Apache Open Office

Version 4.1

Calc Guide

AOO Documentation Team

Chapter 12

Calc Macros

Automating repetitive tasks

Copyright

This document is Copyright © 2024 by the Apache Software Foundation®. You may distribute it and/or modify it under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/3.0/), version 3.0 or later.

All trademarks within this guide belong to their legitimate owners.

Acknowledgments

This document is updated from a previous version by OOoAuthors.

Feedback

Please direct any comments or suggestions about this document to: doc@openoffice.apache.org

Publication date and software version

Published <date>. Based on Apache OpenOffice 4.1.

Note for Mac Users

Some keystrokes and menu items are different on a Mac from those used in Windows and Linux. The table below gives some common substitutions for the instructions in this chapter. For a more detailed list, see the application Help.

Windows/Linux	***Mac equivalent***	***Effect***
Tools > Options menu selection	**OpenOffice > Preferences**	Access setup options
Right-click	*Control+click*	Open context menu
Ctrl (Control)	⌘ (Command)	Used with other keys
F5	*Shift+⌘+F5*	Open the Navigator
F11	*⌘+T*	Open Styles & Formatting window

Contents

Chapter 12
Calc Macros....1

Copyright....2
Note for Mac users....2
Introduction....4
The OpenOffice Basic Language....5
Using the macro recorder....7
Write your own functions....11
  Using a macro as a function....13
  Passing arguments to a macro....16
  Arguments are passed as values....18
  Writing macros that act like built-in functions....18
Accessing cells directly....18
Sorting....20
Conclusion....21

Introduction

A macro is a saved sequence of instructions that have been stored for later use. An example of a simple macro is one that “types” your address. OpenOffice macros are very flexible, allowing automation of a range of tasks, from simple to very complex. In the right situation, a macro can improve both the speed and accuracy of using a spreadsheet. It is important to remember, however, that Calc includes powerful tools like Pivot Tables and Database Ranges. It is wise to investigate whether a built in tool can do what you need before you decide on making a macro.

Macros are especially useful to repeat a task the same way over and over again. This chapter briefly discusses common problems related to macro programming using Calc. Macros can be written in several programming languages, but the most common one is a language called Basic. Before we get into the details of making Calc macros, we will very briefly explain some features of the OpenOffice Basic language.

The OpenOffice Basic Language

OpenOffice Basic is a simple programming language, both in its ease of use and in its relatively small number of features. It is still far too large a topic to cover in any detail here. An entire top level section of the Help documentation is devoted to it, just as there are sections for Writer, Calc and the other major applications. There is also a Basic Programming Guide linked at the end of this chapter. Here, we will very briefly mention a few aspects of Basic to help you understand the macros shown in the rest of this chapter.

Basic was develped in 1964, and is still used today. It allows a programmer to:

- define the type of data being manipulated. A certain value might be defined as an integer or as set of three texts. Such definitions make code easier to read, because the intent is clearer, and prevent misleading results due to bad inputs.
- define what the manipulation will consist of. A macro may do calculations in a spreadsheet, search text documents for certain words, save copies of the document, or almost anything else you can do with a document manually.
- define what conditions or circumstances will trigger the manipulation. Basic allows the code to take different actions depending on user input or on the current content of a document.

The table below shows a few of the keywords you will find in macros. Some of them are discussed in more detail later in this chapter.

Basic	Means
rem	a remark, that is,internal documentation
dim	sets up, that is **dim**ensions, the extent or the type of the item being established
double	The double data type is **capable of holding 64 bits of decimal numbers or floating points**.
integer	The INTEGER data type **stores whole numbers that range from -2,147,483,647 to 2,147,483,647 for 9 or 10 digits of precision**. An INTEGER value is typically used for quantities like counts.
for	A "For" Loop is **used to repeat a specific block of code a known number of times**. For example, if we want to check the grade of every student in a class, we loop our program instructions to start from 1 and to complete when the correct number is reached.
sub, end sub	indicators for the beginning and end of a subroutine, a part of a macro.

It's not necessary for users to be proficient in Basic, or in using it to create macros like those presented later in this chapter. But macros rely on Basic, and on some of the mechanisms it offers. Here is a macro made with the macro recorder. Don't worry about what it does.

```
sub EnterMyName
rem ------------------------------------------------------------
rem define variables
dim document   as object
dim dispatcher as object
rem ------------------------------------------------------------
rem get access to the document
document   = ThisComponent.CurrentController.Frame
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

rem ------------------------------------------------------------
dim args1(0) as new com.sun.star.beans.PropertyValue
args1(0).Name = "Text"
args1(0).Value = "Andrew Pitonyak"

dispatcher.executeDispatch(document, ".uno:InsertText", "", 0, args1())
end sub
```

Notice first that it begins with sub and ends with *end* sub. From the table above, you can learn that the lines that start with *rem* are documentation and the lines that start with *dim* define the extent or type of a variable. Let's focus on the variable *document* that will represent a document. First that variable is defined to be of the type *object.*

That means that the variable will not contain just a number or text but something more complicated, in this case a whole document.

```
dim document   as object
```

Slightly later in the code, there is a comment explaining that the next line assigns something (an object) to the *document* variable that give the macro access to the OpenOffice document and then thee is the line that does that assignment.

```
rem get access to the document
document   = ThisComponent.CurrentController.Frame
```

The final line of the code acts on the *document* variable, inserting some text.

```
dispatcher.executeDispatch(document, ".uno:InsertContents", "", 0, args1())
```

If you are new to programming, those steps are likely to look complicated and confusing. The basic ideas are simple. You define what kind of thing a variable will be, you assign a value to it with something that looks like *variable = Something*, and you may then do things with that variable using functions that appear as a function name followed by parentheses, like *dispatcher.executeDispatch()*. (Not everything that has parentheses after it is a function. It might be a collection of values, an array, such as the args1() variable in the code above.)

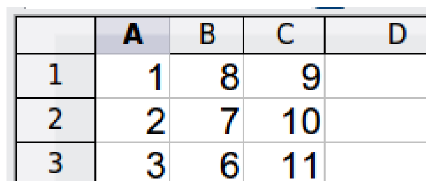
Basic offers two other programming techniques - conditioal statements such as IF NOT, which carry out actions based upon the passing or failing of a test, and loops like FOR – programming instructions that ensure the repetition of an action or actions.

