Calc Guide

Chapter 12
Calc Macros

Automating repetitive tasks

OpenOffice.org

Copyright

This document is Copyright © 2005–2007 by its contributors as listed in the section titled **Authors**. You may distribute it and/or modify it under the terms of either the GNU General Public License, version 2 or later (http://www.gnu.org/licenses/gpl.html), or the Creative Commons Attribution License, version 3.0 (http://creativecommons.org/licenses/by/3.0/) or later.

All trademarks within this guide belong to their legitimate owners.

Authors

Andrew Pitonyak
Gary Schnabl
Jean Hollis Weber

Feedback

Maintainer: Andrew Pitonyak [andrew@pitonyak.org]
Please direct any comments or suggestions about this document to:
authors@user-faq.openoffice.org or andrew@pitonyak.org

Publication date and software version

Published 02 July 2007. Based on OpenOffice.org 2.3.

You can download an editable version of this document from
http://oooauthors.org/en/authors/userguide2/published/

Contents

Copyright....................i
Authors....................i
Feedback....................i
Publication date and software version....................i
Introduction....................1
Using the macro recorder....................1
Write your own functions....................4
Using a macro as a function....................6
Passing arguments to a macro....................9
Arguments are passed as values....................11
Writing macros that act like built-in functions....................11
Accessing cells directly....................11
Sorting....................13

Introduction

A macro is a saved sequence of commands or keystrokes that are stored for later use. An example of a simple macro is one that “types” your address. The OpenOffice.org (OOo) macro language is very flexible, allowing automation of both simple and complex tasks. 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.

Using the macro recorder

Chapter 17 of the *Getting Started* guide (Getting Started with Macros) provides a basis for understanding the general macro capabilities in OpenOffice.org 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 (see Figure 2).

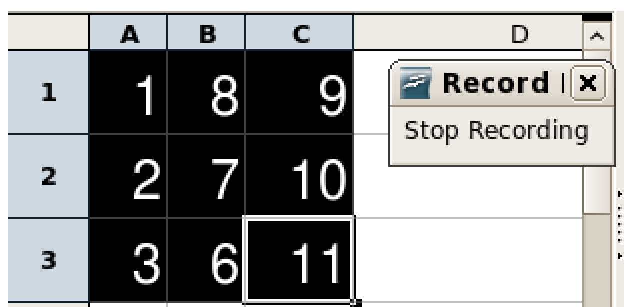


Figure 2: Stop recording button.

6) Use **Edit > Paste Special** to open the Paste Special dialog (see Figure 3).

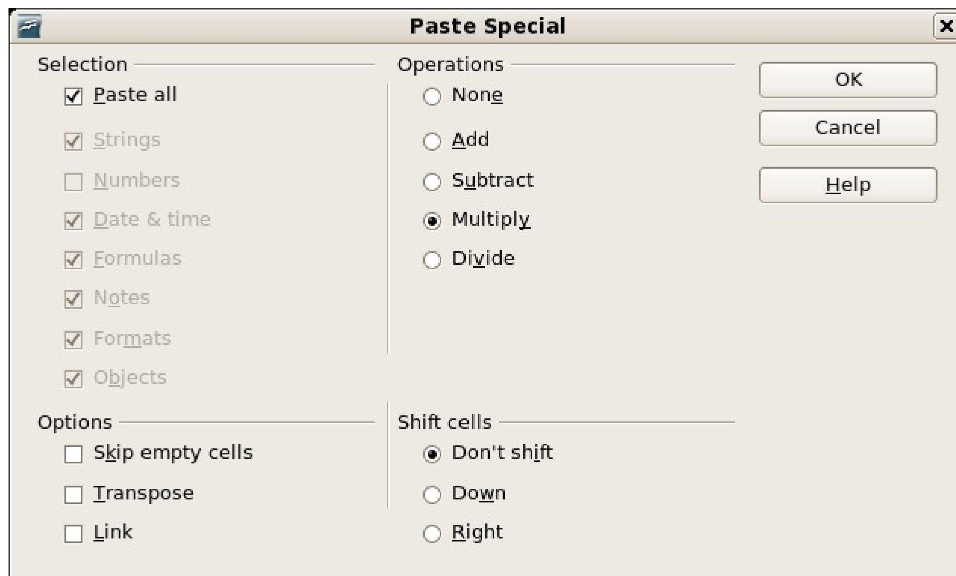


Figure 3: Paste Special dialog.

7) Set the operation to **Multiply** and click **OK**. The cells are now multiplied by 3 (see Figure 4).

