systems, see [wtsls \(p. 519\)](#).

<b>unlink</b>	<a href="#">Command</a>    <a href="#">Model Proc</a>
---------------	---

Break links in series objects or models.

## Syntax

Command: **unlink** *link\_names*

Model Proc: *object.unlink spec*

When used as a command, **unlink** converts link objects to ordinary series or alphas. Follow the keyword with a list of names of links to be converted to ordinary series (values). The list of links may include wildcard characters.

Used as a model proc, `unlink` breaks equation links in the model. Follow the name of the model object by a period, the keyword, and a specification for the variables to unlink.

The *spec* may contain either a list of the endogenous variables to be unlinked, or the keyword “@ALL”, instructing EViews to unlink all equations in the model.

Note: if a link is to another model or a system object, more than one endogenous variable may be associated with the link. If the *spec* contains any of the endogenous variables in a linked model or system, EViews will break the link for all of the variables found in the link.

### Examples

```
unlink gdp income
```

converts the link series GDP and INCOME to ordinary series.

```
unlink *
```

breaks all links in the current workfile page.

Used as a model proc, the expressions:

```
mod1.unlink @all
```

```
mod2.unlink z1 z2
```

unlink all of equations in MOD1, and all of the variables associated with the links for Z1 and Z2 in MOD2.

### Cross-references

See [Chapter 8, “Series Links”, on page 169](#) for a detailed description of link objects. See also [link \(p. 323\)](#) and [linkto \(p. 324\)](#).

See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews. See also [append \(p. 202\)](#), [merge \(p. 351\)](#) and [solve \(p. 451\)](#).

<b>update</b>	<a href="#">Model Proc</a>
---------------	----------------------------

**Update model specification.**

Recompiles the model and updates all links.

### Syntax

Model Proc:      `model.update`

Follow the name of the model object by a period and the keyword `update`.

## Examples

```
mod1.update
```

recompiles and updates all of the links in MOD1.

## Cross-references

See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews. See also [append \(p. 202\)](#), [merge \(p. 351\)](#) and [solve \(p. 451\)](#).

<b>updatecoefs</b>	<a href="#">Equation Proc</a>   <a href="#">Logl Proc</a>   <a href="#">Pool Proc</a>   <a href="#">Sspace Proc</a>   <a href="#">System Proc</a>
--------------------	---

Update coefficient object values from estimation object.

Copies coefficients from the estimation object into the appropriate coefficient vector or vectors.

## Syntax

```
Object Proc:      object.updatecoef
```

Follow the name of the estimation object by a period and the keyword `updatecoef`.

## Examples

```
equation eq1.ls y c x1 x2 x3
equation eq2.ls z c z1 z2 z3
eq1.updatecoef
```

places the coefficients from EQ1 in the default coefficient vector C.

```
coef(3) a
equation eq3.ls y=a(1)+z1^c(1)+log(z2+a(2))+exp(c(4)+z3/a(3))
equation eq2.ls z c z1 z2 z3
eq3.updatecoef
```

updates the coefficient vector A and the default vector C so that both contain the coefficients from EQ3.

## Cross-references

See also [coef \(p. 238\)](#).

**uroot**[Command](#) || [Group View](#) | [Pool View](#) | [Series View](#)

Carries out unit root tests on a single series, pool series, group of series, or panel structured series.

The ordinary, single series unit root tests include Augmented Dickey-Fuller (ADF), GLS detrended Dickey-Fuller (DFGLS), Phillips-Perron (PP), Kwiatkowski, *et. al.* (KPSS), Elliot, Rothenberg, and Stock (ERS) Point Optimal, or Ng and Perron (NP) tests for a unit root in the series (or its first or second difference).

If used on a series in a panel structured workfile, or with a pool series, or group of series, the procedure will perform panel unit root testing. The panel unit root tests include Levin, Lin and Chu (LLC), Breitung, Im, Pesaran, and Shin (IPS), Fisher - ADF, Fisher - PP, and Hadri tests on levels, or first or second differences.

### Syntax

Command:	<code>uroot(options) series_name</code>
Group View:	<code>group_name.uroot(options)</code>
Pool View:	<code>pool_name.uroot(options) pool_series</code>
Series View:	<code>series_name.uroot(options)</code>

Tests on a single series may be carried out in two ways: as a command, or as a procedure off of the series. Accordingly, you may enter the keyword followed by the individual series name, or you may enter the series name followed by a period and the keyword.

For other cases, you should enter the object name followed by a period, the keyword, and in the case of pool testing, the name of a pool “?” series.

### Options

#### *Basic Specification Options*

You should specify the exogenous variables and order of dependent variable differencing in the test equation using the following options:

<code>const (default)</code>	Include a constant in the test equation.
<code>trend</code>	Include a constant and a linear time trend in the test equation.
<code>none</code>	Do not include a constant or time trend (only available for the ADF and PP tests).

---

`dif = integer` Order of differencing of the series prior to running the test. Valid values are {0, 1, 2}.

For backward compatibility, the shortened forms of these options, “c”, “t”, and “n”, are presently supported. For future compatibility we recommend that you use the longer forms.

For ordinary (non-panel) unit root tests, you should specify the test type using one of the following keywords:

<code>adf (default)</code>	Augmented Dickey-Fuller.
<code>dfgls</code>	GLS detrended Dickey-Fuller (Elliot, Rothenberg, and Stock).
<code>pp</code>	Phillips-Perron.
<code>kpss</code>	Kwiatkowski, Phillips, Schmidt, and Shin.
<code>ers</code>	Elliot, Rothenberg, and Stock (Point Optimal).
<code>np</code>	Ng and Perron.

For panel testing, you may use one of the following keywords to specify the test:

<code>sum (default)</code>	Summary of all of the panel unit root tests.
<code>llc</code>	Levin, Lin, and Chu.
<code>breit</code>	Breitung.
<code>ips</code>	Im, Pesaran, and Shin.
<code>adf</code>	Fisher - ADF.
<code>pp</code>	Fisher - PP.
<code>hadri</code>	Hadri.

#### *Options for ordinary (non-panel) unit root tests*

In addition, the following panel specific options are available:

<code>hac = arg</code>	Method of estimating the frequency zero spectrum: “bt” (Bartlett kernel), “pr” (Parzen kernel), “qs” (Quadratic Spectral kernel), “ar” (AR spectral), “ardt” (AR spectral - OLS detrended data), “argls” (AR spectral - GLS detrended data).  Applicable to PP, KPSS, ERS, and NP tests. <i>The default settings are test specific</i> (“bt” for PP and KPSS, “ar” for ERS, “argls” for NP).
<code>band = arg</code> <code>b = arg</code> <code>(default = "nw")</code>	Method of selecting the bandwidth: “nw” (Newey-West automatic variable bandwidth selection), “a” (Andrews automatic selection), “number” (user specified bandwidth).  Applicable to PP, KPSS, ERS, and NP tests when using kernel sums-of-covariances estimators (where “hac = ” is one of {bt, pz, qs}).
<code>lag = arg</code> <code>(default = "a")</code>	Method of selecting lag length (number of first difference terms) to be included in the regression: “a” (automatic information criterion based selection), or “integer” (user-specified lag length).  Applicable to ADF and DFGLS tests, and for the other tests when using AR spectral density estimators (where “hac = ” is one of {ar, ardt, argls}).
<code>info = arg</code> <code>(default = "sic")</code>	Information criterion to use when computing automatic lag length selection: “aic” (Akaike), “sic” (Schwarz), “hqc” (Hannan-Quinn), “msaic” (Modified Akaike), “msic” (Modified Schwarz), “mhqc” (Modified Hannan-Quinn).  Applicable to ADF and DFGLS tests, and for other tests when using AR spectral density estimators (where “hac = ” is one of {ar, ardt, argls}).
<code>maxlag = integer</code>	Maximum lag length to consider when performing automatic lag length selection:  $\text{default} = \text{int}((12T/100)^{0.25})$  Applicable to ADF and DFGLS tests, and for other tests when using AR spectral density estimators (where “hac = ” is one of {ar, ardt, argls}).

### Options for panel unit root tests

The following panel specific options are available:

balance	Use balanced (across cross-sections or series) data when performing test.
hac = <i>arg</i> ( <i>default</i> = “bt”)	Method of estimating the frequency zero spectrum: “bt” (Bartlett kernel), “pr” (Parzen kernel), “qs” (Quadratic Spectral kernel).  Applicable to “Summary”, LLC, Fisher-PP, and Hadri tests.
band = <i>arg</i> , b = <i>arg</i> ( <i>default</i> = “nw”)	Method of selecting the bandwidth: “nw” (Newey-West automatic variable bandwidth selection), “a” (Andrews automatic selection), <i>number</i> (user-specified common bandwidth), <i>vector_name</i> (user-specified individual bandwidth).  Applicable to “Summary”, LLC, Fisher-PP, and Hadri tests.
lag = <i>arg</i>	Method of selecting lag length (number of first difference terms) to be included in the regression: “a” (automatic information criterion based selection), <i>integer</i> (user-specified common lag length), <i>vector_name</i> (user-specific individual lag length).  If the “balance” option is used,

$$\text{default} = \begin{cases} 1 & \text{if } (T_{\min} \leq 60) \\ 2 & \text{if } (60 < T_{\min} \leq 100) \\ 4 & \text{if } (T_{\min} > 100) \end{cases}$$

where  $T_{\min}$  is the length of the shortest cross-section or series, otherwise *default* = “a”.

Applicable to “Summary”, LLC, Breitung, IPS, and Fisher-ADF tests.

info = <i>arg</i> ( <i>default</i> = “sic”)	Information criterion to use when computing automatic lag length selection: “aic” (Akaike), “sic” (Schwarz), “hqc” (Hannan-Quinn), “msaic” (Modified Akaike), “msic” (Modified Schwarz), “mhqc” (Modified Hannan-Quinn). Applicable to “Summary”, LLC, Breitung, IPS, and Fisher-ADF tests.
maxlag = <i>arg</i>	Maximum lag length to consider when performing automatic lag length selection, where <i>arg</i> is an <i>integer</i> (common maximum lag length) or a <i>vector_name</i> (individual maximum lag length) $\text{default} = \text{int}(\min_i(12, T_i/3) \cdot (T_i/100)^{0.25})$ where $T_i$ is the length of the cross-section or series.

## Other options

p	Print output from the test.
---	-----------------------------

## Examples

The command:

```
gnp.uroot(adf,const,lag=3,save=mout)
```

performs an ADF test on the series GDP with the test equation including a constant term and three lagged first-difference terms. Intermediate results are stored in the matrix MOUT.

```
ip.uroot(dfgls,trend,info=sic)
```

runs the DFGLS unit root test on the series IP with a constant and a trend. The number of lagged difference terms is selected automatically using the Schwarz criterion.

```
unemp.uroot(kpss,const,hac=pr,b=2.3)
```

runs the KPSS test on the series UNEMP. The null hypothesis is that the series is stationary around a constant mean. The frequency zero spectrum is estimated using kernel methods (with a Parzen kernel), and a bandwidth of 2.3.

```
sp500.uroot(np,hac=ardt,info=maic)
```

runs the NP test on the series SP500. The frequency zero spectrum is estimated using the OLS AR spectral estimator with the lag length automatically selected using the modified AIC.

## Cross-references

See “[Unit Root Tests](#)” on page 502 of the *User’s Guide* for discussion of standard unit root tests performed on a single series, and “[Panel Unit Root Tests](#)” on page 514 of the *User’s Guide* for discussion of unit roots tests performed on panel structured workfiles, groups of series, or pooled data.

<b>usage</b>	<a href="#">Valmap View</a>
--------------	-----------------------------

Find series and alphas which use the valmap.

Display list of series and alpha objects which use the valmap.

## Syntax

Valmap View:      `valmap_name.stats(options)`

## Options

<code>p</code>	Print the usage table.
----------------	------------------------

## Examples

`map1.usage`

displays a list of series and alphas which use the valmap MAP1.

## Cross-references

For additional details, see “[Value Maps](#)” on page 155 of the *User’s Guide*. See also “[Valmap](#)” (p. 187).

See also [map](#) (p. 349).

<b>valmap</b>	<a href="#">Object Declaration</a>
---------------	------------------------------------

Declare a value map object.

## Syntax

Command:      `valmap valmap_name`

Follow the `valmap` keyword with a name for the object.

## Examples

The commands:

```
valmap mymap  
mymap.append 3 three  
mymap.append 99 "not in universe"
```

declare the valmap MYMAP and add two lines mapping the values 3 and 99 to the strings “three” and “not in universe”.

## Cross-references

For additional details, see “[Value Maps](#)” on page 155 of the *User’s Guide*. See also “[Valmap](#)” (p. 187).

See also [map](#) (p. 349).

<b>var</b>	<a href="#">Object Declaration</a>
------------	------------------------------------

Declare a var (Vector Autoregression) object.

## Syntax

Command:       **var** var\_name  
Command:       **var** var\_name.ls(*options*) lag\_pairs endog\_list [@ exog\_list]  
Command:       **var** var\_name.ec(*trend, n*) lag\_pairs endog\_list [@ exog\_list]

Declare the var as a name, or a name followed by an estimation method and specification.

The [ls](#) (p. 329) method estimates an unrestricted VAR using equation-by-equation OLS. You must specify the order of the VAR (using one or more pairs of lag intervals), and then provide a list of series or groups to be used as endogenous variables. You may include exogenous variables such as trends and seasonal dummies in the VAR by including an “@-sign” followed by a list of series or groups. A constant is automatically added to the list of exogenous variables; to estimate a specification without a constant, you should use the option “noconst”.

See [ec](#) (p. 271) for the error correction specification of a VAR.

## Options

**noconst**       Do not include a constant in the VAR specification  
(when combining declaration with [ls](#) (p. 329)  
method).

**p**               Print the estimation result if the estimation procedure is  
specified.

## Examples

```
var mvar.ls 1 4 8 8 m1 gdp tb3 @trend
```

declares and estimates an unrestricted VAR named MVAR with three endogenous variables (M1, GDP, TB3), five lagged terms (lags 1 through 4, and 8), a constant, and a linear trend.

```
var jvar.ec(c,2) 1 4 m1 gdp tb3
```

declares and estimates an error correction model named JVAR with three endogenous variables (M1, GDP, TB3), four lagged terms (lags 1 through 4), two cointegrating relations. The “c” option assumes a linear trend in data but only a constant in the cointegrating relations.

## Cross-references

See [Chapter 24, “Vector Autoregression and Error Correction Models”, on page 705](#) of the *User’s Guide* for a discussion of vector autoregressions.

See [ls \(p. 329\)](#) for standard VAR estimation, and [ec \(p. 271\)](#) for estimation of error correction models.

<a href="#">vars</a>	<a href="#">Model View</a>
----------------------	----------------------------

View of model organized by variable.

Display the model in variable form with identification of endogenous, exogenous, and identity variables, with dependency tracking.

## Syntax

Model View:      `model_name.vars`

## Cross-references

See [“Variable View” on page 782](#) of the *User’s Guide* for details. See [Chapter 26, “Models”, on page 761](#) of the *User’s Guide* for a general discussion of models.

See also [block \(p. 219\)](#), [text \(p. 482\)](#), and [eqs \(p. 275\)](#) for alternative representations of the model.

<a href="#">vector</a>	<a href="#">Object Declaration</a>
------------------------	------------------------------------

Declare a vector object.

The `vector` command declares and optionally initializes a (column) vector object.

## Syntax

Command:      `vector(size) vector_name`  
Command:      `vector(size) vector_name = assignment`

The keyword should be followed by the name you wish to give the vector. You may also provide an optional argument specifying the size of the vector. If you do not provide a size, EViews will create a single element vector. Once declared, vectors may be resized by repeating the command with a new size.

You may combine vector declaration and assignment. If there is no assignment statement, the vector will initially be filled with zeros.

## Examples

```
vector vec1  
vector(20) vec2=nrnd  
rowvector(10) row3=3  
vector vec3=row3
```

VEC1 is declared as a single element vector initialized to 0. VEC2 is declared as a column vector of size 20 containing a set of draws from the standard normal distribution. Although declared as a column vector, VEC3 is reassigned as a row vector of size 10 with all elements equal to 3.

## Cross-references

See [Chapter 3, “Matrix Language”, on page 23](#) of the *Command and Programming Reference* for a discussion of matrices and vectors in EViews.

See also [coef \(p. 238\)](#) and [rowvector \(p. 404\)](#).

<b>wald</b>	<a href="#">Equation View</a>   <a href="#">Logl View</a>   <a href="#">Pool View</a>   <a href="#">Sspace View</a>   <a href="#">System View</a>
-------------	---

Wald coefficient restriction test.

The `wald` view carries out a Wald test of coefficient restrictions for an estimation object.

## Syntax

Object View:      `object_name.wald restrictions`

Enter the object name, followed by a period, and the keyword. You must provide a list of the coefficient restrictions, with joint (multiple) coefficient restrictions separated by commas.

## Options

p	Print the test results.
---	-------------------------

## Examples

```
eq1.wald c(2)=0, c(3)=0
```

tests the null hypothesis that the second and third coefficients in equation EQ1 are jointly zero.

```
sys1.wald c(2)=c(3)*c(4)
```

tests the non-linear restriction that the second coefficient is equal to the product of the third and fourth coefficients in SYS1.

```
pool panel us uk jpn
panel.ls cons? c inc? @cxreg ar(1)
panel.wald c(3)=c(4)=c(5)
```

declares a pool object with three cross section members (US, UK, JPN), estimates a pooled OLS regression with separate AR(1) coefficients, and tests the null hypothesis that all AR(1) coefficients are equal.

## Cross-references

See “[Wald Test \(Coefficient Restrictions\)](#)” on page 556 of the *User’s Guide* for a discussion of Wald tests.

See also [cellipse](#) (p. 231), [testdrop](#) (p. 477), [testadd](#) (p. 474).

wfcreate	<a href="#">Command</a>
----------	-------------------------

Create a new workfile. The workfile becomes the active workfile.

## Syntax

Command: **wfcreate**(*options*) *frequency start\_date end\_date*  
*[num\_cross\_sections]*

Command: **wfcreate**(*options*) **u** *num\_observations*

The first form of the command may be used to create a new regular frequency workfile with the specified frequency, start, and end date. If you include the optional *num\_cross\_sections*, EViews will create a balanced panel page using integer identifiers for each of the cross-sections. Note that more complex panel structures may be created using [pagestruct](#) (p. 378).

The second form of the command is used to create an unstructured workfile with the specified number of observations.

## Options

`wf = wf_name`      Optional name for the new workfile.

`page = page_name`      Optional name for the page in the new workfile page. If not provided, EViews will use a name given by the page structure (e.g., “Undated”, “Annual”, “Daily5”).

## Examples

```
wfcreate (wf="annual", page="myproject") a 1950 2005  
wfcreate (wf="unstruct") u 1000
```

creates two workfiles. The first is a workfile named ANNUAL containing a single page named MYPROJECT containing annual data from 1950 to 2005; the second is a workfile named UNSTRUCT containing a single page named UNDATED with 1000 unstructured observations.

```
wfcreate (wf="griliches_grunfeld") a 1935 1954 10
```

creates the GRILICHES\_GRUNFELD workfile containing a paged named “ANNUAL” with 10 cross-sections of annual data for the years 1935 to 1954.

## Cross-references

See also [pagecreate \(p. 369\)](#) and [pagedelete \(p. 371\)](#).

<code>wfopen</code>	<a href="#">Command</a>
---------------------	-------------------------

Open a workfile. Reads in a previously saved workfile from disk, or reads the contents of a foreign data source into a new workfile.

The opened workfile becomes the default workfile; existing workfiles in memory remain on the desktop but become inactive.

## Syntax

Command: **`wfopen [path\]workfile_name`**

Command: **`wfopen(options) source_description [@keep keep_list] [@drop drop_list] [@keepmap keepmap_list] [@dropmap dropmap_list] [@selectif condition]`**

Command: **`wfopen(options)source_description table_description [@keep keep_list] [@drop drop_list] [@keepmap keepmap_list] [@dropmap dropmap_list] [@selectif condition]`**

The workfile or external data source is specified as the first argument following the command keyword and options. In most cases, the external data source is a file, so the *source\_description* will be the name of the file. Alternatively, the external data source may be the output from a web server, in which case the URL should be provided as the *source\_description*. Likewise, reading from an ODBC query, the ODBC DSN (data source name) should be used to identify the *source\_description*.

If the *source\_description* contains spaces, it must be enclosed in (double) quotes, to separate it from the table description.

In cases where there is more than one table that could be formed from the specified external data source, a *table\_description* may be provided to select the desired table. For example, when reading from an Excel file, an optional cell range may be provided to specify which data are to be read from the spreadsheet. When reading from an ODBC data source, a SQL query or table name must be used to specify the table of data to be read. When working with a text file, a *table\_description* of how to break up the file into columns and rows is required.

Note that ODBC support is provided only in the EViews 5 Enterprise Edition.

## Excel, HTML, Text, Binary File Table Descriptors

The following statements may be used in the table descriptions for Excel, HTML, text or binary data sources:

### *Excel Files*

- “range = *arg*”, where *arg* is a range of cells to read from the excel workbook, following the standard excel format [*worksheet!*][*topleft\_cell[:bottomright\_cell]*].

If the worksheet name contains spaces, it should be placed in single quotes. If the worksheet name is omitted, the cell range is assumed to refer to the currently active sheet. If only a top left cell is provided, a bottom right cell will be chosen automatically to cover the range of non-empty cells adjacent to the specified top left cell. If only a sheet name is provided, the first set of non-empty cells in the top left corner of the chosen worksheet will be selected automatically. As an alternative to specify-

ing an explicit range, a name which has been defined inside the excel workbook to refer to a range or cell may be used to specify the cells to read.

#### *HTML Pages*

- “table = *arg*”, where *arg* specifies which table to read in an HTML file/page containing multiple tables.

When specifying *arg*, you should remember that tables are named automatically following the pattern “Table01”, “Table02”, “Table03”, etc. If no table name is specified, the largest table found in the file will be chosen by default. Note that the table numbering may include trivial tables that are part of the HTML content of the file, but would not normally be considered as data tables by a person viewing the page.

- “skip = *int*”, where *int* is the number of rows to discard from the top of the HTML table.

#### *Text/Binary Files*

- “ftype = [ascii|binary]” specifies whether numbers and dates in the file are stored in a human readable text (ASCII), or machine readable (Binary) form.
- “rectype = [crlf|fixed|streamed]” describes the record structure of the file:
  - “crlf”, each row in the output table is formed using a fixed number of lines from the file (where lines are separated by carriage return/line feed sequences). This is the default setting.

“fixed”, each row in the output table is formed using a fixed number of characters from the file (specified in “recflen = *arg*”). This setting is typically used for files that contain no line breaks.

“streamed”, each row in the output table is formed by reading a fixed number of fields, skipping across lines if necessary. This option is typically used for files that contain line breaks, but where the line breaks are not relevant to how rows from the data should be formed.

- “reclines = *int*”, number of lines to use in forming each row when “rectype = crlf” (default is 1).
- “recflen = *int*”, number of bytes to use in forming each row when “rectype = fixed”.
- “recfields = *int*”, number of fields to use in forming each row when “rectype = streamed”.
- “skip = *int*”, number of lines (if rectype is “crlf”) or bytes (if rectype is not “crlf”) to discard from the top of the file.

- “comment = *string*”, where *string* is a double-quoted string, specifies one or more characters to treat as a comment indicator. When a comment indicator is found, everything on the line to the right of where the comment indicator starts is ignored.
- “emptylines = [keep|drop]”, specifies whether empty lines should be ignored (“drop”), or treated as valid lines (“keep”) containing missing values. The default is to ignore empty lines.
- “tabwidth = *int*”, specifies the number of characters between tab stops when tabs are being replaced by spaces (default = 8). Note that tabs are automatically replaced by spaces whenever they are not being treated as a field delimiter.
- “fieldtype = [delim|fixed|streamed|undivided]”, specifies the structure of fields within a record:
  - “Delim”, fields are separated by one or more delimiter characters
  - “Fixed”, each field is a fixed number of characters
  - “Streamed”, fields are read from left to right, with each field starting immediately after the previous field ends.
  - “Undivided”, read entire record as a single series.
- “quotes = [single|double|both|none]”, specifies the character used for quoting fields, where “single” is the apostrophe, “double” is the double quote character, and “both” means that either single or double quotes are allowed (default is “both”). Characters contained within quotes are never treated as delimiters.
- “singlequote”, same as “quotes = single”.
- “delim = [comma|tab|space|dblspace|white|dblwhite]”, specifies the character(s) to treat as a delimiter. “White” means that either a tab or a space is a valid delimiter. You may also use the abbreviation “d =” in place of “delim =”.
- “custom = “*arg1*””, specifies custom delimiter characters in the double quoted string. Use the character “t” for tab, “s” for space and “a” for any character.
- “mult = [on|off]”, to treat multiple delimiters as one. Default value is “on” if “delim” is “space”, “dblspace”, “white”, or “dblwhite”, and “off” otherwise.
- “endian = [big|little]”, selects the endianness of numeric fields contained in binary files.
- “string = [nullterm|nullpad|spacepad]”, specifies how strings are stored in binary files. If “nullterm”, strings shorter than the field width are terminated with a single zero character. If “nullpad”, strings shorter than the field width are followed by extra zero characters up to the field width. If “spacepad”, strings shorter than the field width are followed by extra space characters up to the field width.

The most important part of the Text/Binary table description is the format statement. You may specify the data format using the following descriptors:

- “fformat = (*arg1*, *arg2*, ...)”, Fortran format specification (see syntax below).
- “rformat = (*arg1*, *arg2*, ...)” column range format specification (see syntax below).
- “cformat = “*fmt* “, C printf/scanf format specification.

#### Fortran Syntax

The Fortran syntax follows the rules of the standard Fortran format statement. The syntax follows the general form:

- “fformat = ([*n1*]*Type*[*Width*][.*Precision*], [*n2*]*Type*[*Width*][.*Precision*], ... )”.

where *Type* specifies the underlying data type, and may be one of the following,

- I - integer
- F - fixed precision
- E - scientific
- A - alphanumeric
- X - skip

and *n1*, *n2*, ... are the number of times to read using the descriptor (default = 1). More complicated Fortran compatible variations on this format are possible.

#### Column Range Syntax

- “rformat = ([*ser\_name1*][*type*] *n1[-n2*], [*ser\_name1*] [*type*] *n3[-n4*], ... )”

where optional type is “\$” for string or “#” for number, and *n1*, *n2*, *n3*, *n4*, etc. are the range of columns containing the data.