In many scenarios, users don't need such tools - the macro recorder will be more than sufficient. The recorder easily handles what Merriam-Webster defines as a macro – a single computer instruction that stands for a sequence of operations.

Using the Macro Recorder

Chapter 12 of the *Getting Started* guide (Getting Started with Macros) provides a basis for understanding the general macro capabilities in OpenOffice using the macro recorder. An example is shown here without the explanations in the *Getting Started* guide. The following steps create a macro that performs paste special with multiply.

1) Open a new spreadsheet.
2) Enter numbers into a sheet.



	A	B	C	D
1	1	8	9	
2	2	7	10	
3	3	6	11	

Figure 1: Enter numbers

3) Select cell A3, which contains the number 3, and copy the value to the clipboard.
4) Select the range A1:C3.
5) Use **Tools > Macros > Record Macro** to start the macro recorder. The Record Macro dialog is displayed with a Stop Recording button.

	A	B	C	D
1	1	8	9	Record Ma...
2	2	7	10	
3	3	6	11	Stop Recording

Figure 2: Stop Recording button

6) Use **Edit > Paste Special** to open the Paste Special dialog (Figure 3).
7) Set the operation to **Multiply** and click **OK**. The cells are now multiplied by 3 (Figure 4).
8) Click **Stop Recording** to stop the macro recorder. The OpenOffice Basic Macros dialog (Figure 5) opens.

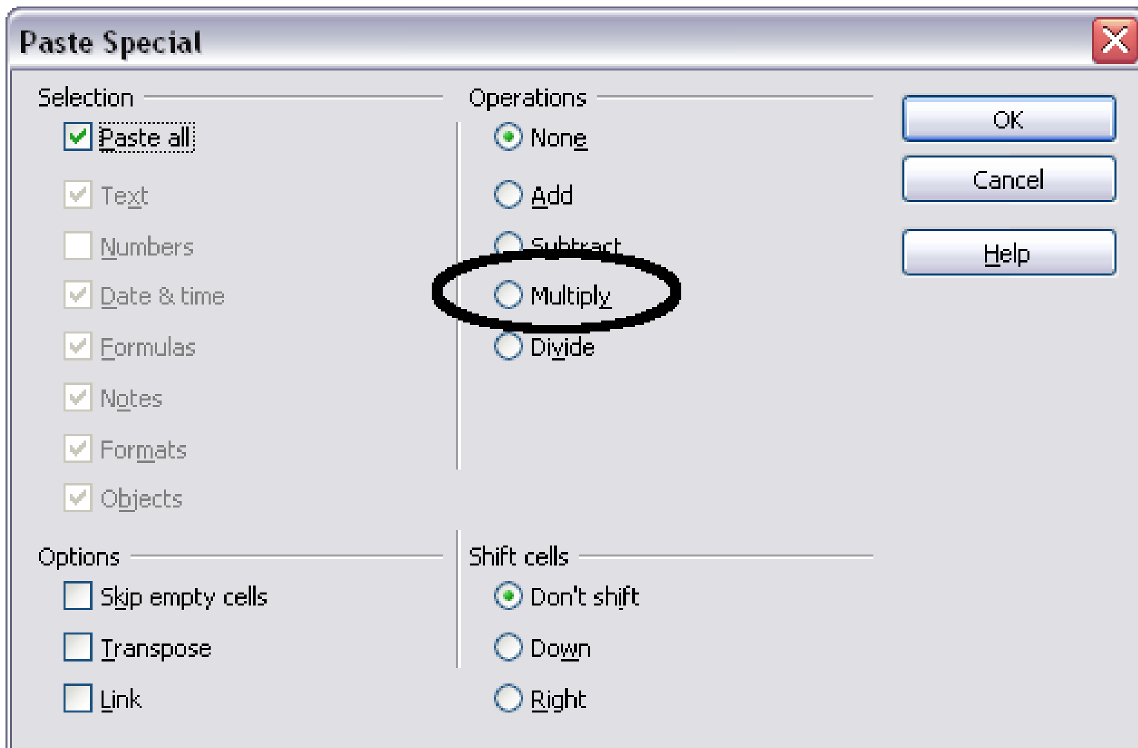
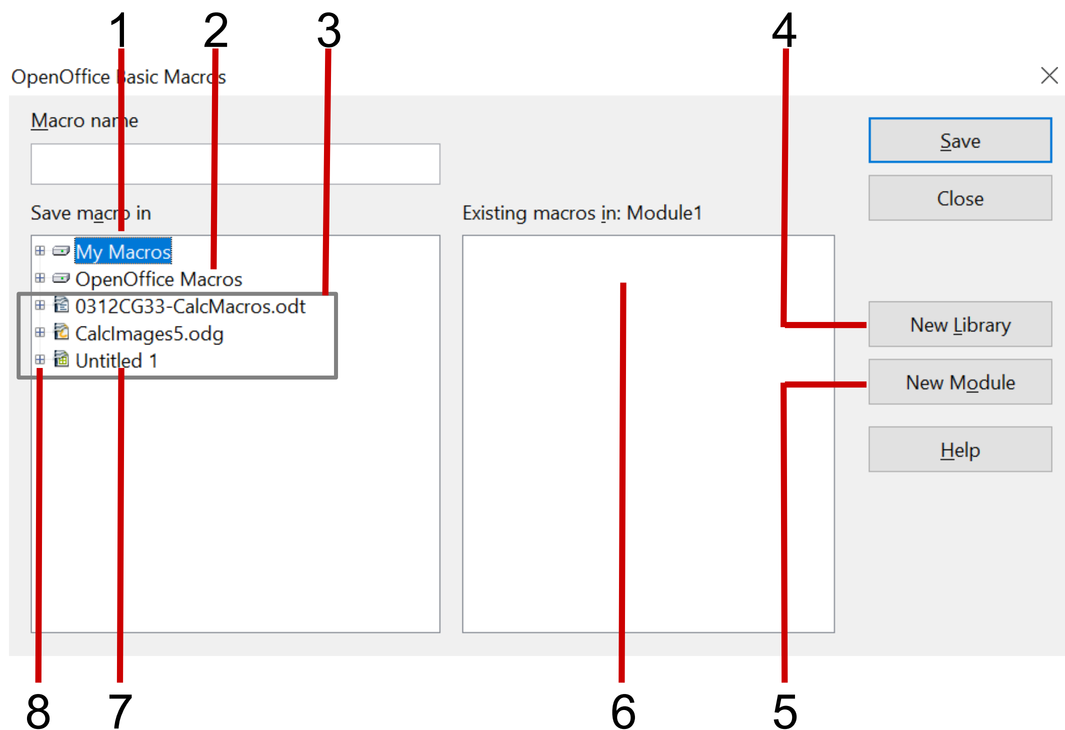