Figure 4: Cells multiplied by 3.

8) Click **Stop Recording** to stop the macro recorder and save the macro.

9) Select the Standard library contained in the current document (see Figure 5). For this example, the current Calc document is *Untitled2*.

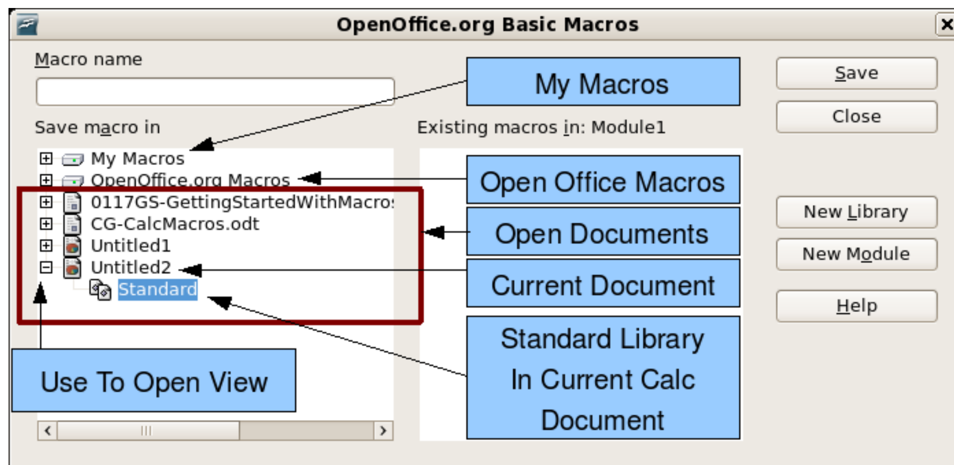
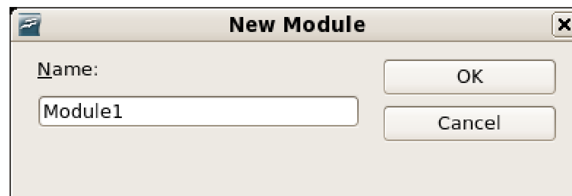


Figure 5: Select the Standard library.

10) Click **New Module** to create a module in the Standard library.



11) Click **OK** to create a module named Module1 (see Figure 6).

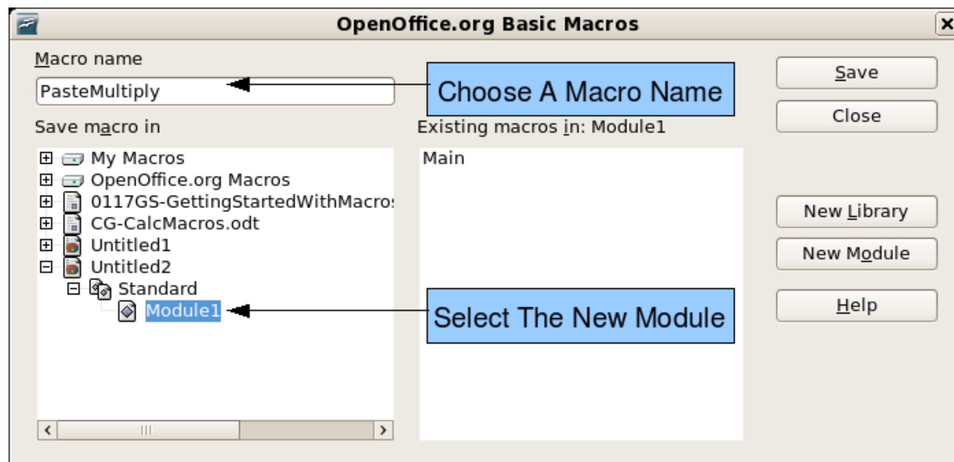


Figure 6: Select the module and name the macro.

12) Select the newly created Module1, enter the macro name *PasteMultiply* and click **Save**.

The created macro is saved in Module1 of the Standard library in the Untitled2 document (see Listing 1).

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 17 of the *Getting Started* guide (Getting Started with Macros); 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 call 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.org Basic** to open the OpenOffice.org Basic Macros dialog (see Figure 7). The *Macro from* box lists available macro library containers. *My Macros* contains macros that you write or add to OOo. *OpenOffice.org Macros* contains macros included with OOo and should not be changed. All other library containers are currently open OOo documents.