#### All Types

- “colhead = *int*”, number of table rows to be treated as column headers.
- “namepos = [first|last|all|none]”, which row(s) of the column headers should be used to form the column name. The setting “first” refers to first line, “last” is last line, “all” is all lines and “none” is no lines. The remaining column header rows will be used to form the column description. The default setting is “all”.
- “Nonames”, the file does not contain a column header (same as “colhead = 0”).
- “names = (“*arg1*”, “*arg2*”,...)”, user specified column names, where *arg1*, *arg2*, ... are names of the first series, the second series, etc. when names are provided, these override any names that would otherwise be formed from the column headers.

- “descriptions = (“*arg1*”, “*arg2*”, …)”, user specified descriptions of the series. If descriptions are provided, these override any names that would otherwise be formed from the column headers.
- “na = “*arg1*””, text used to represent observations that are missing from the file. The text should be enclosed on double quotes.
- “scan = [*int* | all]”, number of rows of the table to scan during automatic format detection (“scan = all” scans the entire file).
- “firstobs = *int*”, first observation to be imported from the table of data (default is 1). This option may be used to start reading rows from partway through the table.
- “lastobs = *int*”, last observation to be read from the table of data (default is last observation of the file). This option may be used to read only part of the file, which may be useful for testing.

## ODBC or Microsoft Access Table Descriptors

When reading from an ODBC or Microsoft Access data source, you must provide a table description to indicate the table of data to be read. You may provide this information on one of two ways: by entering the name of a table in the data source, or by including an SQL query statement enclosed in double quotes.

## Options

<i>type</i> = <i>arg</i> / <i>t</i> = <i>arg</i>	Optional type specification: Access database file (“access”), Aremon-TSD file (“a”, “aremon”, “tsd”), Binary file (“binary”), Excel file (“excel”), Gauss dataset file (“gauss”), GiveWin/PcGive file (“g”, “give”), HTML file/page (“html”), ODBC database (“odbc”), ODBC Dsn file (“dsn”), ODBC query file (“msquery”), MicroTSP workfile (“dos” “microtsp”), MicroTSP Macintosh workfile (“mac”), Rats file (“r”, “rats”), Rats portable/Troll file (“l”, “trl”), SAS program file (“sasprog”), SAS transport file (“sasxport”), SPSS file (“spss”), SPSS portable file (“spssport”), Stata file (“stata”), Text file (“text”), TSP portable file (“t”, “tsp”).
<i>wf</i> = <i>wf_name</i>	Optional name for the new workfile.
<i>page</i> = <i>page_name</i>	Optional name for the page in the new workfile.

## Examples

The following examples illustrate the use of wfopen to open an existing workfiles or to read foreign data into a new workfile.

### *EViews and MicroTSP*

```
wfopen c:\data\macro
```

loads a previously saved EViews workfile MACRO.WF1 from the DATA directory in the C drive.

```
wfopen c:\tsp\nipa.wf
```

loads a MicroTSP workfile NIPA.WF. If you do not use the workfile type option, you should add the extension “.WF” to the workfile name when loading a DOS MicroTSP workfile. An alternative method specifies the type explicitly:

```
wfopen(type=dos) nipa
```

### *Excel*

```
wfopen "c:\data files\data.xls"
```

loads the active sheet of DATA.XLS into a new workfile.

```
wfopen(page=GDP) "c:\data files\data.xls" range="GDP data"  
@drop X
```

reads the data contained in the “GDP data” sheet of DATA.XLS into a new workfile. The data for the series X is dropped, and the name of the new workfile page is “GDP”.

```
wfopen(type=excel, wf=PRCS) c:\data.xls range="prices" @keep s*  
@drop *ic
```

loads into the new PRCS workfile a subset of the data contained in the PRICES data sheet of DATA.XLS. The load keeps only the series that begin with “S”, but does not include the ones that end in “IC”.

```
wfopen c:\data.xls range='sheet1 data'!B2:D30
```

loads into a new workfile the data contained from cells B2 through D30 on worksheet “sheet 1 data” in the workbook DATA.XLS.

### *Stata*

```
wfopen(type=stata) c:\data.dta @dropmap map1 map2
```

load a Stata file DATA.DTA into a new workfile, dropping map MAP1 and MAP2.

**SAS**

```
wfopen(type=sasxport) c:\data.xpt
```

loads a sas transport file data.xpt into a new workfile.

```
wfopen c:\inst.sas
```

creates a workfile by reading from external data using the SAS program statements in INST.SAS. The program may contain a limited set of SAS statements which are commonly used in reading in a data file.

***ASCII text files (.txt, .csv, etc.)***

```
wfopen c:\data.csv skip=5, names=(gdp, inv, cons)
```

reads DATA.CSV into a new workfile page, skipping the first 5 rows and naming the series GDP, INV, and CONS.

```
wfopen(type=text) c:\date.txt delim=comma
```

loads the comma delimited data DATE.TXT into a new workfile.

```
wfopen(type=raw,rectype=fixed) c:\data.txt skip=8, for-  
mat=(F10,X23,A4)
```

loads a text file with fixed length data into a new workfile, skipping the first 8 rows. The reading is done as follows: read the first 10 characters as a fixed precision number, after that, skip the next 23 characters (X23), and then read the next 4 characters as strings (A4).

```
wfopen(type=raw,rectype=fixed) c:\data.txt format=2(4F8,2I2)
```

loads the text file as a workfile using the specified explicit format. The data will be a repeat of four fixed precision numbers of length 8 and two integers of length 2. This is the same description as “format = (F8,F8,F8,F8,I2,I2,F8,F8,F8,I2,I2)”.

```
wfopen(type=raw,rectype=fixed) c:\data.txt rformat=(GDP 1-2 INV  
3 CONS 6-9)
```

loads the text file as a workfile using column range syntax. The reading is done as follows: the first series is located at the first and second character of each row, the second series occupies the 3rd character, the third series is located at character 6 through 9. The series will named GDP, INV, and CONS.

***Html***

```
wfopen "c:\data.html"
```

loads into a new workfile the data located on the HTML file DATA.HTML located on the C:\ drive

```
wfopen(type=html) "http://www.tradingroom.com.au/apps/mkt/  
forex.ac" colhead=3, namepos=first
```

loads into a new workfile the data with the given URL located on the website site “<http://www.tradingroom.com.au>”. The column header is set to three rows, with the first row used as names for columns, and the remaining two lines used to form the descriptions.

#### *ODBC (Enterprise Edition only)*

```
wfopen c:\data.dsn CustomerTable
```

opens in a new workfile the table named CUSTOMERTABLE from the ODBC database described in the DATA.DSN file.

```
wfopen(type=odbc) "my server" "select * from customers where  
id>30" @keep p*
```

opens in a new workfile with SQL query from database using the server “MY SERVER”, keeping only variables that begin with P. The query selects all variables from the table CUSTOMERS where the ID variable takes a value greater than 30.

#### Cross-references

See [Chapter 3, “Workfile Basics”, on page 43](#) of the *User’s Guide* for a basic discussion of workfiles.

See also [pageload \(p. 371\)](#), [read \(p. 391\)](#), [fetch \(p. 279\)](#), [wfsave \(p. 512\)](#), and [page-save \(p. 373\)](#).

wfsave	<a href="#">Command</a>
--------	-------------------------

Save the default workfile as an EViews workfile (.wf1 file) or as a foreign file or data source.

#### Syntax

- Command:        *wfsave(options) [path\]filename*  
Command:        *wfsave(options) source\_description [@keep keep\_list] [@drop drop\_list] [@keepmap keepmap\_list] [@dropmap dropmap\_list] [@smpl smpl\_spec]*  
Command:        *wfsave(options) source\_description table\_description [@keep keep\_list] [@drop drop\_list] [@keepmap keepmap\_list] [@dropmap dropmap\_list] [@smpl smpl\_spec]*

saves the active workfile in the specified directory using *filename*. By default, the workfile is saved as an EViews workfile, but options may be used to save all or part of the active

page in a foreign file or data source. See [wfopen \(p. 504\)](#) for details on the syntax for *source\_descriptions* and *table\_descriptions*.

## Options

### *Workfile Save Options*

1	Save using single precision.
2	Save using double precision.
c	Save compressed workfile (not compatible with EViews versions prior to 5.0).

The default workfile save settings use the Global Options.

### *Foreign Source Save Options*

type = <i>arg</i> , t = <i>arg</i>	Optional type specification. Access database file (“access”), Aremon-TSD file (“a”, “aremon”, “tsd”), Binary file (“binary”), EViews database file (“e”, “evdb”), Excel file (“excel”), Gauss dataset file (“gauss”), GiveWin/PcGive file (“g”, “give”), HTML file/page (“html”), ODBC database (“odbc”), ODBC Dsn file (“dsn”), ODBC query file (“msquery”), MicroTSP workfile (“dos”, “microtsp”), MicroTSP Macintosh workfile (“mac”), Rats file (“r”, “rats”), Rats portable/Troll file (“l”, “trl”), SAS transport file (“sasxport”), SPSS file (“spss”), SPSS portable file (“spss-port”), Stata file (“stata”), Text file (“text”), TSP portable file (“t”, “tsp”).
mode = create	Create new file only; error on attempt to overwrite.
maptype = <i>arg</i>	Write selected maps as: numeric (“n”), character (“c”), both numeric and character (“b”).
mapval	Write mapped values for series with attached value labels.

## Cross-references

See also [pagesave \(p. 373\)](#), [wfopen \(p. 504\)](#), and [pageload \(p. 371\)](#).

wfselect	<a href="#">Command</a>
----------	-------------------------

Make the selected workfile page the active workfile page.

### Syntax

Command:      `wfselect wfname\|pgname`

where *wfname* is the name of a workfile that has been loaded into memory. You may optionally provide the name of a page in the new default workfile that you wish to be made active.

### Examples

```
wfselect myproject  
wfselect myproject\page2
```

both change the default workfile to MYPROJECT. The first command uses the default active page, while the second changes the page to PAGE2.

### Examples

#### Cross-references

See also [pageselect \(p. 374\)](#).

white	<a href="#">Equation View</a>   <a href="#">Var View</a>
-------	--

Performs White's test for heteroskedasticity of residuals.

Carries out White's test for heteroskedasticity of the residuals of the specified equation or Var object. By default, the test is computed without the cross-product terms (using only the terms involving the original variables and squares of the original variables). You may elect to compute the original form of the White test that includes the cross-products.

White's test is not available for equations estimated by `binary`, `ordered`, `censored`, or `count`. For a Var object, the command computes the multivariate version of the test.

### Syntax

Equation View:    `eq_name.white(options)`

Var View:          `var_name.white(options)`

## Options

c	Include all possible nonredundant cross-product terms in the test regression.
p	Print the test results.

### Options for Var View

name = <i>arg</i>	Save test statistics in named matrix object. See below for a description of the statistics stored in the matrix.
-------------------	--

For var views, the “name =” option stores the results in a  $(r + 1) \times 5$  matrix, where  $r$  is the number of unique residual cross-product terms. For a VAR with  $k$  endogenous variables,  $r = k(k + 1)/2$ . The first  $r$  rows contain statistics for each individual test equation, where the first column is the regression R-squared, the second column is the  $F$ -statistic, the third column is the  $p$ -value of  $F$ -statistic, the 4th column is the  $T \times R^2 \chi^2$  statistic, and the fifth column is the  $p$ -value of the  $\chi^2$  statistic.

The numerator and denominator degrees of freedom of the  $F$ -statistic are stored in the third and fourth columns, respectively, of the  $(r + 1)$ -st row, while the  $\chi^2$  degrees of freedom is stored in the fifth column of the  $(r + 1)$ -st row.

In the  $(r + 1)$ -st row and first column contains the joint (system) LM chi-square statistic and the second column contains the degrees of freedom of this  $\chi^2$  statistic.

## Examples

```
eq1.white(c)
```

carries out the White test of heteroskedasticity including all possible cross-product terms.

## Cross-references

See “[White's Heteroskedasticity Test](#)” on page 566 for a discussion of White’s test. For the multivariate version of this test, see “[White Heteroskedasticity Test](#)” on page 712 of the *User’s Guide*.

wls	<a href="#">System Method</a>
-----	-------------------------------

Estimates a system of equations using weighted least squares.

To perform weighted least squares in single equation estimation, see [ls \(p. 329\)](#).

## Syntax

System Method: `system_name.wls(options)`

## Options

i	Iterate simultaneously over the weighting matrix and coefficient vector.
s	Iterate sequentially over the computation of the weighting matrix and the estimation of the coefficient vector.
o ( <i>default</i> )	Iterate the estimate of the coefficient vector to convergence following one-iteration of the weighting matrix.
c	One step (iteration) of the coefficient vector estimates following one iteration of the weighting matrix.
<i>m = integer</i>	Maximum number of iterations.
<i>c = number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<i>l = number</i>	Set maximum number of iterations on the first-stage coefficient estimation to get one-step weighting matrix.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
deriv = <i>keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
p	Print the estimation results.

## Examples

`sys1.wls`

estimates the system of equations in SYS1 by weighted least squares.

## Cross-references

See [Chapter 23, “System Estimation”, on page 679](#) of the *User’s Guide* for a discussion of system estimation.

See also the available options for weighted least squares in [ls \(p. 329\)](#).

<b>workfile</b>	<a href="#">Command</a>
-----------------	-------------------------

Create or change workfiles.

No longer supported; provided for backward compatibility. This command has been replaced by [wfcreate \(p. 503\)](#) and [pageselect \(p. 374\)](#).

<b>write</b>	<a href="#">Command</a>    <a href="#">Coef Proc</a>   <a href="#">Matrix Proc</a>   <a href="#">Pool Proc</a>   <a href="#">Rowvector Proc</a>   <a href="#">Sym Proc</a>   <a href="#">Vector Proc</a>
--------------	--

Write EViews data to a text (ASCII), Excel, or Lotus file on disk.

Creates a foreign format disk file containing EViews data. May be used to export EViews data to another program.

## Syntax

Command:	<code>write(options) [path\]filename arg1 [arg2 arg3 ...]</code>
Pool Proc:	<code>pool_name.write(options) [path\]filename pool_series1 [pool_series2 pool_series3 ...]</code>
Coef Proc:	<code>coef_name.write(options) [path\]filename</code>
Matrix Proc:	<code>matrix_name.write(options) [path\]filename</code>

Follow the keyword by a name for the output file and list the series to be written. The optional path name may be on the local machine, or may point to a network drive. If the path name contains spaces, enclose the entire expression in double quotation marks. To write matrix objects, simply provide a filename; the entire matrix will be exported.

Note that EViews cannot, at present, write into an existing file. The file that you select will, if it exists, be replaced.

## Options

Options are specified in parentheses after the keyword and are used to specify the format of the output file.

### File type

<code>t = dat, txt</code>	ASCII (plain text) files.
<code>t = wk1, wk3</code>	Lotus spreadsheet files.
<code>t = xls</code>	Excel spreadsheet files.

If you omit the “t =” option, EViews will determine the type based on the file extension. Unrecognized extensions will be treated as ASCII files. For Lotus and Excel spreadsheet files specified without the “t =” option, EViews will automatically append the appropriate extension if it is not otherwise specified.

#### ASCII text files

na = <i>string</i>	Specify text string for NAs. Default is “NA”.
names ( <i>default</i> ) / nonames	[Write / Do not write] series names.
dates ( <i>default</i> ) / nodates	[Write / Do not write] dates/obs and (pool) cross-section identifiers (only for pool write).
id	Write cross-section identifier (only for pool write).
d = <i>arg</i>	Specify delimiter ( <i>default</i> is tab): “s” (space), “c” (comma).
t	Write by series (or transpose the data for matrix objects). Default is to read by obs with series in columns.

#### Spreadsheet (Lotus, Excel) files

letter_number	Coordinate of the upper-left cell containing data.
names ( <i>default</i> ) / nonames	[Write / Do not write] series names.
dates ( <i>default</i> ) / nodates	[Write / Do not write] dates/obs and (pool) cross-section identifiers (only for pool write)
id	Write cross-section identifier (only for pool write).
dates = <i>arg</i>	Excel format for writing date: “first” (convert to the first day of the corresponding observation if necessary), “last” (convert to the last day of the corresponding observation).
t	Write by series (or transpose the data for matrix objects). Default is to write by obs with each series in columns.

#### Pooled data writing

bycross ( <i>default</i> ) / byper	Stack pool data by [cross-section / date or period] (only for pool write).
------------------------------------	--

## Examples

```
write(t=txt,na=.,d=c,dates) a:\dat1.csv hat1 hat_se1
```

Writes the two series HAT1 and HAT\_SE1 into an ASCII file named DAT1.CSV on the A drive. The data file is listed by observations, NAs are coded as “.” (dot), each series is separated by a comma, and the date/observation numbers are written together with the series names.

```
write(t=txt,na=.,d=c,dates) dat1.csv hat1 hat_se1
```

writes the same file in the default directory.

```
mypool.write(t=xls,per) "\\network\drive a\growth" gdp? edu?
```

writes an Excel file GROWTH.XLS in the specified directory. The data are organized by observation, and are listed by period/time.

## Cross-references

See “[Exporting to a Spreadsheet or Text File](#)” on page 106 of the *User’s Guide* for a discussion. Pool writing is discussed in “[Exporting Pooled Data](#)” on page 828 of the *User’s Guide*.

See also [pagesave \(p. 373\)](#) and [read \(p. 391\)](#).

wtsls	<a href="#">System Method</a>
-------	-------------------------------

Perform weighted two-stage least squares estimation of a system of equations.

To estimate single equation weighted two-stage least squares using an equation object, see [ls \(p. 329\)](#).

## Syntax

System Method:    `system_name.wtsls(options)`

## Options

i	Iterate simultaneously over the weighting matrix and coefficient vector.
s	Iterate sequentially over the computation of the weighting matrix and the estimation of the coefficient vector.
o (default)	Iterate the coefficient vector to convergence following one-iteration of the weighting matrix.

c	One step (iteration) of the coefficient vector following one iteration of the weighting matrix.
<i>m = integer</i>	Maximum number of iterations.
<i>c = number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<i>l = number</i>	Set maximum number of iterations on the first-stage iteration to get the one-step weighting matrix.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<i>deriv = keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
p	Print estimation results.

## Examples

```
sys1.wtsls
```

estimates the system of equations in SYS1 by weighted two-stage least squares.

## Cross-references

See “[Weighted Two-Stage Least Squares](#)” on page 681 of the *User’s Guide* for further discussion.

See also [tsls](#) (p. 487) for both unweighted and weighted single equation 2SLS.

x11	<a href="#">Series Proc</a>
-----	-----------------------------

Seasonally adjust series using the Census X11.2 method.

## Syntax

Series Proc:      *series\_name.x11(options) adj\_name [fac\_name]*

The X11 procedure carries out Census X11.2 seasonal adjustment. Enter the name of the original series followed by a period, the keyword, and then provide a name for the season-

ally adjusted series. You may optionally list a second series name for the seasonal factors. The seasonal adjustment method is specified as an option in parentheses after the `x11` keyword.

*The X11 procedure is available only for quarterly and monthly series.* The procedure requires at least four full years of data, and can adjust up to 20 years of monthly data and 30 years of quarterly data.

## Options

m	Multiplicative seasonals.
a	Additive seasonals.
s	Use sliding spans.
h	Adjustment for all holidays (only for monthly data specified with the <code>m</code> option).
i	Adjustment for holidays if significant (only for monthly data specified with the “ <code>m</code> ” option).
t	Adjustment for all trading days (only for monthly data).
q	Adjustment for trading days if significant (only for monthly data).
p	Print the X11 results.

## Examples

```
sales.x11(m,h) salesx11 salesfac
```

seasonally adjusts the SALES series and saves the adjusted series as SALESX11 and the seasonal factors as SALESFAC. The adjustment assumes multiplicative seasonals and makes adjustment for all holidays.

## Cross-references

See “[Census X11 \(Historical\)](#)” on page 332 of the *User’s Guide* for a discussion of Census X11 seasonal adjustment method.

Note that the X11 routines are separate programs provided by the Census and are installed in the EViews directory in the files X11Q2.EXE and X11SS.EXE. Additional documentation for these programs can also be found in your EViews directory in the text files X11DOC1.TXT through X11DOC3.TXT.

See also [seas](#) (p. 417), [seasplot](#) (p. 418), [tramoseats](#) (p. 484), and [x12](#) (p. 522).

x12	Series Proc
-----	-------------

Seasonally adjust series using the Census X12 method.

x12 is available only for quarterly and monthly series. The procedure requires at least 3 full years of data and can adjust up to 600 observations (50 years of monthly data or 150 years of quarterly data).

### Syntax

Series Proc:      *series\_name.x12(options) base\_name*

Enter the name of the original series followed by a dot, the keyword, and a base name (no more than the maximum length of a series name minus 4) for the saved series. If you do not provide a base name, the original series name will be used as a base name. See the description in “*save =*” option below for the naming convention used to save series.

### Options

#### *Commonly Used Options*

mode = <i>arg</i> ( <i>default</i> = “m”)	Seasonal adjustment method: “m” (multiplicative adjustment; <i>Series must take only non-negative values</i> ), “a” (additive adjustment), “p” (pseudo-additive adjustment), “l” (log-additive seasonal adjustment; <i>Series must take only positive values</i> ).
filter = <i>arg</i> ( <i>default</i> = “msr”)	Seasonal filter: “msr” (automatic, moving seasonality ratio), “x11” (X11 default), “stable” (stable), “s3x1” (3x1 moving average), “s3x3” (3x3 moving average), “s3x5” (3x5 moving average), “s3x9” (3x9 moving average), “s3x15” (3x15 moving average seasonal filter; <i>Series must have at least 20 years of data</i> ).

save = "arg"	<p>Optionally saved series keyword enclosed in quotes. List the extension (given in Table 6-8, p.71 of the <i>X12-ARIMA Reference Manual</i>) for the series you want to save. The created series will use names of the form <i>basename</i>, followed by a series keyword specific suffix. Commonly used options and suffixes are: ""d10"" (final seasonal factors, saved with suffix <i>_sf</i>), ""d11"" (final seasonally adjusted series using <i>_sa</i>), ""d12"" (final trend-cycle component using <i>_tc</i>), ""d13"" (final irregular component using <i>_ir</i>).</p> <p>All other options are named using the option symbol. For example "save = "d16"" will store a series named <i>basename_D16</i>.</p> <p>To save more than two series, separate the list with a space. For example, "save = "d10 d12"" saves the seasonal factors and the trend-cycle series.</p>
tf = arg	Transformation for regARIMA: "logit" (Logit transformation), "auto" (automatically choose between no transformation and log transformation), <i>number</i> (Box-Cox power transformation using specified parameter; use "tf = 0" for log transformation).
sspan	Sliding spans stability analysis. <i>Cannot be used along with the "h" option.</i>
history	Historical record of seasonal adjustment revisions. <i>Cannot be used along with the "sspan" option.</i>
check	Check residuals of regARIMA.
outlier	Outlier analysis of regARIMA.
x11reg = arg	Regressors to model the irregular component in seasonal adjustment. Regressors must be chosen from the predefined list in Table 6-14, p. 88 of the <i>X12-ARIMA Reference Manual</i> . To specify more than one regressor, separate by a space within the double quotes.
reg = arg_list	Regressors for the regARIMA model. Regressors must be chosen from the predefined list in Table 6-17, pp. 100-101 of the <i>X12-ARIMA Reference Manual</i> . To specify more than one regressor, separate by a space within the double quotes.

arima = <i>arg</i>	ARIMA spec of the regARIMA model. Must follow the X12 ARIMA specification syntax. <i>Cannot be used together with the “amdl = ” option.</i>
amdl = f	Automatically choose the ARIMA spec. Use forecasts from the chosen model in seasonal adjustment. <i>Cannot be used together with the arima = option and must be used together with the mfile = option.</i>
amdl = b	Automatically choose the ARIMA spec. Use forecasts and backcasts from the chosen model in seasonal adjustment. <i>Cannot be used together with the “arima = ” option and must be used together with the “mfile = ” option.</i>
best	Sets the method option of the auto model spec to best (default is first). Also sets the identify option of the auto model spec to all (default is first). <i>Must be used together with the “amdl = ” option.</i>
modelsmpl = <i>arg</i>	Sets the subsample for fitting the ARIMA model. Either specify a sample object name or a sample range. <i>The model sample must be a subsample of the current workfile sample and should not contain any breaks.</i>
mfile = <i>arg</i>	Specifies the file name (include the extension, if any) that contains a list of ARIMA specifications to choose from. <i>Must be used together with the “amdl = ” option.</i> The default is the X12A.MDL file provided by the Census.
outsmpl	Use out-of-sample forecasts for automatic model selection. Default is in-sample forecasts. <i>Must be used together with the “amdl = ” option.</i>
plotspectra	Save graph of spectra for differenced seasonally adjusted series and outlier modified irregular series. The saved graph will be named GR_ <i>seriesname</i> _SP.
p	Print X12 procedure results.

### Other Options

hma = <i>integer</i>	Specifies the Henderson moving average to estimate the final trend-cycle. The X12 default is automatically selected based on the data. To override this default, specify an <i>odd integer between 1 and 101</i> .
sigl = <i>arg</i>	Specifies the lower sigma limit used to downweight extreme irregulars in the seasonal adjustment. The default is 1.5 and you can specify any positive real number.
sigh = <i>arg</i>	Specifies the upper sigma limit used to downweight extreme irregulars in the seasonal adjustment. The default is 2.5 and you can specify any positive real number less than the lower sigma limit.
ea	Nonparametric Easter holiday adjustment (x11easter). <i>Cannot be used together with the “easter[w]” regressor in the “reg = ” or “x11reg = ” options.</i>
f	Appends forecasts up to one year to some optionally saved series. Forecasts are appended only to the following series specified in the “save = ” option: ““b1”” (original series, adjusted for prior effects), ““d10”” (final seasonal factors), ““d16”” (combined seasonal and trading day factors).
flead = <i>integer</i>	Specifies the number of periods to forecast (to be used in the seasonal adjustment procedure). The default is one year and you can specify an integer up to 60.
fback = <i>integer</i>	Specifies the number of periods to backcast (to be used in the seasonal adjustment procedure). The default is 0 and you can specify an integer up to 60. No backcasts are produced for series more than 15 years long.
aicx11	Test (based on AIC) whether to retain the regressors specified in “x11reg = ”. <i>Must be used together with the “x11reg = ” option.</i>
aicreg	Test (based on AIC) whether to retain the regressors specified in “reg = ”. <i>Must be used together with the “reg = ” option.</i>

`sfile = arg` Path/name (including extension, if any) of user provided specification file. The file must follow a specific format; see the discussion below.

#### *User provided spec file*

EViews provides most of the basic options available in the X12 program. For users who need to access the full set of options, we have provided the ability to pass your own X12 specification file from EViews. The advantage of using this method (as opposed to running the X12 program in DOS) is that EViews will automatically handle the data in the input and output series.

