# MacroEconometric Forecasting

Topic:

EViews Basics

Presented by Munisa Yashnarbekova

# The EViews Command Language

# The EViews Command Language

The EViews command language may be split into four groups:

- Commands
- Functions
- Object Views and Procs
- Object Data Members

# EViews Commands

EViews commands tell EViews to:

- Manipulate object containers (Workfiles, Pages, Databases).
- Create new objects.
- Show, copy, delete, rename or close objects.
- Set Program options.

A list of all EViews commands can be found in Chapter 12 of the EViews 9 Command and Programming Reference (PDF available from the EViews Help Menu).

# EViews Commands: Container Manipulation

The most common type of EViews commands are those that create, open, or interact with EViews workfiles.

In general these commands start with "wf":

- `wfopen` – open an existing workfile on disk, or a foreign file.
- `wfcreate` – create a new workfile.
- `wfclose` – close a workfile.
- `wfsave` – save the current workfile as an EViews file or a foreign format.
- `wfselect` – change the current active workfile.
- `wfrefresh` – refresh the links in the current workfile.

# EViews Commands: Container Manipulation

Commands that manipulate pages generally start with "page":

- `pageload` – load a new page in the current workfile.
- `pagecreate` – create a new page in the current workfile.
- `pagedelete` – delete a page in the current workfile.
- `pageselect` – change the current active workfile page.
- `pagestruct` – restructure the current page.
- `pagerename` – rename a page in the current workfile.

# EViews Commands: Container Manipulation

Similarly, commands that work on databases start with "db".

Other object container commands include:

- `smpl` – change the sample of the current workfile page.
- `import` – import data into the current workfile page.
- `fetch` – fetch data from a database into the current workfile page.
- `store` – store data from the current workfile page into a database.

# EViews Commands: Object Creation

Object creation commands (often called object declaration commands) are used to create new objects in the current workfile. The commands are, generally, simply the type of object, followed by the name of the new object:

- `series x` – create a new series called "X".
- `equation eq1` – create a new equation called "EQ1".
- `group g` – create a new group called "G".

# EViews Commands: Object Creation

Often you can follow the command with further specification for the object:

- `series x = @nrnd` – create a new series called "X" filled with normal random numbers.

- `group g x y z` – create a new group called "G" containing the series "X", "Y" and "Z".

The full syntax for each object's declaration command can be found at the start of each object's section of Chapter 1. of the EViews 8 Object Reference.

# EViews Commands: Object show/copy/rename/delete/close

The following commands work on workfile objects

- `show` – display a view of an object.
- `freeze` – freeze the view of an object into a new object.
- `copy` – copy the object from workfile to workfile, or page to page, or create a copy in the current page.
- `rename` – rename an object.
- `delete` – delete an object.
- `close` – close an object (if it object is currently being shown).

# EViews Functions

EViews functions are used to assign values to Series or Alpha objects, or Matrix and Scalar objects, or program variables.

Functions generally start with an "@" symbol, and are preceded by an object name (and, possibly declaration) and an "=". For example:

```
series x = @nrnd
```

Here the @nrnd function is used to assign standard normal random numbers to the series X.

A list of all EViews functions can be found in Chapter 14 of the EViews 9 Command and Programming Reference. Full details of the functions can be found in Chapter 13.

# EViews Functions

There are many types of EViews functions:

- Basic mathematical
- Time series
- Financial
- Descriptive Statistics
- Cumulative Statistics
- Moving Statistics
- Group Row
- By-Group
- Trigonometric
- Statistical Distribution
- String
- Date
- Indicator
- Workfile & Information

# EViews Functions

## Some examples:

| | |
|---|---|
| `series x = @log(y)` | Create a series, X, and assign the values of log(Y) to it. |
| `series x = @pch(y)` | Assign the one-period percentage change in Y to X. |
| `series x = @pv(r,n,y)` | Present value of Y, given a rate of R and N periods. |
| `scalar x = @mean(y)` | Create a scalar, X, equal to the mean of Y. |
| `series x = @cumsum(y)` | Cumulative sum of Y. |
| `series x = @movav(y,3)` | Three period moving average of Y. |
| `series x = @rsum(g)` | Row-sum of the group, G. |
| `series x = @minsby(y,s)` | Minimum values of Y for each category of S. |
| `!x = @tan(y)` | Assign the tangent of the program variable !y to !x. |
| `scalar x = @cnorm(y)` | Cumulative normal distribution at value Y. |
| `!x = @instr(%y, "he")` | Find the position of the phrase "he" in the string %y. |
| `string x = @strnow` | Create a string object containing the current date/time. |
| `series x = @trend` | Create a trend series. |

# Object Views and Procs

The EViews workfile is a collection of objects. Each object type has different Views and Procs available to it, and they are generally accessed by clicking on either the View menu or the Proc menu when an object is open.

Each object view and proc has a command line equivalent. The general syntax of object views and procs is:

```
object.view(options) arguments
```

i.e. the name of the workfile object, followed by a dot, then the view/proc name, followed by options and any arguments.

The Views and Procs available to each object type are listed in the EViews 9 Object Reference.

# Object Views and Procs

## Some examples:

| | |
|---|---|
| `show eq01.stats` | Show the regression output of the equation object EQ01. |
| `show gdp.line` | Show a line graph of the series GDP. |
| `freeze(gr1) gdp.line` | Freeze the graph of GDP into a new graph object, GR1. |
| `eq01.forecast yf` | Forecast EQ01, storing the forecast values into series YF. |
| `unemp.smooth unemps` | Exponential smoothing on UNEMP, and save to UNEMPS. |
| `show gp.coint(s)` | Show the summary cointegration test results for group GP. |

# Object Data Members

Along with the Views and Procs available to each object type, objects also contain "data members". These are retrievable objects that contain information about the parent object.

Data members are only available via command. There is no mouse equivalent. The syntax for retrieving a data member is always:

```
=object.@member
```

Data members always return strings, scalars, or matrix objects. Note, they never return series or alpha objects.

# Object Data Members

Data members are stored with the object. EViews, in general, does not need to perform calculations to retrieve the members (unlike Views and Procs).

Each object's data members are listed at the start of the object's section of the EViews 9 Object Reference.

