Using LibreOffice on macOS

Some keystrokes and menu items are different on macOS 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.

<table>
<thead>
<tr>
<th>Windows or Linux</th>
<th>macOS equivalent</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools &gt; Options menu</td>
<td>LibreOffice &gt; Preferences</td>
<td>Access setup options</td>
</tr>
<tr>
<td>Right-click</td>
<td>Control + click or right-click depending on computer setup</td>
<td>Open a context menu</td>
</tr>
<tr>
<td>Ctrl (Control)</td>
<td>⌘ (Command)</td>
<td>Used with other keys</td>
</tr>
<tr>
<td>F5</td>
<td>Shift + ⌘ + F5</td>
<td>Open the Navigator</td>
</tr>
<tr>
<td>F11</td>
<td>⌘ + T</td>
<td>Open the sidebar Styles panel</td>
</tr>
</tbody>
</table>
Contents

Copyright..............................................................................................................................2
Contributors..........................................................................................................................2
To this edition......................................................................................................................2
To previous editions.........................................................................................................2
Feedback.............................................................................................................................2
Publication date and software version.............................................................................2
Using LibreOffice on macOS...........................................................................................2

Introduction......................................................................................................................4

Using the macro recorder.................................................................................................4

Write your own functions.................................................................................................8
Create function macro.......................................................................................................8
Using a macro as a function.............................................................................................12
Macro security warnings.................................................................................................12
Loaded / unloaded libraries............................................................................................13
Passing arguments to a macro.........................................................................................15
Arguments are passed as values......................................................................................16
Writing macros that act like built-in functions...............................................................16

Accessing cells directly.................................................................................................17

Sorting...............................................................................................................................18

Overview of BeanShell, JavaScript, and Python macros................................................19
Introduction.....................................................................................................................19
BeanShell macros............................................................................................................20
JavaScript macros..........................................................................................................22
Python macros...............................................................................................................24

Conclusion.......................................................................................................................25
Introduction

Chapter 13 of the *Getting Started Guide* (entitled *Getting Started with Macros*) is an introduction to the macro facilities that are available in LibreOffice. The current chapter provides further introductory information about the use of macros within a Calc spreadsheet.

A macro is a set of commands or keystrokes that are stored for later use. An example of a simple macro is one that enters your address into the current cell of an open spreadsheet. You can use macros to automate both simple and complex tasks, and they enable you to introduce new features that are not built into Calc.

The simplest way to create a macro is to record a series of actions through Calc’s user interface. Calc saves recorded macros using the open source LibreOffice Basic scripting language, which is a dialect of the well-known BASIC programming language. Such macros can be edited and enhanced after recording using the built-in LibreOffice Basic Integrated Development Environment (IDE).

The most powerful macros in Calc are created by writing code using one of the four supported scripting languages (LibreOffice Basic, BeanShell, JavaScript, and Python). This chapter provides an overview of Calc’s macro facilities, mostly focused on its default macro scripting language, LibreOffice Basic. Some examples are included for the BeanShell, JavaScript and Python scripting languages but fuller descriptions of the facilities for these languages are beyond the scope of this document.

Using the macro recorder

Chapter 13 of the *Getting Started Guide* includes examples showing how to use the macro recorder and understand the generated LibreOffice Basic scripts. The following steps give a further example, specific to a Calc spreadsheet, without the more detailed explanations of the *Getting Started Guide*. A macro is created and saved which performs a paste special with multiply operation across a range of spreadsheet cells.

1) Use Tools > Options > LibreOffice > Advanced from the Menu bar and select the Enable macro recording option to enable the macro recorder.

2) Use File > New > Spreadsheet from the Menu bar to create a new spreadsheet.

3) Enter the numbers shown in Figure 1 into cells A1:C3 of the first sheet in the new spreadsheet.

![Figure 1: Enter numbers into cells A1:C3](image)

4) Select cell A3, which contains the number 3, and use Edit > Copy from the Menu bar to copy the value to the clipboard.

5) Select all cells in the range A1:C3.

6) Use Tools > Macros > Record Macro from the Menu bar to start the macro recorder. Calc displays the Record Macro dialog, which includes a Stop Recording button (Figure 2).

![Figure 2: Record Macro dialog with Stop Recording button](image)
7) Use **Edit > Paste Special > Paste Special** from the Menu bar to open the Paste Special dialog (Figure 3).

![Paste Special dialog](image)

**Figure 3: Paste Special dialog**

8) Select the **Paste all** option in the **Selection** area and the **Multiply** option in the **Operations** area, and click **OK**. The values in cells A1:C3 are now multiplied by 3 (Figure 4).

![Cells A1:C3 multiplied by 3](image)

**Figure 4: Cells A1:C3 multiplied by 3**

9) Click the **Stop Recording** button to stop the macro recorder. Calc displays a variant of the LibreOffice Basic Macros dialog (Figure 5).