To provide your own specification file, specify the path/name of your file using the “`sfile =`” option in the `x12` proc. Your specification file should follow the format of an X12 specification file as described in the *X12-ARIMA Reference Manual*, with the following exceptions:

- the specification file should have neither a series spec nor a composite spec.
- the `x11` spec must include a save option for `D11` (the final seasonally adjusted series) in addition to any other extensions you want to store. EViews will always look for `D11`, and will error if it is not found.
- to read back data for a “`save`” option other than `D11`, you must include the “`save =`” option in the `x12` proc. For example, to obtain the final trend-cycle series (`D12`) into EViews, you must have a “`save =`” option for `D12` (and `D11`) in the `x11` spec of your specification file and a “`save = d12`” option in the EViews `x12` proc.

Note that when you use an “`sfile =`” option, EViews will ignore any other options in the `x12` proc, except for the “`save =`” option.

#### *Difference between the dialog and command line*

The options corresponding to the **Trading Day/Holiday** and **Outliers** tab in the X12 dialog should be specified by listing the appropriate regressors in the “`x11reg =`” and “`reg =`” options.

## Examples

The command:

```
sales.x12(mode=m,save="d10 d12") salesx12
```

seasonally adjusts the SALES series in multiplicative mode. The seasonal factors (`d10`) are stored as `SALESX12_SF` and the trend-cycles series is stored as `SALESX12_TC`.

```
sales.x12(tf=0,arima="(0 0 1)",reg="const td")
```

specifies a regARIMA model with a constant, trading day effect variables, and MA(1) using a log transformation. This command does not store any series.

```
freeze(x12out) sales.x12(tf=auto, amdl=f, mfile=
    "c:\eviews\mymdl.txt")
```

stores the output from X12 in a text object named X12OUT. The options specify an automatic transformation and an automatic model selection from the file MYMDL.TXT.

```
revenue.x12(tf=auto,sfile="c:\eviews\spec1.txt",save="d12
    d13")
```

adjusts the series REVENUE using the options given in the SPEC1.TXT file. Note the following: (1) the “tf = auto” option will be ignored (you should instead specify this option in your specification file) and (2) EViews will save two series REVENUE\_TC and REVENUE\_IR which will be filled with NAs unless you provided the “save = ” option for D12 and D13 in your specification file.

```
freeze(x12out) sales.x12(tf=auto, amdl=f, mfile=
    "c:\eviews\mymdl.txt")
```

stores the output from X12 in the text object X12OUT. The options specify an automatic transformation and an automatic model selection from the file MYMDL.TXT. The seasonally adjusted series is stored as SALES\_SA by default.

```
revenue.x12(tf=auto,sfile="c:\eviews\spec1.txt",save="d12
    d13")
```

adjusts the series REVENUE using the options given in the SPEC1.TXT file. Note the following: (1) the “tf = auto” option will again be ignored (you should instead specify this in your specification file) and (2) EViews will error if you did not specify a “save = ” option for D11, D12, and D13 in your specification file.

## Cross-references

See [“Census X12” on page 325](#) of the *User’s Guide* for a discussion of the Census X12 program. The documentation for X12, *X12-ARIMA Reference Manual*, may be found in the DOCS subdirectory of your EViews directory, in the PDF files FINALPT1.PDF and FINALPT2.PDF.

See also [seas \(p. 417\)](#) and [x11 \(p. 520\)](#).

**xy**[Command](#) || [Graph Command](#) | [Group View](#) | [Matrix View](#) | [Sym View](#)

Display XY graph of object, or change existing graph object type to XY (if possible).

By default, the first series or column of data will be located along the horizontal axis and the remaining data on the vertical axis. You may optionally choose to plot the data in pairs, where the first two series or columns are plotted against each other, the second two series or columns are plotted against each other, and so forth.

### Syntax

Command:      `xy(options) arg1 [arg2 arg3 ...]`

Object View:    `group_name.xy(options)`

Graph Proc:    `graph_name.xy(options)`

If used as a command, follow the keyword by a list of series and group objects, or by a matrix object. There must be at least two series or columns in the data to be graphed.

If changing the type of a graph, the default behavior is to use the existing settings for lines and symbols in the graph.

See [scat \(p. 412\)](#) and [xylne \(p. 530\)](#) if you wish to create an XY graph with specific line/symbol settings, or use [setelem \(p. 426\)](#) to change the settings after the graph is created.

### Options

Options may be specified in parentheses after the keyword.

#### *Template and printing options*

`o = graph_name`   Use appearance options from the specified graph.

`t = graph_name`   Use appearance options and copy text and shading from the specified graph.

`p`               Print the XY line graph.

Note that use of a template will override the existing line and symbol settings.

#### *Scale options*

`a (default)`   Automatic single scale.

`b`               Plot series or columns in pairs (the first two against each other, the second two against each other, and so forth).

n	Normalized scale (zero mean and unit standard deviation). May not be used with the “s” option.
d	Dual scaling with no crossing.
x	Dual scaling with possible crossing.
m	Display XY plots in multiple graphs (will override the “s” option). Not for use with an existing graph object.

### Panel options

The following options apply when graphing panel structured data:

panel = <i>arg</i> (default taken from global set- tings)	Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “medmxmn” (plot median, max and min).
--	--

### Examples

```
group g1 inf unemp gdp inv
g1.xy(o=gra1)
```

plots INF on the horizontal axis and UNEMP, GDP and INV on the vertical axis, using the graph object GRA1 as a template.

```
g1.xy(b)
g1.xy(b,m)
```

plots INF against UNEMP and GDP against INV in first in a single graph, and then in multiple graphs.

If there is an existing graph GRAPH01, the expression:

`graph01.xy(m, n)`

changes its type to an XY plot displayed in multiple graphs with normalized scales, with the remaining XY graph settings at their default values, and using the existing GRAPH01 settings for line and symbol display.

## Cross-references

See “[XY Line](#)” on page 363 of the *User’s Guide* for additional details.

See [graph \(p. 303\)](#) for graph declaration and additional graph types and [options \(p. 358\)](#) for graph options. [scat \(p. 412\)](#) and [xyline \(p. 530\)](#) are specialized forms of XY graphs.

<b>xyline</b>	<a href="#">Command</a>    <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Sym View</a>
---------------	--

Display XY line graph, or change existing graph object type to XY line (if possible).

By default, the first series or column of data will be located along the horizontal axis and the remaining data on the vertical axis. You may optionally choose to plot the data in pairs, where the first two series or columns are plotted against each other, the second two series or columns are plotted against each other, and so forth.

## Syntax

Command:      `xyline(options) arg1 [arg2 arg3 ...]`

Object View:    `group_name.xyline(options)`

Graph Proc:    `graph_name.xyline(options)`

If used as a command, follow the keyword by a list of series and group objects, or by a matrix object. There must be at least two series or columns in the data to be graphed.

XY line graphs are simply XY plots (see [xy \(p. 528\)](#)) with lines turned on, and symbols turned off (see [setelem \(p. 426\)](#)).

## Options

Options may be specified in parentheses after the keyword:

### *Template and printing options*

`o = graph_name`   Use appearance options from the specified graph.

`t = graph_name`   Use appearance options and copy text and shading from the specified graph.

`p`               Print the XY graph.

Note that use of the template option will override the lines setting.

### Scale options

a (default)	Automatic single scale.
b	Plot series or columns in pairs (the first two against each other, the second two against each other, and so forth).
n	Normalized scale (zero mean and unit standard deviation). May not be used with the “s” option.
d	Dual scaling with no crossing.
x	Dual scaling with possible crossing.
m	Display XY plots in multiple graphs (will override the “s” option). Not for use with an existing graph object.

### Panel options

The following options apply when graphing panel structured data.

panel = arg (default taken from global settings)	Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “medmxmn” (plot median, max and min).
---	--

### Examples

```
group g1 inf unemp gdp inv
g1.xyline(o=gra1)
```

plots, in a single graph, INF (on the vertical axis) against UNEMP and GDP against INV, using the graph object GRA1 as a template.

```
g1.xyline
```

```
g1.xyline(m)
```

plots INF against UNEMP and GDP against INV, first in a single graph, and then in multiple graphs.

If there is an existing graph GRAPH01, the expression:

```
graph01.xyline(d)
```

changes its type to an XY *line* plot with dual scales and no crossing, with the remaining XY graph settings at their default values.

### Cross-references

See “[XY Line](#)” on page 363 of the *User’s Guide* for additional details.

See [xy](#) (p. 528) and [graph](#) (p. 303) for graph declaration and additional graph types and [options](#) (p. 358) for graph options. See [scat](#) (p. 412) for XY scatterplots.

xypair	<a href="#">Command</a>    <a href="#">Graph Command</a>   <a href="#">Group View</a>   <a href="#">Matrix View</a>   <a href="#">Rowvector View</a>   <a href="#">Sym View</a>
--------	---

Display XY pairs graph, or change existing graph object type to XY pairs (if possible).

Plots the data in pairs, where the first two series or columns are plotted against each other, the second two series or columns are plotted against each other, and so forth.

### Syntax

Command:        `xypair(options) arg1 [arg2 arg3 ...]`

Object View:     `group_name.xypair(options)`

Graph Proc:     `graph_name.xypair(options)`

If used as a command, follow the keyword by a list of series and group objects, or by a matrix object. There must be at least two series or columns in the data to be graphed.

If changing the type of a graph, the default behavior is to use the existing settings for lines and symbols in the graph.

This graph type is equivalent to using [xy](#) (p. 528) with the “b” option indicating that the data should be graphed in pairs.

### Options

Options may be specified in parentheses after the keyword:

### *Template and printing options*

<i>o = graph_name</i>	Use appearance options from the specified graph.
<i>t = graph_name</i>	Use appearance options and copy text and shading from the specified graph.
<b>p</b>	Print the XY graph.

Note that the use of a template option will override the pairs setting.

### *Scale options*

<b>a (default)</b>	Automatic single scale.
<b>b</b>	Plot series or columns in pairs (the first two against each other, the second two against each other, and so forth).
<b>n</b>	Normalized scale (zero mean and unit standard deviation). May not be used with the “s” option.
<b>d</b>	Dual scaling with no crossing.
<b>x</b>	Dual scaling with possible crossing.
<b>m</b>	Display XY plots in multiple graphs (will override the “s” option). Not for use with an existing graph object.

### *Panel options*

The following options apply when graphing panel structured data:

<b>panel = arg</b> (default taken from global set- tings)	Panel data display: “stack” (stack the cross-sections), “individual” or “1” (separate graph for each cross-section), “combine” or “c” (combine each cross-section in single graph; one time axis), “mean” (plot means across cross-sections), “mean1se” (plot mean and +/- 1 standard deviation summaries), “mean2sd” (plot mean and +/- 2 s.d. summaries), “mean3sd” (plot mean and +/- 3 s.d. summaries), “median” (plot median across cross-sections), “med25” (plot median and +/- .25 quantiles), “med10” (plot median and +/- .10 quantiles), “med05” (plot median +/- .05 quantiles), “med025” (plot median +/- .025 quantiles), “med005” (plot median +/- .005 quantiles), “med-mxmn” (plot median, max and min).
--	---

## Examples

```
group g1 inf unemp gdp inv  
g1.xypair(o=gra1)
```

plots, in a single graph, INF (on the vertical axis) against UNEMP and GDP against INV, using the graph object GRA1 as a template.

```
g1.xypair  
g1.xypair(m)
```

plots INF against UNEMP and GDP against INV, first in a single graph, and then in multiple graphs.

If there is an existing graph GRAPH01, the expression

```
graph01.xypair(m, n)
```

changes its type to an XY *pair* plot displayed in multiple graphs with normalized scales, with the remaining XY graph settings at their default values, and using the existing GRAPH01 settings for line and symbol display.

## Cross-references

See “[XY Line](#)” on page 363 of the *User’s Guide* for additional details.

See [graph \(p. 303\)](#) for graph declaration and additional graph types and [options \(p. 358\)](#) for graph options. See [xy \(p. 528\)](#) for the creation of general XY graphs, and [scat \(p. 412\)](#) for scatterplots.

## Appendix C. Special Expression Reference

---

This Appendix provides an alphabetical listing of special expressions that may be used in series assignment and generation, or as terms in estimation specifications.

ar	Equation Expression
----	---------------------

Autoregressive error specification.

The AR specification can appear in an [ls \(p. 329\)](#) or [tsls \(p. 487\)](#) specification to indicate an autoregressive component. `ar(1)` indicates the first order component, `ar(2)` indicates the second order component, and so on.

### Examples

The command:

```
ls m1 c tb3 tb3(-1) ar(1) ar(4)
```

regresses M1 on a constant, TB3, and TB3 lagged once with a first order and fourth order autoregressive component. The command:

```
tsls sale c adv ar(1) ar(2) ar(3) ar(4) @ c gdp
```

performs two-stage least squares of SALE on a constant and ADV with up to fourth order autoregressive components using a constant and GDP as instruments.

### Cross-references

See [Chapter 17, “Time Series Regression”, on page 477](#) of the *User’s Guide* for details on ARMA and seasonal ARMA modeling.

See also [sar \(p. 541\)](#), [ma \(p. 537\)](#), and [sma \(p. 542\)](#).

@expand	Equation Expression
---------	---------------------

Automatic dummy variables.

The `@expand` expression may be added in estimation to indicate the use of one or more automatically created dummy variables.

### Syntax

Expression:      `@expand(ser1[, ser2, ser3, ...][, drop_spec])`

creates a set of dummy variables that span the unique values of the input series *ser1*, *ser2*, etc.

The optional *drop\_spec* may be used to drop one or more of the dummy variables. *drop\_spec* may contain the keyword “@DROPFIRST” (indicating that you wish to drop the first category), “@DROPLAST” (to drop the last category), or a description of an explicit category, using the syntax:

```
@DROP(val1[, val2, val3,...])
```

where each argument corresponds to a category in @EXPAND. You may use the wild card “\*” to indicate all values of a corresponding category.

### Example

For example, consider the following two variables:

- SEX is a numeric series which takes the values 1 and 0.
- REGION is an alpha series which takes the values “North”, “South”, “East”, and “West”.

The command:

```
eq.ls income @expand(sex) age
```

regresses INCOME on two dummy variables, one for “SEX = 0” and one for “SEX = 1” as well as the simple regressor AGE.

The @EXPAND statement in,

```
eq.ls income @expand(sex, region) age
```

creates 8 dummy variables corresponding to :

```
sex = 0, region = "North"  
sex = 0, region = "South"  
sex = 0, region = "East"  
sex = 0, region = "West"  
sex = 1, region = "North"  
sex = 1, region = "South"  
sex = 1, region = "East"  
sex = 1, region = "West"
```

The expression:

```
@expand(sex, region, @dropfirst)
```

creates the set of dummy variables defined above, but no dummy is created for “SEX = 0, REGION = “North””. In the expression:

```
@expand(sex, region, @droplast)
```

no dummy is created for “SEX = 1, REGION = “WEST””.

The expression:

```
@expand(sex, region, @drop(0, "West"), @drop(1, "North"))
```

creates a set of dummy variables from SEX and REGION pairs, but no dummy is created for “SEX = 0, REGION = “West”” and “SEX = 1, REGION = “North””.

```
@expand(sex, region, @drop(1, *))
```

specifies that dummy variables for all values of REGION where “SEX = 1” should be dropped.

## Cross-references

See “[Automatic Categorical Dummy Variables](#)” on page 450 of the *User’s Guide* for further discussion.

ma	Equation Expression
----	---------------------

Moving average error specification.

The `ma` specification may be added in an [ls \(p. 329\)](#) or [tsls \(p. 487\)](#) specification to indicate a moving average error component. `ma(1)` indicates the first order component, `ma(2)` indicates the second order component, and so on.

## Examples

```
ls(z) m1 c tb3 tb3(-1) ma(1) ma(2)
```

regresses M1 on a constant, TB3, and TB3 lagged once with first order and second order moving average error components. The “z” option turns off backcasting in estimation.

## Cross-references

See “[Time Series Regression](#)” on page 477 of the *User’s Guide* for details on ARMA and seasonal ARMA modeling.

See also [sma \(p. 542\)](#), [ar \(p. 535\)](#), and [sar \(p. 541\)](#).

na	Series Expression
----	-------------------

Not available code. “NA” is used to represent missing observations.

### Examples

```
smp1 if y >= 0  
series z = y  
smp1 if y < 0  
z = na
```

generates a series Z containing the contents of Y, but with all negative values of Y set to “NA”.

NA values will also be generated by mathematical operations that are undefined:

```
series y = nrnd  
y = log(y)
```

will replace all positive value of Y with log(Y) and all negative values with “NA”.

```
series test = (yt <> na)
```

creates the series TEST which takes the value one for nonmissing observations of the series YT. A zero value of TEST indicates missing values of the series YT.

Note that the behavior of missing values has changed since EViews 2. Previously, NA values were coded as 1e-37. This implied that in EViews 2, you could use the expression:

```
series z = (y>=0)*x + (y<0)*na
```

to return the value of Y for non-negative values of Y and “NA” for negative values of Y. This expression will now generate the value “NA” for all values of Y, since mathematical expressions involving missing values always return “NA”. You must now use the smp1 statement as in the first example above, or the @recode or @nan function.

### Cross-references

See “[Missing Values](#)” on page 126 of the *User’s Guide* for a discussion of working with missing values in EViews.

<b>nrnd</b>	Series Expression
-------------	-------------------

Normal random number generator.

When used in a series expression, `nrnd` generates (pseudo) random draws from a normal distribution with zero mean and unit variance.

### Examples

```
smp1 @first @first
series y = 0
smp1 @first+1 @last
series y = .6*y(-1)+.5*nrnd
```

generates a Y series that follows an AR(1) process with initial value zero. The innovations are normally distributed with mean zero and standard deviation 0.5.

```
series u = 10+@sqr(3)*nrnd
series z = u+.5*u(-1)
```

generates a Z series that follows an MA(1) process. The innovations are normally distributed with mean 10 and variance 3.

```
series x = nrnd^2+nrnd^2+nrnd^2
```

generates an X series as the sum of squares of three *independent* standard normal random variables, which has a  $\chi^2(3)$  distribution. Note that adding the sum of the three series is not the same as issuing the command:

```
series x=3*nrnd^2
```

since the latter involves the generation of a single random variable.

The command:

```
series x=@qchisq(rnd, 3)
```

provides an alternative method of simulating random draws from a  $\chi^2(3)$  distribution.

### Cross-references

See “[Statistical Distribution Functions](#)” on page 554 for a list of other random number generating functions from various distributions.

See also `rnd` (p. 541), `rndint` (p. 402) and `rndseed` (p. 403).

pdl	Equation Expression
-----	---------------------

**Polynomial distributed lag specification.**

This expression allows you to estimate polynomial distributed lag specifications in ls or tsls estimation. pdl forces the coefficients of a distributed lag to lie on a polynomial. The expression can only be used in estimation by list.

### Syntax

Expression:      **pdl(series\_name, lags, order[,options])**

### Options

The PDL specification must be provided in parentheses after the keyword pdl in the following order: the name of the series to which to fit a polynomial lag, the number of lags to include, the order (degree) of polynomial to fit, and an option number to constrain the PDL. By default, EViews does not constrain the endpoints of the PDL.

The constraint options are:

- 1           Constrain the near end of the distribution to zero.
- 2           Constrain the far end of the distribution to zero.
- 3           Constrain both the near and far end of the distribution to zero.

### Examples

```
ls sale c pdl(order,8,3) ar(1) ar(2)
```

fits a third degree polynomial to the coefficients of eight lags of the regressor ORDER.

```
tsls sale c pdl(order,12,3,2) @ c pdl(rain,12,6)
```

fits a third degree polynomial to the coefficients of twelve lags of ORDER, constraining the far end to be zero. Estimation is by two-stage least squares, using a constant and a sixth degree polynomial fit to twelve lags of RAIN.

```
tsls y c x1 x2 pdl(z,12,3,2) @ c pdl(*) z2 z3 z4
```

When the PDL variable is exogenous in 2SLS, you may use “pdl(\*)” in the instrument list instead of repeating the full PDL specification.

## Cross-references

See “[Polynomial Distributed Lags \(PDLs\)](#)” on page [445](#) of the *User’s Guide* for further discussion.

<b>rnd</b>	Series Expression
------------	-------------------

Uniform random number generator.

Generates (pseudo) random draws from a uniform distribution on (0,1). The expression may be included in a series expression or in an equation to be used in `solve`.

## Examples

```
series u=5+(12-5)*rnd
```

generates a U series drawn from a uniform distribution on (5, 12).

## Cross-references

See the list of available random number generators in [Appendix D, “Operator and Function Reference”, beginning on page 543](#).

See also [nrnd](#) (p. 539), [rndint](#) (p. 402) and [rndseed](#) (p. 403).

<b>sar</b>	Equation Expression
------------	---------------------

Seasonal autoregressive error specification.

`sar` can be included in `ls` or `tsls` specification to specify a multiplicative seasonal autoregressive term. A `sar(p)` term can be included in your equation specification to represent a seasonal autoregressive term with lag  $p$ . The lag polynomial used in estimation is the product of that specified by the `ar` terms and that specified by the `sar` terms. The purpose of the `sar` expression is to allow you to form the product of lag polynomials.

## Examples

```
ls tb3 c ar(1) ar(2) sar(4)
```

TB3 is modeled as a second order autoregressive process with a multiplicative seasonal autoregressive term at lag four.

```
tsls sale c adv ar(1) sar(12) sar(24) @ c gdp
```

In this two-stage least squares specification, the error term is a first order autoregressive process with multiplicative seasonal autoregressive terms at lags 12 and 24.

## Cross-references

See “[ARIMA Theory](#)” beginning on page 485 of the *User’s Guide* for details on ARMA and seasonal ARMA modeling.

See also [sma \(p. 542\)](#), [ar \(p. 535\)](#), and [ma \(p. 537\)](#).

sma	Equation Expression
-----	---------------------

Seasonal moving average error specification.

sma can be included in a ls or tsls specification to specify a multiplicative seasonal moving average term. A sma(*p*) term can be included in your equation specification to represent a seasonal moving average term of order *p*. The lag polynomial used in estimation is the product of that specified by the ma terms and that specified by the sma terms. The purpose of the sma expression is to allow you to form the product of lag polynomials.

## Examples

```
ls tb3 c ma(1) ma(2) sma(4)
```

TB3 is modeled as a second order moving average process with a multiplicative seasonal moving average term at lag four.

```
tsls(z) sale c adv ma(1) sma(12) sma(24) @ c gdp
```

In this two-stage least squares specification, the error term is a first order moving average process with multiplicative seasonal moving average terms at lags 12 and 24. The “z” option turns off backcasting.

## Cross-references

See “[ARIMA Theory](#)” beginning on page 485 of the *User’s Guide* for details on ARMA and seasonal ARMA modeling.

See also [sar \(p. 541\)](#), [ar \(p. 535\)](#), and [ma \(p. 537\)](#).

## Appendix D. Operator and Function Reference

---

The reference material in this section describes basic operators and functions that may be used with series and (in some cases) matrix objects. A general description of the use of these operators and functions may be found in [Chapter 6, “Working with Data”, beginning on page 121](#) of the *User’s Guide*.

This material is divided into several topics:

- [Operators](#).
- [Basic mathematical functions](#).
- [Time series functions](#).
- [Descriptive statistics](#).
- [By-group statistics](#).
- [Additional and special functions](#).
- [Trigonometric functions](#).
- [Statistical distribution functions](#).
- [String functions](#).
- [Date functions](#).
- [Workfile functions](#).
- [Value map functions](#).
- [Matrix functions](#).

Documentation on libraries of more specialized functions is provided elsewhere:

- For a description of functions related to string and date manipulation, see [“String Function Summary” on page 127](#) and [“Date Function Summary” on page 150](#).
- For details on workfile functions that provide information about each observation of a workfile based on information contained in the structure of the workfile, see [Appendix E, “Workfile Functions”, on page 559](#).
- Functions for working with value maps are documented in [“Valmap Functions” on page 165](#) of the *User’s Guide*.

- For a list of functions specific to matrices, see “[Matrix Function and Command Summary](#)” on page 44.

## Operators

All of the operators described below may be used in expressions involving series and scalar values. When applied to a series expression, the operation is performed for each observation in the current sample. The precedence of evaluation is listed in [“Operators” on page 121](#) of the *User’s Guide*. Note that you can enforce order-of-evaluation using parentheses.

Expression	Operator	Description
+	add	$x+y$ adds the contents of X and Y.
-	subtract	$x-y$ subtracts the contents of Y from X.
*	multiply	$x*y$ multiplies the contents of X by Y.
/	divide	$x/y$ divides the contents of X by Y.
$\wedge$	raise to the power	$x^y$ raises X to the power of Y.
>	greater than	$x>y$ takes the value 1 if X exceeds Y, and 0 otherwise.
<	less than	$x<y$ takes the value 1 if Y exceeds X, and 0 otherwise.
=	equal to	$x=y$ takes the value 1 if X and Y are equal, and 0 otherwise.
$\neq$	not equal to	$x\neq y$ takes the value 1 if X and Y are not equal, and 0 if they are equal.
$\leq$	less than or equal to	$x\leq y$ takes the value 1 if X does not exceed Y, and 0 otherwise.
$\geq$	greater than or equal to	$x\geq y$ takes the value 1 if Y does not exceed X, and 0 otherwise.
and	logical and	$x \text{ and } y$ takes the value 1 if both X and Y are nonzero, and 0 otherwise.
or	logical or	$x \text{ or } y$ takes the value 1 if either X or Y is nonzero, and 0 otherwise.

In addition, EViews provides special functions to perform comparisons using special rules for handling missing values (see [“Missing Values” on page 126](#)) :

---

<code>@eqna (x, y)</code>	equal to	takes the value 1 if X and Y are equal, and 0 otherwise. NAs are treated as ordinary values for purposes of comparison.
<code>@isna (x)</code>	equal to NA	takes the value 1 if X is equal to NA and 0 otherwise.
<code>@neqna (x, y)</code>	not equal to	takes the value 1 if X and Y are not equal, and 0 if they are equal. NAs are treated as ordinary values for purposes of comparison.

---

## Basic Mathematical Functions

These functions perform basic mathematical operations. When applied to a series, they return a value for every observation in the current sample. When applied to a matrix object, they return a value for every element of the matrix object. The functions will return NA values for observations where the input values are NAs, or where the input values are not valid. For example, the square-root function `@sqrt`, will return NAs for all observations less than zero.