# Object Data Members

## Some examples:

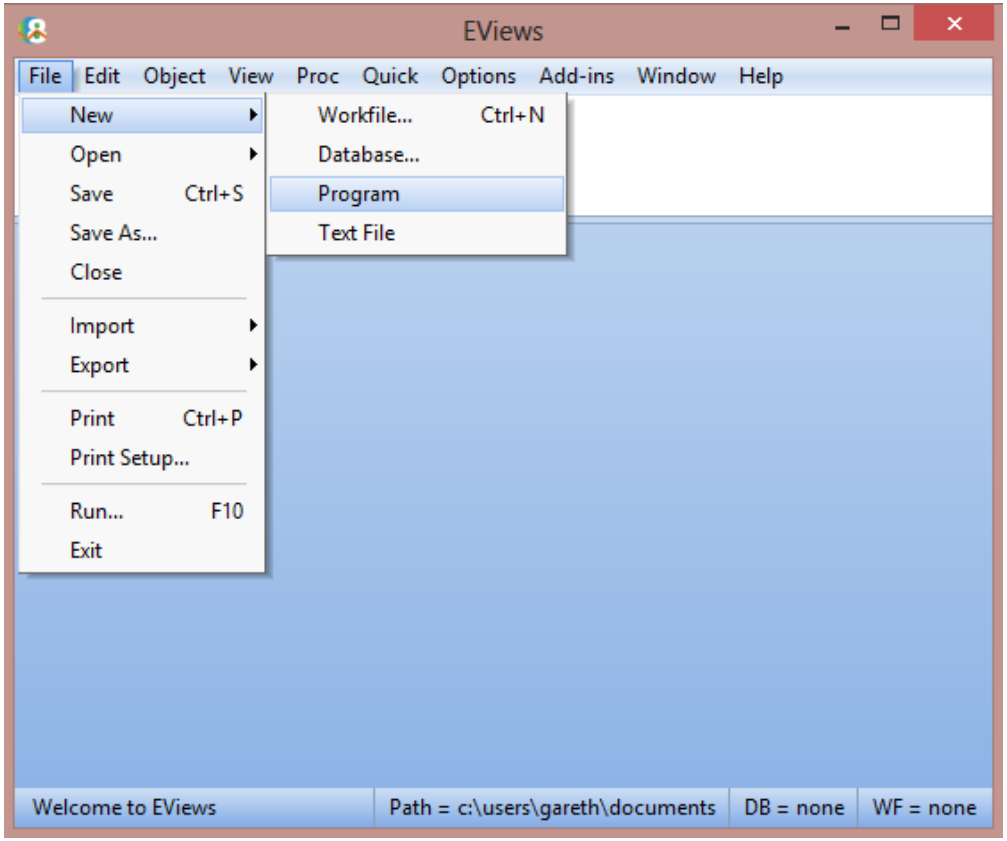
| | |
|---|---|
| `=unemp.@displayname` | Returns the UNEMP series' display name. |
| `=gdp.@first` | Returns the date of the first non-NA in GDP. |
| `scalar x = eq01.@r2` | Stores the R-squared from EQ01 into the scalar X. |
| `matrix x = eq01.@coefcov` | Saves EQ01's coefficient covariance matrix into matrix X. |
| `vector x = eq01.@tstats` | Saves the t-statistics into vector X. |
| `=g1.@count` | Returns the number of series in the group G1. |
| `=g1.@seriesname(1)` | Returns the name of the first series in the group G1. |

# EViews Program Basics

# Creating a New Program

EViews programs are merely a collection EViews commands collected together in a text file named with a .prg extension. As such any text editor can be used to create an EViews program.

However the easiest way to create one is to use EViews itself, by clicking on File->New->Program.



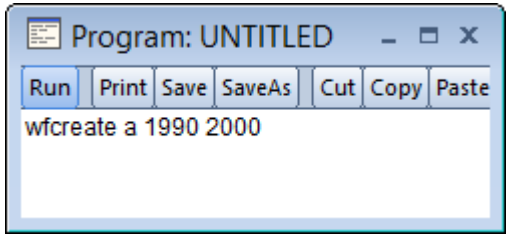
# Loading and Saving Programs

To open an existing program, simply click on File->Open->Program.

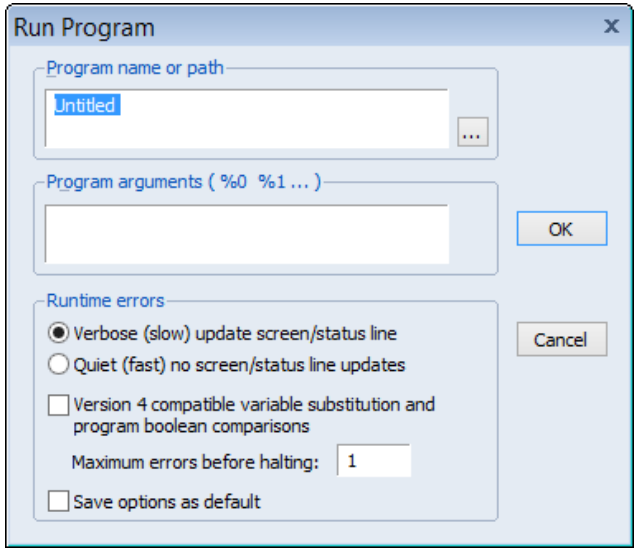
To save an open program, click on File->Save.

# Running a Program

Once you have created your program, you can run it by clicking on the Run button.

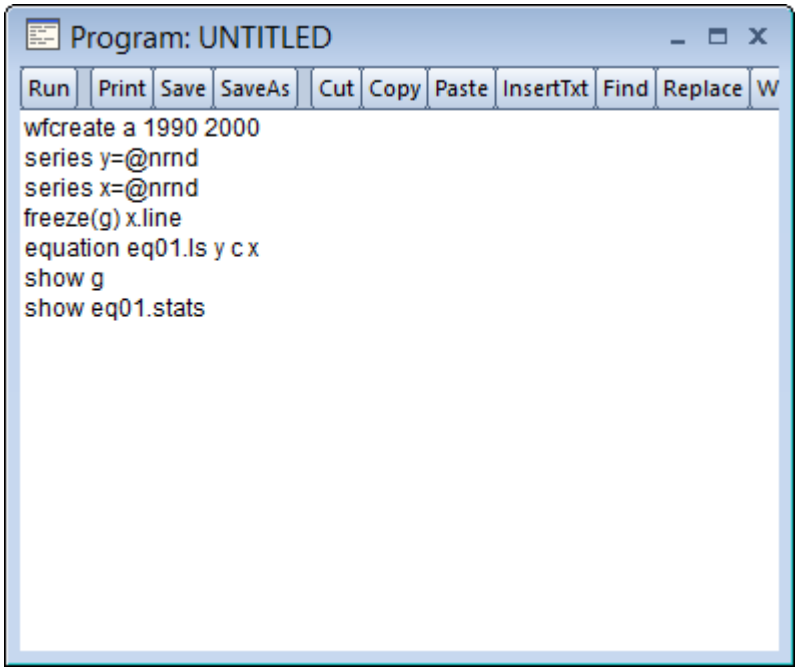



This brings up the Run dialog, which contains some program execution options. We'll ignore those for now, and just hit the OK button.