Figure 3: Paste Special dialog

	A	B	C	D
1	3	24	27	Record Ma...
2	6	21	30	
3	9	18	33	Stop Recording

Figure 4: Cells multiplied by 3

9) Select the current document (see Figure 5). For this example, the current Calc document is *Untitled 1*. Existing documents show a library named Standard. This library is not created until the document is saved or the library is needed, so at this point your new document does not contain a library. You can create a new library to contain the macro, but this is not necessary.



1 My Macros

2 OpenOffice Macros

3 Open documents

4 Create new library

5 Create new module in library

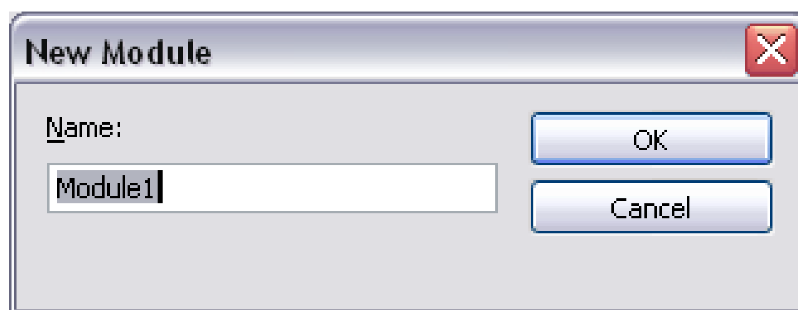
6 Macros in selected library

7 Current document

8 Expand/collapse list

Figure 5: Parts of the OpenOffice Basic Macros dialog

10) Click **New Module**. If no libraries exist, then the Standard library is automatically created and used. In the New Module dialog, type a name for the new module or leave the name as Module1.



11) Click **OK** to create a new module named Module1. Select the newly created Module1, type `PasteMultiply` in the *Macro name* box at the upper left, and click **Save**. (See Figure 6.)

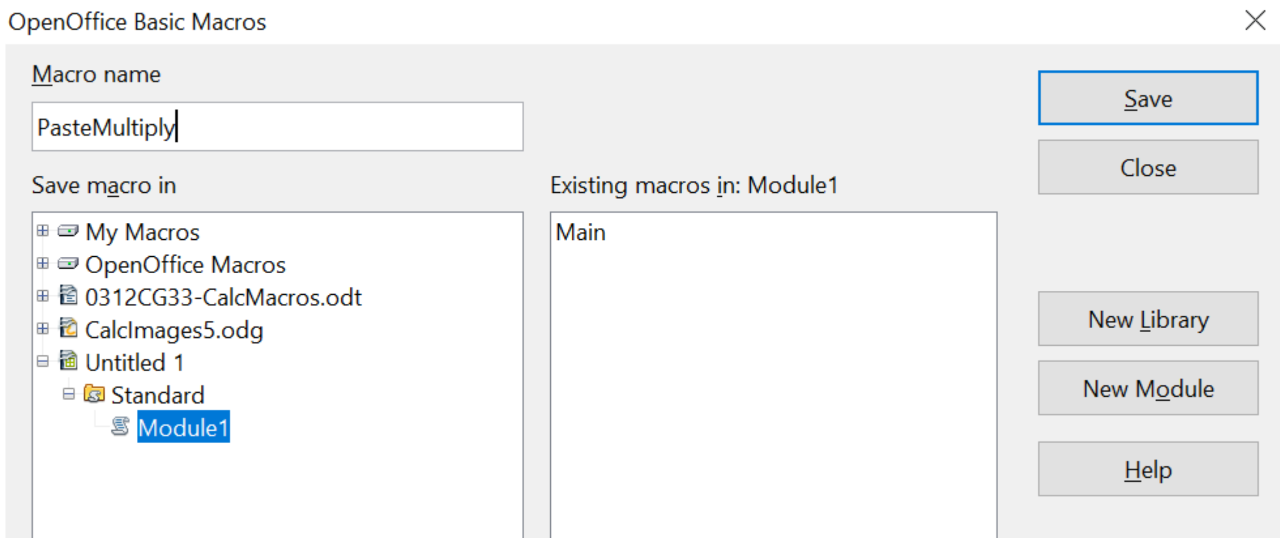


Figure 6: Select the module and name the macro

The created macro is saved in Module1 of the Standard library in the *Untitled 1* document. Listing 1 shows the contents of the macro.

Listing 1. Paste special with multiply.

```
sub PasteMultiply
   rem --------------------------------------------------------------
   rem define variables
   dim document   as object
   dim dispatcher as object
   rem --------------------------------------------------------------
   rem get access to the document
   document   = ThisComponent.CurrentController.Frame
   dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

   rem --------------------------------------------------------------
   dim args1(5) as new com.sun.star.beans.PropertyValue
   args1(0).Name = "Flags"
   args1(0).Value = "A"
   args1(1).Name = "FormulaCommand"
   args1(1).Value = 3
   args1(2).Name = "SkipEmptyCells"
   args1(2).Value = false
   args1(3).Name = "Transpose"
   args1(3).Value = false
   args1(4).Name = "AsLink"
   args1(4).Value = false
   args1(5).Name = "MoveMode"
   args1(5).Value = 4

   dispatcher.executeDispatch(document, ".uno:InsertContents", "", 0,
   args1())
end sub
```

More detail on recording macros is provided in Chapter 12 (Getting Started with Macros) in the *Getting Started* guide; we recommend you read it if you have not already done so. More detail is also provided in the following sections, but not as related to recording macros.