---

Name	Function	Examples/Description
<code>@abs (x) , abs (x)</code>	absolute value	<code>@abs(-3) = 3 .</code>
<code>@ceiling (x)</code>	smallest integer not less than	<code>@ceiling(2.34) = 3 ; @ceiling(4) = 4 .</code>
<code>@exp (x) , exp (x)</code>	exponential, $e^x$	<code>@exp(1) ≈ 2.71813 .</code>
<code>@fact (x)</code>	factorial, $x!$	<code>@fact(3) = 6 ; @fact(0) = 1 .</code>
<code>@factlog (x)</code>	natural logarithm of the factorial, $\log_e(x!)$	<code>@factlog(3) = 1.7918 ; @factlog(0) = 0 .</code>
<code>@floor (x)</code>	largest integer not greater than	<code>@floor(1.23) = 1 ; @floor(-3.1) = -4 .</code>
<code>@inv (x)</code>	reciprocal, $1/x$	<code>@inv(2) = 0.5 .</code>
<code>@mod (x, y)</code>	floating point remainder	returns the remainder of $x/y$ with the same sign as $x$ . If $y = 0$ the result is 0.
<code>@log (x) , log (x)</code>	natural logarithm, $\log_e(x)$	<code>@log(2) ≈ 0.693 ; log(2.71813) ≈ 1 .</code>
<code>@log10 (x)</code>	base-10 logarithm, $\log_{10}(x)$	<code>@log10(100) = 2 .</code>

---

<code>@logx(x, b)</code>	base- <i>b</i> logarithm, $\log_b(x)$	$\text{@log}(256, 2) = 8.$
<code>@nan(x, y)</code>	recode NAs in X to Y	returns <i>x</i> if <i>x</i> < > NA , and <i>y</i> if <i>x</i> = NA .
<code>@recode(s, x, y)</code>	recode by condition	returns <i>x</i> if condition <i>s</i> is true; otherwise returns <i>y</i> .
<code>@round(x)</code>	round to the nearest integer	$\text{@round}(-97.5) = -98;$ $\text{@round}(3.5) = 4.$
<code>@sqrt(x), sqr(x)</code>	square root	$\text{@sqrt}(9) = 3.$

## Time Series Functions

The following functions facilitate working with time series data. Note that NAs will be returned for observations for which lagged values are not available. For example, `d(x)` returns a missing value for the first observation in the workfile, since the lagged value is not available.

Name	Function	Description
<code>d(x)</code>	first difference	$(1 - L)X = X - X(-1)$ where <i>L</i> is the lag operator.
<code>d(x, n)</code>	<i>n</i> -th order difference	$(1 - L)^n X.$
<code>d(x, n, s)</code>	<i>n</i> -th order difference with a seasonal difference at <i>s</i>	$(1 - L)^n(1 - L^s)X.$
<code>dlog(x)</code>	first difference of the logarithm	$(1 - L)\log(X)$ $= \log(X) - \log(X(-1))$ .
<code>dlog(x, n)</code>	<i>n</i> -th order difference of the logarithm	$(1 - L)^n\log(X).$
<code>dlog(x, n, s)</code>	<i>n</i> -th order difference of the logarithm with a seasonal difference at <i>s</i>	$(1 - L)^n(1 - L^s)\log(X).$
<code>@movav(x, n)</code>	<i>n</i> -period backward moving average	$\text{@movav}(x, 3)$ $= (X + X(-1) + X(-2))/3$
<code>@movsum(x, n)</code>	<i>n</i> -period backward moving sum	$\text{@movsum}(x, 3)$ $= (X + X(-1) + X(-2))$

<code>@pc(x)</code>	one-period percentage change (in percent)	equals <code>@pch(x) * 100</code>
<code>@pch(x)</code>	one-period percentage change (in decimal)	$(X - X(-1))/X(-1)$
<code>@pca(x)</code>	one-period percentage change—annualized (in per- cent)	equals <code>@pcha(x) * 100</code>
<code>@pcha(x)</code>	one-period percentage change—annualized (in deci- mal)	$\begin{aligned} @pcha(x) \\ = (1 + @pch(x))^n - 1 \end{aligned}$
		where $n$ is the lag associated with one-year ( $n = 4$ ) for quarterly data, etc.).
<code>@pcy(x)</code>	one-year percentage change (in percent)	equals <code>@pchy(x) * 100</code>
<code>@pchy(x)</code>	one-year percentage change (in decimal)	$(X - X(-n))/X(-n)$ , where $n$ is the lag associated with one- year ( $n = 12$ ) for annual data, etc.).

## Descriptive Statistics

These functions compute descriptive statistics for a specified sample, excluding missing values if necessary. The default sample is the current workfile sample. If you are performing these computations on a series and placing the results into a series, you can specify a sample as the last argument of the descriptive statistic function, either as a string (in double quotes) or using the name of a sample object. For example:

```
series z = @mean(x, "1945m01 1979m12")
```

or

```
w = @var(y, s2)
```

where S2 is the name of a sample object and W and X are series. Note that you may not use a sample argument if the results are assigned into a matrix, vector, or scalar object. For example, the following assignment:

```
vector(2) a
series x
a(1) = @mean(x, "1945m01 1979m12")
```

is not valid since the target  $A(1)$  is a vector element. To perform this latter computation, you must explicitly set the global sample prior to performing the calculation performing the assignment:

```
smp1 1945:01 1979:12  
a(1) = @mean(x)
```

To determine the number of observations available for a given series, use the `@obs` function. Note that where appropriate, EViews will perform casewise exclusion of data with missing values. For example, `@cov(x, y)` and `@cor(x, y)` will use only observations for which data on both X and Y are valid.

In the following table, arguments in square brackets [ ] are optional arguments:

- [s]: sample expression in double quotes or name of a sample object. *The optional sample argument may only be used if the result is assigned to a series.* For `@quantile`, you must provide the method option argument in order to include the optional sample argument.

If the desired sample expression contains the double quote character, it may be entered using the double quote as an escape character. Thus, if you wish to use the equivalent of,

```
smp1 if name = "Smith"
```

in your `@MEAN` function, you should enter the sample condition as:

```
series y = @mean(x, "if name=""Smith""")
```

The pairs of double quotes in the sample expression are treated as a single double quote.

Function	Name	Description
<code>@cor(x, y[, s])</code>	correlation	the correlation between X and Y.
<code>@cov(x, y[, s])</code>	covariance	the covariance between X and Y.
<code>@inner(x, y[, s])</code>	inner product	the inner product of X and Y.
<code>@obs(x[, s])</code>	number of observations	the number of non-missing observations for X in the current sample.
<code>@mean(x[, s])</code>	mean	average of the values in X.
<code>@median(x[, s])</code>	median	computes the median of the X (uses the average of middle two observations if the number of observations is even).
<code>@min(x[, s])</code>	minimum	minimum of the values in X.
<code>@max(x[, s])</code>	maximum	maximum of the values in X.

---

<code>@quantile(x, q[, m, s])</code>	quantile	the $q$ -th quantile of the series X. $m$ is an optional integer argument for specifying the quantile method: 1 (rankit - default), 2 (ordinary), 3 (van der Waerden), 4 (Blom), 5 (Tukey).
<code>@stdev(x[, s])</code>	standard deviation	square root of the unbiased sample variance (sum-of-squared residuals divided by $n - 1$ ).
<code>@sum(x[, s])</code>	sum	the sum of X.
<code>@sumsq(x[, s])</code>	sum-of-squares	sum of the squares of X.
<code>@var(x[, s])</code>	variance	variance of the values in X (division by $n$ ).

---

## By-Group Statistics

The following “by group”-statistics are available in EViews. They may be used as part of a series expression statements to compute the statistic for subgroups, and to assign the value of the relevant statistic to each observation.

---

Function	Description
<code>@obsby(arg1, arg2[, s])</code>	Number of non-NA $arg1$ observations for each $arg2$ group.
<code>@sumsby(arg1, arg2[, s])</code>	Sum of $arg1$ observations for each $arg2$ group.
<code>@meansby(arg1, arg2[, s])</code>	Mean of $arg1$ observations for each $arg2$ group.
<code>@minsby(arg1, arg2[, s])</code>	Minimum value of $arg1$ observations for each $arg2$ group.
<code>@maxsby(arg1, arg2[, s])</code>	Maximum value of $arg1$ observations for each $arg2$ group.
<code>@mediansby(arg1, arg2[, s])</code>	Median of $arg1$ observations for each $arg2$ group.
<code>@varsby(arg1, arg2[, s])</code>	Variance of $arg1$ observations for each $arg2$ group.
<code>@stdevsby(arg1, arg2[, s])</code>	Standard deviation of $arg1$ observations for each $arg2$ group.
<code>@sumsqsbysby(arg1, arg2[, s])</code>	Sum of squares of $arg1$ observations for each $arg2$ group.

---

<code>@quantilesby(arg1, arg2[, s])</code>	Quantiles of <i>arg1</i> observations for each <i>arg2</i> group.
<code>@skewsby(arg1, arg2[, s])</code>	Skewness of <i>arg1</i> observations for each <i>arg2</i> group.
<code>@kurtsby(arg1, arg2[, s])</code>	Kurtosis of <i>arg1</i> observations for each <i>arg2</i> group.
<code>@nasby(arg1, arg2[, s])</code>	Number of <i>arg1</i> NA values for each <i>arg2</i> group.

With the exception of `@QUANTILEBY`, all of the functions take the form:

`@STATBY(arg1, arg2[, s])`

where `@STATBY` is one of the by-group function keyword names, *arg1* is a series expression, *arg2* is a numeric or alpha series expression identifying the subgroups, and *s* is an optional sample literal (a quoted sample expression) or a named sample. With the exception of `@OBSBY`, *arg1* must be a numeric series.

By default, EViews will use the workfile sample when computing the descriptive statistics. You may provide the optional sample *s* as a literal (quoted) sample expression or a named sample.

The `@QUANTILEBY` function requires an additional argument for the quantile value that you wish to compute:

`@QUANTILEBY(arg1, arg2, q[, s])`

For example, to compute the first quartile, you should use the *q* value of .25.

There are two related functions of note,

`@GROUPID(arg[, smpl])`

returns an integer indexing the group ID for each observation of the series or alpha expression *arg*, while:

`@NGROUPS(arg[, smpl])`

returns a scalar indicating the number of groups (distinct values) for the series or alpha expression *arg*.

## Special Functions

EViews provides a number of special functions used in evaluating the properties of various statistical distributions or for returning special mathematical values such as Euler's constant. For further details on special functions, see the extensive discussions in Temme (1996), Abramowitz and Stegun (1964), and Press, *et al.* (1992).

Function	Description
<code>@beta(a,b)</code>	beta integral (Euler integral of the second kind): $B(a, b) = \int_0^1 t^{a-1} (1-t)^{b-1} dt = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$ for $a, b > 0$ .
<code>@betainc(x,a,b)</code>	incomplete beta integral: $\frac{1}{B(a,b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$ for $0 \leq x \leq 1$ and $a, b > 0$ .
<code>@betaincder(x,a,b,s)</code>	derivative of the incomplete beta integral: evaluates the derivatives of the incomplete beta integral $B(x, a, b)$ , where $s$ is an integer from 1 to 9 corresponding to the desired derivative: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} \frac{\partial B}{\partial x} & \frac{\partial B}{\partial a} & \frac{\partial B}{\partial b} \\ \frac{\partial^2 B}{\partial x^2} & \frac{\partial^2 B}{\partial x \partial a} & \frac{\partial^2 B}{\partial x \partial b} \\ \frac{\partial^2 B}{\partial a^2} & \frac{\partial^2 B}{\partial a \partial b} & \frac{\partial^2 B}{\partial b^2} \end{bmatrix}$
<code>@betaincinv(p,a,b)</code>	inverse of the incomplete beta integral: returns an $x$ satisfying: $p = \frac{1}{B(a,b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$ for $0 \leq p \leq 1$ and $a, b > 0$ .
<code>@betalog(a,b)</code>	natural logarithm of the beta integral: $\log B(a, b) = \log \Gamma(a) + \log \Gamma(b) - \log \Gamma(a+b).$
<code>@binom(n,x)</code>	binomial coefficient: $\binom{n}{x} = \frac{n!}{x!(n-x)!}$ for $n$ and $x$ positive integers, $0 \leq x \leq n$ .

<code>@binomlog(n, x)</code>	natural logarithm of the binomial coefficient: $\log(n!) - \log(x!) - \log((n-x)!)$
<code>@cloglog(x)</code>	complementary log-log function: $\log(-\log(1-x))$
See also <code>@qextreme</code> .	
<code>@digamma(x), @psi(x)</code>	first derivative of the log gamma function: $(x) = \frac{d \log \Gamma(x)}{dx} = \frac{1}{\Gamma(x)} \frac{d \Gamma(x)}{dx}$
for $x \geq 0$ .	
<code>@erfc(x)</code>	complementary error function: $\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt = 1 - \operatorname{erf}(x).$
for $x \geq 0$ .	
<code>@gamma(x)</code>	(complete) gamma function: $\Gamma(x) = \int_0^{\infty} e^{-t} t^{x-1} dt$
for $x \geq 0$ .	
<code>@gammader(x)</code>	first derivative of the gamma function: $\Gamma'(x) = d\Gamma(x)/(dx)$
Note: Euler's constant, $\gamma \approx 0.5772$ , may be evaluated as $\gamma = -@\gammader(1)$ . See also <code>@digamma</code> and <code>@trigamma</code> .	
<code>@gammainc(x, a)</code>	incomplete gamma function: $G(x, a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$
for $x \geq 0$ and $a > 0$ .	

---

<code>@gammaincder(x, a, n)</code>	derivative of the incomplete gamma function: Evaluates the derivatives of the incomplete gamma integral $G(x, a)$ , where $n$ is an integer from 1 to 5 corresponding to the desired derivative:
	$\begin{bmatrix} 1 & 2 & - \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} \frac{\partial G}{\partial x} & \frac{\partial G}{\partial a} & - \\ \frac{\partial^2 G}{\partial x^2} & \frac{\partial^2 G}{\partial x \partial a} & \frac{\partial^2 G}{\partial a^2} \end{bmatrix}$
<code>@gammaincinv(p, a)</code>	inverse of the incomplete gamma function: find the value of $x$ satisfying:
	$p = G(x, a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$ for $0 \leq p < 1$ and $a > 0$ .
<code>@gammalog(x)</code>	logarithm of the gamma function: $\log \Gamma(x)$ . For derivatives of this function see <code>@digamma</code> and <code>@trigamma</code> .
<code>@logit(x)</code>	logistic transform:
	$\frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$
<code>@psi(x)</code>	see <code>@digamma</code> .
<code>@trigamma(x)</code>	second derivative of the log gamma function:
	$'(x) = \frac{d^2 \log \Gamma(x)}{dx^2}$

---

## Trigonometric Functions

When applied to a series, all of the trigonometric functions operate on every observation in the current sample and return a value for every observation. Where relevant, the input and results should/will be expressed in radians. All results are real valued—complex values will return NAs.

---

Function	Name	Examples/Description
<code>@acos(x)</code>	arc cosine (real results in radians)	<code>@acos(-1) = π</code>
<code>@asin(x)</code>	arc sine (real results in radians)	<code>@asin(-1) = π/2</code>
<code>@atan(x)</code>	arc tangent (results in radians)	<code>@atan(1) = π/4</code>

---

<code>@cos(x)</code>	cosine (argument in radians)	<code>@cos(3.14159) ≈ -1</code>
<code>@sin(x)</code>	sine (argument in radians)	<code>@sin(3.14159) ≈ 0</code>
<code>@tan(x)</code>	tangent (argument in radians)	<code>@tan(1) ≈ 1.5574</code>

## Statistical Distribution Functions

The following functions provide access to the density or probability functions, cumulative distribution, quantile functions, and random number generators for a number of standard statistical distributions.

There are four functions associated with each distribution. The first character of each function name identifies the type of function:

Function Type	Beginning of Name
Cumulative distribution (CDF)	<code>@c</code>
Density or probability	<code>@d</code>
Quantile (inverse CDF)	<code>@q</code>
Random number generator	<code>@r</code>

The remainder of the function name identifies the distribution. For example, the functions for the beta distribution are `@cbeta`, `@dbeta`, `@qbeta` and `@rbeta`.

When used with series arguments, EViews will evaluate the function for each observation in the current sample. As with other functions, NA or invalid inputs will yield NA values. For values outside of the support, the functions will return zero.

Note that the CDFs are assumed to be right-continuous:  $F_X(k) = \Pr(X \leq k)$ . The quantile functions will return the smallest value where the CDF evaluated at the value equals or exceeds the probability of interest:  $q_X(p) = q^*$ , where  $F_X(q^*) \geq p$ . The inequalities are only relevant for discrete distributions.

The information provided below should be sufficient to identify the meaning of the parameters for each distribution. For further details, see the *Command and Programming Reference*.

Distribution	Functions	Density/Probability Function
Beta	<code>@cbeta(x, a, b)</code> , <code>@dbeta(x, a, b)</code> , <code>@qbeta(p, a, b)</code> , <code>@rbeta(a, b)</code>	$f(x, a, b) = \frac{x^{a-1}(1-x)^{b-1}}{B(a, b)}$ for $0 \leq p \leq 1$ and for $a, b > 0$ , where $B$ is the <code>@beta</code> function.

Binomial	<code>@cbinom(x, n, p), @dbinom(x, n, p), @qbinom(s, n, p), @rbinom(n, p)</code>	$\Pr(x, n, p) = \binom{n}{x} p^x (1-p)^{n-x}$ if $x = 0, 1, \dots, n, \dots$ , and 0 otherwise, for $0 \leq p \leq 1$ .
Chi-square	<code>@cchisq(x, v), @dchisq(x, v), @qchisq(p, v), @rchisq(v)</code>	$f(x, v) = \frac{1}{2^{v/2} \Gamma(v/2)} x^{v/2-1} e^{-x/2}$ where $x \geq 0$ , and $v > 0$ . Note that the degrees of freedom parameter $v$ need not be an integer.
Exponential	<code>@cexp(x, m), @dexp(x, m), @qexp(p, m), @rexp(m)</code>	$f(x, m) = \frac{1}{m} e^{-x/m}$ for $x \geq 0$ , and $m > 0$ .
Extreme Value (Type I-minimum)	<code>@cextreme(x), @dextreme(x), @qextreme(p), @cloglog(p), @rextreme</code>	$f(x) = \exp(x - e^x)$ for $-\infty < x < \infty$ .
<i>F</i> -distribution	<code>@cfdist(x, v1, v2), @dfdist(x, v1, v2), @qfdist(p, v1, v2), @rfdist(v1, v1)</code>	$f(x, v_1, v_2) = \frac{x^{(v_1-2)/2} v_2^{v_2/2}}{B(v_1/2, v_2/2)}$ $x^{(v_1-2)/2} (v_2 + v_1 x)^{-(v_1+v_2)/2}$ where $x \geq 0$ , and $v_1, v_2 > 0$ . Note that the functions allow for fractional degrees of freedom parameters $v_1$ and $v_2$ .
Gamma	<code>@cgamma(x, b, r), @dgamma(x, b, r), @qgamma(p, b, r), @rgamma(b, r)</code>	$f(x, b, r) = b^{-r} x^{r-1} e^{-x/b} / \Gamma(r)$ where $x \geq 0$ , and $b, r > 0$ .
Generalized Error	<code>@cgded(x, r), @dgded(x, r), @qgded(p, r), @rgded(r)</code>	$f(x, r) = \frac{r I\left(\frac{3}{r}\right)^{1/2}}{2 I\left(\frac{1}{r}\right)^{3/2}} \exp\left(- x ^r \left(\frac{I\left(\frac{3}{r}\right)}{I\left(\frac{1}{r}\right)}\right)^{r/2}\right)$ where $-\infty < x < \infty$ , and $r > 0$ .

Laplace	<code>@claplace(x), @dlaplace(x), @qlaplace(x), @rlaplace</code>	$f(x) = \frac{1}{2}e^{- x }$ for $-\infty < x < \infty$ .
Logistic	<code>@clogistic(x), @dlogistic(x), @qlogistic(p), @rlogistic</code>	$f(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$ for $-\infty < x < \infty$ .
Log-normal	<code>@clognorm(x,m,s), @dlognorm(x,m,s), @qlognorm(p,m,s), @rlognorm(m,s)</code>	$f(x, m, s) = \frac{1}{x \sqrt{2\pi s^2}} e^{-(\log x - m)^2 / (2s^2)}$ $x > 0, -\infty < m < \infty, \text{ and } s > 0$ .
Negative Binomial	<code>@cnegbin(x,n,p), @dnegbin(x,n,p), @qnegbin(s,n,p), @rnegbin(n,p)</code>	$\Pr(x, n, p) = \frac{\Gamma(x+n)}{\Gamma(x+1)\Gamma(n)} p^n (1-p)^{x-n}$ if $x = 0, 1, \dots, n, \dots$ , and 0 otherwise, for $0 \leq x \leq 1$ .
Normal (Gaussian)	<code>@cnorm(x), @dnorm(x), @qnorm(p), @rnorm, nrnd</code>	$f(x) = (2\pi)^{-1/2} e^{-x^2/2}$ for $-\infty < x < \infty$ .
Poisson	<code>@cpoisson(x,m), @dpoisson(x,m), @qpoisson(p,m), @rpoisson(m)</code>	$\Pr(x, m) = m^x e^{-m} / x!$ if $x = 0, 1, \dots, n, \dots$ , and 0 otherwise, for $m > 0$ .
Pareto	<code>@cpareto(x,a,k), @dpareto(x,a,k), @qpareto(p,a,k), @rpareto(a,k)</code>	$f(x, a, k) = (ak^a) / x^{a+1}$ for $a > 0$ , and $0 \leq k \leq x$ .
Student's t-distribution	<code>@ctdist(x,v), @tdist(x,v), @qtdist(p,v), @rtdist(v)</code>	$f(x, v) = \frac{\Gamma(\frac{v+1}{2})}{\frac{1}{(v\pi)^{\frac{v}{2}}} \Gamma(\frac{v}{2})} \left(1 + \left(\frac{x^2}{v}\right)\right)^{\frac{-(v+1)}{2}}$ for $-\infty < x < \infty$ , and $v > 0$ . Note that $v = 1$ , yields the Cauchy distribution.

---

Uniform	<code>@cunif(x,a,b),</code> <code>@dunif(x,a,b),</code> <code>@qunif(p,a,b),</code> <code>@runif(a,b), rnd</code>	$f(x) = \frac{1}{b-a}$ for $a < x < b$ and $b > a$ .
Weibull	<code>@cweib(x,m,a),</code> <code>@dweib(x,m,a),</code> <code>@qweib(p,m,a),</code> <code>@rweib(m,a)</code>	$f(x, m, a) = am^{-a}x^{a-1}e^{-(x/m)^a}$ where $-\infty < x < \infty$ , and $m, a > 0$ .

---

## Additional Distribution Related Functions

The following utility functions were designed to facilitate the computation of  $p$ -values for common statistical tests. While these results may be derived using the distributional functions above, they are retained for convenience and backward compatibility.

---

<b>Function</b>	<b>Distribution</b>	<b>Description</b>
<code>@chisq(x,v)</code>	Chi-square	Returns the probability that a Chi-squared statistic with $v$ degrees of freedom exceeds $x$ : $@chisq(x,v) = 1 - @cchisq(x,d)$
<code>@fdist(x,v1,v2)</code>	$F$ -distribution	Probability that an $F$ -statistic with $v_1$ numerator degrees of freedom and $v_2$ denominator degrees of freedom exceeds $x$ : $@fdist(x,v1,v2) = 1 - @cfdist(x,v1,v2)$
<code>@tdist(x,v)</code>	$t$ -distribution	Probability that a $t$ -statistic with $v$ degrees of freedom exceeds $x$ in absolute value (two-sided $p$ -value): $@tdist(x,v) = 2 * (1 - @ctdist(x,v))$

---

of the *User's Guide*



# Appendix E. Workfile Functions

---

EViews workfile functions provide information about each observation of a workfile based on information contained in the structure of the workfile.

These functions can be viewed in two ways. First, they are simply virtual series available within each workfile that can be used wherever a regular series might be used. Alternatively, they may be thought of as special functions that take two additional implicit arguments: the workfile within which the function is being used, and the observation number within this workfile for which to return a value.

Since workfile functions are based on the structure of a workfile, they may only be used in operations where a workfile is implicitly involved. They may be used in statements that generate workfile series, in statements that set the workfile sample, and in expressions used in estimation. They cannot be used when manipulating scalar variables or vectors and matrices in EViews programs.

## Basic Workfile Information

The `@OBSNUM` function provides information on basic observation numbering for the workfile, returning the observation number of the current observation in the workfile:

- `@OBSNUM`: returns the observation number of the current observation in the workfile.

The observation number starts at one for the first row, and increments by one for each subsequent row of the workfile.

For example:

```
series idnum = @obsnum
```

creates a series IDNUM that contains sequential values from one to the number of observations in the workfile.

Additional functions provide scalar values associated with the workfile and default workfile sample:

- `@ELEM(x, arg)`: returns the value of the series *x* at observation or date *arg*. If the workfile is undated, *arg* should be an integer corresponding to the observation ID as given in `@OBSNUM`. If the workfile is dated, *arg* should be a string representation of a date in the workfile, enclosed in double quotes. Note that `@ELEM` is not available in panel structured workfiles.

- `@OBSRANGE`: returns number of observations in the current active workfile range (0 if no workfile in memory).
- `@OBSSMPL`: returns number of observations in the current active workfile sample (0 if no workfile in memory).

## Dated Workfile Information

### Basic Date Functions

There is a set of functions that provides information about the dates in your dated workfiles. The first two functions return the start and end date of the period of time (interval) associated with the current observation of the workfile:

- `@DATE`: returns the start date of the period of time of the current observation of the workfile.
- `@ENDDATE`: returns the end date of the period of time associated with the current observation of the workfile.

Each date is returned in a number using standard EViews date representation (fractional days since 1st Jan 1AD, see “[Dates](#)” beginning on page 127).

A period is considered to end during the last millisecond contained within the period. In a regular frequency workfile, each period will end immediately before the start of the next period. In an irregular workfile there may be gaps between the end of one period and the start of the following period due to observations that were omitted in the workfile.

The `@DATE` and `@ENDDATE` functions can be combined with the EViews date manipulation functions to provide a wide variety of calendar information about a dated workfile.

For example, if we had a monthly workfile containing sales data for a product, we might expect the total sales that occurred in a given month to be related to the number of business days (Mondays to Fridays) that occurred within the month. We could create a new series in the workfile containing the number of business days in each month by using:

```
series busdays = @datediff(@date(+1), @date, "B")
```

If the workfile contained irregular data, we would need to use a more complicated expression since in this case we can not assume that the start of the next period occurs immediately after the end of the current period. For a monthly irregular file, we could use:

```
series busdays = @datediff(@dateadd(@date, 1, "M"), @date, "B")
```

Similarly, when working with a workfile containing daily share price data, we might be interested in whether price volatility is different in the days surrounding a holiday for which the market is closed. We can use the first formula given above to determine the

number of business days between adjacent observations in the workfile, then use this result to create two dummy variables that indicate whether a particular observation is before or after a holiday day.

```
series before_holiday = (busdays > 1)
series after_holiday = (busdays(-1) > 1)
```

We could then use these dummy variables as exogenous regressors in the variance equation of a GARCH estimation to estimate the impact of holidays on price volatility.