# A Simple Program

The simplest program is just a collection of simple commands:



# Program Comments

EViews uses the apostrophe as a comment character in programs. Any text following a ' *and on the same line* will be ignored by the program.

Example:

```
series x = @nrnd  'this is a comment
```

Only the "series x = @nrnd" part of this line is executed by EViews.

# Program Variables

# Program Variables

Program variables are variables that only exist when the program is running. Once the program has finished, the variables disappear. They do not form part of your workfile.

There are two types of program variables. Numeric and string variables. Numeric variables start with an “!”.  String variables start with a “%”.

Tip: Due to their temporary nature, displaying program variables is difficult. For debugging purposes it is often useful to temporarily assign program variables to workfile objects (such as scalars, strings or tables) to help you follow what is happening to your program variables.

# ! Variables

! variables are numeric scalar program variables. They can be used in most mathematical calculations in a program.

Examples:

```
!x = 3
!y = 3+2
!z = !x + !y
!pi = 4*@atan(1)
series y = @sqrt(!z)
```

# % Variables

% variables are character/string program variables. They contain quoted text.

You can concatenate two strings with a + sign. You may also use string functions to assign to a % variable.

Examples:

```
%x = "hello"
%y = "my name is" + %name
%pi = "3.142"
%z = "The date/time is " + @strnow
```

# Replacement Variables

You can use % variables in two ways. The first is as an string actual string – i.e. anywhere EViews expects a string value, you can use a % variable instead.

The second way is as a "replacement". Replacement variables are used to substitute the variable with its string value.

To instruct EViews to use a % variable as a replacement, rather than as a string, enclose the % variable inside braces {}.

# Replacement Variables

Example:

```
%x = "gdp"
series y = %x
series z = {%x}
```

The first line assigns "gdp" to the % variable %x.

The second line is read by EViews as:

```
series y = "gdp"
```

This will error (series are numeric and cannot contain text).

The third line, however, is read by EViews as:

```
series z = gdp
```

# Replacement Variables

A good rule of thumb to follow when deciding whether you should enclose your % variable inside braces or not, is to think "Does EViews expect quotes in this command?" If EViews is expecting quotes, leave the % variable as it is. If it is not expecting quotes, uses the braces.

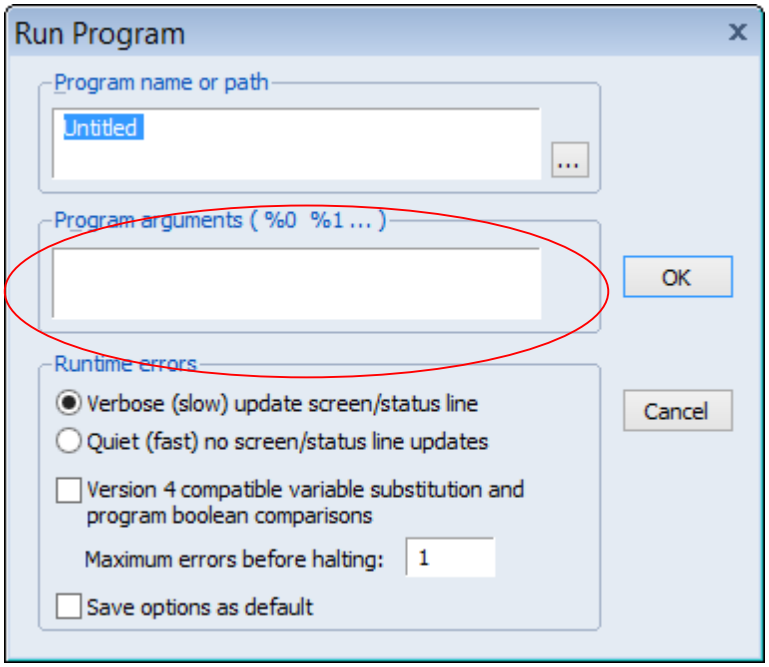
```
series y = {%z} + {%z}
scalar p = @instr(%z, %y)
%z = "name"
alpha a = %z
alpha b = {%z}
```

Note the difference between the last two – the first sets A equal to "name", and the second sets B equal to the alpha series called NAME.

# Program Arguments

Program arguments are special % string variables whose value may be changed every time you run the program. The individual arguments are called "%0", "%1", "%2" and so on.

You can set the arguments using the Run dialog:




Simply enter a space delimited list of values in the Program arguments box.

# Program Arguments

For example, take the following program line:

```
equation eq1.ls {%0} C {%1}
```

If you enter "Y X" into the Program arguments box on the run dialog, then the program will execute the line:

```
equation eq1.ls Y C X
```

Whereas entering "GDP UNEMP" would execute:

```
equation eq1.ls GDP C UNEMP
```

# Control of Execution

# If/else/endif statements

If statements are used when you wish to execute a line of code only if a certain condition is met. The basic syntax is:

```
if [condition] then
'line of code to execute
endif
```

[condition] must be an expression that evaluates to a scalar value of 1 or 0, or a scalar true/false. Note this means you *cannot*, in general, use a series expression as part of an if statement.

# If/else/endif statements

Examples:

```
if 1+1=2 then
wfcreate a 1990 2000
endif
```

Since 1+1 does equal 2, the wfcreate command will be executed (creating an annual workfile between 1990 and 2000).

```
if !p>3 then
equation eq1.ls y c x1 x2
endif
```

Only if the program variable, !p, is greater than three will equation EQ1 be estimated.

# If/else/endif statements

```
if @instr(%x, "gdp")>0 then
var v1.ls 1 3 {%x} unemp m2
endif
```

Only if the program variable %x contains the string "gdp" will the VAR be estimated.

```
if @max(sales)>1000000 then
show price.hist
endif
```

If the maximum value of SALES is greater than 1,000,000, show the histogram and descriptive statistics of the series PRICE.

# If/else/endif statements

You may use an else statement to tell EViews what to do if the condition is not met. The basic syntax is:

```
if [condition] then
'line of code to execute if true
else
'line of code to execute if not true
endif
```

# If/else/endif statements

Example:

```
if !p>3 then
equation eq1.ls y c x1 x2 x3
else
equation eq1.ls y c z1 z2 z3
endif
```

If !p is greater than 3, equation Y is regressed against X1, X2 and X3. Otherwise Y is regressed against Z1, Z2 and Z3.