Write Your Own Functions

Calc can consider and execute macros as Calc functions. Use the following steps to create a simple macro.

1) Create a new Calc document named `CalcTestMacros.ods`.
2) Use **Tools > Macros > Organize Macros > OpenOffice Basic** to open the OpenOffice Basic Macros dialog. The *Macro from* box lists available macro library containers including currently open documents. *My Macros* contains macros that you write or add to OpenOffice. *OpenOffice Macros* contains macros included with OpenOffice and should not be changed.

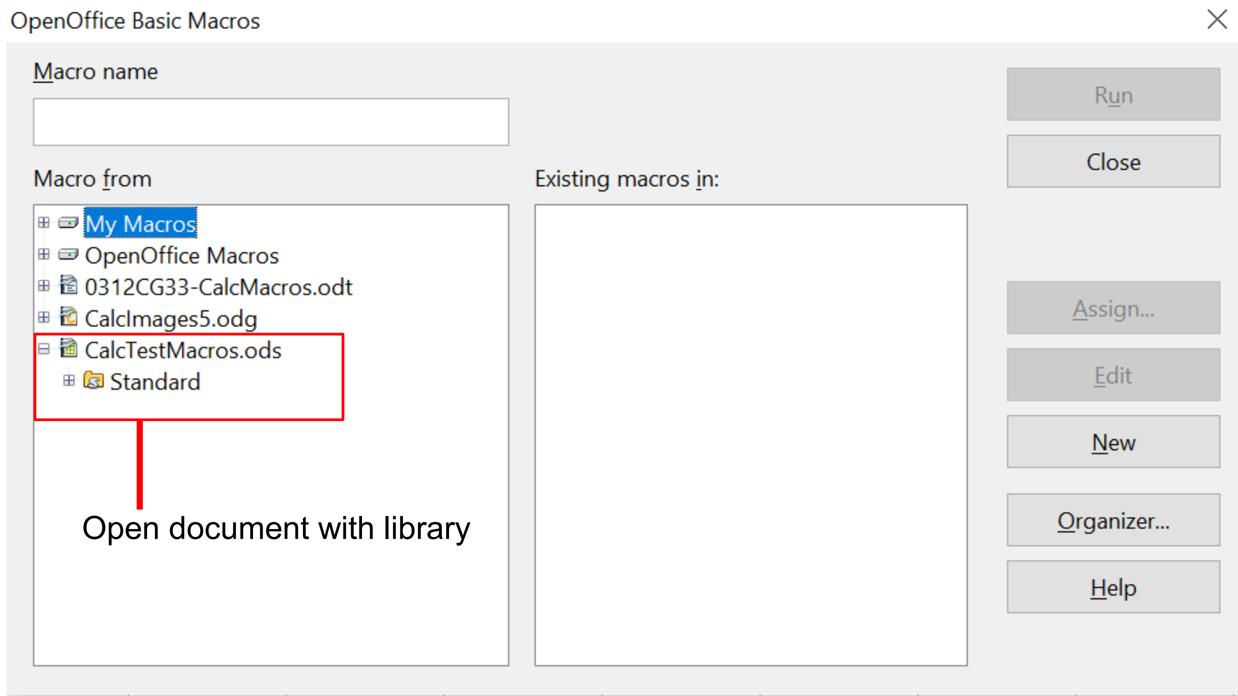


Figure 7: OpenOffice Basic Macros dialog

3) Click **Organizer** to open the Basic Macro Organizer dialog (Figure 8). On the Libraries tab, select the document to contain the macro.

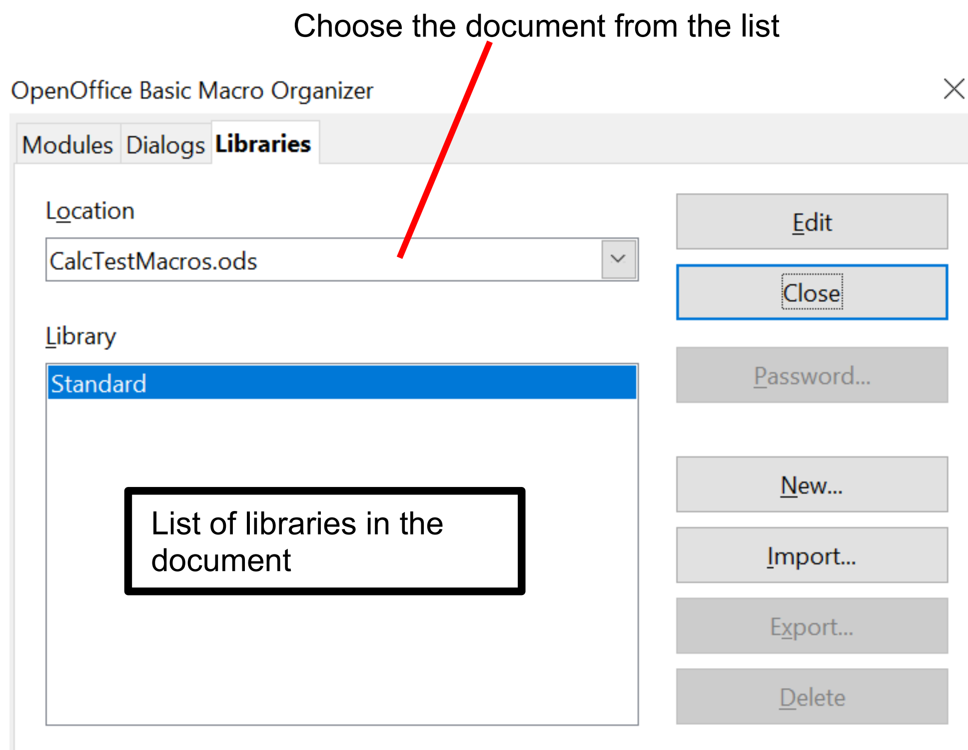


Figure 8: OpenOffice Basic Macro Organizer

4) Click **New** to open the New Library dialog.