In many cases, you may wish to transform the date numbers returned by @DATE so that the information is contained in an alternate format. EViews provides workfile functions that bundle common translations of date numbers to usable information. These functions include:

- @YEAR: returns the four digit year in which the current observation begins. It is equivalent to “@DATEPART(@DATE, “YYYY””).
- @QUARTER: returns the quarter of the year in which the current observation begins. It is equivalent to “@DATEPART(@DATE, “Q”)”.
- @MONTH: returns the month of the year in which the current observation begins. It is equivalent to “@DATEPART(@DATE, “MM””).
- @DAY: returns the day of the month in which the current observation begins. It is equivalent to “@DATEPART(@DATE, “DD””).
- @WEEKDAY: returns the day of the week in which the current observation begins, where Monday is given the number 1 and Sunday is given the number 7. It is equivalent to “@DATEPART(@DATE, “W”)”.
- @STRDATE(*fmt*): returns the set of workfile row dates as strings, using the date format string *fmt*. See [“Date Formats” on page 130](#) for a discussion of date format strings.
- @SEAS(*season\_number*): returns a dummy variable based on the period within the current year in which the current observation occurs, where the year is divided up according to the workfile frequency. For example, in a quarterly file, “@SEAS(1)”, “@SEAS(2)”, “@SEAS(3)”, and “@SEAS(4)” correspond to the set of dummy variables for the four quarters of the year. These expressions are equivalent (in the quarterly workfile) to “@QUARTER = 1”, “@QUARTER = 2”, “@QUARTER = 3”, and “@QUARTER = 4”, respectively.
- @ISPERIOD(*arg*): returns a dummy variable for whether the observation is in the specified period, where *arg* is a double quoted date or period number. Note that in dated workfiles, *arg* is rounded down to the workfile frequency prior to computation.

Additional information on working with dates is provided in “[Dates](#)” beginning on [page 127](#).

## Trend Functions

One common task in time series analysis is the creation of variables that represent time trends. EViews provides two distinct functions for this purpose:

- `@TREND(["base_date"])`: returns a time trend that increases by one for each observation of the workfile. The optional *base\_date* may be provided to indicate the starting date for the trend.
- `@TRENDC(["base_date"])`: returns a calendar time trend that increases based on the number of calendar periods between successive observations. The optional *base\_date* may be provided to indicate the starting date for the trend.

The functions `@TREND` and `@TRENDC` are used to represent two different types of time trends that differ in some circumstances.

In a regular frequency workfile, `@TREND` and `@TRENDC` both return a simple trend that increases by one for each observation of the workfile.

In an irregular workfile, `@TREND` provides an observation trend as before, but `@TRENDC` now returns a calendar trend that increases based on the number of calendar periods between adjacent observations. For example, in a daily irregular file where a Thursday has been omitted because it was a holiday, the `@TRENDC` value would increase by two between the Wednesday before and the Friday after the holiday, while the `@TREND` will increase by only one.

Both `@TREND` and `@TRENDC` functions may be used with an argument consisting of a string containing the date at which the trend has the value of zero. If this argument is omitted, the first observation in the workfile will be given the value of zero.

The decision of which form of time trend is appropriate in a particular context should be based on what role the time trend is playing in the analysis. When used in estimation, a time trend is usually used to represent some sort of omitted variable. If the omitted variable is something that continues to increase independently of whether the workfile data is observed or not, then the appropriate trend would be the calendar trend. If the omitted variable is something that increases only during periods when the workfile data is observed, then the appropriate trend would be the observation trend.

An example of the former sort of variable would be where the trend is used to represent population growth, which continues to increase whether or not, for example, financial markets are open on a particular day. An example of the second sort of variable might be some type of learning effect, where learning only occurs when the activity is actually undertaken.

Note that while these two trends are provided as built in functions, other types of trends may also be generated based on the calendar data of the workfile. For example, in a file containing monthly sales data, a trend based on either the number of days in the month or the number of business days in the month might be more appropriate than a trend that increments by one for each observation.

These sorts of trends can be readily generated using @DATE and the @DATEDIFF functions. For example, to generate a trend based on the number of days elapsed between the start date of the current observation and the base date of 1st Jan 2000, we can use:

```
series daytrend = @datediff(@date, @dateval("1/1/2000"), "d")
```

When used in a monthly file, this series would provide a trend that adjusts for the fact that some months contain more days than others.

Note that trends in panel structured workfiles follow special rules. See “[Panel Trend Functions](#)” on page 563 for details.

## Panel Workfile Functions

### Panel Identifier Functions

Additional information is available in panel structured workfiles. EViews provides workfile functions that provide information about the cross-section, cell, and observation IDs associated with each observation in a panel workfile:

- @CROSSID: returns the cross-section index (cross-section number) of the current workfile observation.
- @CELLID: returns the inner dimension index value for the current observation in the workfile. The index numbers identify the unique values of the inner dimension observed across all cross-sections. Thus, if the first cross-section has annual observations for 1990, 1992, 1994, and 1995, and the second cross-section has observations for 1990, 1995, and 1997, the corresponding @CELLID values will be (1, 2, 3, 4) and (1, 4, 5), respectively.
- @OBSID: returns the observation number within each panel cross section. @OBSID is similar to @OBSNUM except that it resets to one whenever it crosses the seam between two adjacent cross sections.

See “[Identifier Indices](#)” on page 862 of the *User’s Guide* for additional discussion.

### Panel Trend Functions

Central to the notion of a panel trend is the notion that the trend values are initialized at the start of a cross-section, increase for successive observations in the specific cross-section, and are reset at the start of the next-cross section.

Beyond that, there are several notions of a time trend that may be employed. EViews provides four different functions that may be used to create a trend series: `@OBSID`, `@TRENDC`, `@CELLID`, and `@TREND`.

The `@OBSID` function may be used to return the simplest notion of a trend in which the values for each cross-section begin at one and increase by one for successive observations in the cross-section. To begin your trends at zero, simply use the expression “`@OBSID-1`”. Note that such a trend does not use information about the cell ID values in determining the value increment.

The calendar trend function, `@TRENDC`, computes trends in which values for observations with the earliest observed date are normalized to zero, and successive observations are incremented based on the calendar for the workfile frequency.

Lastly, `@CELLID` and `@TREND` compute trends using the observed dates in the panel:

- `@CELLID`, which returns an index into the unique values of the cell ID, returns a form of time trend in which the values increase based on the number of cell ID values between successive observations.
- `@TREND` function is equivalent to “`@CELLID-1`”.

In fully balanced workfiles (workfiles with the same set of cell identifier in each cross-section), the expressions “`@OBSID-1`”, “`@CELLID-1`”, and “`@TREND`” all return the same values. Additionally, if the workfile follows a regular frequency, then the `@TRENDC` function returns the same values as `@TREND`.

Note that because of the way they employ information computed across cross-sections, `@TREND` and `@TRENDC` may not take the optional *base\_date* argument in panel structured workfiles (see “[Trend Functions](#)” on page 562 ).

## Appendix F. String and Date Function Reference

---

The following is an alphabetical listing of the functions used when working with strings and dates in EViews.

@dateadd	Date Function
----------	---------------

Syntax:            @dateadd(*d, offset[, u]*)  
Argument 1:        date number, *d*  
Argument 2:        number of time units, *offset*  
Argument 3:        time unit, *u*  
Return:            date number

Returns the date number given by *d* offset by *offset* time units as specified by the time unit string *u*. If no time unit is specified, EViews will use the workfile regular frequency, if available.

Example:

Suppose that the value of *d* is 730088.0 (midnight, December 1, 1999). Then we can add and subtract 10 days from the date by using the functions

```
@dateadd(730088.0, 10, "dd")  
@dateadd(730088.0, -10, "dd")
```

which return 730098.0 (December 11, 1999) and (730078.0) (November 21, 1999). Note that these results could have been obtained by taking the original numeric value plus or minus 10.

To add 5 weeks to the existing date, simply specify “W” or “WW” as the time unit string:

```
@dateadd(730088.0, 5, "ww")
```

returns 730123.0 (January 5, 2000).

<b>@datediff</b>	<a href="#">Date Function</a>
------------------	-------------------------------

Syntax:            `@datediff(d1, d2[, u])`

Argument 1:        date number, *d1*

Argument 2:        date number, *d2*

Argument 3:        time unit, *u*

Return:            date number

Returns the difference between two date numbers *d1* and *d2*, measured by time units specified by the time unit string *u*. If no time unit is specified, EViews will use the workfile regular frequency, if available.

Example:

Suppose that *date1* is 730088.0 (December 1, 1999) and *date2* is 729754.0 (January 1, 1999), then,

```
@datediff(730088.0, 729754.0, "dd")
```

returns 334 for the number of days between the two dates. Note that this result is simply the difference between the two numbers.

The following expressions calculate differences in months and weeks:

```
@datediff(730088.0, 729754.0, "mm")
```

```
@datediff(730088.0, 729754.0, "ww")
```

return 11 and 47 for the number of months and weeks between the dates.

<b>@datefloor</b>	<a href="#">Date Function</a>
-------------------	-------------------------------

Syntax:            `@datefloor(d1, u[, step])`

Argument 1:        date number, *d1*

Argument 2:        time unit, *u*

Argument 3:        offset, *step*

Return:            date number

Finds the first possible date number associated with *d1* in the given time unit *u*, with an optional step offset.

Example:

Suppose that  $d1$  is 730110.5 (12 noon, December 23, 1999). Then the @DATEFLOOR values

```
@datefloor(730110.5, "dd")
@datefloor(730110.5, "mm")
```

yield 730110.0 (midnight, December 23, 1999) and 730088.0 (midnight, December 1, 1999).

```
@datefloor(730098.5, "q")
@datefloor(730110.5, "y", 1)
```

returns 730027.0 (midnight, October 1, 1999), and 729754.0 (midnight, January 1, 2000).

@datepart	Date Function
-----------	---------------

Syntax:            `@datepart( $d1, u$ )`

Argument 1:        date number,  $d1$

Argument 2:        time unit,  $u$

Return:            number

Returns a numeric part of a date number given by  $u$ , where  $u$  is a time unit string.

Example:

Consider the  $d1$  date value 730110.5 (noon, December 23, 1999). The @DATEPART values for

```
@datepart(730110.5, "dd")
@datepart(730110.5, "w")
@datepart(730110.5, "ww")
@datepart(730110.5, "mm")
@datepart(730110.5, "yy")
```

are 23 (day of the month), 4 (week in the month), 52 (week in the year), 12 (month in the year), and 1999 (year), respectively.

@datestr	Date Function   String Function
----------	---------------------------------

Syntax:            `@datestr( $d[, fmt]$ )`

Argument 1:        date number,  $d$

Argument 2:        date format string,  $fmt$

Return:            string

Convert the numeric date value, *d*, into a string representation of a date, *str*, using the optional format string *fmt*.

Example:

```
@datestr(730088, "mm/dd/yy")
```

will return “12/1/99”,

```
@datestr(730088, "DD/mm/yyyy")
```

will return “01/12/1999”, and

```
@datestr(730088, "Month dd, yyyy")
```

will return “December 1, 1999”, and

```
@datestr(730088, "w")
```

will produce the string “3”, representing the weekday number for December 1, 1999. See “[Dates](#)” on page 127 of the *User’s Guide* for additional details on date numbers and date format strings. See also [@strdate \(p. 577\)](#).

<b>@dateval</b>	<a href="#">Date Function</a>   <a href="#">String Function</a>
-----------------	---

Syntax:            `@dateval(str1[, fmt])`

Argument 1:        string, *str1*

Argument 2:        date format string, *fmt*

Return:            date number

Convert the string representation of a date, *str*, into a date number using the optional format string *fmt*.

Example:

```
@dateval("12/1/1999", "mm/dd/yyyy")
```

will return the date number for December 1, 1999 (730088) while

```
@dateval("12/1/1999", "dd/mm/yyyy")
```

will return the date number for January 12, 1999 (729765).

See “[Dates](#)” on page 127 of the *User’s Guide* for additional details on date numbers and date format strings.

<b>@dtoo</b>	Date Function   String Function
--------------	---------------------------------

Syntax:            `@dtoo(str)`

Argument:        `string, str`

Return:            integer

Date-TO-Observation. Returns the observation number corresponding to the date contained in the string. The observation number is relative to the start of the current workfile range, not the current sample. Observation numbers may be used to select a particular element in a vector that has been converted from a series, provided that NAs are preserved (see [stomna \(p. 597\)](#)).

Examples:

```
scalar obnum = @dtoo("1994:01")
vec1(1) = gdp(@dtoo("1955:01") + 10)
```

Suppose that the workfile contains quarterly data. Then the second example places the 1957:02 value of the GDP series in the first element of VEC1.

Note that `@DTOO` will generate an error if used in a panel structured workfile.

See also [@otod \(p. 575\)](#).

<b>@eqna</b>	String Function
--------------	-----------------

Syntax:            `@eqna(str1, str2)`

Argument 1:        `string, str1`

Argument 2:        `string, str2`

Return:            integer

Tests for equality of `str1` and `str2`, treating null strings as ordinary blank strings, and not as missing values. Strings which test as equal return a 1, and 0 otherwise.

Example:

```
@eqna("abc", "abc")
```

returns a 1, while

```
@eqna("", "def")
```

returns a 0.

See also [@isempty \(p. 571\)](#) and [@neqna \(p. 574\)](#).

<b>@insert</b>	<a href="#">String Function</a>
----------------	---------------------------------

Syntax:            `@insert(str1, str2, n)`  
Argument 1:        string, *str1*  
Argument 2:        string, *str2*  
Argument 3:        integer, *n*  
Return:            string

Inserts the string *str2* into the base string *str1* at the position given by the integer *n*.

Example:

```
@insert("I believe it can be done", "not ", 16)  
returns "I believe it cannot be done".
```

See also [@replace \(p. 575\)](#), [@instr \(p. 570\)](#) and [@mid \(p. 573\)](#).

<b>@instr</b>	<a href="#">String Function</a>
---------------	---------------------------------

Syntax:            `@instr(str1, str2[, n])`  
Argument 1:        string, *str1*  
Argument 2:        string, *str2*  
Argument 3:        integer, *n*  
Return:            string

Finds the starting position of the target string *str2* in the base string *str1*. By default, the function returns the location of the first occurrence of *str2* in *str1*. You may provide an optional integer *n* to change the occurrence. If the occurrence of the target string is not found, @INSTR will return a 0.

Example:

```
@instr("1.23415", "34")  
return the value 4, since the substring "34" appears beginning in the fourth character of  
the base string.
```

See also [@mid \(p. 573\)](#).

**@isempty****String Function**

Syntax:            `@isempty(str)`

Argument:        string, *str*

Return:           integer

Tests for whether *str* is a blank string, returning a 1 if *str* is a null string, and 0 otherwise.

Example:

```
@isempty("1.23415")
```

returns a 0, while

```
@isempty("")
```

return the value 1.

See also [@len](#), [@length](#) (p. 572).

**@left****String Function**

Syntax:            `@left(str, n)`

Argument 1:       string, *str*

Argument 2:       integer, *n*

Return:           string

Returns a string containing *n* characters from the left end of *str*. If the string is shorter than *n* characters, this function returns all of the characters in the source string.

Example:

```
scalar scl = @left("I did not do it",5)
```

returns "I did".

See also [@right](#) (p. 576), and [@mid](#) (p. 573).

<b>@len, @length</b>	<a href="#">String Function</a>
----------------------	---------------------------------

Syntax:            `@len(str), @length(str)`

Argument:        `string, str`

Return:            integer

Returns an integer value for the length of the string *str*.

Example:

```
@length("I did not do it")
```

returns the value 15.

See also [@mid \(p. 573\)](#).

<b>@lower</b>	<a href="#">String Function</a>
---------------	---------------------------------

Syntax:            `@lower(str)`

Argument:        `string, str`

Return:            string

Returns the lowercase representation of the string *str*.

Example:

```
@length("I did NOT do it")
```

returns the string “i did not do it”.

See also [@upper \(p. 578\)](#).

<b>@ltrim</b>	<a href="#">String Function</a>
---------------	---------------------------------

Syntax:            `@ltrim(str)`

Argument:        `string, str`

Return:            string

Returns the string *str* with spaces trimmed from the left.

Example:

```
@ltrim(" I doubt that I did it. ")
```

returns “I doubt that I did it. ”. Note that the spaces on the right remain.

See also [@rtrim \(p. 576\)](#) and [@trim \(p. 578\)](#).

<b>@makedate</b>	<a href="#">Date Function</a>
------------------	-------------------------------

Syntax:            `@makedate(arg1[, arg2[,arg3]], fmt)`  
 Argument 1:        number, *arg1*  
 Argument 2:        number(s) *arg2, arg3*  
 Argument 3:        date format, *fmt*  
 Return:            date number

Takes the numeric values given by the arguments *arg1*, and optionally, *arg2*, etc. and returns a date number using the required format string, *fmt*.

Example:

The expressions,

```
@makedate(1999, "yyyy")
@makedate(99, "yy")
```

both return the date number 729754.0 corresponding to 12 midnight on January 1, 1999.

```
@makedate(199003, "yyyymm")
@makedate(1990.3, "yyyy.mm")
@makedate(1031990, "ddmmYYYY")
@makedate(30190, "mmddyy")
```

all return the value 726526.0, representing March 1, 1990.

See “[Translating Ordinary Numbers into Date Numbers](#)” on page 139 for additional details.

<b>@mid</b>	<a href="#">String Function</a>
-------------	---------------------------------

Syntax:            `@mid(str, n1[, n2])`  
 Argument 1:        string, *str*  
 Argument 2:        integer, *n1*  
 Argument 3:        integer, *n2*  
 Return:            string

Returns *n2* characters from *str*, starting at location *n1* and continuing to the right. If you omit *n2*, it will return all of the remaining characters in the string.

Example:

```
%1 = @mid("I doubt it", 9, 2)  
%2 = @mid("I doubt it", 9)
```

See also [@left \(p. 571\)](#) and [@right \(p. 576\)](#).

<b>@neqna</b>	<a href="#">String Function</a>
---------------	---------------------------------

Syntax:            `@neqna(str1, str2)`

Argument 1:        string, *str1*

Argument 2:        string, *str2*

Return:            integer

Tests for inequality of *str1* and *str2*, treating null strings as ordinary blank strings, and not as missing values. Strings which test as equal return a 0, and 1 otherwise.

Example:

```
@neqna ("abc", "abc")
```

returns a 0, while

```
@eqna ("", "def")
```

returns a 1.

See also [@isempty \(p. 571\)](#) and [@eqna \(p. 569\)](#).

<b>@now</b>	<a href="#">Date Function</a>
-------------	-------------------------------

Syntax:            `@now`

Return:            date number

Returns the date number associated with the current time.

<b>@otod</b>	<a href="#">Date Function</a>   <a href="#">String Function</a>
--------------	---

Syntax:            `@otod(n)`

Argument:        integer, *n*

Return:            string

Observation-TO-Date. Returns a string containing the date or observation corresponding to observation number *n*, counted from the beginning of the current workfile. For example, consider the string assignment

```
%1 = @otod(5)
```

For an annual workfile dated 1991–2000, %1 will contain the string “1995”. For a quarterly workfile dated 1970:1–2000:4, %1 will contain the string “1971:1”. Note that `@otod(1)` returns the date or observation label for the start of the workfile.

See also [@dtoo](#) (p. 569).

<b>@replace</b>	<a href="#">String Function</a>
-----------------	---------------------------------

Syntax:            `@replace(str1, str2, str3[,n])`

Argument 1:       string, *str1*

Argument 2:       string, *str2*

Argument 3:       string, *str3*

Argument 4:       integer, *n*

Return:            string

Returns the base string *str1*, with the replacement *str3* substituted for the target string *str2*. By default, all occurrences of *str2* will be replaced, but you may provide an optional integer *n* to specify the number of occurrences to be replaced.

Example:

```
@replace("Do you think that you can do it?", "you", "I")
```

returns the string “Do I think that I can do it?”, while

```
@replace("Do you think that you can do it?", "you", "I", 1)
```

returns “Do I think that you can do it?”.

See also [@insert](#) (p. 570) and [@mid](#) (p. 573).

<b>@right</b>	<a href="#">String Function</a>
---------------	---------------------------------

Syntax:            `@right(str, n)`

Argument 1:        string, *str*

Argument 2:        integer, *n*

Return:            same as source

Returns a string containing *n* characters from the right end of *str*. If the source is shorter than *n*, the entire string is returned. Example:

```
%1 = @right("I doubt it",8)
```

returns the string “doubt it”.

See also [@left \(p. 571\)](#), [@mid \(p. 573\)](#).

<b>@rtrim</b>	<a href="#">String Function</a>
---------------	---------------------------------

Syntax:            `@rtrim(str)`

Argument:        string, *str*

Return:            string

Returns the string *str* with spaces trimmed from the right.

Example:

```
@rtrim(" I doubt that I did it. ")
```

returns the string “ I doubt that I did it.”. Note that the spaces on the left remain.

See also [@ltrim \(p. 572\)](#) and [@trim \(p. 578\)](#).

<b>@str</b>	<a href="#">String Function</a>
-------------	---------------------------------

Syntax:            `@str(d,[fmt])`

Argument 1:        scalar, *d*

Argument 2:        numeric format string, *fmt*

Return:            string

Returns a string representing the given number. You may provide an optional format string.

Example:

```
%1 = @str(15.23456)
alpha newa = @str(32.56)
```

See also [@val \(p. 578\)](#).

<b>@strdate</b>	<a href="#">Date Function</a>   <a href="#">String Function</a>
-----------------	---

Syntax:            `@strdate(fmt)`

Argument:        date format string, *fmt*

Return:            string corresponding to each element in workfile

Return the set of workfile row dates as strings, using the date format string *fmt*.

See also [@datestr \(p. 567\)](#) and [@strnow \(p. 577\)](#).

<b>@strlen</b>	<a href="#">String Function</a>
----------------	---------------------------------

Syntax:            `@strlen(s)`

Argument:        string, *s*

Return:            number *n*

Same as [@len](#), [@length \(p. 572\)](#).

<b>@strnow</b>	<a href="#">Date Function</a>   <a href="#">String Function</a>
----------------	---

Syntax:            `@strnow(fmt)`

Argument:        date format string, *fmt*

Return:            string

Returns a string representation of the current date number (at the moment the function is evaluated) using the date format string, *fmt*.

Example:

```
@strnow("DD/mm/yyyy")
```

returns the date associated with the current time in string form with 2-digit days, months, and 4-digit years separated by a slash, “24/12/2003”.

See also [@strdate \(p. 577\)](#) and [@datestr \(p. 567\)](#).

<b>@trim</b>	<a href="#">String Function</a>
--------------	---------------------------------

Syntax:            `@trim(str)`

Argument:        `string, str`

Return:           `string`

Returns the string *str* with spaces trimmed from the both the left and the right.

Example:

```
@rtrim(" I doubt that I did it. ")
```

returns the string “I doubt that I did it.”.

See also [@ltrim \(p. 572\)](#) and [@rtrim \(p. 576\)](#).

<b>@upper</b>	<a href="#">String Function</a>
---------------	---------------------------------

Syntax:            `@upper(str)`

Argument:        `string, str`

Return:           `string`

Returns the uppercase representation of the string *str*.

Example:

```
@length("I did not do it")
```

returns the string “I DID NOT DO IT”.

See also [@lower \(p. 572\)](#).

<b>@val</b>	<a href="#">String Function</a>
-------------	---------------------------------

Syntax:            `@val(str[, fmt])`

Argument 1:       `string, str`

Argument 2:       numeric format string, *fmt*

Return:           `scalar`

Returns the number that a string *str* represents. You may provide an optional numeric format string *fmt*. By default, EViews will attempt to interpret the string as a number using standard formats; if this is not possible, the function returns “NA”.

Example:

```
scalar sc1 = @val("17.4648")
scalar sc2 = @val(tab1(3,4))
scalar sc3 = @val("-234.35")
```

See also [@str \(p. 576\)](#).



## Appendix G. Matrix Reference

---

The following is an alphabetical listing of the functions and commands used when working with the EViews matrix language. For a description of the EViews matrix language, see [Chapter 3, “Matrix Language”, on page 23](#). For a quick summary of these entries, see [“Matrix Function and Command Summary” on page 44](#).

<b>@cholesky</b>	<a href="#">Matrix Algebra Function</a>
------------------	---

Syntax:            `@cholesky(s)`

Argument:        `sym, s`

Return:           `matrix`

Returns a matrix containing the Cholesky factorization of  $s$ . The Cholesky factorization finds the lower triangular matrix  $L$  such that  $LL'$  is equal to the symmetric source matrix. Example:

```
matrix fact = @cholesky(s1)
matrix orig = fact*@transpose(fact)
```

<b>colplace</b>	<a href="#">Matrix Utility Command</a>
-----------------	--

Syntax:            `colplace(m, v, n)`

Argument 1:       `matrix, m`

Argument 2:       `vector, v`

Argument 3:       `integer, n`

Places the column vector  $v$  into the matrix  $m$  at column  $n$ . The number of rows of  $m$  and  $v$  must match, and the destination column must already exist within  $m$ . Example:

```
colplace(m1,v1,3)
```

The third column of M1 will be set equal to the vector V1.

See also [rowplace \(p. 596\)](#).

<b>@columnextract</b>	<a href="#">Matrix Utility Function</a>
-----------------------	---

Syntax:            `@columnextract(m, n)`  
Argument 1:        matrix or sym, *m*  
Argument 2:        integer, *n*  
Return:            vector

Extract a vector from column *n* of the matrix object *m*, where *m* is a matrix or sym. Example:

```
vector v1 = @columnextract(m1, 3)
```

Note that while you may extract the first column of a column vector, or any column of a row vector, the command is more easily executed using simple element or vector assignment in these cases.

See also [@rowextract \(p. 595\)](#).

<b>@columns</b>	<a href="#">Matrix Utility Function</a>
-----------------	---

Syntax:            `@columns(o)`  
Argument:          matrix, vector, rowvector, sym, scalar, or series, *o*  
Return:            integer

Returns the number of columns in the matrix object *o*. Example:

```
scalar sc2 = @columns(m1)
vec1(2) = @columns(s1)
```

See also [@rows \(p. 596\)](#).

<b>@cond</b>	<a href="#">Matrix Algebra Function</a>
--------------	---

Syntax:            `@cond(o, n)`  
Argument 1:        matrix or sym, *o*  
Argument 2:        (optional) scalar *n*  
Return:            scalar

Returns the condition number of a square matrix or sym, *o*. If no norm is specified, the infinity norm is used to determine the condition number. The condition number is the

product of the norm of the matrix divided by the norm of the inverse. Possible norms are “1” for the infinity norm, “0” for the Frobenius norm, and an integer “n” for the  $L^n$  norm.