# If statements and series/samples

As already mentioned, you may not use an if statement on a series expression. If you want to conditionally assign values to a series you must either use a sample, or the @recode function.

Examples:

```
smpl if x<0
series y = 100
smpl if x<=0
series y = 200
smpl @all
```

```
series z = @recode(x<0, 100, 200)
```

# For Loops

EViews supports two types of for loop; numerical and string.

Numerical for loops take the form:

```
for !i=1 to 10
'lines to be repeatedly executed
next
```

A scalar variable (either a ! variable, as shown here, or a scalar object in the workfile) is used to control the loop. Each time through the loop the scalar variable is incremented by 1. The loop stops once the variable reaches its terminal value (here 10).

# For Loops

Examples:

```
for !i=1 to 5
series x{!i} = @nrnd
next
```

This loop generates 5 random normal series, X1, X2, X3, X4 and X5.

```
for !i=1 to 10
equation eq1.ls y c x{!i}
next
```

Five equations are created, each regressing Y against a single X variable.

# For Loops

You may optionally add a *step* statement to change how much the control variable increases at each iteration of the loop:

```
for !i=1 to 10 step 2
series x{!i} = @nrnd
next
```

This loop generates 5 random normal series, X1, X3, X5, X7 and X9.

```
for !i=20 to 1 step -5
equation eq1.ls y c x{!i}
next
```

Four equations are created, each regressing Y against a single X variable, first X20, then X15, then X10, then X5

# For Loops

String for loops simply loop a string control variable over different string values.

String for loops take the form:

```
for %j [space delimited list of values]
'lines to be repeatedly executed
next
```

A string variable (either a % variable, as shown here, or a string object in the workfile) is used to control the loop. Each time through the loop the control variable is set equal to the next value in the given list.

# For Loops

Examples:

```
for %j gdp unemp time
series {%j} = @nrnd
next
```

This loop generates 3 random normal series, GDP, UNEMP and TIME.

```
%regs = "sales_uk sales_usa sales_fra"
for %k {%regs}
equation eq1.ls {%k} c demand_global
next
```

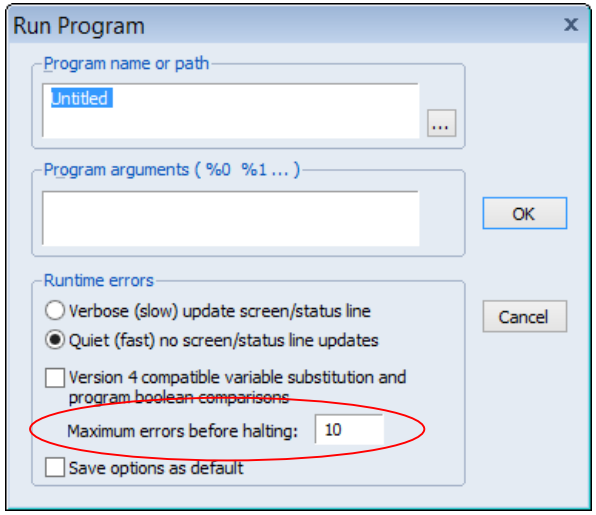
Three equations are created, each with a different dependent variable, but the same independent variable.

# Error Handling

# Maximum Number of Errors

By default, an EViews program will stop as soon as EViews issues an error. There are two methods available to override this behavior:

- Use the Run dialog to increase the "Maximum errors before halting"




- Use the `setmaxerrs` command within the program to dynamically change the number of errors allowed.

# Maximum Number of Errors

Example:

```
wfcreate m 1990 2000
series y=@nrnd
series x=@nrnd
series w=@nrnd
series z=3
for %reg x z w
      equation eq_{%reg}.ls y c {%reg}
next
```

This program loops through the series X, Z and W, performing a regression of Y against a constant and each of those series, one at a time. It will cause an error when it regresses against Z, since Z is perfectly collinear with the constant.

# Maximum Number of Errors

Increasing the maximum number of errors to something greater than 1 in the Run dialog will let the program continue past the error.

Similarly, adding a `setmaxerrs` line to the program will allow it to run:

```
wfcreate m 1990 2000
series y=@nrnd
series x=@nrnd
series w=@nrnd
series z=3
setmaxerrs 100
for %reg x z w
      equation eq_{%reg}.ls y c {%reg}
next
```

# Program Execution

# Running/Executing Programs

There are a number of ways to instruct EViews to run a program:

- The Run button/dialog.
- The *Run* command.
- The *Exec* command.
- The *Include* command.

# Run Command

The *Run* command can be issued from the EViews command line to instruct EViews to open and run a program. The syntax is:

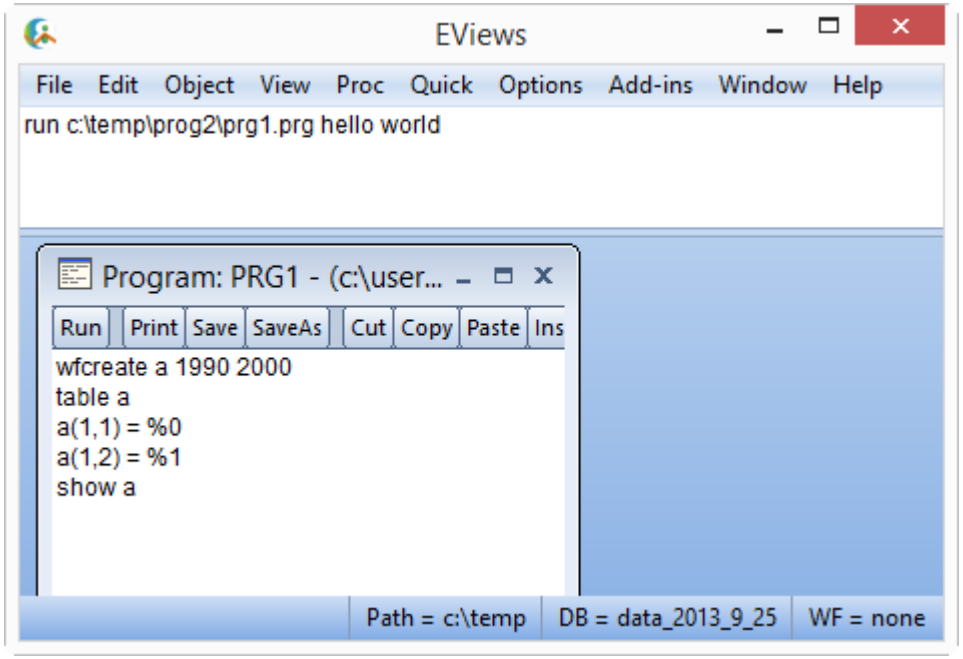
```
run myprogram.prg [arguments]
```

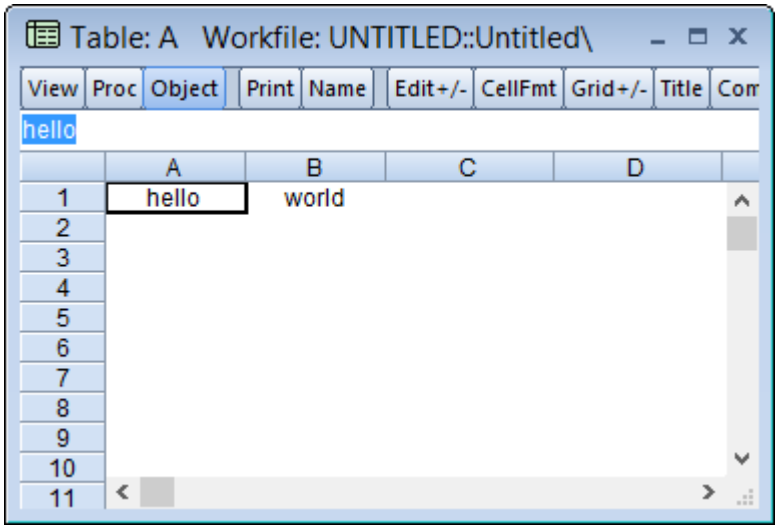
Simply give the name of the program (with a path, if required) following the *Run* command.

You may add arguments as a space delimited list following the name of the program.

# Run Command

## For example:

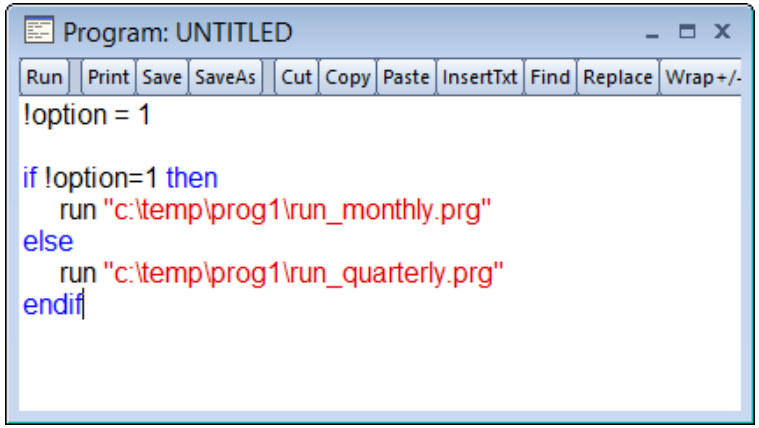


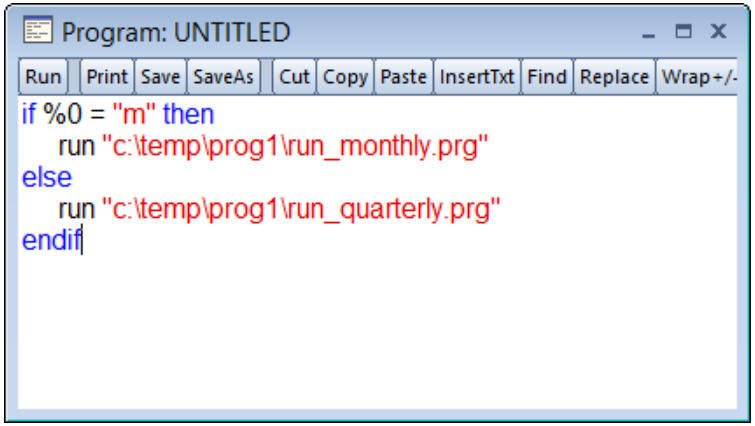



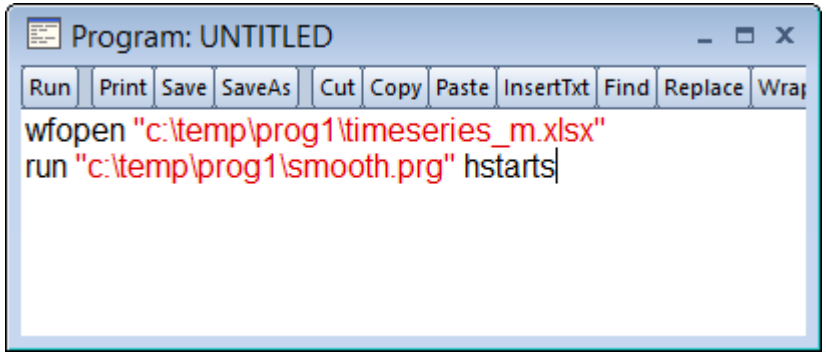
# Run Command

You may also use the *Run* command inside a program to launch and run a second program.

For example:

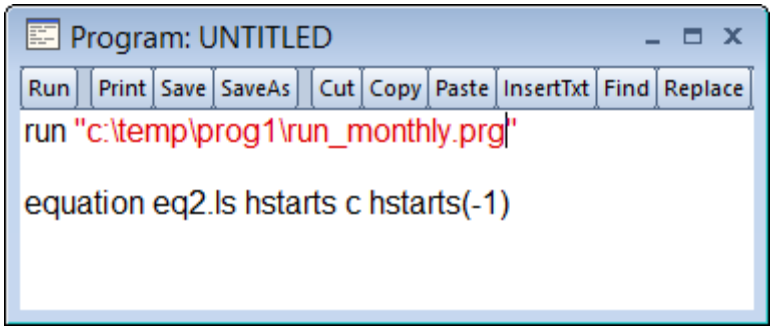


# Run Command

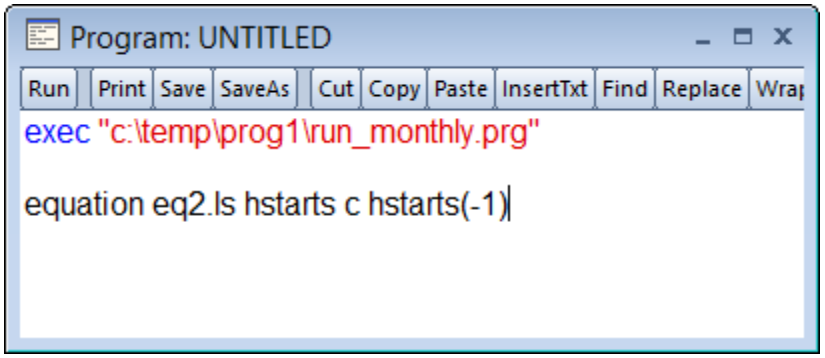
However, the *Run* command has one drawback when being used inside a program: EViews will not return to the calling program once the child program has finished running. For example:




In this program the line estimating equation *eq2* will never execute. EViews stops all program execution once *run_monthly.prg* has finished.