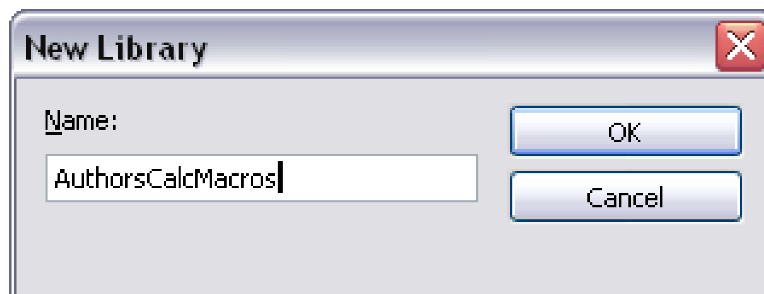


Figure 9: New Library dialog

5) Enter a descriptive library name (such as AuthorsCalcMacros) and click **OK** to create the library. The new library name is shown the library list, but the dialog may show only a portion of the name.

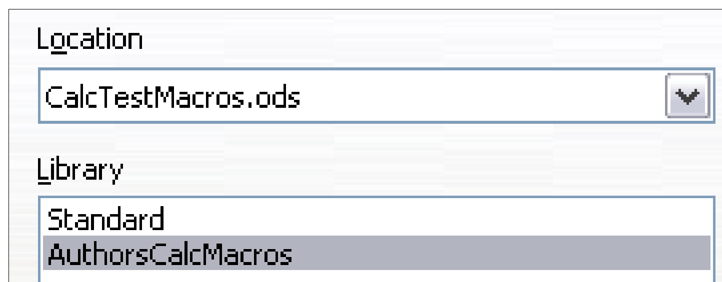


Figure 10: The library is shown in the organizer

6) Select AuthorsCalcMacros and click **Edit** to edit the library. Calc automatically creates a module named Module1 and a macro named Main.

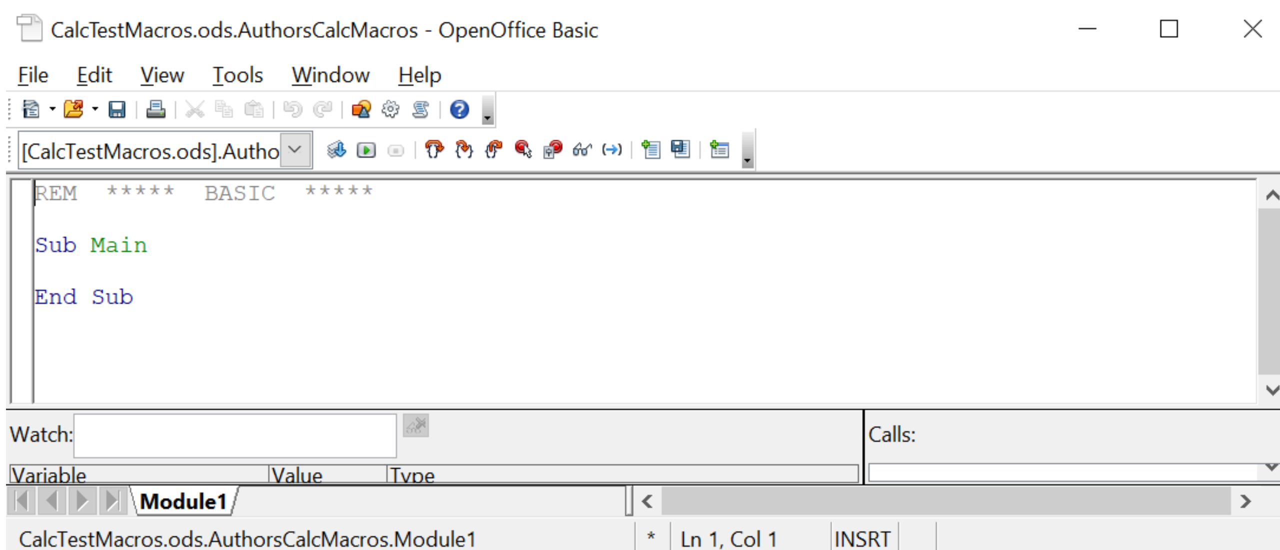


Figure 11: Basic Integrated Development Environment (IDE)

7) Modify the code so that it is the same as that shown in Listing 2. The important addition is the creation of the `NumberFive` function, which returns the number five. The `Option Explicit` statement forces all variables to be declared before they are used. If Option Explicit is omitted, variables are automatically defined at first use as type Variant.
8) Save the modified Module1.

Listing 2. Function that returns five.

```
REM  *****  BASIC  *****
Option Explicit

Sub Main

End Sub

Function NumberFive()
  NumberFive = 5
End Function
```

Using a Macro as a Function

Using the newly created Calc document `CalcTestMacros.ods`, enter the formula `=NumberFive()` (see Figure 12). Calc finds the macro and calls it.

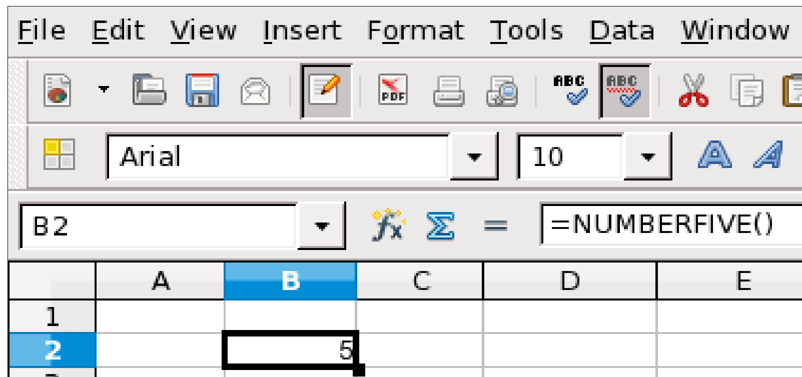


Figure 12: Use the NumberFive() Macro as a Calc function

Tip	Function names are not case sensitive. In Figure 12, you can enter `=NumberFive()` and Calc clearly shows `=NUMBERFIVE()`.