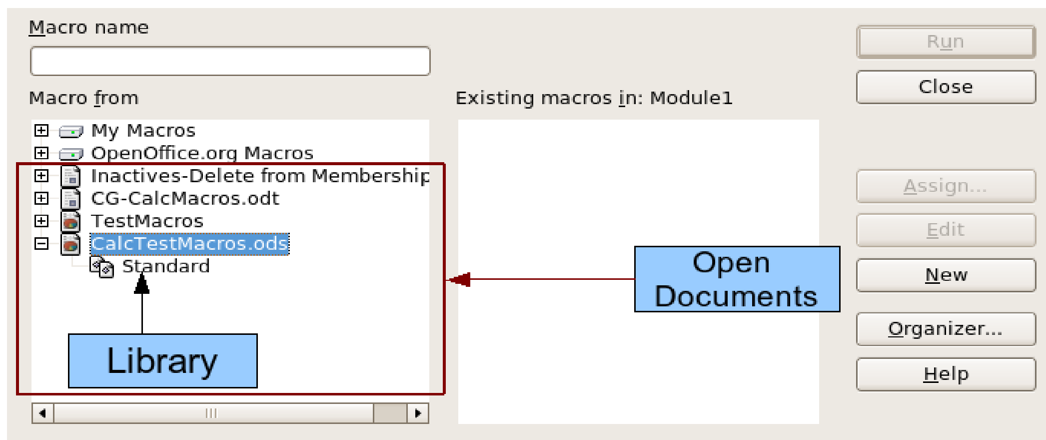


Figure 7. OpenOffice.org Basic Macros dialog.

3) Click **Organizer** to open the OpenOffice.org Basic Macro Organizer dialog (see Figure 8).

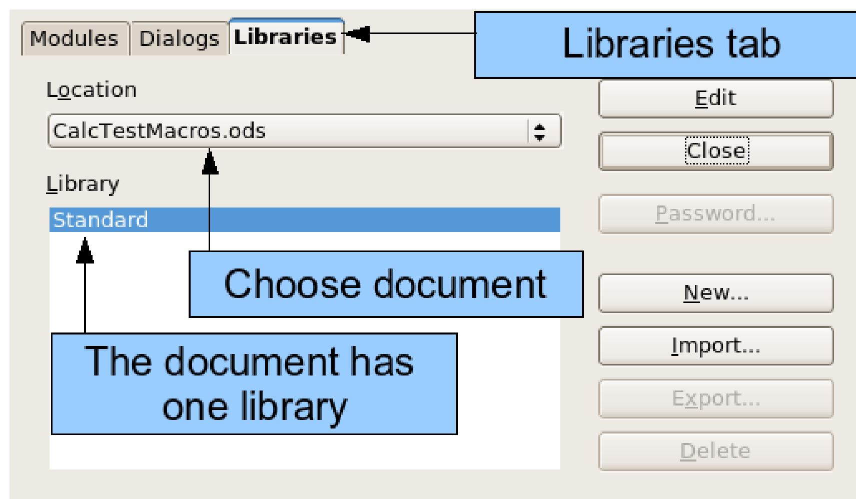


Figure 8. OpenOffice.org Basic Macro Organizer

4) Click the **Libraries** tab.

5) Select the document to contain the macro.

6) Click **New** to open the New Library dialog (see Figure 9).

Figure 9. New Library dialog.

7) Enter a descriptive library name (such as AuthorsCalcMacros) and click **OK** to create the library (see Figure 10). The new library name is shown the library list, but the dialog may show only a portion of the name.

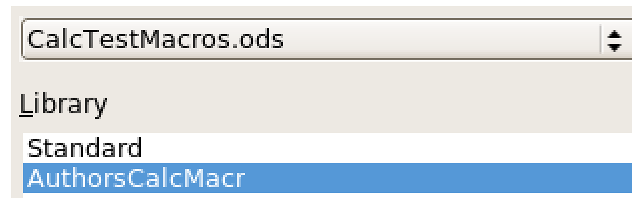


Figure 10. The library is shown in the organizer.

8) Select AuthorsCalcMacros (see Figure 10) and click **Edit** to edit the library. OOo automatically creates a module named Module1 and a macro named Main (see Figure 11).

