



Workbook Functions in Dynamic Text

Zephyr Associates, Inc.
P.O. Box 12368
Zephyr Cove, NV 89448

775-588-0654
800-789-5323
Fax 775-588-8423

www.styleadvisor.com

Introduction to Workbook Functions:

Suppose you're making a report using Dynamic Text (DT) in tables on a Notes page. You've figured out how to do something fancy with DT (fancier than just displaying a value from a Scan...something involving calculation) and you find yourself doing it over and over in a lot of cells. You're repeating yourself, writing duplicate code, hoping not to make any cut-and-paste errors.

There is a way to package a specific block of code so that it can be used anywhere in your workbook. You can make a Workbook Function. Once it's defined and debugged a Workbook Function can be used just like any of the built-in functions that make DT useful. As a quick aside, let me say what I mean by "block of code". Everything between a begin marker (<%) and an end marker (%>) is a block of code.

This tutorial assumes that you're familiar with the Display() function, variables, and if/then statements as they're used in DT, but don't know about or are hazy on the concept of functions. If you're already hip to the concepts and just want to find out how DT does it, you can probably just look at the code samples below and read the last page of this doc. Everybody else, hang on tight. We're going to go over the following, in detail:

- What is a function?
- How do functions work? Not just how are they used, but how they behave.
- How to write a function and why to write a function.
- How to make a function available to every block of DT in a workbook.

What the heck's a function?

If you've used Dynamic Text (DT) at all, you've already used a function. Almost everything in the "Add Function" menu in the Edit Dynamic Text dialog is a function.

So, what's a function? It's a self-contained block of code that can be called by name and which has a pre-defined job (maybe that's why it's called a function?). It also has a pre-defined number of arguments. Zero is a legitimate number of arguments if a function doesn't need any input. In DT, all functions look like this: `FunctionName()`. Inside the parentheses is where you supply the arguments. Even if the function doesn't take any arguments, the parentheses are required. That's how DT recognizes functions among all the other stuff you can put in your code and scripts.

For example, `ManagerName()` is a function that returns the name of a manager. It requires one argument that specifies which manager's name you want to see. Like most built-in functions in DT, `ManagerName()` can also take an optional argument to specify which chart the manager's names appears in. We call that optional argument the "context". If you don't supply a context to `ManagerName()` it will use the workbook analysis parameters. `ManagerName(1)` returns the name of the first manager in the workbook parameters. `ManagerName(1,"Page 2|Style Table #1")` returns the name of the first manager in the first Style table on Page 2.

You may have noticed me using the term "returns" when I'm referring to the output of the function. I'm doing that deliberately because there's a "return" command for use in functions that tells the function what it should output. Everything that happens in a function stays in the function unless you tell it to return something. It's kind of like Vegas that way.

Here's what a function definition looks like in Dynamic Text:

```
function FunctionName(var1,var2,...)
    regular DT code that does something to the variables var1, var2, and
    so forth goes here
    return answer
end
```

This is fake code to show the format, not a real working function. We're going to write a real function later. Function definitions all start with the word "function" followed by a unique name and parentheses containing variable names for any arguments the function needs to work. Most functions use "return" to specify the answer the function is intended to provide when it's called. All functions end with the word "end".

One oddity about homemade functions is that you define them at the top of your script and use them later in the script, so your script no longer reads from top to bottom in the order of execution. The stuff between "function" and "end" will only be executed when the function is used, then the script jumps up to the definition, runs through to the "end" then, having gotten whatever value the function returns, jumps back down to the line where you used the function. That's probably about as clear as mud at this point, but an example should help.

Here's a script that doesn't do much:

```
<%  
  
function Three()  
    return 3  
end  
  
var1 = 2  
var2 = Three()  
  
Display(var1 + var2)  
  
%>
```

When you run this script the first line that gets executed is the one where `var1` is assigned. DT will skip over the function definition (the red part) until the function gets called. In the next line we use the function `Three()`, so DT will jump up to the line that says “function `Three()`” and do whatever the function definition tells it. The second line of the definition returns the number 3. The third line of the definition is “end” so DT now skips back down to the `var2` definition line with that 3 and assigns it to `var2`. Then DT goes to the `Display` line and adds 2 and 3 and displays a 5. Every time this script includes “`Three()`” DT will have to go back to the top, run everything between “function” and “end” then snap back to the line where `Three()` appeared and replace it with a 3.

A function can be as simple as this one or very complicated. The execution is the same. DT ignores the function definition until it sees a line where the function name appears, then it finds the definition, runs everything between “function” and “end”, and finally goes back and replaces the function name with the value the function returns.

Who needs functions?

Here's a simple example program that compares two variables and tells you if the first one is higher than the second one. Okay, okay, I know, it's not very useful, but it's just an example.

```
<%  
  
variable1 = 0.6  
variable2 = 1.1  
variable3 = 0.743  
variable4 = 0.234  
  
if variable1 > variable2 then  
  Display("In the first test, ")  
  Display("first is higher|")  
end  
  
if variable3 > variable4 then  
  Display("In the second test, ")  
  Display("first is higher|")  
end  
  
if variable2 > variable3 then  
  Display("In the third test, ")  
  Display("first is higher|")  
end  
  
%>
```

Here's what the output of this script will look like:

In the second test, first is higher
In the third test, first is higher

Things to notice are that I'm basically doing the same test three times in a row, just using different variables, and that the output is identical for each of the three tests except for the string that says what test is being displayed.

If you think I'm leading up to telling you there's a better way to do this, well, you're right.

We're going to make a function that does this test and tells you which variable won the comparison.

Really.