Save the Calc document, close it, and open it again. Depending on your settings in **Tools > Options > OpenOffice > Security > Macro Security**, Calc will display the warning shown in Figure 13 or the one shown in Figure 14. You will need to click **Enable Macros**, or Calc will not allow any macros to be run inside the document. If you do not expect a document to contain a macro, it is safer to click **Disable Macros** in case the macro is a virus.

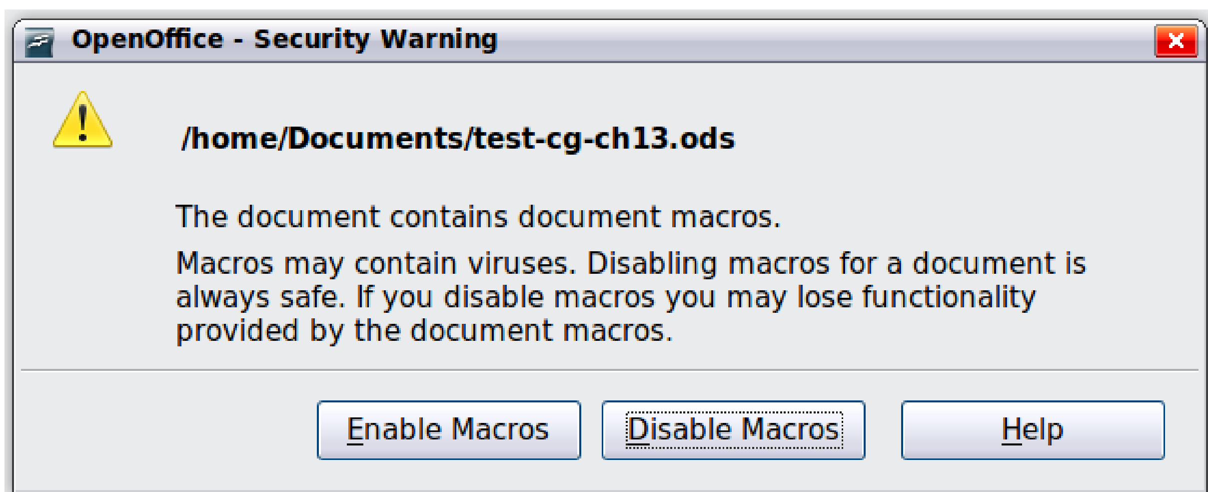


Figure 13: OpenOffice warns you that a document contains macros

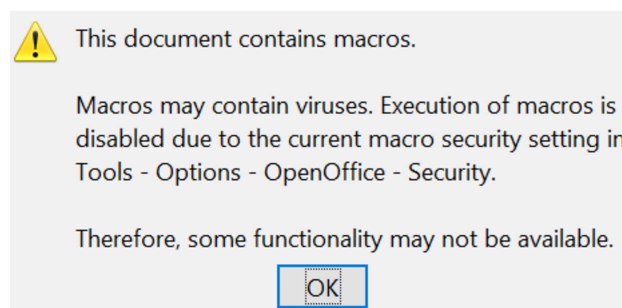


Figure 14: Warning if macros are disabled

Caution!	If you chose to disable macros, when the relevant document loads, Calc will no longer be able to find your function.

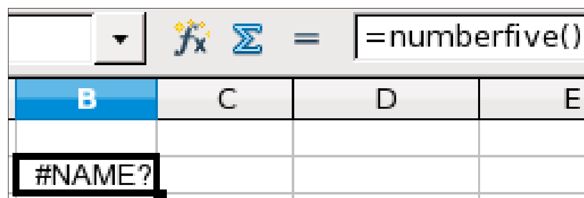


Figure 15: The function is gone

When a document is created and saved, it automatically contains a library named Standard. The Standard library is automatically loaded when the document is opened. No other library is automatically loaded.

Calc does not contain a function named NumberFive(), so it checks all opened and visible macro libraries for the function. Libraries in *OpenOffice Macros, My Macros*, and the Calc document are checked for an appropriately named function (see Figure 7). The NumberFive() function is stored in the AuthorsCalcMacros library, which is not automatically loaded when the document is opened.

Use **Tools > Macros > Organize Macros > OpenOffice Basic** to open the Basic Macros dialog (see Figure 16). Expand CalcTestMacros and find AuthorsCalcMacros. The icon for a loaded library is a different color from the icon for a library that is not loaded.

Click the expansion symbol (usually a plus or a triangle) next to AuthorsCalcMacros to load the library. The icon changes color to indicate that the library is now loaded. Click **Close** to close the dialog.

Unfortunately, the cells containing =NumberFive() are in error. Calc does not recalculate cells in error unless you edit them or somehow change them. The usual solution is to store macros used as functions in the Standard library. If the macro is large or if there are many macros, a stub with the desired name is stored in the Standard library. The stub macro loads the library containing the implementation and then calls the implementation.

1) Use **Tools > Macros > Organize Macros > OpenOffice Basic** to open theBasic Macros dialog. Select the NumberFive macro and click **Edit** to open the macro for editing.