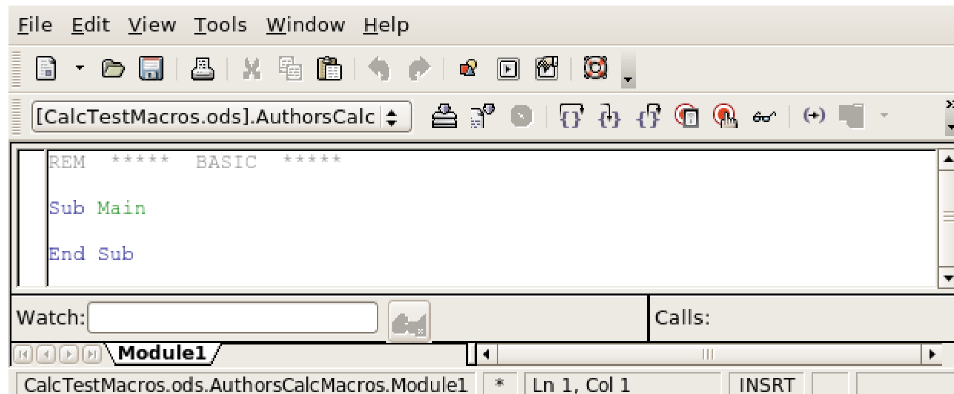


Figure 11. Basic Integrated Development Environment (IDE).

9) 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 statement Option Explicit 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.

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.

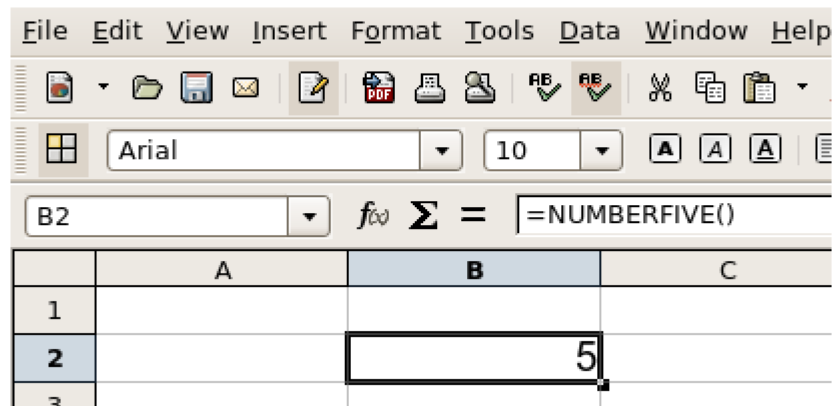


Figure 12. Use the NumberFive() Macro as a Calc function.

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

Save the Calc document, close it, and open it again. By default, OOo will warn you when you open a document that contains a macro (see Figure 13). Click **Enable Macros**, or OOo 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.

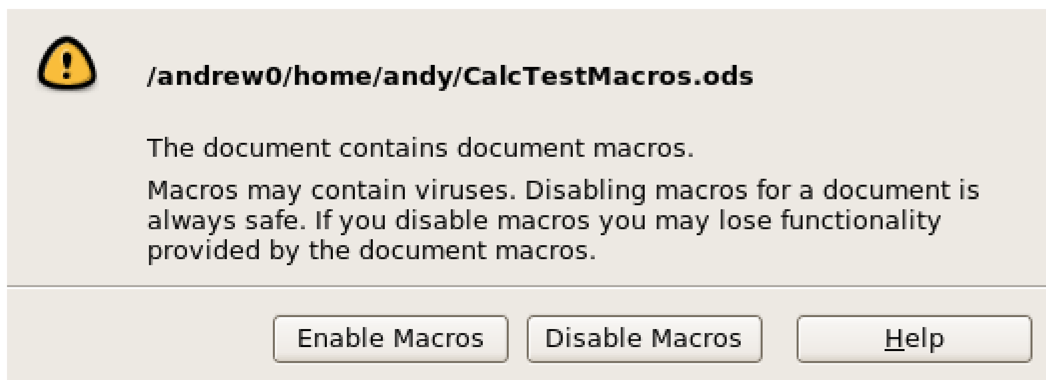


Figure 13. OOo warns you that a document contains macros.

When the document loads, Calc can no longer find the function (see Figure 14).

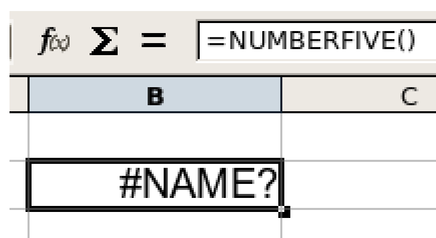


Figure 14. The function is gone.

Every OOo document type, except for Base, can contain macros—and this may change in the future. When the document is created, it automatically contains a library named Standard. The Standard library is special in that it is automatically loaded when the document is opened. No other library is automatically opened.