# Exec Command

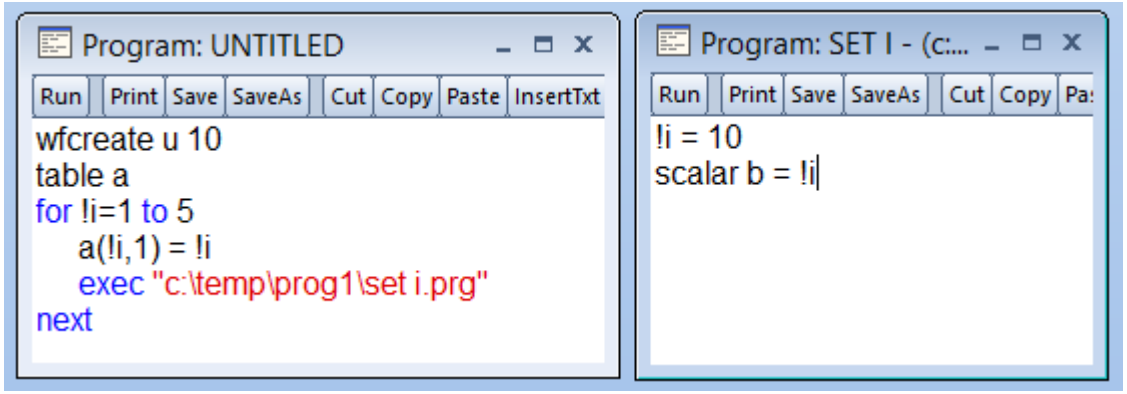
The *Exec* command works in the same was as the *Run* command, with one main differences; unlike the *Run* command, EViews will continuing executing the calling program after an *Exec.*




This program will execute the *run_monthly.prg* program, and then continue to execute the following lines, creating the equation eq2.

# Exec Command

Note that with the *Exec* command (and *Run*) each program's program variables are isolated. i.e. setting a variable, such as !i, in the parent program will not interfere with a variable with the same name in the child program.



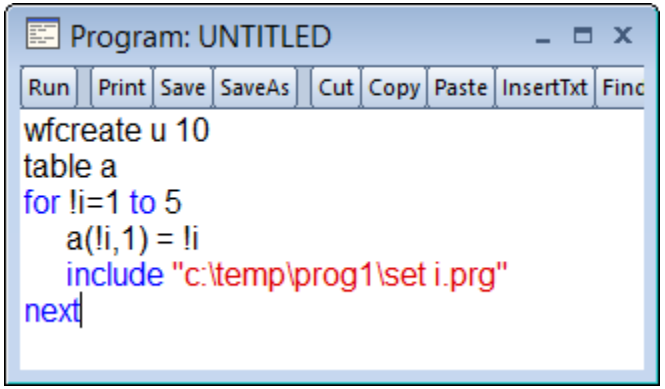

Similarly, any subroutines defined in the child program will not be available in the calling program.

# Include

The final way to execute a program from within a program is with the *include* command.

*Include* is similar to *Exec* with a couple of differences:

- Unlike *Exec*, program variables are not isolated – the child program can interfere with program variables used in the parent




- Subroutines are available to the parent program.

# Tips

- *Run* should never be used in a program
- *Include* should only be used when loading a "library" of subroutines – the included program should not actually execute any code, merely provide subroutines for use in the parent program.
- *Exec* should be used for all other times you wish to run a child program.

# Subroutines

# Subroutine Definition

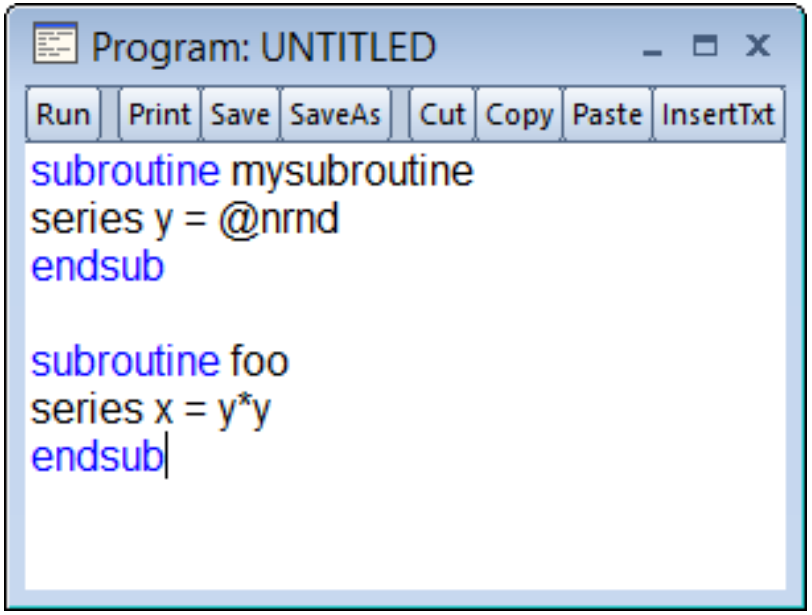
EViews subroutines are user-defined functions that exist within the scope of the EViews programming language.

Subroutines allow you to define a set of commands in one part of your program and then call (or execute) those commands repeatedly in other parts of your program.

The basic syntax to define the start of a subroutine is to use the `subroutine` keyword, followed by the name of the subroutine. The subroutine ends with the `endsub` keyword.