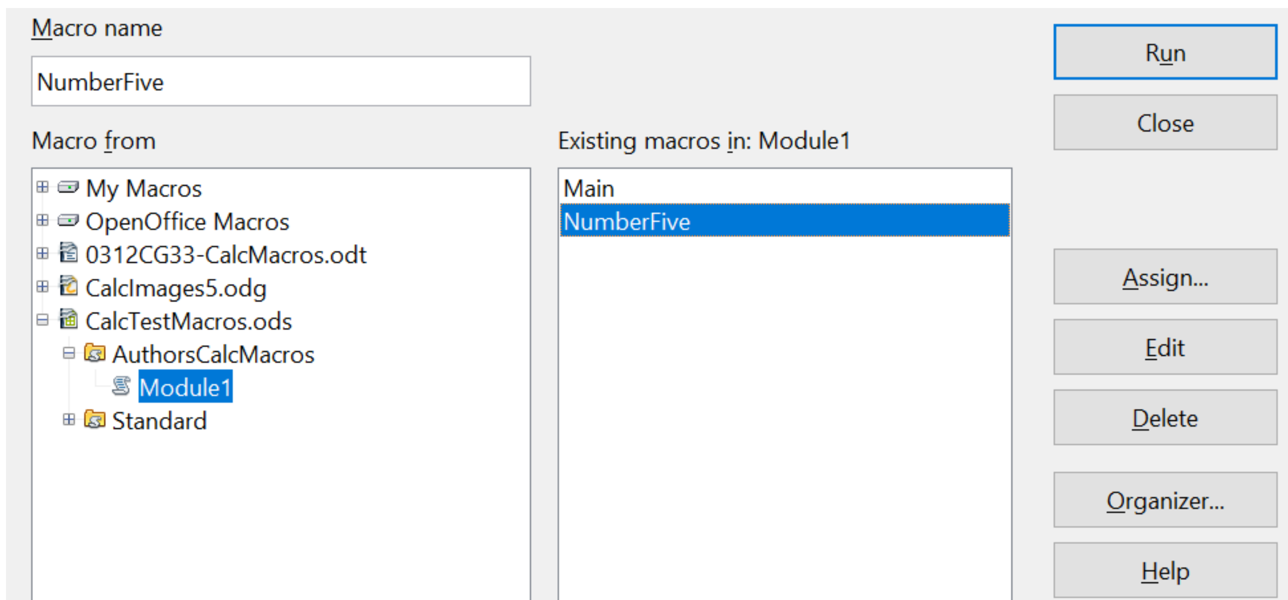


Figure 16: Select a macro and click Edit

2) Change the name of NumberFive to NumberFive_Implementation (Listing 3).

Listing 3. Change the name of NumberFive to NumberFive_Implementation

```
Function NumberFive_Implementation()
  NumberFive_Implementation() = 5
End Function
```

3) In the Basic IDE (see Figure 11), hover the mouse cursor over the toolbar buttons to display the tool tips. Click the **Select Macro** button to open the OpenOffice Basic Macros dialog (see Figure 16).
4) Select the Standard library in the CalcTestMacros document and click **New** to create a new module. Enter a meaningful name such as CalcFunctions and click **OK**. OpenOffice automatically creates a macro named Main and opens the module for editing.
5) Create a macro in the Standard library that calls the implementation function (see Listing 4). The new macro loads the AuthorsCalcMacros library if it is not already loaded, and then calls the implementation function.
6) Save, close, and reopen the Calc document. This time, the NumberFive() function works.

Listing 4. Change the name of NumberFive to NumberFive_Implementation.

```
Function NumberFive()
  If NOT BasicLibraries.isLibraryLoaded("AuthorsCalcMacros") Then
    BasicLibraries.LoadLibrary("AuthorsCalcMacros")
  End If
  NumberFive = NumberFive_Implementation()
End Function
```

Passing Arguments to a Macro

To illustrate a function that accepts arguments, we will write a macro that calculates the sum of its arguments that are positive, while ignoring arguments that are less than zero (see Listing 5).

Listing 5. PositiveSum calculates the sum of the positive arguments.

```
Function PositiveSum(Optional x)
  Dim TheSum As Double
  Dim iRow As Integer
  Dim iCol As Integer

  TheSum = 0.0
  If NOT IsMissing(x) Then
    If NOT IsArray(x) Then
      If x > 0 Then TheSum = x
    Else
      For iRow = LBound(x, 1) To UBound(x, 1)
        For iCol = LBound(x, 2) To UBound(x, 2)
          If x(iRow, iCol) > 0 Then TheSum = TheSum + x(iRow, iCol)
        Next
      Next
    End If
  End If
  PositiveSum = TheSum
End Function
```

The macro in Listing 5 demonstrates some important techniques:

1) The argument `x` is optional. When an argument is not optional and the function is called without it, OpenOffice prints a warning message every time the macro is called. If Calc calls the function many times, then the error is displayed many times.
2) `IsMissing` checks that an argument was passed before the argument is used.
3) `IsArray` checks to see if the argument is a single value, or an array. For example, `=PositiveSum(A1)` or `=PositiveSum(A1:A4)`. In the first case, the value of A1 is passed as a simple number to the argument, and in the second case, the values of A1:A4 are passed as an array to the function.
4) If a range is passed to the function, it is passed as a two-dimensional array of values, even if the cells are in a single column or row; for example, `=PositiveSum(A2:B5)`. `LBound` and `UBound` are used to determine the array bounds that are used. Although the lower bound is one, it is considered safer to use `LBound` in case it changes in the future.

Tip	The macro in Listing 5 is careful and checks to see if the argument is an array or a single argument. The macro does not verify that each value is numeric. You may be as careful as you like. The more things you check, the more robust the macro is, and the slower it runs.

Passing one argument is as easy as passing two: add another argument to the function definition (see Listing 6). When calling a function with two arguments, separate the arguments with a semicolon; for example, `=TestMax(3; -4)`.

Listing 6. TestMax accepts two arguments and returns the larger of the two.

```
Function TestMax(x, y)
  If x >= y Then
    TestMax = x
  Else
```

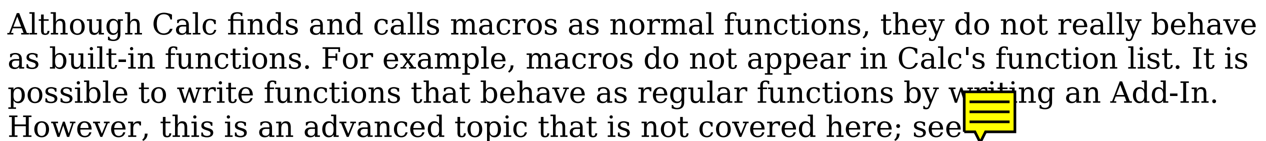
```
    TestMax = y
  End If
End Function
```

Arguments are Passed as Values

Arguments passed to a macro from Calc are always values; it's simply not possible to know what cells, if any, are used. For example, `=PositiveSum(A3)` passes the value stored in A3. The function PositiveSum has no way of knowing that cell A3 was used. If you must know which cells are referenced, send the range as a string to the function, and then determine the value in the referenced cell.