Calc does not contain a function named NumberFive(), so it checks all opened and visible macro libraries for the function. Libraries in *OpenOffice.org 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.org Basic** to open the OpenOffice.org Basic Macros dialog (see Figure 7). Expand CalcTestMacros and find AuthorsCalcMacros. The icon for a loaded library is different than the icon for a library that is not loaded (see Figure 15).

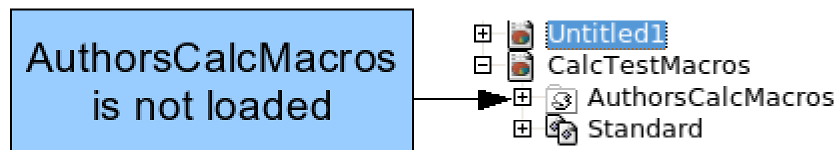


Figure 15. Unloaded macro library.

Click the plus (+) next to AuthorsCalcMacros to load the library. The icon changes to indicate that the library is now loaded (see Figure 16). Click **Close** to close the dialog.

Figure 16: Loaded macro library uses a different icon.

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.org Basic** to open the OpenOffice.org Basic Macros dialog (see Figure 17). Select the NumberFive macro and click **Edit** to open the macro for editing.

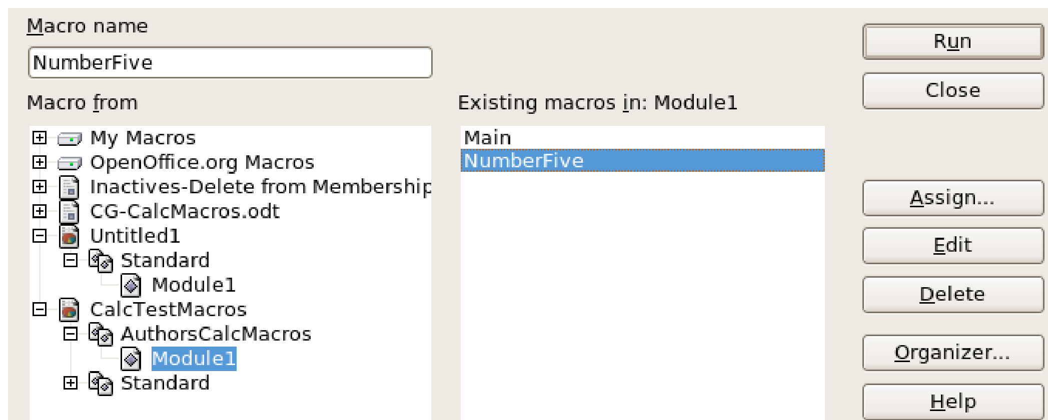


Figure 17. Select a macro and Click Edit.

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

Listing 3. Change the name of NumberFive to NumberFive_Implementation

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

2) 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.org Basic Macros dialog (see Figure 17).

3) 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**. OOo automatically creates a macro named Main and opens the module for editing.

4) 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.

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
```

5) Save, close, and reopen the Calc document. This time, the NumberFive() function works.

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 —it will ignore 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 a couple of important techniques.

1) The argument x is optional. If the argument is not optional and it is called without an argument, OOo prints a warning message every time that the macro is called. If Calc calls the function many times, then the error is displayed many times.
2) The IsMissing method is used to verify that an argument was passed before it is used.
3) The IsArray method is used to see if the function was called with with a single value. For example, =PositiveSum(7) or =PositiveSum(A4). In the first case, the number 7 is passed as an argument, and in the second case, the value of cell A4 is passed to the function.
4) If a range is passed to the function, it is passed as a two-dimensional array of values; 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 the argument to see if it is an array or a single argument. The macro does not, however, verify that each value is numeric. You may be as careful as you desire. The more things you check, the more robust the macro will be, and the slower it will run.

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 is not possible to know what cells, if any, are used. For example, =PositiveSum(A3) passes the value of cell A3, and PositiveSum has no way of knowing that cell A3 was used. If you must know which cells are referenced rather than the values in the cells, pass the range as a string, parse the string, and obtain the values in the referenced cells.

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 the function lists. 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 http://wiki.services.openoffice.org/wiki/SimpleCalcAddIn.

Accessing cells directly

You can access the OOo 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 StarBasic when the macro starts to reference the current document. A Calc document contains sheets: ThisComponent.getSheets(). 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, 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 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 from which the macro was called.

Sorting

Consider sorting the data in Figure 18. First, sort on column B descending and then column A ascending.

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

	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 18: Sort column B descending and column A ascending.

Listing 9. Add cell A2:C5 in every sheet.

```
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
```