Example:

```
scalar scl = @cond(m1)
mat1(1, 4) = @cond(s1, 2)
```

<b>@convert</b>	Matrix Utility Function
-----------------	-------------------------

Syntax:	<code>@convert(o, smp)</code>
Argument 1:	series or group, <i>o</i>
Argument 2:	(optional) sample, <i>smp</i>
Return:	vector or matrix

If *o* is a series, `@convert` returns a vector from the values of *o* using the optional sample *smp* or the current workfile sample. If any observation has the value “NA”, the observation will be omitted from the vector. Examples:

```
vector v2 = @convert(ser1)
vector v3 = @convert(ser2, smp1)
```

If *o* is a group, `@convert` returns a matrix from the values of *o* using the optional sample object *smp* or the current workfile sample. The series in *o* are placed in the columns of the resulting matrix in the order they appear in the group spreadsheet. If any of the series for a given observation has the value “NA”, the observation will be omitted for all series. For example:

```
matrix m1=@convert(grp1)
matrix m2=@convert(grp1, smp1)
```

For a conversion method that preserves NAs, see [stomna \(p. 597\)](#).

<b>@COR</b>	Matrix Descriptive Statistic
-------------	------------------------------

Syntax:             $\text{@cor}(v1, v2)$   
Argument 1:        vector, rowvector, or series,  $v1$   
Argument 2:        vector, rowvector, or series,  $v2$   
Return:            scalar

Syntax:             $\text{@cor}(o)$   
Argument:          matrix object or group,  $o$   
Return:            sym

If used with two vector or series objects,  $v1$  and  $v2$ , `@cor` returns the correlation between the two vectors or series. Examples:

```
scalar sc1 = @cor(v1,v2)
s1(1,2) = @cor(v1,r1)
```

If used with a matrix object or group,  $o$ , `@cor` calculates the correlation matrix between the columns of the matrix object.

```
scalar sc2 = @cor(v1,v2)
mat3(4,2) = 100*@cor(r1,v1)
```

For series and group calculations, EViews will use the current workfile sample. See also [@cov \(p. 584\)](#).

<b>@COV</b>	Matrix Descriptive Statistic
-------------	------------------------------

Syntax:             $\text{@cov}(v1, v2)$   
Argument 1:        vector, rowvector, or series,  $v1$   
Argument 2:        vector, rowvector, or series,  $v2$   
Return:            scalar

Syntax:             $\text{@cov}(o)$   
Argument:          matrix object or group,  $o$   
Return:            sym

If used with two vector or series objects,  $v1$  and  $v2$ , `@cov` returns the covariance between the two vectors or series. Examples:

---

```
scalar scl = @cov(v1, v2)
s1(1,2) = @cov(v1, r1)
```

If used with a matrix object or group, *o*, `@cov` calculates the covariance matrix between the columns of the matrix object.

```
!1 = @cov(v1, v2)
mat3(4,2) = 100*@cov(r1, v1)
```

For series and group calculations, EViews will use the current workfile sample. See also [@cor \(p. 584\)](#).

<a href="#">@det</a>	<a href="#">Matrix Algebra Function</a>
----------------------	---

Syntax:            `@det(m)`  
 Argument:        matrix or sym, *m*  
 Return:           scalar

Calculate the determinant of the square matrix or sym, *m*. The determinant is nonzero for a nonsingular matrix and 0 for a singular matrix. Example:

```
scalar scl = @det(m1)
vec4(2) = @det(s2)
```

See also [@rank \(p. 594\)](#).

<a href="#">@eigenvalues</a>	<a href="#">Matrix Algebra Function</a>
------------------------------	---

Syntax:            `@eigenvalues(s)`  
 Argument:        sym, *s*  
 Return:           vector

Returns a vector containing the eigenvalues of the sym. The eigenvalues are those scalars  $\lambda$  that satisfy  $Sx = \lambda x$  where *S* is the sym associated with the argument *s*. Associated with each eigenvalue is an eigenvector (see [@eigenvectors \(p. 586\)](#)). The eigenvalues are arranged in ascending order.

Example:

```
vector v1 = @eigenvalues(s1)
```

<b>@eigenvectors</b>	<a href="#">Matrix Algebra Function</a>
----------------------	---

Syntax:            [@eigenvectors\(s\)](#)

Argument:        `sym, s`

Return:           `matrix`

Returns a square matrix, of the same dimension as the `sym`, whose columns are the eigenvectors of the source matrix. Each eigenvector  $v$  satisfies  $Sv = nv$ , where  $S$  is the symmetric matrix given by  $s$ , and where  $n$  is the eigenvalue associated with the eigenvector  $v$ . The eigenvalues are arranged in ascending order, and the columns of the eigenvector matrix correspond to the sorted eigenvalues. Example:

```
matrix m2 = @eigenvectors(s1)
```

See also the function [@eigenvalues \(p. 585\)](#).

<b>@explode</b>	<a href="#">Matrix Utility Function</a>
-----------------	---

Syntax:            [@explode\(s\)](#)

Argument:        `sym, s`

Return:           `matrix`

Creates a square matrix from a `sym, s`, by duplicating the lower triangle elements into the upper triangle. Example:

```
matrix m2 = @explode(s1)
```

See also [@implode \(p. 588\)](#).

<b>@filledmatrix</b>	<a href="#">Matrix Utility Function</a>
----------------------	---

Syntax:            [@filledmatrix\(n1, n2, n3\)](#)

Argument 1:       `integer, n1`

Argument 2:       `integer, n2`

Argument 3:       `scalar, n3`

Return:           `matrix`

Returns a matrix with  $n1$  rows and  $n2$  columns, where each element contains the value  $n3$ . Example:

```
matrix m2 = @filledmatrix(3, 2, 7)
```

creates a  $3 \times 2$  matrix where each element is set to 7. See also [fill \(p. 282\)](#).

<b>@filledrowvector</b>	<a href="#">Matrix Utility Function</a>
-------------------------	---

Syntax:            `@filledrowvector( $n1, n2$ )`

Argument 1:        integer,  $n1$

Argument 2:        scalar,  $n2$

Return:            rowvector

Returns a rowvector of length  $n1$ , where each element contains the value  $n2$ . Example:

```
rowvector r1 = @filledrowvector(3, 1)
```

creates a 3 element rowvector where each element is set to 1. See also [fill \(p. 282\)](#).

<b>@filledsym</b>	<a href="#">Matrix Utility Function</a>
-------------------	---

Syntax:            `@filledsym( $n1, n2$ )`

Argument 1:        integer,  $n1$

Argument 2:        scalar,  $n2$

Return:            sym

Returns an  $n1 \times n1$  sym, where each element contains  $n2$ . Example:

```
sym s2= @filledsym(3, 9)
```

creates a  $3 \times 3$  sym where each element is set to 9. See also [fill \(p. 282\)](#).

<b>@filledvector</b>	<a href="#">Matrix Utility Function</a>
----------------------	---

Syntax:            `@filledvector( $n1, n2$ )`

Argument 1:        integer,  $n1$

Argument 2:        scalar,  $n2$

Return:            vector

Returns a vector of length  $n1$ , where each element contains the value  $n2$ . Example:

```
vector r1 = @filledvector(5, 6)
```

creates a 5 element column vector where each element is set to 6. See also [fill \(p. 282\)](#).

<b>@getmaindiagonal</b>	<a href="#">Matrix Utility Function</a>
-------------------------	---

Syntax:            `@getmaindiagonal(m)`

Argument:        matrix or sym, *m*

Return:           vector

Returns a vector created from the main diagonal of the matrix or sym object. Example:

```
vector v1 = @getmaindiagonal(m1)
```

```
vector v2 = @getmaindiagonal(s1)
```

<b>@identity</b>	<a href="#">Matrix Utility Function</a>
------------------	---

Syntax:            `@identity(n)`

Argument:        integer, *n*

Return:           matrix

Returns a square  $n \times n$  identity matrix. Example:

```
matrix i1 = @identity(4)
```

<b>@implode</b>	<a href="#">Matrix Utility Function</a>
-----------------	---

Syntax:            `@implode(m)`

Argument:        square matrix, *m*

Return:           sym

Forms a sym by copying the lower triangle of a square input matrix, *m*. Where possible, you should use a sym in place of a matrix to take advantage of computational efficiencies. Be sure you know what you are doing if the original matrix is not symmetric—this function does not check for symmetry. Example:

```
sym s2 = @implode(m1)
```

See also [@explode \(p. 586\)](#).

<b>@inner</b>	Matrix Algebra Function
---------------	-------------------------

Syntax:            `@inner(v1, v2, smp)`

Argument 1:      vector, rowvector, or series, *v1*

Argument 2:      vector, rowvector, or series, *v2*

Argument 3:     (optional) sample, *smp*

Return:          scalar

Syntax:            `@inner(o, smp)`

Argument 1:      matrix object or group, *o*

Argument 2:     (optional) sample, *smp*

Return:          scalar

If used with two vectors, *v1* and *v2*, `@inner` returns the dot or inner product of the two vectors. Examples:

```
scalar scl = @inner(v1, v2)
s1(1,2) = @inner(v1, r1)
```

If used with two series, `@inner` returns the inner product of the series using observations in the workfile sample. You may provide an optional sample.

If used with a matrix or sym, *o*, `@inner` returns the inner product, or moment matrix, *o'o*. Each element of the result is the vector inner product of two columns of the source matrix. The size of the resulting sym is the number of columns in *o*. Examples:

```
scalar scl = @inner(v1)
sym sym1 = @inner(m1)
```

If used with a group, `@inner` returns the uncentered second moment matrix of the data in the group, *g*, using the observations in the sample, *smp*. If no sample is provided, the workfile sample is used. Examples:

```
sym s2 = @inner(gr1)
sym s3 = @inner(gr1, smp1)
```

See also [@outer \(p. 593\)](#).

<b>@inverse</b>	<a href="#">Matrix Algebra Function</a>
-----------------	---

Syntax:            `@inverse(m)`  
Argument:        square matrix or sym, *m*  
Return:            matrix or sym

Returns the inverse of a square matrix object or sym. The inverse has the property that the product of the source matrix and its inverse is the identity matrix. The inverse of a matrix returns a matrix, while the inverse of a sym returns a sym. Note that inverting a sym is much faster than inverting a matrix.

Examples:

```
matrix m2 = @inverse(m1)
sym s2 = @inverse(s1)
sym s3 = @inverse(@implode(m2))
```

See [@solvesystem \(p. 596\)](#).

<b>@issingular</b>	<a href="#">Matrix Algebra Function</a>
--------------------	---

Syntax:            `@issingular(o)`  
Argument:        matrix or sym, *o*  
Return:            integer

Returns “1” if the square matrix or sym, *o*, is singular, and “0” otherwise. A singular matrix has a determinant of 0, and cannot be inverted. Example:

```
scalar scl = @issingular(m1)
```

<b>@kronecker</b>	<a href="#">Matrix Algebra Function</a>
-------------------	---

Syntax:            `@kronecker(o1, o2)`  
Argument 1:       matrix object, *o1*  
Argument 2:       matrix object, *o2*  
Return:            matrix

Calculates the Kronecker product of the two matrix objects, *o1* and *o2*. The resulting matrix has a number of rows equal to the product of the numbers of rows of the two

matrix objects and a number of columns equal to the product of the numbers of columns of the two matrix objects. The elements of the resulting matrix consist of submatrices consisting of one element of the first matrix object multiplied by the entire second matrix object. Example:

```
matrix m3 = @kronecker(m1,m2)
```

<b>@makediagonal</b>	<a href="#">Matrix Utility Function</a>
----------------------	---

Syntax:	<code>@makediagonal(v, k)</code>
Argument 1:	vector or rowvector, <i>v</i>
Argument 2:	(optional) integer, <i>k</i>
Return:	matrix

Creates a square matrix with the specified vector or rowvector, *v*, in the *k*-th diagonal relative to the main diagonal, and zeroes off the diagonal. If no *k* value is provided or if *k* is set to 0, the resulting matrix will have the same number of rows and columns as the length of *v*, and will have *v* in the main diagonal. If a value for *k* is provided, the matrix has the same number of rows and columns as the number of elements in the vector plus *k*, and will place *v* in the diagonal offset from the main by *k*.

Examples:

```
matrix m1 = @makediagonal(v1)
matrix m2 = @makediagonal(v1,1)
matrix m4 = @makediagonal(r1,-3)
```

M1 will contain V1 in the main diagonal; M2 will contain V1 in the diagonal immediately above the main diagonal; M4 will contain R1 in the diagonal 3 positions below the main diagonal. Using the optional *k* parameter may be useful in creating covariance matrices for AR models. For example, you can create an AR(1) correlation matrix by issuing the commands:

```
matrix(10,10) m1
vector(9) rho = .3
m1 = @makediagonal(rho,-1) + @makediagonal(rho,+1)
m1 = m1 + @identity(10)
```

<b>matplace</b>	<a href="#">Matrix Utility Command</a>
-----------------	--

Syntax:            `matplace(m1, m2, n1, n2)`

Argument 1:        matrix, *m1*

Argument 2:        matrix, *m2*

Argument 3:        integer, *n1*

Argument 4:        integer, *n2*

Places the matrix object *m2* into *m1* at row *n1* and column *n2*. The sizes of the two matrices do not matter, as long as *m1* is large enough to contain all of *m2* with the upper left cell of *m2* placed at row *n1* and column *n2*.

Example:

```
matrix(100,5) m1  
matrix(100,2) m2  
matplace(m1,m2,1,1)
```

<b>mtos</b>	<a href="#">Matrix Utility Command</a>
-------------	--

Convert matrix to a series or group. Fills a series or group with the data from a vector or matrix.

### Syntax

Vector Proc:        `mtos(vector, series[, sample])`

Matrix Proc:        `mtos(matrix, group[, sample])`

Matrix-TO-Series Object. Include the vector or matrix name in parentheses, followed by a comma and then the series or group name. The number of included observations in the sample must match the row size of the matrix to be converted. If no sample is provided, the matrix is written into the series using the current workfile sample. Example:

```
mtos(mom,gr1)
```

converts the first column of the matrix MOM to the first series in the group GR1, the second column of MOM to the second series in GR1, and so on. The current workfile sample length must match the row length of the matrix MOM. If GR1 is an existing group object, the number of series in GR1 must match the number of columns of MOM. If a group object named GR1 does not exist, EViews creates GR1 with the first series named SER1, the second series named SER2, and so on.

```

series col1
series col2
group g1 col1 col2
sample s1 1951 1990
mtos(m1,g1,s1)

```

The first two lines declare series objects, the third line declares a group object, the fourth line declares a sample object, and the fifth line converts the columns of the matrix M1 to series in group G1 using sample S1. This command will generate an error if M1 is not a  $40 \times 2$  matrix.

### Cross-references

See [Chapter 3, “Matrix Language”, on page 23](#) for further discussions and examples of matrices.

See also [stom \(p. 597\)](#) and [stomna \(p. 597\)](#).

<b>@norm</b>	<a href="#">Matrix Algebra Function</a>
--------------	---

Syntax:	<code>@norm(o, n)</code>
Argument 1:	matrix, vector, rowvector, sym, scalar, or series, <i>o</i>
Argument 2:	(optional) integer, <i>n</i>
Return:	scalar

Returns the value of the norm of any matrix object, *o*. Possible choices for the norm type *n* include “-1” for the infinity norm, “0” for the Frobenius norm, and an integer “*n*” for the  $L^n$  norm. If no norm type is provided, this function returns the infinity norm.

Examples:

```

scalar sc1 = @norm(m1)
scalar sc2 = @norm(v1,1)

```

<b>@outer</b>	<a href="#">Matrix Algebra Function</a>
---------------	---

Syntax:	<code>@outer(v1, v2)</code>
Argument 1:	vector, rowvector, or series, <i>v1</i>
Argument 2:	vector, rowvector, or series, <i>v2</i>
Return:	matrix

Calculates the cross product of  $v1$  and  $v2$ . Vectors may be either row or column vectors. The outer product is the product of  $v1$  (treated as a column vector) and  $v2$  (treated as a row vector), and is a square matrix of every possible product of the elements of the two inputs. Example:

```
matrix m1=@outer(v1,v2)
matrix m4=@outer(r1,r2)
```

See also [@inner \(p. 589\)](#).

<b>@permute</b>	<a href="#">Matrix Utility Function</a>
-----------------	---

Syntax:            `@permute(m1)`

Input:            matrix  $m1$

Return:            matrix

This function returns a matrix whose rows are randomly drawn without replacement from rows of the input matrix  $m1$ . The output matrix has the same size as the input matrix.

```
matrix xp = @permute(x)
```

To draw with replacement from rows of a matrix, use [@resample \(p. 595\)](#).

<b>@rank</b>	<a href="#">Matrix Algebra Function</a>
--------------	---

Syntax:            `@rank(o, n)`

Argument 1:        vector, rowvector, matrix, sym, or series,  $o$

Argument 2:        (optional) integer,  $n$

Return:            integer

Returns the rank of the matrix object  $o$ . The rank is calculated by counting the number of singular values of the matrix which are smaller in absolute value than the tolerance, which is given by the argument  $n$ . If  $n$  is not provided, EViews uses the value given by the largest dimension of the matrix multiplied by the norm of the matrix multiplied by machine epsilon (the smallest representable number).

```
scalar rank1 = @rank(m1)
scalar rank2 = @rank(s1)
```

See also [@svd \(p. 599\)](#).

<b>@resample</b>	Matrix Utility Function
------------------	-------------------------

Syntax:            `@resample(m1, n2, n3, v4)`  
 Input 1:            matrix *m1*  
 Input 2:            (optional) integer *n2*  
 Input 3:            (optional) positive integer *n3*  
 Input 4:            (optional) vector *v4*  
 Output:            matrix

This function returns a matrix whose rows are randomly drawn with replacement from rows of the input matrix.

*n2* represents the number of “extra” rows to be drawn from the matrix. If the input matrix has *r* rows and *c* columns, the output matrix will have  $r + n2$  rows and *c* columns. By default,  $n2 = 0$ .

*n3* represents the block size for the resample procedure. If you specify  $n3 > 1$ , then blocks of consecutive rows of length *n3* will be drawn with replacement from the first  $r - n3 + 1$  rows of the input matrix.

You may provide a name for the vector *v4* to be used for weighted resampling. The weighting vector must have length *r* and all elements must be non-missing and non-negative. If you provide a weighting vector, each row of the input matrix will be drawn with probability proportional to the weights in the corresponding row of the weighting vector. (The weights need not sum to 1. EViews will automatically normalize the weights).

```
matrix xb = @bootstrap(x)
```

To draw without replacement from rows of a matrix, use [@permute \(p. 594\)](#).

<b>@rowextract</b>	Matrix Utility Function
--------------------	-------------------------

Syntax:            `@rowextract(m, n)`  
 Argument 1:        matrix or sym, *m*  
 Argument 2:        integer, *n*  
 Return:            rowvector

Extracts a rowvector from row *n* of the matrix object *m*. Example:

```
rowvector r1 = @rowextract(m1, 3)
```

See also [@columnextract \(p. 582\)](#).

<b>rowplace</b>	<a href="#">Matrix Utility Command</a>
-----------------	--

Syntax:            `rowplace(m, r, n)`

Argument 1:        matrix, *m*

Argument 2:        rowvector, *r*

Argument 3:        integer

Places the rowvector *r* into the matrix *m* at row *n*. The number of columns in *m* and *r* must match, and row *n* must exist within *m*. Example:

```
rowplace (m1, r1, 4)
```

See also [colplace \(p. 581\)](#).

<b>@rows</b>	<a href="#">Matrix Utility Function</a>
--------------	---

Syntax:            `@rows(o)`

Argument:        matrix, vector, rowvector, sym, series, or group, *o*

Return:           scalar

Returns the number of rows in the matrix object, *o*.

Example:

```
scalar scl=@rows(m1)
scalar size=@rows(m1)*@columns(m1)
```

For series and groups [@rows \(p. 596\)](#) returns the number of observations in the workfile range. See also [@columns \(p. 582\)](#).

<b>@solvesystem</b>	<a href="#">Matrix Algebra Function</a>
---------------------	---

Syntax:            `@solvesystem(o, v)`

Argument 1:        matrix or sym, *o*

Argument 2:        vector, *v*

Return:           vector

Returns the vector  $x$  that solves the equation  $Mx = p$  where the matrix or sym  $M$  is given by the argument  $o$ . Example:

```
vector v2 = @solvesystem(m1,v1)
```

See also [@inverse \(p. 590\)](#).

<b>stom</b>	<a href="#">Matrix Utility Command</a>
-------------	--

- |             |                         |
|-------------|-------------------------|
| Syntax:     | stom( $o1, o2, smp$ )   |
| Argument 1: | series or group, $o1$   |
| Argument 2: | vector or matrix, $o2$  |
| Argument 3: | (optional) sample $smp$ |

Series-TO-Matrix Object. If  $o1$  is a series, stom fills the vector  $o2$  with data from the  $o1$  using the optional sample object  $smp$  or the workfile sample.  $o2$  will be resized accordingly. If any observation has the value “NA”, the observation will be omitted from the vector. Example:

```
stom(ser1,v1)
stom(ser1,v2,smp1)
```

If  $o1$  is a group, stom fills the matrix  $o2$  with data from  $o1$  using the optional sample object  $smp$  or the workfile sample.  $o2$  will be resized accordingly. The series in  $o1$  are placed in the columns of  $o2$  in the order they appear in the group spreadsheet. If any of the series in the group has the value “NA” for a given observation, the observation will be omitted for all series. Example:

```
stom(grp1,m1)
stom(grp1,m2,smp1)
```

For a conversion method that preserves NAs, see [stomna \(p. 597\)](#).

<b>stomna</b>	<a href="#">Matrix Utility Command</a>
---------------	--

- |             |                         |
|-------------|-------------------------|
| Syntax:     | stomna( $o1, o2, smp$ ) |
| Argument 1: | series or group, $o1$   |
| Argument 2: | vector or matrix, $o2$  |
| Argument 3: | (optional) sample $smp$ |

Series-TO-Matrix Object with NAs. If *o1* is a series, *stom* fills the vector *o2* with data from *o1* using the optional sample object *smp* or the workfile sample. *o2* will be resized accordingly. All “NA” values in the series will be assigned to the corresponding vector elements.

Example:

```
stom(ser1,v1)  
stom(ser1,v2,smp1)
```

If *o1* is a group, *stom* fills the matrix *o2* with data from *o1* using the optional sample object *smp* or the workfile sample. *o2* will be resized accordingly. The series in *o1* are placed in the columns of *o2* in the order they appear in the group spreadsheet. All NAs will be assigned to the corresponding matrix elements. Example:

```
stomna(grp1,m1)  
stomna(grp1,m2,smp1)
```

For conversion methods that automatically remove observations with NAs, see [@convert \(p. 583\)](#) and [stom \(p. 597\)](#).

@subextract	Matrix Utility Function
-------------	-------------------------

Syntax:	<code>@subextract(o, n1, n2, n3, n4)</code>
Argument 1:	vector, rowvector, matrix or sym, <i>o</i>
Argument 2:	integer, <i>n1</i>
Argument 3:	integer, <i>n2</i>
Argument 4:	(optional) integer, <i>n3</i>
Argument 5:	(optional) integer, <i>n4</i>
Return:	matrix

Returns a submatrix of a specified matrix, *o*. *n1* is the row and *n2* is the column of the upper left element to be extracted. The optional arguments *n3* and *n4* provide the row and column location of the lower right corner of the matrix. Unless *n3* and *n4* are provided this function returns a matrix containing all of the elements below and to the right of the starting element.

Examples:

```
matrix m2 = @subextract(m1,5,9,6,11)  
matrix m2 = @subextract(m1,5,9)
```

<b>@svd</b>	Matrix Algebra Function
-------------	-------------------------

Syntax:            `@svd(m1, v1, m2)`

Argument 1:      matrix or sym, *m1*

Argument 2:      vector, *v1*

Argument 3:      matrix or sym, *m2*

Return:            matrix

Performs a singular value decomposition of the matrix *m1*. The matrix *U* is returned by the function, the vector *v1* will be filled (resized if necessary) with the singular values and the matrix *m2* will be assigned (resized if necessary) the other matrix, *V*, of the decomposition. The singular value decomposition satisfies:

$$\begin{aligned} m1 &= UWV \\ U'U &= V'V = I \end{aligned} \tag{G.1}$$

where *W* is a diagonal matrix with the singular values along the diagonal. Singular values close to zero indicate that the matrix may not be of full rank. See the [@rank \(p. 594\)](#) function for a related discussion.

Examples:

```
matrix m2
vector v1
matrix m3 = @svd(m1,v1,m2)
```

<b>@trace</b>	Matrix Algebra Function
---------------	-------------------------

Syntax:            `@trace(m)`

Argument:        matrix or sym, *m*

Return:            scalar

Returns the trace (the sum of the diagonal elements) of a square matrix or sym, *m*. Example:

```
scalar scl = @trace(m1)
```

<b>@transpose</b>	<a href="#">Matrix Algebra Function</a>
-------------------	---

Syntax:            `@transpose(o)`

Argument:        matrix, vector, rowvector, or sym, *o*

Return:          matrix, rowvector, vector, or sym

Forms the transpose of a matrix object, *o*. *o* may be a vector, rowvector, matrix, or a sym. The result is a matrix object with a number of rows equal to the number of columns in the original matrix and number of columns equal to the number of rows in the original matrix. This function is an identity function for a sym, since a sym by definition is equal to its transpose. Example:

```
matrix m2 = @transpose(m1)
rowvector r2 = @transpose(v1)
```

<b>@unitvector</b>	<a href="#">Matrix Utility Function</a>
--------------------	---

Syntax:            `@unitvector(n1, n2)`

Argument 1:       integer, *n1*

Argument 2:       integer, *n2*

Return:          vector

Creates an *n1* element vector with a “1” in the *n2*-th element, and “0” elsewhere. Example:

```
vec v1 = @unitvector(8, 5)
```

creates an 8 element vector with a “1” in the fifth element and “0” for the other 7 elements. Note: if you wish to create an *n1* element vector of ones, you should use a declaration statement of the form:

```
vector(n1) v1=1
```

<b>@vec</b>	<a href="#">Matrix Utility Function</a>
-------------	---

Syntax:            `@vec(o)`

Argument:        matrix, sym, *o*

Return:          vector

Creates a vector from the columns of the given matrix stacked one on top of each other. The vector will have the same number of elements as the source matrix.

