

# What to do when your Dynamic Text doesn't work

## (Debugging<sup>1</sup> Dynamic Text)

First take note of *how* it doesn't work. Do you get the wrong answer, no answer, or an error message?

If you get the wrong answer the problem is likely to be with your logic. These can be the hardest to figure out because the code may be doing exactly what you told it to do.

If you get no answer it could be a logic error, but it could also be a problem with the data. The most common of these is attempting to mix types<sup>2</sup> of data that don't mix. Dynamic Text tries to be smart about mixing types but doesn't always come up with the type you wanted.

If you get an error message you could have a data problem but chances are good that you have a syntax error. A syntax error means that you've given StyleADVISOR an instruction that it doesn't understand. Punctuation, capitalization, incorrect number of arguments<sup>3</sup>...any of these can result in an error. Dynamic Text error messages include a line number to help you look for the problem. This appears in parentheses at the end of the message. Numbers include blank lines, so count carefully.

### Ways to debug in Dynamic Text.

Some basics about the Edit Dynamic Text dialog in case you're just starting out. This dialog can be resized. If your script uses the Value() function chances are some of your lines are wider than the default dialog. Make it as big as you can and you'll be happier. Next, remember that this dialog has an UNDO button. This makes it easy to delete something to see what happens, then to put it back. This dialog also has a CANCEL button. If you've completely messed up your script in your attempts to debug, just hit CANCEL and it'll revert to the way it was before you started messing with it. On some monitors the colors in the Edit dialog are difficult to distinguish, but begin and end markers (<%, %>) display in blue, functions and variables in purple, strings and arguments in black, and comments in green. When your cursor is within a function you'll get a brief description of that function in the Help section at the bottom right. It's pretty brief, but can often save you a trip to the manual if you just need a reminder of how something works. The most useful area for debugging is the Current Value section at the bottom. This displays the results of the script. It can only show two lines, but most scripts only output one line. If your script returns more than two lines the scroll arrows will activate to let you see the rest of your results.

Unfortunately, sometimes an error will appear to StyleADVISOR to occur far away from the actual mistake so the line number (if there is one) may not always point to the exact location of the problem. For example, if you forget to include a close quote at the end of a string, everything after that line will look to StyleADVISOR like part of the string...including the end marker (%>) so your code will look like an incomplete block and the alert you'll get is "Unexpected "<%>" found." In a situation like this, look for things that aren't the color you expected them to be. If the end marker isn't blue, chances are you have an unpaired quote somewhere.

If you suspect an unpaired quote, a good place to start looking is at concatenated<sup>4</sup> strings. The syntax for concatenating strings can result in a line that's difficult to read, and thus easy to do incorrectly.

You may have noticed by now that there is no way to display line numbers in the Edit Dynamic Text dialog. This is a very good reason to try to keep your scripts short. I can probably find the fifth line from the top of a script, but how am I going to be sure I counted correctly if I'm looking for the fiftieth?

## Simplifying code

A good debugging practice is to simplify your code so you don't have to wade through whole blocks of stuff you already know isn't involved in the problem. Say you added new code to a working block of Dynamic Text and now it doesn't work. You already know the old stuff works.

In a situation like this you can usually comment<sup>5</sup> out stuff that is not involved. To comment something out means to add two hyphens at the beginning of a line to render it non-functional. Remember that if you comment out the first line of an "If" statement you'll have to comment out the rest of the lines down to the "End" or you'll cause a new syntax error. For example, to comment out the following snippet of code:

```
if Today() == "March 17, 2006" then
  Display("Happy St. Patrick's Day!")
end
```

This won't be enough:

```
--if Today() == "March 17, 2006" then
  Display("Happy St. Patrick's Day!")
end
```

You'll need to do this to properly comment this out:

```
--if Today() == "March 17, 2006" then
  --Display("Happy St. Patrick's Day!")
--end
```

Another technique is to create a new block of Dynamic Text and paste the new code into that and do your debugging there. Once you get it working you can paste it back into your old code, confident that it'll work. Then you can delete the scratch block you used for debugging.