We write a function to CompareVariables()

Here's an example that has the same output as the previous one, but uses a function to do the comparison.

```
<%  
  
function CompareVariables(var1,var2,test)  
  
    if var1 > var2 then  
        return "In the "..test.." test, first is higher|"  
    end  
  
end  
  
variable1 = 0.6  
variable2 = 1.1  
variable3 = 0.743  
variable4 = 0.234  
  
CompareVariables(variable1,variable2,"first")  
  
CompareVariables(variable3,variable4,"second")  
  
CompareVariables(variable2,variable4,"third")  
  
%>
```

And here's what the output will look like:

```
In the second test, first is higher  
In the third test, first is higher
```

I know what you're thinking. "That's not any simpler looking than the other one. In fact, it's not much shorter than the first example." Those are good observations, but stick with me and I'll explain why this might be a better way to do the same work.

First, let's look at the code. You should see the function definition above the variable definitions (again, I've highlighted them in red). This function takes a pair of variables and a string telling it which test is being run as arguments.

The CompareVariables() function takes two variables and a string then compares the variables. If the first is higher, it displays a string that says "In the nth test, first is higher" where nth is whatever string you want it to use. If the second one is higher CompareVariables() does nothing.

You call a function by typing its name and putting values that you want it to act on in the parentheses as arguments:

```
CompareVariables(variable1,variable2,"first")
```

This will output the string "In the first test, first is higher" if and only if variable1 is higher than variable2. To duplicate the previous example we call it three times with three different sets of variables.

Note that I made the function name long and self-explanatory. I could have called it CV(), but next time I read this script I may not remember what that stands for. Use long names. If you get tired of typing long names whenever you use the function, copy its name from the definition line and paste it where you want. That way you know you spelled it correctly.

Great, now change the logic

Here's a hypothetical situation. Somebody you have to obey looks at your original script and says, "Great! But I'm really more interested in whether or not the first value is lower, not higher. Just switch it around and it'll be perfect!"

Well, that should be fairly simple, shouldn't it? All you need to do is change > to < and the word "higher" to "lower". Right?

Well, yes, but the way the code was originally written, you have to make that change in each if/then block and you have to remember to make six changes instead of just one.

```
<%  
  
variable1 = 0.6  
variable2 = 1.1  
variable3 = 0.743  
variable4 = 0.234  
  
if variable1 < variable2 then  
  Display("In the first test, ")  
  Display("first is lower|")  
end  
  
if variable3 < variable4 then  
  Display("In the second test, ")  
  Display("first is lower|")  
end  
  
if variable2 < variable3 then  
  Display("In the third test, ")  
  Display("first is lower|")  
end  
  
%>
```

I had to find every if/then and change the > to <. There were three. Since they're right next to each other they were all easy to find. In a real script they could be far enough apart that I might not catch them all.

Then I had to remember to find all the instances of the word "higher" and change those to "lower". There were three of those, too, and the same thing applies. I have six chances to forget to make a change and mess up my script. I highlighted all the changes I had to make in blue.

Here's what the output will look like if we made the changes correctly:

In the first test, first is lower

Edit CompareVariables()

Here's what has to be done to change the example that uses CompareVariables(). I go into the function definition at the top of the script and change one > to < and change one "higher" to "lower".

```
<%  
  
function CompareVariables(var1,var2,test)  
  
    if var1 < var2 then  
        return "In the "..test.." test, first is lower|"  
    end  
  
end  
  
variable1 = 0.6  
variable2 = 1.1  
variable3 = 0.743  
variable4 = 0.234  
  
CompareVariables(variable1,variable2,"first")  
  
CompareVariables(variable3,variable4,"second")  
  
CompareVariables(variable2,variable4,"third")  
  
%>
```

That's it. I don't have to find every time CompareVariables() gets used in the script. They all work the new way without having to mess with them.

See? It really was easier this way.

Workbook Functions:

Okay, now there's one more step that makes homemade functions make a whole lot more sense. In the example we've created, `CompareVariables()` is only useful within the block of code where it's defined. If your workbook contains other blocks of code, they will not be able to use `CompareVariables()` in its current state.

Wouldn't it be cool if there was a way for other blocks of code to take advantage of our fabulously useful function `CompareVariables()` without copying the entire thing into each place we want to use it? Well, there is.

We're going to use `Edit Dynamic Text Values for Workbook` to convert `CompareVariables()` to a function that can be used in any block of code in your workbook.

First, copy everything between "function" and "end" then find `Edit Dynamic Text Values for Workbook` in the `Edit` menu. Click on the `New` button and enter `CompareVariables` in the `Name` field. Now click on the `Edit` button next to your newly created but undefined function. This brings up an `Edit Dynamic Text` dialog just like the one you're used to. Paste. You may have noticed that you've already supplied a name, so delete `CompareVariables` from the first line so that it reads `function (var1,var2,test)".` Your `Edit` dialog should look like this:

```
function (var1,var2,test)
  if var1 < var2 then
    return "In the "..test.." test, first is lower|"
  end
end
```

Note that there must not be any begin markers (`<%`) or end markers (`%>`) in a `Workbook` function. It's made to go inside a block of code. Click on the `OK` button in the `Edit` dialog then click on the `OK` in the `Dynamic Text Values` dialog. That looks like nothing happened, doesn't it? You just created a `Workbook` function, but you're not using it yet.

Go back to that block of code you modified on the `Edits To Sample Function` page and double-click on it to bring up the `Edit` dialog. Now cut out the function (everything between "function" and "end") and hit `OK`. If you've done everything correctly it should look again as if nothing happened. The block should come up with exactly the same output, but now it's using the `Workbook` version of the function `CompareVariables()`. Now you can use `CompareVariables()` anywhere in this workbook as if it was one of `StyleADVISOR`'s built-in functions.