Simple and Series Functions in Easy Language®

Introduction

When creating a function in Easy Language, you have a choice between declaring it as Simple, Series, or "auto-detect". While some functions appear to be unaffected by whether they are stored as simple or series, there may be times when it will have a profound affect on how your function behaves. Moreover, the auto-detect feature in TradeStation's verifier may not ascertain which function type you intended and compile it the wrong way. This article will contrast the properties of series and simple functions so you can override the auto-detect feature and explicitly select the correct type for each of your functions.

NOTE – This material was written for TradeStation 4. TradeStation 2000 has slightly changed the way series and simple functions work. However, it is still safe to use this guide, and your code should work the same way on both platforms.

Function Basics

A function is a piece of code that, once created, may be accessed inside studies or other functions. For example, the code below uses the function RSI to calculate the 9 bar Relative Strength Index, then adds 7 and places the sum into local variable MyVar. Using this function is much more efficient than writing your own code to calculate RSI.

```
MyVar = 7 + RSI(close,9);
```

EL has many pre-written functions, including SQUAREROOT and XAVERAGE. The reader can create his own functions too. The rules for coding a function are similar to those for coding indicators, with the following exceptions:

1. A function can neither plot on a chart, as do indicators, nor make trades, as do systems. Instead they calculate a value and return it back to the code that called the function. In the example above, the AVERAGE function makes it calculation and returns the result to the line of code shown above, where it is then added to 7 and the sum is placed into local variable MyVar.
2. Inside a function, you specify the value that is to be returned by equating the value to a variable with the exact same name as the function itself. For example, if your function was called MyFunc, you would specify it is to return the value (H+L)/2 by including the following line of code inside your function …

```
MyFunc = (H+L)/2;
```

Case Studies

We now get to the core issue: simple and series functions may behave very differently even when the code appears similar. For example, the following function was explicitly specified to be type-series:

```
var: xav(close) ;
xav = 0.1*Close + 0.9*xav ;
AvgSeries = xav ;
```

and the following identical function was explicitly specified to be type-simple:

```
var: xav(close) ;
xav = 0.1*Close + 0.9*xav ;
AvgSimple = xav ;
```

Place both functions within the following indicator, which is designed to begin plotting only after the arbitrarily selected date of 01May84. Let's call this indicator "Demo_1".

```
If date > 840501 then
      begin
      Plot1 ( AvgSeries, "series" ) ;
      Plot2 ( AvgSimple, "simple" ) ;
      end ;
```

The code of both functions appear identical and both are used the same way in the indicator. Will they produce the same plots? As figure 1 shows, the two plots begin from very different locations and eventually converge. Surprised?

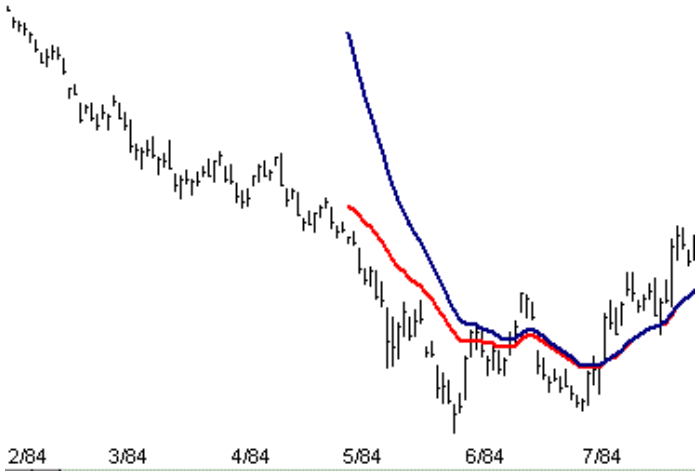

Figure 1 - Plots from identical simple and series functions inside indicator Demo_1

As another example, let's embed the same two functions in the following indicator, which is designed to calculate a new plot on every other bar. Let's call ithis indicator "Demo_2".

```
If mod (CurrentBar, 2) = 1 then
      begin
      Plot1 ( AvgSeries, "series" ) ;
      Plot2 ( AvgSimple, "simple" ) ;
      end ;
```

Although both functions appear to be employed in an identical manner, figure 2 shows their plots start at the same place and immediately diverge!

Figure 2 - Plots from identical simple and series functions inside indicator Demo_2

These two examples illustrate how functions can perform not as anticipated. This can be very frustrating especially when the misbehaving function is embedded in a study containing hundreds of lines of code. Why this happens is the result of several properties of EL, which are discussed next.

Simple versus Series

EL lets you refer to historical values of any time-series data by using the square bracket [ ] syntax. For example, the closing price N bars ago would be represented by "close[5]". All data produced by the server (e.g. O, H, L, C, V, and OI) are time-series in nature, and so their historical values may be accessed this way. In addition to data produced by the server, any study (e.g. Indicator, ShowMe, PaintBar or System) can access historical values of any variable coded within the same study, often referred to as "local variables". For example, the code below places a past value of one local variable into the current value of another.

```
MyOtherVar = MyVar[3];
```

EL stores variables in computer memory in either of two ways. Series variables have with them a limited amount of history that you can access by using the bracket syntax. In contrast, simple variables have no history and do not accept the bracket syntax.