**Caution**

An earlier bug caused versions of LibreOffice 6.2 to crash when you clicked the **Stop Recording** button, but this bug is fixed in LibreOffice 6.3. See entry #122598 in The Document Foundation's Bugzilla defect tracking system for more details.
The Save macro in area of the LibreOffice Basic Macros dialog shows the existing LibreOffice Basic macros, hierarchically structured into library containers, libraries, modules, and macros as described in Chapter 13 of the Getting Started Guide. Figure 5 shows the My Macros library container, the LibreOffice Macros library container, the library container for the open Balance.ods file, and the library container for the untitled file created at step 2. Use the expand / collapse icons to the left of each library container name to view the libraries, modules and macros within that container.

10) Select the entry for the current document in the Save macro in area. As the current document in this example has not been saved, it is referred to by its default name Untitled 1.

Documents that have been saved include a macro library named Standard. This library is not created until the document is saved or the library is needed, so at this point in the example procedure your new document does not contain a library. You can create a new library to contain the macro you have just created, but this is not necessary.

11) Click the New Module button. Calc displays the New Module dialog (Figure 6). Type a name for the new module or leave the name as the default Module1.
Note

The libraries, modules and macro names must follow some strict rules. Following the main rules, the names must:

- Begin with a letter
- Comprise lower case letters (a..z), upper case letters (A..Z), digits (0..9), and underscore characters (_)
- Not contain any other spaces, punctuation symbols, or special characters (including accents)

12) Click the **OK** button to create a new module. As no macro libraries exist in our current document, Calc automatically creates and uses a **Standard** library.

13) On the LibreOffice Basic Macros dialog, select the entry for the newly created module in the **Save macro in** area, type the text **PasteMultiply** in the **Macro Name** box, and click the **Save** button (Figure 7).

The macro is saved with the name **PasteMultiply** in the newly created module within the **Standard** library of the **Untitled 1** document. Listing 1 shows the contents of the macro.
Listing 1. Paste special with multiply macro

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

Write your own functions

Create function macro

You can write a macro and then call it as you would call a Calc function. Use the following steps to create a simple function macro:

1) Create a new spreadsheet, save it with the name CalcTestMacros.ods, and leave it open in Calc.
2) Use Tools > Macros > Organize Macros > LibreOffice Basic from the Menu bar to open the LibreOffice Basic Macros dialog (Figure 8). Note that the layout of the LibreOffice Basic Macros dialog in this circumstance is different from the version that Calc displays when the user clicks the Stop Recording button on the Record Macro dialog (see Figure 5). The Macro From area lists the available macro library containers, including those relating to any LibreOffice documents that are currently open. My Macros contains macros that you write or add to LibreOffice and are available to more than one document. LibreOffice Macros contains macros that were included with your LibreOffice installation and should not be changed.
3) Click **Organizer** to open the LibreOffice Basic Macro Organizer dialog (Figure 9).

![Figure 8: LibreOffice Basic Macros dialog](image1.png)

![Figure 9: LibreOffice Basic Macro Organizer](image2.png)

Click on the **Libraries** tab and, in the **Location:** area, select the entry for the name of the current document. The **Library:** area updates to show the name of the empty **Standard** library.
4) Click New to open the New Library dialog to create a new library for this document (Figure 10).

![New Library dialog](image)

Figure 10: New Library dialog

5) Enter a descriptive library name (such as AuthorsCalcMacros) and click OK to create the library. The Library: area of the LibreOffice Basic Macro Organizer dialog updates to include the name of the newly created library. A library name can comprise up to 30 characters. Note that in some cases, the dialog may show only a portion of the name.

![LibreOffice Basic Macro Organizer](image)

Figure 11: The new library is shown in the Library: area

Select the AuthorsCalcMacros entry in the Library: area and click Edit to edit the library. Calc automatically creates a module named Module1 and a macro named Main. Calc displays the LibreOffice Basic Integrated Development Environment (IDE), shown in Figure 12.

10 | Create function macro
Figure 12: LibreOffice Basic Integrated Development Environment

Figure 12 shows the default configuration for the LibreOffice Basic IDE. This comprises:

- A menu bar.
- Three toolbars (Language, Macro, and Standard). The Macro toolbar provides various icons for editing and testing programs.
- The Object Catalog, enabling selection of the required library container, library, module, and macro.
- The Editor Window, in which you can edit the LibreOffice Basic program code. The column on the left side is used to set breakpoints in the program code.
- The Watch Window (located at the left, below the Object Catalog and Editor Window) displays the contents of variables or arrays during a single step process.
- The Call Stack Window (located to the right, below the Object Catalog and Editor Window) provides information about the call stack of procedures and functions when a program runs.
- A tab control area.
- A status bar.

The LibreOffice Basic IDE provides powerful facilities for the development and debugging of LibreOffice Basic macros. A fuller description of this facility is beyond the scope of this document.

6) In the Editor Window, 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 value 5.
Tip

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`.

Listing 2. Function that returns the value 5

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

Sub Main
End Sub

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

7) Use the `Save` button of the Standard toolbar within the LibreOffice Basic IDE to save the modified Module1.

Using a macro as a function

Using your newly created CalcTestMacros.ods spreadsheet, select a cell and enter the formula `=NumberFive()` (Figure 13). Calc finds the macro, calls it, and displays the result (5) in that cell.