Writing Macros that Act Like Built-in Functions

Although Calc finds and calls macros as normal functions, they do not really behave as built-in functions. For example, macros do not appear in Calc's function list. It is possible to write functions that behave as regular functions by writing an Add-In. However, this is an advanced topic that is not covered here; see



https://wiki.openoffice.org/wiki/SimpleCalcAddIn

Accessing Cells Directly

You can access the OpenOffice internal objects directly to manipulate a Calc document. For example, the macro in Listing 7 adds the values in cell A2 from every sheet in the current document. `ThisComponent` is set by OpenOffice Basic when the macro starts to reference the current document. A Calc document has as a sub-object a collection of sheets: `ThisComponent.getSheets()`. You can get a particular sheet from that collection with the `getByIndex()` method. (There is also a `getByName()` method.) Use `getCellByPosition(col, row)` to return a cell at a specific row and column.

Listing 7. Add cell A2 in every sheet.

```
Function SumCellsAllSheets()
  Dim TheSum As Double
  Dim i As integer
  Dim oSheets
  Dim oSheet
  Dim oCell

  oSheets = ThisComponent.getSheets()
  For i = 0 To oSheets.getCount() - 1
    oSheet = oSheets.getByIndex(i)
    oCell = oSheet.getCellByPosition(0, 1) ' GetCell A2
    TheSum = TheSum + oCell.getValue()
  Next
  SumCellsAllSheets = TheSum
End Function
```

Tip	A cell object supports the methods `getValue()`, `getString()`, and `getFormula()` to get the numerical value, the string value (what is visible in the cell), or the formula used in a cell. Use the corresponding set functions to set appropriate values.

Use `oSheet.getCellRangeByName("A2")` to return a range of cells by name. If a single cell is referenced, then a cell object is returned. If a cell range is given, then an entire range of cells is returned (see Listing 8). Notice that a cell range returns data as an array of arrays, which is more cumbersome than treating it as an array with two dimensions as is done in Listing 5.

Listing 8. Add cell A2:C5 in every sheet

```
Function SumCellsAllSheets()
  Dim TheSum As Double
  Dim iRow As Integer, iCol As Integer, i As Integer
  Dim oSheets, oSheet, oCells
  Dim oRow(), oRows()

  oSheets = ThisComponent.getSheets()
  For i = 0 To oSheets.getCount() - 1
    oSheet = oSheets.getByIndex(i)
    oCells = oSheet.getCellRangeByName("A2:C5")
    REM getDataArray() returns the data as variant so strings
    REM are also returned.
    REM getData() returns data as type Double, so only
    REM numbers are returned.
    oRows() = oCells.getData()
    For iRow = LBound(oRows()) To UBound(oRows())
      oRow() = oRows(iRow)
      For iCol = LBound(oRow()) To UBound(oRow())
        TheSum = TheSum + oRow(iCol)
      Next
    Next
  Next
  SumCellsAllSheets = TheSum
End Function
```

Tip	When a macro is called as a Calc function, the macro cannot modify any value in the sheet except the value of the cell that called the function.

Sorting

Consider sorting the data in Figure 17. First, sort on column B descending and then column A ascending.

	A	B	C
1	1	5	One
2	4	1	Two
3	3	1	Three
4	7	8	Four
5	4	2	Five

Becomes

	A	B	C
1	7	8	Four
2	1	5	One
3	4	2	Five
4	3	1	Three
5	4	1	Two

Figure 17: Sort column B descending and column A ascending

The example in Listing 9 demonstrates how to sort on two columns.

Listing 9. Sort cells A1:C5 on Sheet 1.

```
Sub SortRange
  Dim oSheet          ' Calc sheet containing data to sort.
  Dim oCellRange      ' Data range to sort.

  REM An array of sort fields determines the columns that are
  REM sorted. This is an array with two elements, 0 and 1.
  REM To sort on only one column, use:
  REM Dim oSortFields(0) As New com.sun.star.util.SortField
  Dim oSortFields(1) As New com.sun.star.util.SortField

  REM The sort descriptor is an array of properties.
  REM The primary property contains the sort fields.
  Dim oSortDesc(0) As New com.sun.star.beans.PropertyValue

  REM Get the sheet named "Sheet1"
  oSheet = ThisComponent.Sheets.getByName("Sheet1")

  REM Get the cell range to sort
  oCellRange = oSheet.getCellRangeByName("A1:C5")

  REM Select the range to sort.
  REM The only purpose would be to emphasize the sorted data.
  'ThisComponent.getCurrentController.select(oCellRange)

  REM The columns are numbered starting with 0, so
  REM column A is 0, column B is 1, etc.
  REM Sort column B (column 1) descending.
  oSortFields(0).Field = 1
  oSortFields(0).SortAscending = FALSE

  REM If column B has two cells with the same value,
  REM then use column A ascending to decide the order.
  oSortFields(1).Field = 0
  oSortFields(1).SortAscending = True

  REM Setup the sort descriptor.
```

```
    oSortDesc(0).Name = "SortFields"
    oSortDesc(0).Value = oSortFields()

    REM Sort the range.
    oCellRange.Sort(oSortDesc())
  End Sub
```

As this macro is a subroutine, you execute it with **Tools > Macros > Run Macro** and then open **CalcTestMacros > AuthorsCalcMacros > Module1**. Click on the macro **SortRange** and then **Run**.

Conclusion

-

We have briefly covered how macros are organized into libraries and modules, how to use the macro recorder, using macros as Calc functions, and writing macros using the Application Programming Interface (API) rather than the recorder. The macro recorder simply stores the effects of keystrokes or mouse clicks and it can be used without any knowledge of programming. It cannot record every possible user action but it can often automate tedious, repetitive tasks, saving time and reducing errors. Writing functions and macros using the API are more advanced topics that require some programming knowledge. They can be used to make very powerful tools but that may require spending a lot of time learning the API, writing code and testing it. When the situation requires it, OpenOffice provides the infrastructure to implement complex and robust macros.