Example:

```
vector v1 = @vec(m1)
```

<b>@vech</b>	<a href="#">Matrix Utility Function</a>
--------------	---

Syntax:              **@vech(*o*)**

Argument:          matrix, sym, *o*

Return:             vector

Creates a vector from the columns of the lower triangle of the source square matrix *o* stacked on top of each another. The vector has the same number of elements as the source matrix has in its lower triangle. Example:

```
vector v1 = @vech(m1)
```



## Appendix H. Programming Language Reference

---

The following reference is an alphabetical listing of the program statements and support functions used by the EViews programming language.

For details on the EViews programming language, see [Chapter 6, “EViews Programming”, on page 83](#). For a quick summary of these entries, see [“Programming Summary” on page 115](#).

<b>call</b>	<a href="#">Program Statement</a>
-------------	-----------------------------------

Call a subroutine within a program.

The call statement is used to call a subroutine within a program.

### Cross-references

See [“Calling Subroutines” on page 110](#). See also [subroutine \(p. 611\)](#), [endsub \(p. 604\)](#).

<b>@date</b>	<a href="#">Support Function</a>
--------------	----------------------------------

Syntax:            **@date**

Return:            string

Returns a string containing the current date in “mm/dd/yy” format.

### Examples

```
%y = @date
```

assigns a string of the form “10/10/00”

### Cross-references

See also [@time \(p. 612\)](#).

<b>else</b>	Program Statement
-------------	-------------------

ELSE clause of IF statement in a program.

Starts a sequence of commands to be executed when the IF condition is false. The `else` keyword must be terminated with an `endif`.

### Syntax

```
if [condition] then  
    [commands to be executed if condition is true]  
else  
    [commands to be executed if condition is false]  
endif
```

### Cross-references

See “[IF Statements](#)” on page 99. See also, [if](#) (p. 606), [endif](#) (p. 604), [then](#) (p. 612).

<b>endif</b>	Program Statement
--------------	-------------------

End of IF statement. Marks the end of an IF, or an IF-ELSE statement.

### Syntax

```
if [condition] then  
    [commands if condition true]  
endif
```

### Cross-references

See “[IF Statements](#)” on page 99. See also, [if](#) (p. 606), [else](#) (p. 604), [then](#) (p. 612).

<b>endsub</b>	Program Statement
---------------	-------------------

Mark the end of a subroutine.

### Syntax

```
subroutine name(arguments)  
    commands
```

---

```
endsub
```

### Cross-references

See “[Defining Subroutines](#)” beginning on page 108. See also, [subroutine \(p. 611\)](#), [return \(p. 609\)](#).

<b>@errorcount</b>	<a href="#">Support Function</a>
--------------------	----------------------------------

Syntax:           **@errorcount**

Argument:       none

Return:           integer

Number of errors encountered. Returns a scalar containing the number of errors encountered during program execution.

<b>@evpath</b>	<a href="#">Support Function</a>
----------------	----------------------------------

Syntax:           **@evpath**

Return:           string

Returns a string containing the directory path for the EViews executable.

### Examples

If your currently executing copy of EViews is installed in “D:\\EVIEWS”, then

```
%y = @evpath
```

assigns a string of the form “D:\\EVIEWS”.

### Cross-references

See also [cd \(p. 229\)](#) and [@temppath \(p. 611\)](#).

<b>exitloop</b>	<a href="#">Program Statement</a>
-----------------	-----------------------------------

Exit from current loop in programs.

`exitloop` causes the program to break out of the current FOR or WHILE loop.

### Syntax

Command:       **exitloop**

## Examples

```
for !i=1 to 107  
    if !i>6 then exitloop  
next
```

## Cross-references

See “[The FOR Loop](#)” on page 101. See also, [stop \(p. 611\)](#), [return \(p. 609\)](#), [for \(p. 606\)](#), [next \(p. 607\)](#), [step \(p. 610\)](#).

<b>for</b>	<a href="#">Program Statement</a>
------------	-----------------------------------

**FOR loop in a program.**

The **for** statement is the beginning of a FOR...NEXT loop in a program.

## Syntax

```
for counter = start to end [step stepsize]  
    [commands]  
next
```

## Cross-references

See “[The FOR Loop](#)” on page 101. See also, [exitloop \(p. 605\)](#), [next \(p. 607\)](#), [step \(p. 610\)](#).

<b>if</b>	<a href="#">Program Statement</a>
-----------	-----------------------------------

**IF statement in a program.**

The **if** statement marks the beginning of a condition and commands to be executed if the statement is true. The statement must be terminated with the beginning of an ELSE clause, or an **endif**.

## Syntax

```
if [condition] then  
    [commands if condition true]  
endif
```

## Cross-references

See “[IF Statements](#)” on page 99. See also [else \(p. 604\)](#), [endif \(p. 604\)](#), [then \(p. 612\)](#).

<b>include</b>	<a href="#">Program Statement</a>
----------------	-----------------------------------

Include another file in a program.

The `include` statement is used to include the contents of another file in a program file.

## Syntax

`include filename`

## Cross-references

See “[Multiple Program Files](#)” on page 107. See also [call \(p. 603\)](#).

<b>@isobject</b>	<a href="#">Support Function</a>
------------------	----------------------------------

Syntax:            `@isobject(str)`

Argument:        `string, str`

Return:            `integer`

Check for an object’s existence. Returns a “1” if the object exists in the current workfile, and a “0” if it does not exist.

<b>next</b>	<a href="#">Program Statement</a>
-------------	-----------------------------------

End of FOR loop. `next` marks the end of a FOR loop in a program.

## Syntax

`for [conditions of the FOR loop]`

`[commands]`

`next`

## Cross-references

See “[The FOR Loop](#)” beginning on page 101. See also, [exitloop \(p. 605\)](#), [for \(p. 606\)](#), [step \(p. 610\)](#).

open	<a href="#">Command</a>
------	-------------------------

Open a file. Opens a workfile, database, program file, or ASCII text file.

See [open \(p. 356\)](#).

output	<a href="#">Command</a>
--------	-------------------------

Redirects printer output or display estimation output.

See [output \(p. 361\)](#).

poff	<a href="#">Program Statement</a>
------	-----------------------------------

Turn off automatic printing in programs.

`poff` turns off automatic printing of all output. In programs, `poff` is used in conjunction with `pon` to control automatic printing; these commands have no effect in interactive use.

### Syntax

Command:      `poff`

### Cross-references

See “[Print Setup](#)” on page 919 for a discussion of printer control.

See also [pon \(p. 608\)](#).

pon	<a href="#">Program Statement</a>
-----	-----------------------------------

Turn on automatic printing in programs.

`pon` instructs EViews to send all statistical and data display output to the printer (or the redirected printer destination; see [output \(p. 361\)](#)). It is equivalent to including the “p” option in all commands that generate output. `pon` and `poff` only work in programs; they have no effect in interactive use.

### Syntax

Command:      `pon`

## Cross-references

See “[Print Setup](#)” on page 919 of the *User’s Guide* for a discussion of printer control.

See also [poff \(p. 608\)](#).

<b>program</b>	<a href="#">Command</a>
----------------	-------------------------

Create a program.

See [program \(p. 389\)](#).

<b>return</b>	<a href="#">Program Statement</a>
---------------	-----------------------------------

Exit subroutine.

The `return` statement forces an exit from a subroutine within a program. A common use of `return` is to exit from the subroutine if an unanticipated error has occurred.

## Syntax

```
if [condition] then
    return
endif
```

## Cross-references

See “[Subroutines](#)” beginning on page 108. See also [exitloop \(p. 605\)](#), [stop \(p. 611\)](#).

<b>run</b>	<a href="#">Command</a>
------------	-------------------------

Run a program. The `run` command executes a program. The program may be located in memory or stored in a program file on disk.

See [run \(p. 405\)](#).

<b>statusline</b>	<a href="#">Command</a>
-------------------	-------------------------

Send a text message to the EViews status line.

## Syntax

```
statusline string
```

### Example

```
for !i = 1 to 10
    statusline Iteration !i
next
```

step	Program Statement
------	-------------------

Step size of a FOR loop.

### Syntax

```
for !i=a to b step n
    [commands]
next
```

step may be used in a FOR loop to specify the size of the step in the looping variable. If no step is provided, for assumes a step of “+1”.

If a given step exceeds the end value b in the FOR loop specification, the contents of the loop will not be executed.

### Examples

```
for !j=5 to 1 step -1
    series x = nrnd*!j
next
```

repeatedly executes the commands in the loop with the control variable !J set to “5”, “4”, “3”, “2”, “1”.

```
for !j=0 to 10 step 3
    series z = z/!j
next
```

Loops the commands with the control variable !J set to “0”, “3”, “6”, and “9”.

You should take care when using non-integer values for the stepsize since round-off error may yield unanticipated results. For example:

```
for !j=0 to 1 step .01
    series w = !j
next
```

may stop before executing the loop for the value !J = 1 due to round-off error.

## Cross-references

See “[The FOR Loop](#)” beginning on page 101. See also [exitloop \(p. 605\)](#), [for \(p. 606\)](#), [next \(p. 607\)](#).

<code>stop</code>	<a href="#">Program Statement</a>
-------------------	-----------------------------------

Break out of program.

The `stop` command halts execution of a program. It has the same effect as hitting the F1 (break) key.

## Syntax

Command:      `stop`

## Cross-references

See also, [exitloop \(p. 605\)](#), [return \(p. 609\)](#).

<code>subroutine</code>	<a href="#">Program Statement</a>
-------------------------	-----------------------------------

Declare a subroutine within a program.

The `subroutine` statement marks the start of a subroutine.

## Syntax

`subroutine name(arguments)`

*commands*

`endsub`

## Cross-references

See “[Subroutines](#)” beginning on page 108. See also [endsub \(p. 604\)](#).

<code>@temppath</code>	<a href="#">Support Function</a>
------------------------	----------------------------------

Syntax:      `@temppath`

Return:      string

Returns a string containing the directory path for the EViews temporary files as specified in the global options **File Locations....** menu.

## Examples

If your currently executing copy of EViews puts temporary files in “D:\EVIEWS”, then:

```
%y = @tempPath
```

assigns a string of the form “D:\EVIEWS”.

## Cross-references

See also [cd \(p. 229\)](#) and [@evpath \(p. 605\)](#).

<b>then</b>	<a href="#">Program Statement</a>
-------------	-----------------------------------

Part of IF statement.

then marks the beginning of commands to be executed if the condition given in the IF statement is satisfied.

## Syntax

```
if [condition] then  
    [commands if condition true]  
endif
```

## Cross-references

See “[IF Statements](#)” on page 99. See also, [else \(p. 604\)](#), [endif \(p. 604\)](#), [if \(p. 606\)](#).

<b>@time</b>	<a href="#">Support Function</a>
--------------	----------------------------------

Syntax:            **@time**  
Return:            string

Returns a string containing the current time in “hh:mm” format.

## Examples

```
%y = @time
```

assigns a string of the form “15:35”.

## Cross-references

See also [@date \(p. 603\)](#).

to	Expression    Program Statement
----	---------------------------------

Upper limit of for loop OR lag range specifier.

`to` is required in the specification of a FOR loop to specify the upper limit of the control variable; see “[The FOR Loop](#)” on page 101.

When used as a lag specifier, `to` may be used to specify a range of lags to be used in estimation.

### Syntax

*Used in a FOR loop:*

```
for !i = n to m
```

*[commands]*

next

*Used as a Lag specifier:*

```
series_name(n to m)
```

### Examples

```
ls cs c gdp(0 to -12)
```

Runs an OLS regression of CS on a constant, and the variables GDP, GDP(-1), GDP(-2), ..., GDP(-11), GDP(-12).

### Cross-references

See “[The FOR Loop](#)” beginning on page 101. See also, [exitloop](#) (p. 605), [for](#) (p. 606), [next](#) (p. 607).

@toc	Support Function
------	------------------

Syntax:              [@toc](#)

Return:              integer

Compute elapsed time (since timer reset) in seconds.

### Examples

```
tic
```

*[some commands]*

```
!elapsed = @toc
```

resets the timer, executes commands, and saves the elapsed time in the control variable !ELAPSED.

### Cross-references

See also [tic \(p. 482\)](#) and [toc \(p. 483\)](#).

<b>wend</b>	<a href="#">Program Statement</a>
-------------	-----------------------------------

End of WHILE clause.

wend marks the end of a set of program commands that are executed under the control of a WHILE statement.

### Syntax

```
while [condition]
      [commands while condition true]
wend
```

### Cross-references

See “The WHILE Loop” on page 104. See also [while \(p. 614\)](#).

<b>while</b>	<a href="#">Program Statement</a>
--------------	-----------------------------------

Conditional control statement. The while statement marks the beginning of a WHILE loop.

The commands between the `while` keyword and the `wend` keyword will be executed repeatedly until the condition in the `while` statement is false.

### Syntax

```
while [condition]
      [commands while condition true]
wend
```

### Cross-references

See “The WHILE Loop” on page 104. See also [wend \(p. 614\)](#).

# Index

---

## Symbols

- ! (exclamation) control variable [88](#)
- % (percent sign)
  - program arguments [96, 97](#)
  - string variable [89](#)
- \* (asterisk) multiplication [35](#)
- + (plus)
  - addition [35](#)
- / (slash) division [36](#)
- @abs [545](#)
- @all [450](#)
- @beta [551](#)
- @betainc [551](#)
- @betaincder [551](#)
- @betaincinv [551](#)
- @betalog [551](#)
- @binom [551](#)
- @binomlog [552](#)
- @ceiling [545](#)
- @cellid [563](#)
- @cholesky [581](#)
- @cloglog [552](#)
- @columnextract [582](#)
- @columns [582](#)
- @cond [582](#)
- @convert [583](#)
- @cor [548, 584](#)
- @cov [548, 584](#)
- @crossid [563](#)
- @date [146, 560, 603](#)
- @dateadd [144, 565](#)
- @datediff [144, 566](#)
- @datefloor [144, 566](#)
- @datepart [145, 567](#)
- @datestr [139, 567](#)
- @dateval [123, 138, 568](#)
- @day [146, 561](#)
- @det [585](#)
- @digamma [552](#)
- @dtoc [569](#)
- @eigenvalues [585](#)
- @eigenvectors [586](#)
- @elem [559](#)
- @enddate [560](#)
- @eqna [122, 545, 569](#)
- @erf [552](#)
- @erfc [552](#)
- @errorcount [605](#)
- @evpath [605](#)
- @exp [545](#)
- @expand [535](#)
- @explode [586](#)
- @fact [545](#)
- @factlog [545](#)
- @filledmatrix [586](#)
- @filledrowvector [587](#)
- @filledsym [587](#)
- @filledvector [587](#)
- @first [450](#)
- @firstmax [450](#)
- @firstmin [450](#)
- @floor [545](#)
- @-functions
  - by-group statistics [549](#)
  - descriptive statistics [547](#)
  - integrals and other special functions [550](#)
  - mathematical functions [545](#)
  - special p-value functions [557](#)
  - statistical distribution functions [554](#)
  - time series functions [546](#)
  - trigonometric functions [553](#)
- @gamma [552](#)
- @gammader [552](#)
- @gammainc [552](#)
- @gammaincder [553](#)
- @gammaincinv [553](#)
- @gammalog [553](#)
- @getmaindiagonal [588](#)
- @groupid [550](#)
- @identity [588](#)
- @implode [588](#)

@inner 548, 589  
@insert 124, 570  
@instr 122, 570  
@inv 545  
@inverse 590  
@isempty 122, 571  
@isna 545  
@isobject 607  
@isperiod 561  
@issingular 590  
@kronecker 590  
@kurtsby 550  
@last 450  
@lastmax 450  
@lastmin 450  
@left 123, 571, 576  
@length 122, 572  
@log 545  
@log10 545  
@logit 553  
@logx 546  
@lower 125, 572  
@ltrim 124, 572  
@makedate 140, 573  
@makediagonal 591  
@max 548  
@maxsby 549  
@meansby 549  
@median 548  
@mediansby 549  
@mid 124, 573  
@min 548  
@minsby 549  
@mod 545  
@month 146, 561  
@movav 546  
@movsum 546  
@nan 546  
@nasby 550  
@neqna 123, 545, 574  
@ngroups 550  
@norm 593  
@now 145, 574  
@obs 548  
@obsby 549  
@obsid 563  
@obsnum 559  
@obsrange 560  
@obssmpl 560  
@otod 126, 575  
@outer 593  
@pca 547  
@pch 547  
@pcha 547  
@pchy 547  
@pcy 547  
@permute 594  
@psi 552  
@quantile 549  
@quantilesby 550  
@quarter 146, 561  
@rank 594  
@recode 546  
@replace 124, 575  
@resample 595  
@right 123  
@round 546  
@rowextract 595  
@rows 596  
@rtrim 124, 576  
@seas 561  
@skewsby 550  
@solvesystem 596  
@sqrt 546  
@stdev 549  
@stdevsby 549  
@str 125, 576  
@strdate 126, 146, 561, 577  
@strlen 577  
@strnow 126, 577  
@subextract 598  
@sum 549  
@sumsby 549  
@sumsq 549  
@sumsqsbys 549  
@svd 599  
@temppath 611  
@time 612

@toc 613  
 @trace 599  
 @transpose 600  
 @trend 562, 564  
 @trend in panel 564  
 @trendc 562, 564  
 @trigamma 553  
 @trim 124, 578  
 @unitvector 600  
 @upper 125, 578  
 @val 122, 578  
 @var 549  
 @varsby 549  
 @vec 600  
 @vech 601  
 @weekday 146, 561  
 @year 146, 561  
 – (continuation character) 84  
 – (dash)  
     negation 34  
     subtraction 35

## Numerics

2sls (Two-Stage Least Squares) 487  
 3sls (Three-stage least squares) 194

## A

abs 545  
 add 196  
 Add factor  
     initialization 197  
 Add text to graph 199  
 addassign 196  
 addinit 197  
 Addition operator (+) 35  
 addtext 199  
 align 200  
 Align multiple graphs 200  
 Alpha 152  
     data members 152  
     declare 201  
     element functions 152  
     procs 152  
     views 152  
 alpha 201

And 544  
 Anderson-Darling test 272  
 Andrews test 479  
 append 202  
 Append specification line 202  
 AR  
     autoregressive error 535  
     inverse roots of polynomial 212  
     seasonal 541  
 ar 535  
 Arc cosine 553  
 Arc sine 553  
 Arc tangent 553  
 ARCH  
     residual LM test for ARCH 206  
     see also GARCH. 203  
 archtest 206  
 area 207  
 Area graph 207  
 Arguments  
     in programs 97  
     in subroutines 109  
 arlm 209  
 ARMA  
     structure view 210  
 arroots 212  
 ASCII file  
     open as workfile 504  
 Assign values to matrix objects 24  
     by element 24  
     converting series or group 30  
     copy 27  
     copy submatrix 29  
     fill procedure 25  
 Augmented Dickey-Fuller test 494  
 auto 212  
 Autocorrelation  
     compute and display 246  
     multivariate VAR residual test 390  
 Autoregressive error. See AR.  
 Autoregressive conditional heteroskedasticity  
     See ARCH and GARCH.  
 Autowrap text 83  
 Auxiliary commands 10  
     summary 16  
 Axis

- rename label 354  
set axis characteristics 213  
  set scaling in graph 410  
axis 213
- B**
- Background color 429  
Band-pass filter 223  
Bar graph 215  
Batch mode  
  See Program.  
Batch mode for program 86  
Baxter-King band-pass filter 223  
BDS test 217  
bdstest 217  
Beta  
  distribution 554  
  integral 551  
  integral, logarithm of 551  
Binary  
  dependent variables 217  
  model prediction table 387  
binary 217  
Binary file 504  
Binomial  
  coefficient function 551  
  coefficients, logarithm of 552  
  distribution 555  
block 219  
Bootstrap rows of matrix 595  
Boxplot 219, 221  
  customize 226  
  customize individual elements 421  
bplabel 226  
Breusch-Godfrey test  
  See also Serial correlation.  
By-group statistics 549
- C**
- call 603  
Call subroutine 110  
Causality test 228  
cause 228  
ccopy 229  
cd 229
- cdfplot 230  
Cell  
  background color 429  
  borders 440  
  display format 432  
  font selection 431  
  height 436  
  indentation 437  
  justification 438  
  merging multiple 441  
  set text color 443  
  width 444  
censored 233  
Censored dependent variable models 233  
cfetch 234  
Change default directory 229, 235  
checkderivs 235  
Chi-square distribution 555  
Cholesky factor  
  function to compute 581  
chow 236  
Chow test 236  
Christiano-Fitzgerald band-pass filter 223  
clabel 237  
cleartext 237  
Close  
  EViews application 279  
  window 238  
close 238  
Coef 153  
  data members 154  
  declare 238  
  fill values 282  
  procs 154  
  views 153  
coef 238  
cofcov 239  
Coefficient  
  covariance matrix of estimates 239  
  See Coef.  
  update default coef vector 493  
coint 240  
Cointegration  
  make cointegrating relations from VEC 334  
  test 240  
colplace 581

---

Column  
 extract from matrix [582](#)  
 number of columns in matrix [582](#)  
 place in matrix [581](#)  
 stack matrix [600](#)  
 stack matrix (lower triangle) [601](#)  
 width [50, 423, 444](#)

Commands  
 auxiliary [10, 16](#)  
 basic command summary [16](#)  
 batch mode [2](#)  
 execute without opening window [267](#)  
 interactive mode [1](#)  
 object assignment [9](#)  
 object command [6](#)  
 object declaration [5, 9](#)  
 save record of [2](#)  
 window [1](#)

comment [242](#)

Comments in tables [242](#)

Comparison operators [36](#)

Complementary log-log function [552](#)

Condition number of matrix [582](#)

Conditional standard deviation  
 display graph of [295](#)

Conditional variance  
 make series from ARCH [336](#)

Confidence ellipse [231](#)

Container [11](#)  
 database [12](#)  
 workfile [11](#)

Continuation character in programs [84](#)

control [243](#)

Control variable [88](#)  
 as replacement variable [92](#)  
 in while statement [105](#)  
 save value to scalar [88](#)

Convert  
 date to observation number [569](#)  
 matrix object to series or group [592](#)  
 matrix objects [30, 42, 583](#)  
 matrix to sym [588](#)  
 observation number to date [126, 575](#)  
 scalar to string [576](#)  
 series or group to matrix (drop NAs) [597](#)  
 series or group to matrix (keep NAs) [597](#)  
 string to scalar [90, 122, 578](#)

sym to matrix [586](#)

Coordinates  
 for legend in graph [319](#)

Copy  
 database [255](#)  
 objects [13, 244](#)  
 workfile page [367](#)

cor [246](#)

correl [246](#)

Correlation [548, 584](#)  
 cross [251](#)  
 matrix [246](#)

Correlogram [246, 251](#)  
 squared residuals [247](#)

correlog [247](#)

Cosine [554](#)

count [248](#)

Count models [248](#)

cov [250](#)

Covariance [548, 584](#)  
 matrix [250](#)

Cramer-von Mises test [272](#)

Create  
 database [255](#)  
 workfile [503](#)  
 workfile page [369](#)

cross [251](#)

Cross correlations [251](#)

Cross product [593](#)

Cross section member  
 add to pool [196](#)  
 define list of [262](#)

Current date function [603](#)

Current time function [612](#)

CUSUM test [401](#)  
 of squares [401](#)

**D**

Data  
 enter from keyboard [252](#)

data [252](#)

Data members  
 alpha series [152](#)  
 coef [154](#)  
 equation [157](#)  
 group [162](#)

matrix 167  
pool 171  
rowvector 173  
series 176  
sspace 178  
sym 182  
system 183  
table 186  
var 189  
vector 191

Database  
copy 13, 255  
create 12, 255  
delete 13, 256  
fetch 279  
Haver Analytics 305, 306, 309  
open 13  
open existing 257  
open or create 254  
pack 258  
rebuild 258  
rename 13, 259  
repair 260  
store object in 465

Date  
arithmetic 565, 566, 573  
current date 574  
extract portion of date 567

Dated data table 270, 339

datelabel 252

Dates  
arithmetic 144  
convert from observation number 126, 575  
convert string to date 568  
convert to observation number 569  
converting from numbers 140  
current as string 577  
current date and time 145  
current date as a string 126  
date arithmetic 144  
date associated with observation 146  
date numbers 130  
date operators 143  
date strings 128  
extract portion of 145  
format strings 130  
rounding 144  
string representation 568

string representations 123, 128, 138  
strings representations 567  
two-digit years 135, 148  
workfile dates as strings 126, 146, 561, 577

dates 254  
db 254  
dbcopy 255  
dbcreate 255  
dbdelete 256  
dbopen 257  
dbpack 258  
dbrebuild 258  
dbrename 259  
dbrepair 260

Declare  
matrix 23  
object 5, 9  
decomp 260  
define 262

Delete  
database 256  
object 15  
objects or pool identifiers 263  
workfile page 371

delete 263

Derivatives  
examine derivs of specification 264  
make series or group containing 335

derivs 264

describe 265

Descriptive statistics 462, 549  
@-functions 547, 549  
by category of dependent variable 350  
by classification 457  
make series 347  
matrix functions 45  
pool 265

Determinant 585

Diagonal matrix 591

Dickey-Fuller test 494

Difference 546

Directory  
change working 229, 235  
EViews executable 605

Display  
action 6

---

and print 8  
 objects 445  
 spreadsheet tables 444  
**displayname** 266  
**Distribution function**  
 empirical cumulative, survivor and quantiles 230  
 Quantile-quantile plot 389  
**Division operator (/)** 36  
**do** 267  
 Double exponential smoothing 447  
**draw** 267  
 Draw lines in graph 267  
**DRI database**  
 convert to EViews database 269  
 copy from 229  
 fetch series 234  
 read series description 237  
**driconvert** 269  
**Drop**  
 group series or cross-section from pool definition 270  
**drop** 270  
**dtable** 270  
 Dummy variables  
 automatic creation 535  
 Durbin's h 108  
 Dynamic forecast 287

**E**

**ec** 271  
**edftest** 272  
 Eigenvalues 585  
 Eigenvectors 586  
 Elapsed time 483, 613  
**Element**  
 assign in matrix 24  
 matrix functions 45  
**else** 604  
 Else clause in if statement 100, 604  
 Empirical distribution functions 230  
 Empirical distribution test 272  
 Empty string 571  
**endif** 604  
**endog** 274