The output of a series function behaves like a series variable, whose history is accessible using the bracket syntax, as in the following example:

```
value1 = Xaverage(close,9)[8] ;
```

Under certain circumstances, you can achieve similar results by accessing past values of the input variable instead, as in the following example:

```
value1 = Xaverage(close[8],9) ;
```

Both versions will plot identical lines. This alternate method only works if all the time series variables referenced by the function are specified as input parameters and all these parameters are bracketed the same way. This substitution trick becomes a lifesaver when dealing with type-simple functions because they produce only simple output values, devoid of any history. For example, the first line of code below produces a grammar error message, whereas the alternative expression will verify.

```
value1 = squareroot(close)[4] ;
value1 = squareroot(close[4]) ;
```

The above trick assumes the input parameter is a series variable. When that is not the case, try this approach: convert the function output into a series variable, then access its history, as in the following example:

```
temp = squareroot( some_complex_calculation ) ;
value1 = temp[4] ;
```

The caveat to this trick is that it does not work inside a simple function. The reason is all local variables declared inside a series function are type-series, and all local variables declared inside a simple function are simple. Consequently, the same local variable VALUE1 is of type-series inside the previously defined function AVGSERIES, and is type-simple inside function AVGSIMPLE. In the above line of code, "temp[4]" is not valid inside simple functions because the variable temp is simple, and has no history.

If you must have this result occur inside a simple function, then extract the code segment represented by "some_complex_calculation" and make it a separate series function. Assuming its name is "MyComplexCalc", you can then use the following:

```
value1 = squareroot( MyComplexCalc[4] ) ;
```

Sequence of Evaluation

The order in which functions are evaluated is another piece to the puzzle regarding the results in Figures 1 and 2. Consider the following two lines of code:

```
value1 = Phase_of_Moon (date) ;
value2 = AvgSeries (value1,19) ;
```

The second line will not verify. This is because series functions have stringent requirements regarding input parameters and that is due to the order in which EL code is evaluated. Here's why…

Series functions provide a time series data stream whose elements at anytime might be accessed as historical values. This requires the series function to be evaluated on every bar. To guarantee that occurs, all series functions within a study are evaluated before the rest of the code in that same study. This way, they are updated on every bar regardless of the conditional logic surrounding it.

This explains why the second line above will not verify. The series function XAVERAGE is evaluated before the first line of code is even read. However, the function operates on the variable VALUE1, which is not defined until the first line is read. Since all inputs to a function must be defined when the function is evaluated, and VALUE1 is not yet defined, we have an illegal situation. To prevent such deadlock scenarios from occurring, you may not use local variables as inputs to series functions.

The problem above can be resolved by making the function PHASE_OF_MOON an input parameter to XAVERAGE. This way, PHASE_OF_MOON is evaluated first, then XAVERAGE is evaluated next, and then the result is assigned to VALUE2.

```
value2 = AvgSeries ( Phase_of_Moon(date) , 19 ) ;
```

If you must use a local variable as input to a function, then the function must be type-simple. This provides another way to resolve the problem, as shown in the following code:

```
value1 = Phase_of_Moon (date) ;
value2 = AvgSimple (value1,19) ;
```

Case Studies Revisited

We now return to the case studies. Indicator DEMO_1 produced Figure 1 by plotting only after some specified date. The indicator plotted functions AVGSIMPLE (black line) and AVGSERIES (red line). Because the simple function did not begin processing until after the specified date, it missed its opportunity to initialize the local variable when Currentbar = 1. That is why it starts high (because it was initialized to the closing price of the first bar) and rapidly adjusts downward to the current closing prices. In contrast, series function AVGSERIES was evaluated on every bar, regardless of the conditional logic surrounding it. That is why it appears to be right on target when its plot first appears. Both lines converge because the incorrect initial conditions in AVGSIMPLE eventually fades into oblivion.

Indicator DEMO_2 produced Figure 2 by evaluating a new plot on every other bar. Since both functions are updated on the very first bar, we see they start at the same location. However, the simple function is evaluated on every other bar and the series function is, once again, evaluated on every bar. This means that the local variable inside AVGSERIES is being updated twice as frequently as AVGSIMPLE, causing the function overall to have only half the lag.

Which to Use?

Because series functions are guaranteed to be evaluated on every bar, use the series type unless either of the following overriding conditions exist:

- The function is to be evaluated on a conditional basis (e.g. inside an IF clause).
- At least one input parameter is a local variable. (So it can be adjusted on-the-fly)

In order to make the built-in function RSI accept local variables, the author created a type simple version, which the reader is invited to try out. The indicator, called "Adaptive RSI", can be freely downloaded from http://www.jurikres.com/freebies/mainfree.htm#software. Instructions are provided.