## Displaying more information

The technique you'll probably use more than any other, though, involves making your code display extra information that will help you understand what it's up to. This stuff will eventually be deleted once you get your code working the way you want.

The simplest example that I use all the time is to put a Display() line inside an If/Then statement to see if my code is getting in there when I expect it. If I wanted to see if the following block, which doesn't normally display anything, was being executed

```
if variable1 < variable2 then
  variable1 = variable1 + 1
end
```

I could insert a Display() function to output a string saying "This comes from inside the If statement" like this:

```
if variable1 < variable2 then
  Display("This comes from inside the If statement")
  variable1 = variable1 + 1
end
```

Then look at the output at the bottom of the Edit Dynamic Text dialog to see if that string appears. If it doesn't, then my test is failing and I can try to find out why variable1 is not smaller than variable2. The next step might be to see what values variable1 and variable2 have just before doing the test:

```
Display(variable1)
Display(variable2)
if variable1 < variable2 then
  Display("This comes from inside the If statement")
  variable1 = variable1 + 1
end
```

This will result in the current values of variable1 and variable2 showing up in your output right next to each other. If, say, variable1 is equal to 2 and variable2 is equal to 1 you'll see a 21 somewhere in your output. Since it's possible that one or the other might be equal to 21 you might want to add some text to describe what you're looking at like this:

```
Display("variable1 equals: "..variable1.." variable2 equals: "..variable2)
if variable1 < variable2 then
  Display("This comes from inside the If statement")
  variable1 = variable1 + 1
end
```

This will output the message "variable1 equals: 2 variable2 equals: 1" and will make a whole lot more sense.

## Hardcoding Variables

Usually you'll use variables because you want the values in them to vary. If you want to test your logic with a variety of sample cases to be sure it'll do the right thing, it's usually easier to just set the variable to the test value you want rather than try to simulate all the conditions your script is likely to encounter.

For example, if you have a script that needs to do something different depending on a value from a Scan Table you'd have to find managers with all the appropriate test values (positive, negative, zero, N/A, whatever) and change managers several times to be sure the script is working for all cases.

```
FromScan = Value(1,"Cumulative Return","Scan #1|Data Source")
if FromScan < 0 then
  Display("Losing")
elseif FromScan == 0 then
  Display("Broke Even")
else
  Display("Made Money")
end
```

There are three possible classes of values for the variable FromScan that you need to check. You could test this by identifying managers with the appropriate Cumulative Return for the time period and change managers three times...or...you could comment out the line that gets the value and just set FromScan to -1 and see if it does the right thing, then change FromScan to 0 and see what happens, then change FromScan to 1 and see if it works in that case. Once you're sure the logic is working you can delete the hardcoded value and uncomment the line that sets it with Value().

```
--FromScan = Value(1,"Cumulative Return","Scan #1|Data Source")
FromScan = -1
if FromScan < 0 then
  Display("Losing")
elseif FromScan == 0 then
```

```

    Display("Broke Even")
else
    Display("Made Money")
end

```

Note that this snippet of code doesn't have a test for "N/A" (even though I mentioned it as a possibility in an earlier paragraph) so a missing value would pass the test for displaying "Made Money". That's what I mean by a logic error. The code does exactly what I asked it to do, but I didn't plan for a missing value in the Scan. If your script makes assumptions about something you can't control (like whether or not there's a value in the Scan) there's a good chance that your logic won't always be right. It's best not to assume anything, not even if it's obvious. Don't assume, for example, that percentages will always be between 0 and 100 inclusive if you're not setting them yourself. Put in a test for invalid input. It's a computer. It doesn't mind doing stupid tests that usually don't catch a problem, but your users will be glad you thought of it in the rare cases where the input is wacky.

## Checking Type

Mixing types of data in calculations is easy to do by mistake. It's also possible for StyleADVISOR to fail to assign the correct (or expected) type of a value. StyleADVISOR tries to be clever about assigning types but won't always set the type the way you want. In most cases, if you have a string that looks like a number you can add it to a number and get the expected answer, but sometimes the result can be surprising or even an error.