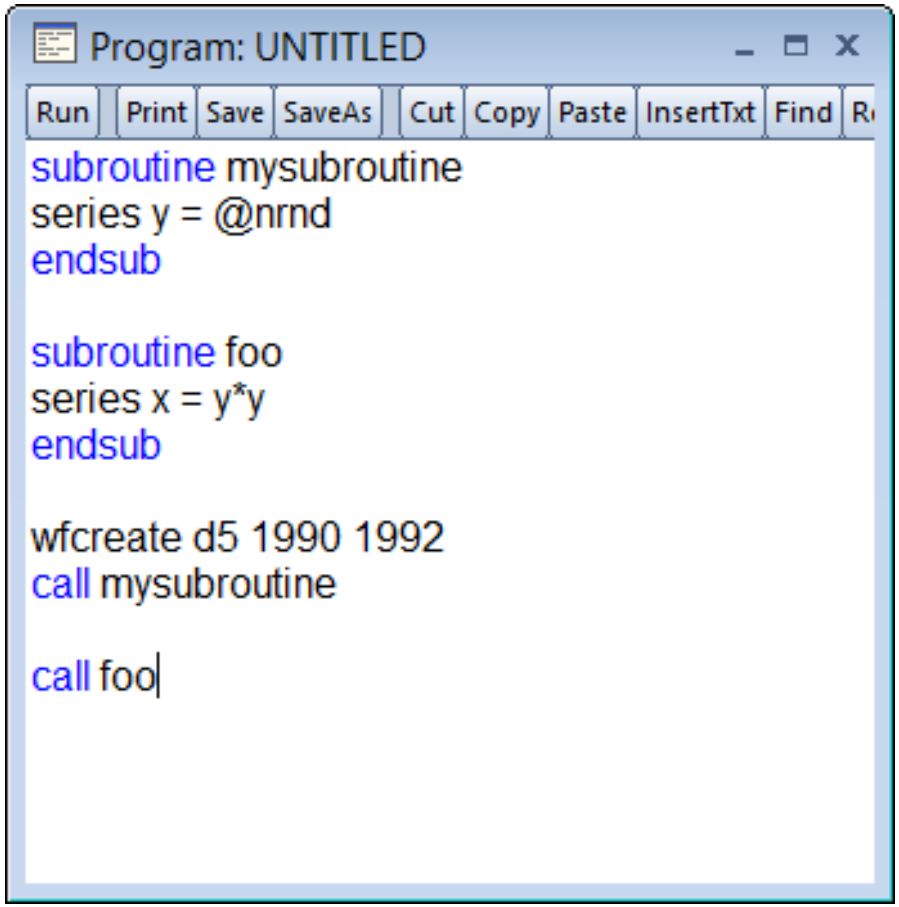
# Subroutine Definition

Example:



# Calling Subroutines

To execute the code inside a subroutine, the `call` keyword is used, followed by the name of the subroutine you wish to execute.



# Subroutine Arguments

Subroutines may include arguments, allowing you to define the variables used within the subroutine.

To declare arguments with your subroutine you define the type of object being passed in, as well as its name:

```
subroutine name(obj_type1 arg1, obj_type2 arg2, …)
```

The name of the arguments in your subroutine do not (and should not) be the name of objects in your workfile.

# Subroutine Examples

Program: UNTITLED

Run | Print | Save | SaveAs | Cut | Copy | Paste

```
subroutine square(series in)
    series out = in^2
endsub

wfcreate d5 1990 2000
series y=@nrnd
call square(y)
```

Program: UNTITLED

Run | Print | Save | SaveAs | Cut | Copy | Paste | InsertTxt | Find | Repla

```
subroutine AplusBeqC(series a, series b, series c)
    c = a + b
endsub

wfcreate d5 1990 2000
series y=@nrnd
series z=@nrnd
series x
call aplusbeqc(y, z, x)
```

Program: STOCK1 - (c:\users\viu35889\dropbox\eviews elearning...

Run | Print | Save | SaveAs | Cut | Copy | Paste | InsertTxt | Find | Replace | Wrap+/- | LineNum+/- | Encryp

```
subroutine ema(series out, series in, scalar n) 'calculate exponential moving average
    out = in
    !p = 2/(n+1)
    smpl @first+1 @last
    out = !p*in + (1-!p)*out(-1)
endsub

subroutine bollinger(series out, series lowout, series highout, series price, scalar n, scalar k)
'calculate bollinger bands
    out = @movav(price,n)
    lowout = out - k*@movstdev(price,n)
    highout = out + k*@movstdev(price,n)
endsub

wfcreate d5 1990 2000
series stock = 10+@rnd

series stock_ema
call ema(stock_ema, stock, 3)

series stock_boll
series stock_boll_l
series stock_boll_h
call bollinger(stock_boll, stock_boll_l, stock_boll_h, stock, 5, 3)
group boll stock_boll stock_boll_l stock_boll_h
show boll.line
```

# Subroutine and Argument Placement

Subroutines can be placed anywhere in your program: at the start, at the end, or spread throughout the program. They need not be defined above the *call* statement that calls them.

Subroutine arguments can be inputs (i.e. objects that the subroutine uses) or outputs (i.e. objects that the subroutine produces), or both. There is no way to indicate that an object is an input or an output.

# Tips

- Subroutine should all be placed at the start of the program file for easy debugging access.
- If you have many subroutines, define them in a separate program file and then *include* that file at the top of your program.
- Define your subroutines such that outputs come first in the argument list (or come last). Don't mix inputs and outputs.
- For even greater clarity, name the arguments with a name that indicates whether the argument is an input or output.

# Subroutine Strings and Scalars

When defining a subroutine, you will often want to use a string or a scalar value as one of the arguments. You can choose whether to use a string/scalar workfile object, or a program variable as the argument:

```
subroutine foo(string a, scalar b)

subroutine goo(string %a, scalar !b)
```

Here foo uses workfile objects, and goo uses program variables (note the use of % and ! in the argument names).

# Subroutine Strings and Scalars

When calling a subroutine with a string or scalar argument you can use a program variable, a workfile object, or a value, whether you defined the subroutine with or without a program variable.

```
subroutine foo(scalar b)
subroutine goo(scalar !b)
call foo(myscalar)
call foo(!x)
call foo(3)
call goo(myscalar)
call goo(!x)
call goo(3)
```

# Subroutine Strings and Scalars

Thus there is very little difference between defining your subroutines with either objects or program variables when using strings or scalars.

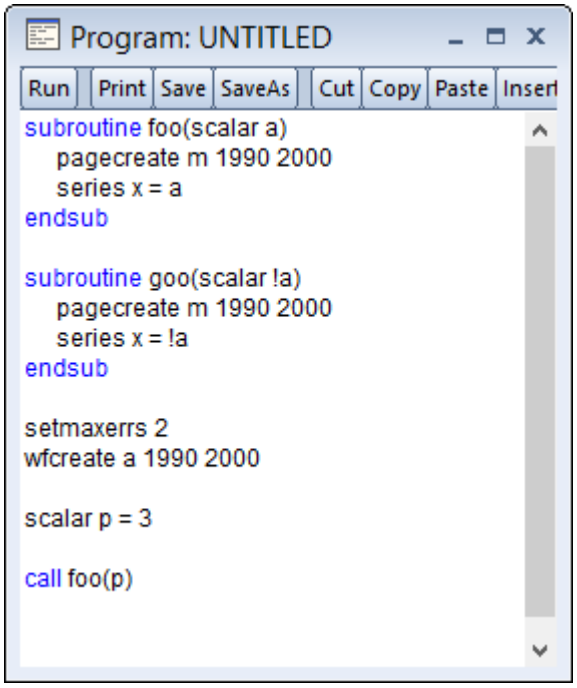
The one exception is if the subroutine changes workfile page.