Endogenous variables  
 make series or group 335  
 of specification 274  
**endsub** 604  
**eqs** 275  
 equals comparison 36  
**Equation** 155  
 data members 157  
 declare 275  
 methods 155  
 procs 156  
 views 155  
**equation** 275  
**errbar** 276  
 Error bar graph 276  
 Error correction model  
 See VEC and VAR.Vector error correction model  
**Error function** 552  
 complementary 552  
**Errors**  
 count in programs 605  
 handling in programs 106  
**Estimation methods**  
 (single) equation 155  
 2sls 487  
 3sls 194  
 least squares 329  
 nonlinear least squares 329  
 system (of equations) 182  
 VAR 188  
 Euler's constant 552  
**Excel file** 504  
 export data to file 517  
 importing data into workfile 391  
**exclude** 278  
 Exclude variables from model solution 278  
**Execute program** 84  
 abort 86  
 quiet mode 85  
 verbose mode 85  
 with arguments 96, 97  
**Exit**  
 from EViews 279  
 loop 107, 605  
 subroutine 609  
**exit** 279  
**exitloop** 605

exp 545  
expand 279  
Expand workfile 279  
Exponential  
    distribution 555  
    function 545  
Exponential smoothing 447  
Export  
    matrix 41  
    workfile data to file 517  
    workfile page 373  
Extract  
    row vector 595  
    submatrix from matrix 598  
Extreme value distribution 555

**F**

Factorial 545  
F-distribution 555  
fetch 279  
Fetch object 16, 279  
Files  
    open program or text file 356  
    temporary location 611  
Fill  
    matrix 25  
    object 282  
    row vector 587  
    symmetric matrix 587  
    vector 587  
fill 282  
Filled  
    matrix 586  
fiml 284  
fit 285  
Font selection 431  
for 606  
For loop  
    accessing elements of a series 102  
    accessing elements of a vector 102  
    changing samples within 101  
    define using control variables 101  
    define using scalars 103  
    define using string variables 103  
    exit loop 107  
    mark end 607

nesting 102  
roundoff error in 610  
start loop 606  
step size 610  
upper limit 613

Forecast  
    dynamic (multi-period) 287  
    static (one-period ahead) 285  
forecast 287  
Format number 49  
Formula series 293  
Freeze  
    table 47  
freeze 290  
Freeze view 290  
freq 291  
Frequency conversion 244, 279, 323  
    default method for series 424  
    using links 324  
Frequency table  
    one-way 291  
frml 293  
Full information maximum likelihood 284

**G**

Gamma  
    distribution 555  
Gamma function 552  
    derivative 552  
    incomplete 552  
    incomplete derivative 553  
    incomplete inverse 553  
    logarithm 553  
    second derivative 553  
GARCH  
    display conditional standard deviation 295  
    estimate model 203  
    generate conditional variance series 336  
garch 295  
Gauss file 504  
Gaussian distribution 556  
Generalized autoregressive conditional heteroskedasticity  
    See ARCH and GARCH.  
Generalized error distribution 555  
Generate series

for pool 295  
**genr** 295  
 See also series.  
**Global**  
 subroutine 111  
 variable 111  
**GMM**  
 estimate by 297  
**gmm** 297  
**Gompit models** 217  
 Goodness of fit (for binary models) 479  
**Gradients**  
 display 302  
 saving in series 337  
**grads** 302  
 Granger causality test 228, 478  
**Graph** 159  
 align multiple graphs 200  
 area graph 207  
 axis labeling 252, 254  
 bar graph 215  
 change legend or axis name 354  
 commands 159  
 create 59  
 create using freeze 290  
 declaring 303  
 drawing lines and shaded areas 267  
 error bar 276  
 high-low-open-close 307  
 legend appearance and placement 319  
 line graph 320  
 merge multiple 351  
 merging graphs 303  
 options for bottom axis in boxplotst 226  
 options for individual elements 426  
 options for individual elements of a boxplot 421  
 pie graph 384  
 place text in 199  
 procs 160  
 save to disk 407  
 scatterplot graph 412  
 set axis characteristics 213  
 set axis scale 410  
 set options 358  
 spike graph 455  
 templates 473  
 views 159

XY graph 528  
 XY line graph 530  
 XY pairs graph 532  
**graph** 303  
 greater than comparison 36  
 greater than or equal to comparison 36  
**Group** 161  
 add series 196  
 convert to matrix 463, 464, 592  
 convert to matrix (with NAs) 597  
 data members 162  
 declare 304  
 procs 162  
 views 161  
**group** 304

**H**

Haver Analytics Database  
 convert to EViews database 305  
 display series description 309  
 fetch series from 306  
**hconvert** 305  
 Heteroskedasticity test (White) 514  
**hfetchn** 306  
 High-Low-(Open-Close) graph 307  
**hilo** 307  
**hist** 308  
 Histogram 308  
**hlabel** 309  
 Hodrick-Prescott filter 310  
 Holt-Winters 447  
 Hosmer-Lemeshow test 479  
**hpfn** 310  
**HTML**  
 open page as workfile 504  
 save table as 57

**I**

Identity matrix 588  
 extract column 600  
**if** 606  
**If statement** 99  
 else clause 100, 604  
 end of condition 604  
 start of condition 606  
 then 612

Import data  
    matrix 40

Import data from file 391

impulse 311

Impulse response function 311

Include  
    file in a program file 607  
    program file 107

include 607

Incomplete beta 551  
    derivative 551  
    integral 551  
    inverse 551

Incomplete gamma 552  
    derivative 553  
    inverse 553

Independence test 217

Initial parameter values 383

Inner product 548, 589

Insertion point in command line 1

Integer random number 402

Interactive mode 1

Inverse of matrix 590

**J**

Jarque-Bera  
    multivariate normality test 313

jbera 313

Johansen cointegration test 240

**K**

Kalman filter 336

kdensity 315

kerfit 316

Kernel  
    bivariate regression 316  
    density 315

Keyboard focus  
    defaults 2

Kolmogorov-Smirnov test 272

KPSS unit root test 494

Kronecker product 590

**L**

label 317

Label object 266, 317

Label values 349

Lag  
    specify as range 613  
    VAR lag order selection 318

Lag exclusion test 480

laglen 318

Lagrange multiplier  
    test for ARCH in residuals 206

Landscape printing 8

Laplace distribution 556

Least squares estimation 329

Legend  
    appearance and placement 319  
    rename 354

legend 319

Length of string 572

less than comparison 36

less than or equal to comparison 36

Lilliefors test 272

line 320

Line drawing 267

Line graph 320

linefit 322

Link 163  
    declare 323  
    procs 164  
    specification 324

link 323

Link object  
    convert to ordinary series 491

linkto 324

load 327

Local  
    subroutine 113  
    variable 111

log  
    arbitrary base 546  
    base 10 545  
    natural 545

Log difference 546

Logistic  
    logit function 553

Logistic distribution 556

logit 328

Logit models 217

---

**Log** [164](#)  
 append specification line [202](#)  
 check user-supplied derivatives [235](#)  
 data members [165](#)  
 declare [328](#)  
 method [164](#)  
 procs [164](#)  
 statements [165](#)  
 views [164](#)  
**logl** [328](#)  
 Log-normal distribution [556](#)  
**Loop**  
 exit loop [107, 605](#)  
 for (control variables) [101](#)  
 for (scalars) [103](#)  
 for (string variables) [103](#)  
 nest [102](#)  
 over matrix elements [41, 102](#)  
 while [104](#)  
**Lotus file**  
 export data to file [517](#)  
**Lowercase** [572](#)  
**ls** [329](#)  
**M**  
**MA**  
 seasonal [542](#)  
**ma** [537](#)  
 Make model object [341](#)  
 makecoint [334](#)  
 makederivs [335](#)  
 makeendog [335](#)  
 makefilter [336](#)  
 makegarch [336](#)  
 makegrads [337](#)  
 makegraph [338](#)  
 makegroup [339](#)  
 makelimits [341](#)  
 makemodel [341](#)  
 makereg [342](#)  
 makeresids [342](#)  
 makesignals [344](#)  
 makestates [345](#)  
 makestats [347](#)  
 makesystem [348](#)  
**map** [349](#)  
**Match merge** [323, 324](#)  
**Mathematical functions** [545](#)  
**matplace** [592](#)  
**Matrix**  
 assign values [24](#)  
 condition number [582](#)  
 convert to other matrix objects [42](#)  
 convert to series or group [30](#)  
 copy [27](#)  
 copy submatrix [29](#)  
 data members [167](#)  
 declare [23, 350](#)  
 export data [41](#)  
 fill values [282](#)  
 filled [586](#)  
 import data [40](#)  
 main diagonal [588](#)  
 objects [23](#)  
 permute rows of [594](#)  
 place submatrix [592](#)  
 procs [167](#)  
 resample rows from [595](#)  
 singular value decomposition [599](#)  
 stack columns [600](#)  
 stack lower triangular columns [601](#)  
 views [40, 166](#)  
**matrix** [350](#)  
**Matrix commands and functions**  
 commands [38](#)  
 descriptive statistics [37, 45](#)  
 difference [38](#)  
 element [37, 45](#)  
 functions [38](#)  
 matrix algebra [37, 45](#)  
 missing values [39](#)  
 utility [37, 44](#)  
**Matrix operators**  
 addition (+) [35](#)  
 and loop operators [41](#)  
 comparison operators [36](#)  
 division (/) [36](#)  
 multiplication (\*) [35](#)  
 negation (-) [34](#)  
 order of evaluation [34](#)  
 subtraction (-) [35](#)  
**Maximum** [548](#)  
**Maximum likelihood estimation** [352](#)  
**Logl** [164](#)

- state space 177
- Mean 548  
equality test 475, 476
- means 350
- Median 548  
equality test 475, 476
- Merge  
graph objects 303, 351  
into model 351  
using links 324
- merge 351
- Messages  
model solution 354  
suppress during program execution 85
- Minimum 548
- Missing values 39, 545  
code for missing 538  
mathematical functions 545  
recoding 546
- ml 352
- Model 168  
append specification line 202  
break all model links 491  
declare 353  
equation view 275  
merge into 351  
procs 168  
scenarios 415  
update specification 492  
variable view 501  
views 168
- model 353
- Models  
add factor assignment and removal 196  
add factor initialization 197  
block structure 219  
exclude variables from solution 278  
make from estimation object 341  
make graph of model series 338  
make group of model series 339  
options for solving 452  
overrides in model solution 363  
solution messages 354  
solve 451  
solve control to match target 243  
text representation 482  
trace iteration history 484
- modulus 545
- Moving average 537, 546
- Moving sum 546
- msg 354
- mtos 592
- Multiplication operator (\*) 35
- N**
- NA  
recode 546
- na 538
- name 354
- Nearest neighbor regression 355
- Negation operator (-) 34
- Negative binomial  
count model 248  
distribution 556
- next 607
- nfit 355
- Nonlinear least squares 329
- Norm of a matrix 593
- Normal distribution 556  
random number 539
- not equal to comparison 36
- nrnd 539
- Number  
evaluate a string 578  
formatting in tables 49
- Number of observations 548
- Numbers  
converting from strings 122
- O**
- Object  
assignment 9  
command 6  
containers 11  
copy 13, 244  
create using freeze 290  
declaration 5, 9  
delete 15  
fetch 16  
fetch from database or databank 279  
label 317  
print 387  
rename 15, 394

---

save 16  
 store 16, 465  
 test for existence 607  
**ODBC** 504  
 OLS (ordinary least squares) 329  
 Omitted variables test 474  
 One-way frequency table 291  
**Open**  
 database 257  
 text or program files 356  
**open** 356  
 Open foreign source data 504  
**Operator** 544  
 relational 119  
**options** 358  
**Or** 544  
**ordered** 360  
 Ordered dependent variable  
     estimating models with 360  
     make vector of limit points from equation 341  
**Outer product** 593  
**Output**  
     display estimation results 361  
     printing 8  
     redirection 361  
**output** 361  
**Output redirection** 608  
**override** 363  
 Override variables in model solution 363

**P**

**Page**  
     contract 366  
     copy from 367  
     create new 369  
     define structure 378  
     delete page 371  
     rename 372  
     resize 378  
     save or export 373  
     set active 374  
     stack 375  
     subset from 366  
**pageappend** 364  
**pagecontract** 366  
**pagecopy** 367

pagecreate 369  
 pagedelete 371  
 pageload 371  
 pagerename 372  
 pagesave 373  
 pageselect 374  
 pagestack 375  
 pagestruct 378  
 pageunstack 381  
**Panel**  
     unit root test 494  
**Panel data**  
     cell identifier 563  
     group identifier 563  
     time trend 564  
     trends 563  
     within-group identifier 563  
**param** 383  
**Parameters** 383  
**Pareto distribution** 556  
**Partial autocorrelation** 246  
**Partial correlation** 246  
**pdl** 540  
**PDL (polynomial distributed lag)** 540  
**Percent change**  
     annualized 547  
**Percentage change** 547  
**Percentage change (one year)** 547  
**Period dummy variable** 561  
**Permute rows of matrix** 594  
**Phillips-Perron test** 494  
**Pi** 553  
**pie** 384  
**Pie graph** 384  
**plot** 386  
**poff** 608  
**Poisson**  
     count model 248  
     distribution 556  
**Polynomial distributed lags** 540  
**pon** 608  
**Pool** 169  
     add cross section member 196  
     data members 171  
     declare 386

- delete identifiers 263  
generate series using identifiers 295  
make group of pool series 339  
members 169  
procs 170  
views 170  
pool 386  
Portrait (print orientation) 8  
Power (raise to) 544  
Precedence of evaluation 34  
predict 387  
Prediction table 387  
Presentation table 270  
Principal components 383  
Print  
    and display 8  
    automatic printing 608  
    landscape 8  
    portrait 8  
    turn off in program 608  
print 387  
probit 388  
Probit models 217  
Program 83  
    abort 86  
    arguments 96, 97  
    call subroutine 110, 603  
    counting execution errors 605  
    create 83  
    create new file 389  
    entering text 83  
    exit loop 107  
    if statement 99  
    include file 107, 607  
    line continuation character 84  
    modes 96  
    open 84  
    place subroutine 109  
    quiet mode 85  
    run 405  
    running 84  
    save 84  
    stop 106  
    stop execution 611  
    verbose mode 85  
    version 4 compatibility 96  
program 389
- P-value functions 557
- Q**
- QQ-plot  
    See Distribution function.  
qqplot 389  
Q-statistic 246  
qstats 390  
Quantile function 549  
Quantile-Quantile plot. See Distribution function  
Quiet mode 85
- R**
- Random number  
    generator for normal 539  
    integer 402  
    seed 403  
    uniform 541  
range 391  
Rank 594  
Read  
    data from foreign file 391  
read 391  
Recode values 546  
Recursive least squares 400  
    CUSUM 401  
    CUSUM of squares 401  
Redirect output to file 8, 361  
Redundant variables test 477  
Regressors  
    make group containing 342  
Rename  
    database 259  
    object 15, 394  
rename 394  
Repair database 260  
Replacement variable 91  
    naming objects 92  
Representations view 395  
Resample  
    observations 395  
    rows from matrix 595  
resample 395  
reset 397  
RESET test 397

---

Reset timer [482](#)  
 residcor [398](#)  
 residcov [398](#)  
 resids [399](#)  
**Residuals**  
     correlation matrix of [398](#)  
     covariance matrix of [398](#)  
     display of [399](#)  
     make series or group containing [342](#)  
 resize page [378](#)  
 Restricted VAR text [237](#)  
**Results**  
     display or retrieve [400](#)  
 results [400](#)  
 return [609](#)  
 Rich Text Format [57](#)  
 rls [400](#)  
 rnd [541](#)  
 rndint [402](#)  
 rndseed [403](#)  
 Roots of the AR polynomial [212](#)  
 Roundoff error in for loops [610](#)  
**Row**  
     height [50](#)  
     numbers [596](#)  
     place in matrix [596](#)  
 rowplace [596](#)  
**Rowvector** [172](#)  
     data members [173](#)  
     declare [404](#)  
     extract from matrix [595](#)  
     filled rowvector function [587](#)  
     procs [173](#)  
     views [172](#)  
 rowvector [404](#)  
**Run**  
     application in batch mode [86](#)  
 run [405](#)  
 Run batch program [405](#)  
 Run program  
     multiple files [107](#)  
**S**  
 Sample  
     change using for loop [101](#)  
     declare [406](#)  
         number of observations in [560](#)  
         procs [174](#)  
         set current [449](#)  
         set sample specification in [420](#)  
     sample [406](#)  
     sar [541](#)  
     SAS file [504](#)  
     Save  
         commands in file [2](#)  
         objects to disk [16](#)  
     save [407](#)  
     Scalar [174](#)  
         declare [410](#)  
     scalar [410](#)  
     scale [410](#)  
     scat [412](#)  
     scatmat [414](#)  
     Scatter diagrams  
         matrix of [414](#)  
     Scatterplot [412](#)  
         with kernel fit [316](#)  
         with nearest neighbor fit [355](#)  
         with regression line fit [322](#)  
     scenario [415](#)  
     seas [417](#)  
     Seasonal adjustment  
         moving average [417](#)  
         Tramo/Seats [484](#)  
         X11 [520](#)  
         X12 [522](#)  
     Seasonal autoregressive error [541](#)  
     Seasonal dummy variable [561](#)  
     Seasonal line graphs [418](#)  
     seasplot [418](#)  
     Second moment matrix [589](#)  
     Seed random number generator [403](#)  
     Seemingly unrelated regression. See SUR  
     Sequential LR tests [107](#)  
     Serial correlation  
         Breusch-Godfrey LM test [212](#)  
         multivariate VAR LM test [209](#)  
     Series [175](#)  
         auto-updating [293](#)  
         convert to matrix [463, 464, 592, 597](#)  
         convert to matrix (with NAs) [597](#)  
         data members [176](#)

declare 418  
element function 176  
extract observation 559  
fill values 282  
formula 293  
frequency conversion default method 424  
procs 176  
views 175  
series 418  
set 420  
Set graph date labeling formats 252, 254  
setbpelem 421  
setcell 422  
setcolwidth 423  
setconvert 424  
setelem 426  
setfillcolor 429  
setfont 431  
setformat 432  
setheight 436  
setindent 437  
setjust 438  
setline 439  
setlines 440  
setmerge 441  
settextcolor 443  
setwidth 444  
Shade region of graph 267  
sheet 444  
show 445  
Show object view 445  
Signal variables  
    display graphs 446  
    saving 344  
signalgraph 446  
Sine 554  
Singular matrix  
    test for 590  
Singular value decomposition 599  
sma 542  
smooth 447  
Smoothing  
    exponential smooth series 447  
    signal series 344  
    state series 345  
smpl 449  
Solve  
    linear system 596  
    simultaneous equations model 451  
solve 451  
Solve. See Models.  
solveopt 452  
sort 453  
Sort workfile 453  
spec 454  
Specification 395  
Specification view 454  
spike 455  
Spike graph 455  
Spreadsheet view 444  
SPSS file 504  
sqrt 546  
Sspace  
    append specification line 202  
    data members 178  
    declare 457  
    display signal graphs 446  
    make Kalman filter from 336  
    method 177  
    procs 178  
    specification display 467  
    state graphs 459  
    views 177  
sspace 457  
Stability test 236  
Stack matrix by column 600  
    lower triangle 601  
Stack workfile page 375  
Standard deviation 549  
Starting values 383  
Stata file 504  
statby 457  
State space  
    specification 467  
State variables  
    display graphs of 459  
    final one-step ahead predictions 460  
    initial values 461  
    smoothed series 345  
statefinal 460  
stategraphs 459

---

stateinit [461](#)  
 Static forecast [285](#)  
 Statistical distribution functions [554](#)  
 Statistics [265](#)  
     compute for subgroups [457](#)  
 stats [462](#)  
 Status line [463](#)  
 statusline [463, 609](#)  
 step [610](#)  
 stom [463, 597](#)  
 stomna [464, 597](#)  
 stop [611](#)  
 Stop program execution [106, 611](#)  
 store [465](#)  
 Store object in database or databank [16, 465](#)  
 String [89](#)  
     assign to table cell [48](#)  
     change case [125](#)  
     concatenation [118](#)  
     convert date string to observation [126](#)  
     convert date value into [567](#)  
     convert date value to string [139](#)  
     convert from a number [576](#)  
     convert into date value [568](#)  
     convert number to string [125](#)  
     convert string to number [122](#)  
     convert to a scalar [578](#)  
     converting string to date number [123, 138](#)  
     converting to numbers [122](#)  
     current date as a string [126](#)  
     extract portion [123, 124](#)  
     find substring [122](#)  
     find substring in [570](#)  
     functions [121](#)  
     insert string into [570](#)  
     insert string into another string [124](#)  
     length [122, 572](#)  
     length of [577](#)  
     lowercase [572](#)  
     missing value [117, 120](#)  
     null string [117](#)  
     operators [118](#)  
     relational comparison [119](#)  
     relational comparisons with empty strings [120](#)  
     replace portion of string with new string [124](#)  
     replace substring in [575](#)  
     substring [571, 576](#)  
         test for blank [571](#)  
         test for blank string [122](#)  
         test for equality [569](#)  
         test for inequality [574](#)  
         test two strings for equality [122](#)  
         test two strings for inequality [123](#)  
         trim spaces [124](#)  
         trim spaces from ends [572, 576, 578](#)  
         uppercase [578](#)  
 String variable [89](#)  
     as replacement variable [91](#)  
     convert to a scalar [90](#)  
     in for loop [103](#)  
     program arguments [96, 97](#)  
 structure [467](#)  
 Structure workfile page [378](#)  
 Subroutine [108](#)  
     arguments [109](#)  
     call [110, 603](#)  
     declare [611](#)  
     define [108](#)  
     global [111](#)  
     local [113](#)  
     mark end [604](#)  
     placement [109](#)  
     return from [108, 609](#)  
 subroutine [611](#)  
 Substring [570, 573](#)  
 Subtraction operator (-) [35](#)  
 Sum [549](#)  
 Sum of squares [549](#)  
 SUR  
     estimating [468](#)  
 sur [468](#)  
 svar [469](#)  
 Sym [180](#)  
     create from lower triangle of square matrix [588](#)  
     create from scalar function [587](#)  
     create square matrix from [586](#)  
     data members [182](#)  
     declare [471](#)  
     procs [181](#)  
     stack columns [601](#)  
     views [181](#)  
 sym [471](#)  
 Symmetric matrix  
     See Sym.

- System 182  
3SLS 194  
append specification line 202  
create from pool or var 348  
data members 183  
declare 472  
FIML estimation 284  
methods 182  
procs 183  
views 183  
weighted least squares 515  
system 472
- T**
- Table 185  
add comment to cell in 242  
assigning numbers to cells 48  
assigning strings to cells 48  
assignment with formatting 49  
background color 54  
borders and line characteristics 440  
borders and lines 54  
cell format 52  
column width 50  
column widths 444  
commands 186  
create using freeze 290  
creating 47  
creating by freezing a view 47  
data members 186  
decimal format code 49  
declare 48, 472  
display format for cells 432  
display justification for cells 438  
fill (background) color for cells 429  
font characteristics 54  
font for text in cells 431  
horizontal line 439  
indentation for cells 437  
justification and indentation 53  
justification code 49  
merge cells 441  
procs 185  
row height 50  
row heights 436  
save to disk 407  
set and format cell contents 422  
set column width 423
- text color 54  
text color for cells 443  
views 185  
write to file 57
- table 472  
Tangent 554  
t-distribution 556  
template 473
- Test  
Chow 236  
CUSUM 401  
CUSUM of squares 401  
exogeneity 478  
for ARCH 206  
for equality of values 545  
for inequality of values 545  
for missing value 545  
for serial correlation 209, 212  
Goodness of fit 479  
Granger causality 228  
heteroskedasticity (White) 514  
Johansen cointegration 240  
lag exclusion (Wald) 480  
mean, median, variance equality 475  
mean, median, variance equality by classification 476  
omitted variables 474  
redundant variables 477  
RESET 397  
simple mean, median, variance hypotheses 481  
unit root 494  
Wald 502
- testadd 474  
testbtw 475  
testby 476  
testdrop 477  
testexog 478  
testfit 479  
testlags 480  
teststat 481
- Text 186  
declare 482  
views 186
- text 482  
Text file  
open as workfile 504
- Then 612

---

Three stage least squares 194  
 tic 482  
 Time  
     current as string 577  
 Timer 482, 483, 613  
     to 613  
 Tobit models 233  
 toc 483  
 trace 484  
 Trace of a matrix 599  
 Tramo/Seats 484  
 tramoseats 484  
 Transpose 600  
 Trigonometric functions 553  
 Truncated dependent variable models 233  
 tsls 487  
 Two-stage least squares  
     see 2sls

**U**

Uniform distribution 557  
 Uniform random number generator 541  
 Unit root test 494  
 Unit vector 600  
 unlink 491  
 Unstack workfile page 381  
 Untitled objects 10  
 update 492  
 updatecoefs 493  
 Uppercase 578  
 uroot 494  
 usage 499

**V**

Valmap 187  
     apply to series 349  
     declare 499  
     find series that use map 499  
     procs 187  
     views 187  
 valmap 499

**VAR**

estimate factorization matrix 469  
 impulse response 311  
 lag exclusion test 480

lag length test 318  
 multivariate autocorrelation test 390  
 variance decomposition 260

**Var** 187

clear restrictions 237  
 data members 189  
 declare 500  
 methods 188  
 procs 188  
 views 188

**var** 500

**Variance** 549

equality test 475, 476  
 Variance decomposition 260

**vars** 501

**VEC**

estimating 271  
**Vector** 190

data members 191  
 declare 501  
 fill 587  
 procs 191  
 views 191

**vector** 501

**Vector autoregression**

See Var.

**Verbose mode** 85

**W**

wald 502  
 Wald test 502  
 Watson test 272  
 Weibull distribution 557  
 Weighted least squares 515  
 Weighted two-stage least squares 519  
 wend 614

**wfcreate** 503

**wfopen** 504

**wfsave** 512

**wfselect** 514

**while** 614

**While loop** 104

abort 105  
 end of 614  
 exit loop 107  
 start of 614

white 514  
Wildcards 14  
wls 515  
Workfile 11, 378  
append contents of workfile page to current page 364  
close 12  
contract page 366  
convert repeated obs to repeated series 381  
convert repeated series to repeated obs 375  
copy from page 367  
create 250  
create new workfile 503  
create page in 369  
date strings 126, 146, 561  
define structured page 378  
delete page 371  
end date of observation interval 560  
expand 279  
export page 512  
frequency 11  
load workfile pages into 371  
number of observations in 560  
observation date functions 146, 561  
observation number 559  
open existing 12, 504  
open foreign source into 504  
panel to pool 381  
period indicators 561  
pool page to panel page 375  
rename page 372  
save 12, 512  
save or export page 373  
seasonal indicators 561  
set active page 374  
set active workfile and page 514  
sort observations 453  
stack page 375  
start date of observation interval 560  
time trend 562, 564  
time trend (calendar based) 562  
unstack page 381  
workfile 517  
Write  
    data to text file 517  
write 517  
wtcls 519

**X**

x11 520  
x12 522  
xy 528  
XY (line) graph 530  
XY (pairs) graph 532  
XY graph 528  
xyline 530  
xypair 532