If you get an alert complaining that you've tried to do an operation on an illegal type the first thing to do is figure out what types of data you're operating on.

```

if variable1 < variable2 then
    variable1 = variable1 + 1
end

```

If this snippet throws an alert that mentions type, use the Type() function to display what type StyleADVISOR believes variable1 and variable2 are set to:

```

Type(variable1)
Type(variable2)
if variable1 < variable2 then
    variable1 = variable1 + 1
end

```

This will cause your code to output a string with the type for each variable, right next to each other...something like "NumberNumber" or "NumberString" or "NumberRecordSet" or even "NumberNil". As before, you can use Display() to make your debugging statements more legible like this:

```

Display("v1 is a "..Type(variable1).. " v2 is a "..Type(variable2))
if variable1 < variable2 then
    variable1 = variable1 + 1
end

```

Now you can tell if you're mixing types. If the type of either variable is Nil that means its value is unset. Either you didn't set it, or you spelled the variable name wrong, or the value you got from elsewhere wasn't set. If you didn't set it, you need to figure out why not. Did you try to set it inside an If/Then statement that may not have worked the way you expected? Now you're back to adding Display() lines to figure out if you're passing the test and if not, why not. If you used Value() to snag a value from a table and got back a Nil, you need to put in a test for Nil before trying to do math on the value.

If the types are a mixture of Number and String you can override StyleADVISOR's guess about what type they should be by using the Number() function to change the String into a Number. Be careful if you do this because forcing a

string that is made up of letters rather than numbers to be of type Number can give you surprising results. If you're sure variable2 is supposed to be a Number and Type() says it's a String here's what you can do:

```
if variable1 < Number(variable2) then
  variable1 = variable1 + 1
end
```

If you're using the Excel() or Query() functions you'll probably encounter variables with the type "Record Set". If a Record Set only contains one record you can often get away with treating it as if it was a number and do math with it, but it's safest to index<sup>6</sup> values out of Record Sets.

```
Result = Excel("file.xls", "SELECT [Std Dev] FROM [ManagerData]")
Display(Result * 100)
```

Might work if there's only one answer that matches the Select, but because the variable Result is going to have the type of Record or Record Set it'd be safer to be more explicit about what you want to multiply by 100 in the next line by indexing the first item out of that Record Set like this:

```
Result = Excel("file.xls", "SELECT [Std Dev] FROM [ManagerData]")
Display(Result[1] * 100)
```

## Checking Length

When you're dealing with the results of the Excel() or Query() functions there is another way to get more information out of your code and that's using the Length() function. When used on a string (for example) the Length() function returns the number of characters in the string, but when it's used on a Record Set, Length() returns information about how many rows the Record Set contains and when it's used on a row Length() returns the number of columns in that row.

In the previous example we assumed that the variable Result would have only one value:

```
Result = Excel("file.xls", "SELECT [Std Dev] FROM [ManagerData]")
```

But if we'd written it like this:

```
Result = Excel("file.xls", "SELECT * FROM [ManagerData]")
```

now we're asking for the entire contents of the named region ManagerData, which may be a single value or a number of rows and columns. In this case, the type of Result will be Record Set and the length will be the number of data rows in the named region. If I'm getting weird results from the above line I'd probably insert a line below it to check the length of the Record Set to be sure I was getting all the data I expected:

```
Result = Excel("file.xls", "SELECT * FROM [ManagerData]")
Display("Result is this long: "..Length(Result))
```

If I get an answer like "46" when I was expecting something on the order of five or six rows I'd insert a check for type because this value could be the length of an error message.

```
Result = Excel("file.xls", "SELECT * FROM [ManagerData]")
Display("Result length: "..Length(Result).." and type:"..Type(Result))
```

If Result turned out to have the type of Error or String then I'd just display it to see what the complaint is. If the length was reasonable I'd check the spreadsheet for things like the extent of the named region to be sure all the data I wanted was in the region I asked for.

I'd also check to see how many columns were in each row returned by my script.

```
Result = Excel("file.xls", "SELECT * FROM [ManagerData]")
Display("Row 1 is this long: "..Length(Result[1]))
```

This should return the number of columns in the first row. If that's only returning 1 and I know there's more than one column, again I'd look at the spreadsheet to see if the region I asked for covers all the data I want.