When using a workfile object as an argument, changing the workfile page means that the object is no longer available inside the subroutine.

# Subroutine Strings and Scalars

For example the following program will error inside *foo* because the scalar *a* is not available on the new page, so *x* cannot be created.

Using *goo* instead will work just fine.



# User Dialogs

# User Dialogs

EViews provides a number of functions for the creation of custom dialogs inside an EViews program.

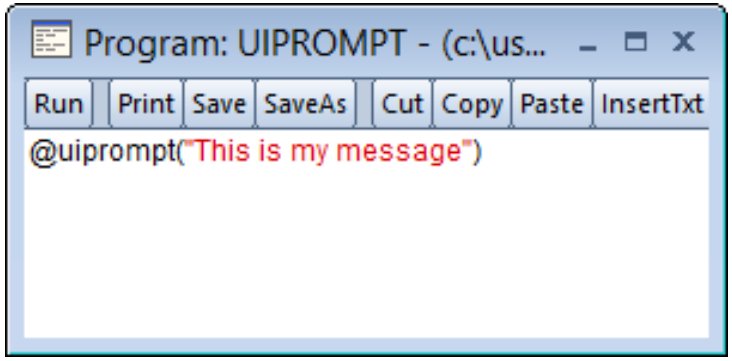
These custom dialogs (or User Dialogs to use the EViews term) allow the program to present information, or collect information, from the person running the program.

They provide a more elegant way of providing program input options than program arguments, program options, or simply letting the user configure some settings at the top of the program.

# User Dialogs

The simplest form of User Dialog is the `@uiprompt("msg")` function.

It simply produces a message box for the user containing the message *msg*:



# User Dialogs

## Other simple User Dialogs functions:

| Function | Description | Full Syntax |
|---|---|---|
| ***@uiedit*** | Produces an edit field. | `@uiedit(string iostring, string prompt)` |
| ***@uilist*** | Produces a list box. Has two versions, one returns a string, the other a scalar. | `@uilist(string iostring, string prompt, string list)`<br>`@uilist(scalar ioscalar, string prompt, string list)` |
| ***@uiradio*** | Produces a set of radio buttons. | `@uiradio(scalar ioscalar, string prompt, string list)` |

The *iostring* or *ioscalar* arguments are both an input and output. They decide what is pre-filled in in the dialog, and they return the user's input to the dialog.

# User Dialogs

All User Dialogs are EViews functions, and as such have return values. The value returned depends upon whether the user clicked on the "OK" button or the "Cancel" button.

If a user clicks "OK" on the User Dialog, the function will return a value of 0. "Cancel" returns -1.

# @uidialog

The final User Dialog function is `@uidialog`, and it is the most complicated, allowing you to build extensive dialogs with many different controls inside them.

To specify a dialog with `@uidialog`, you must enter a list of control types and the arguments for each control type.

# @uidialog

The controls are:

| Control | Arguments | Description |
|---|---|---|
| ***"edit"*** | `string iostring, string prompt` | Edit box |
| ***"list"*** | `string iostring, string prompt, string list`<br>`scalar ioscalar, string prompt, string list` | List box |
| ***"radio"*** | `scalar ioscalar, string prompt, string list` | Set of radio buttons |
| ***"check"*** | `scalar ioscalar, string prompt` | Single check box |
| ***"caption"*** | `string caption` | Dialog title |
| ***"button"*** | `string buttontext` | Custom button |

`@uidialog` also returns a 0 or -1 depending on whether "OK" or "Cancel" was pressed, unless there is a custom button control, in which case pressing the button rather than "OK" or "Cancel" will return a positive integer.

# EViews Add-ins

# EViews Add-ins

EViews Add-ins are ways of extending the features available in EViews via the menu system, or via the command language.

A number of Add-ins are available to download by clicking on the Add-ins menu.

However, the most important thing about Add-ins (at least for this class) is that Add-ins are really nothing more than EViews programs.

Any program you write can be turned into an EViews Add-in, simply by using the Manage Add-ins menu item.

# Adding Add-ins

When you add an Add-in to your copy of EViews, you may specify whether the Add-in is available via the menu system, via a command, or both.

You may also choose whether to make your Add-in Global or Object-specific.

Global add-ins are available at all times, and always appear in the Add-ins menu. Object-specific add-ins only appear in the Add-ins menu when an object of that type is open.

# addin

Rather than using the menu system to register/add an Add-in, you may add one via the `addin` command. The syntax is:

`addin(type=`*`type`*`, docs=`*`docspath`*`, menu=`*`menutext`*`, proc=`*`proctext`*`) programname`

(There are other options also available – see the Command Reference entry for more details).

*Type* should be the object type of the Add-in, or "Global" for a Global Add-in.

*Docspath* should be the full path and file name of any documentation accompanying the Add-in.

*Menutext* should be the text to add to the Add-ins menu (if any).

*Proctext* should be the command line given to the Add-in.

# _this

Object-specific Add-ins work on whatever object is currently open. An equation-specific Add-in might, for example, use the coefficient covariance matrix to perform a post-estimation diagnostic.

The problem is that you probably won't know the name of the currently open object, so referencing the data-members, or accessing views and procs of the currently open object could be tricky.

# _this

Thankfully, EViews provides the _this keyword. _this simply tells EViews that you want to work with the current open object.

All of the object data members, procs and views are available by using _this rather than the object's name.

Thus:

`_this.hist` will show a histogram of the current open object (if the current open object is a series).

`_this.@coefcov` will access the equation's coefficient covariance matrix (if the current open object is an equation).

# Reference and source


- Conceptual Econometrics Using R (ISSN Book 41) 1st Edition, by Hrishikesh D. Vinod (Editor)
- Principles of Macroeconometric Modeling (Volume 36) (Advanced Textbooks in Economics, Volume 36) by L.R. Klein, W. Welfe, et al. | Oct 5, 1999
- Macroeconomic Modeling and Macroeconometric Simulation: Illustrated with a developing economy Model (Macroeconometric model Book 1) Book 1 of 1: Macroeconometric model | by Kannapiran Arjunan | Jun 9, 2020
- Global and National Macroeconometric Modelling: A Long-Run Structural Approach by Anthony Garratt, Kevin Lee, et al. | May 4, 2012
- Simulation of a macroeconometric model with multiple time series considerations (Wayne economic papers) by Rosemary Rossiter | Jan 1, 1982