Okay, suppose I get the amount of rows and columns I want but I still get an error when I try to do some math with the results. Here's a problem we've encountered with Excel data. Suppose the following snippet of code gives me a mixed type alert, even though I've already determined that the data I'm getting is the correct shape (rows and columns) and the value of the specific record I'm using looks correct.

```
Result = Excel("file.xls", "SELECT * FROM [ManagerData]")
MyPercent = Result[10][3] * 100
```

What should be going on here is that I'm multiplying the third column of the tenth row in my named region by 100, but StyleADVISOR is insisting that I'm mixing types. I've already displayed the value of row 10 column 3 and it sure looks like a number. In this case, take a look at the first five or so rows of column 3 in the named region of the spreadsheet. StyleADVISOR doesn't check the type of every cell it extracts from a spreadsheet when you use the Excel() function. It reads the first few entries and sets the type for the entire column based on what it finds there. If a column has a mixture of numbers and strings and the top of the column is mostly strings, chances are StyleADVISOR will set all values extracted from that column to the type of String, even if Excel says most of them are numbers. Remember that N/A will look like a string to StyleADVISOR. If you don't have control over the spreadsheet and so don't know if there may be missing values at the top that look like strings, you can force the result to be a number using the Number() function like this:

```
Result = Excel("file.xls", "SELECT * FROM [ManagerData]")
MyPercent = Number(Result[10][3]) * 100
```

If you create or can edit the spreadsheet you can take steps to be sure that the first values are the type you want by reordering the rows, or by inserting bogus rows at the top with the correct types in all the columns then rewriting your script to ignore those added rows.

## Debugging Workbook Functions<sup>7</sup>

If the script you're debugging uses Workbook functions and you suspect the problem is in the function as opposed to being in the script that uses the function you'll want to move the function into the script before trying to debug it. You can debug a Workbook function without doing this, but that way lies madness. It's slow and difficult and accident-prone. The alternative may sound complex, but it's really the right way to do it.

```
<%
PNum = Page()
if IsOdd(PNum) then
  Display("Page number is odd")
end
%>
```

This script is using a Workbook function called IsOdd() to decide if a page number is odd or not. If you think there's a bug in IsOdd() here's the procedure:

First do Edit|Edit Dynamic Text for Workbook and find the entry for IsOdd(). Poke the Edit button to bring up the edit dialog and reveal the script. Note that the preview at the bottom of the dialog just displays the code instead of the result. You can't debug in this dialog. You need to get this script somewhere useful. Here's what IsOdd looks like:

```

function(nmr)
  Odd = nmr / 2
  Odd = Find(Odd, ".5")
  if Odd then
    return true
  end
end

```

Select everything in the script and copy, then dismiss the edit dialog and the DT for Workbook dialog.

Double-click on the script that was using IsOdd() to bring up the dialog. Create a new line above the first executable line of the script (in the example, the one that says "PNum = Page()") and paste.

Put your cursor between the word "function" and the open parenthesis "(" and type a space then "IsOdd" so that the line looks like this:

```

function IsOdd(nmr)

```

The entire scripts should look like this now:

```

<%
function IsOdd(nmr)
  Odd = nmr / 2
  Odd = Find(Odd, ".5")
  if Odd then
    return true
  end
end
PNum = Page()
if IsOdd(PNum) then
  Display("Page number is odd")
end
%>

```

Now you have a working function within your script that does exactly the same thing as the Workbook version. A function in a script takes precedence over a Workbook function with the same name, so changes made here will show up in the preview area of the Edit dialog. Now you can use all the regular debugging techniques on the function.

Once you have it working the way you like, you need to replace the Workbook function with your corrected version.

First copy all the lines between "function" and the "end" at the end of the function. You can delete them from the script at this time if you want. The original bug should show up when you do because now you're using the Workbook version of IsOdd() which hasn't been fixed yet.

Edit|Edit Dynamic Text for Workbook and hit the Edit button beside IsOdd. Replace all the text with your new version. Remember to delete the IsOdd from the first line so that it looks the way it originally did:

```

function(nmr)

```

OK out of this Edit dialog and OK out of the DT for Workbook dialog and in a few moments your new and improved IsOdd() function should be applied to the entire workbook.

## Notes \_\_\_\_\_

<sup>1</sup> **Debugging** means figuring out what's wrong with a program. Often debugging involves modifying the program to eliminate unnecessary distractions or to cause it to display extra information.

<sup>2</sup> **Data types**, oddly enough, are types of data. The most common types that StyleADVISOR recognizes are:

String	Text like a manager name is considered a String...anything between quotation marks is a String, even if it looks like a number or a punctuation mark.
Boolean	A Boolean has a value of either true or false, or also 1 or 0 respectively. That's all it can do but you'd be surprised how often that comes in handy.
Nil	Nil is a missing or unset value. If you check a variable's type without ever assigning anything to it you should get "Nil" as the type. If you set a variable to a value in a Scan (for example) that doesn't exist, it will be Nil.
Number	This one is pretty easy to identify. If it looks like a number and it isn't a string, it's a Number.
Date	Dates in a variety of formats are a special type. If you use a function like Today() the answer you get back will be a Date.
Time Period	Something expressed in a number of periods (like what you get from the AnalysisLength() function) is a Time Period.
Currency	The results of the Currency() and CurrencyName() functions are given the Currency type.
Record Set	The results of a Query() or Excel() function will be a Record Set.
Record	An individual column in a Record Set is a Record.
Error	An error message returned by a failed Query() or Excel() query may be given the special Error type

<sup>3</sup> **Arguments**. The data between the parentheses that you provide to a function to tell it what to do are called arguments. Arguments in Dynamic Text are separated by commas. Many Dynamic Text functions take no arguments (like Today()). Most only take one (like ManagerName(1)).

<sup>4</sup> **Concatenation** is the process of joining two things to make one. Typical use of concatenation is to merge two or more strings into one. If a function takes one string as an argument and you want to assemble that argument out of two different strings you'll need to concatenate them together. For example, if you want to display a sentence that includes the manager name using the Display() function you can concatenate the results of the ManagerName() function into your sentence using the string concatenate operator ".." like this:

```
Display("My first manager's name is "..ManagerName(1).."")
```

This is the equivalent of

```
Display("My first manager's name is Sample Fund.")
```

<sup>5</sup> **Comments** are non-functional lines in a program that are added to document how the code works. If you're like me, you may not remember from one day to the next what you were trying to do in a program, so adding comments that

---

explain what you're up to isn't just polite to other people trying to read your program, but essential to being able to maintain it yourself. Any text after a pair of hyphens is a comment, even if it looks like executable code.

<sup>6</sup> **Indexing:** A Record Set, as the name implies is a set of records. It can be a single value, a row of values with items in different columns, or several rows and columns. You get at individual items in a Record Set by indexing...supplying row and column numbers to designate the item whose value you want. For a Record Set named "Result" that has only one row you can get the item you want by following the variable name with a number in brackets. Result[1] indicates that you want the first item, Result[2] is the second item and so on. If the Record Set has more than one row you'll have to include row and column numbers like this: Result[1][2] indicates the second item in the first row. Result[2][15] indicates the fifteenth item in the second row.

<sup>7</sup> **Workbook Functions:** You can define your own functions within a block of Dynamic Text. If you have a calculation you're going to do repeatedly, rather than repeat the code each time you want to do the calculation you can define a function that does that calculation then just call it by name later in the script. If you want to use the same function in more than one script you need to define the function for the entire workbook using "Edit Dynamic Text for Workbook." Functions defined there are available anywhere in the workbook. The syntax for regular functions within a script and functions defined for use by the entire workbook (called Workbook Functions in this document) is slightly different. In a script a function definition looks like this:

```
function NameOfFunction(arguments)
    code
end
```

In a Workbook Function, the name is defined separately. The dialog has a space for the script and a space for the function name. Because of this, the script itself should leave out the function name:

```
function(arguments)
    code
end
```