![Figure 13: Use the NumberFive macro as a Calc function](image)

Tip

Function names are not case sensitive. In Figure 13, the function name was entered as `NumberFive()` but Calc displays it as `NUMBERFIVE()` in the Formula bar.

Macro security warnings

You should now save the Calc document, close it, and open it again. Depending on your settings in the Macro Security dialog accessed using `Tools > Options > LibreOffice > Security > Macro Security` from the Menu bar, Calc may display one of the warnings shown in Figures 14 and 15.

In the case of the warning shown in Figure 14, you will need to click `Enable Macros`, or Calc will not allow any macros to be run in 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.
In the case of the warning shown in Figure 15, Calc will not allow any macros to be run in the document and you should click the OK button to remove the warning from the screen.

When the document loads with macros disabled, Calc will not be able to find any macro functions and will indicate an error in any affected cell by displaying the text #NAME? in that cell.

**Loaded / unloaded libraries**

When it opens a spreadsheet, Calc does not open all macro libraries that it can find in the available library containers because this would be a waste of resources. Instead Calc automatically loads just the Standard library within the My Macros library container and the document’s own Standard library. No other libraries are automatically loaded.

When you re-open your CalcTestMacros.ods spreadsheet, Calc does not contain a function named NumberFive(), so it checks all visible, loaded macro libraries for the function. Loaded libraries in LibreOffice Macros, My Macros, and the document are checked for an appropriately named function. In our initial implementation, the NumberFive() function is stored in the AuthorsCalcMacros library, which is not automatically loaded when the document is opened. Hence the NumberFive() function is not found and an error condition appears in the cell where it is called (Figure 16).

![Figure 16: The macro function is not available](image)
Use **Tools > Macros > Organize Macros > LibreOffice Basic** from the Menu bar to open the LibreOffice Basic Macros dialog (Figure 17). The icon for a loaded library (for example, `Standard`) has a different appearance to the icon for a library that is not loaded (for example, `AuthorsCalcMacros`).

![LibreOffice Basic Macros dialog](image)

**Figure 17: Different symbols for loaded and unloaded libraries**

Click the expand icon next to `AuthorsCalcMacros` to load the library. The icon changes appearance to indicate that the library is now loaded. Click **Close** to close the LibreOffice Basic Macros dialog.

Unfortunately, the cell containing `=NumberFive()` in our initial implementation is still 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. The following steps illustrate this method.

1) Use **Tools > Macros > Organize Macros > LibreOffice Basic** in the Menu bar to open the LibreOffice Basic Macros dialog. Select the `NumberFive` macro and click **Edit** to open the macro for editing (Figure 18).

![LibreOffice Basic Macros dialog with selected macro](image)

**Figure 18: Select a macro and click Edit**

2) Calc displays the LibreOffice Basic IDE (Figure 12), with the input cursor in the Editor Window at the line `Function NumberFive ()`. Change the name of `NumberFive` to `NumberFive_Implmentation` so that the function’s code matches Listing 3.
Listing 3. Change the name of `NumberFive` to `NumberFive_Implementation`

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

3) Click the **Select Macro** button in the Standard toolbar of the LibreOffice Basic IDE to open the LibreOffice Basic Macros dialog (Figure 18).

4) Select the **Standard** library in the `CalcTestMacros.ods` document and click the **New** button to create a new module. Enter a meaningful name such as `CalcFunctions` and click **OK**. Calc automatically creates a macro named `Main` and opens the module for editing.

5) Create a macro in the `CalcFunctions` module of the `Standard` library that loads the `AuthorsCalcMacros` library if it is not already loaded, and then calls the implementation function. See Listing 4.

```basic
Listing 4. Create a new `NumberFive` function to call the `NumberFive_Implementation` function

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

6) Save, close, and reopen the Calc document. This time, if macros are enabled, the `NumberFive()` function works as expected.

### 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).

```basic
Listing 5. `PositiveSum` calculates the sum of its 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
```

---

*Passing arguments to a macro | 15*
The macro in Listing 5 demonstrates some important techniques:

1) The argument \( x \) is \textit{Optional}. When an argument is not \textit{Optional} and the function is called without it, Calc outputs a warning message every time the macro is called. If Calc calls the function many times, then the error is displayed many times.

2) The function \textit{IsMissing} checks that an argument was passed before it is used.

3) The function \textit{IsArray} checks to see if the argument is a single value, or an array. For example, \texttt{=PositiveSum(7)} or \texttt{=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. In both these cases, \texttt{IsArray} returns the value \texttt{False}.

4) If a range is passed to the function, it is passed as a two-dimensional array of values; for example, \texttt{=PositiveSum(A2:B5)}. The functions \texttt{LBound} and \texttt{UBound} are used to determine the array bounds that are used. Although the lower bound is one, it is considered safer to use \texttt{LBound} in case it changes in the future.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Tip} & \\
\hline
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, but the slower it runs. & \\
\hline
\end{tabular}
\caption{Tip}
\end{table}

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 comma; for example, \texttt{=TestMax(3, -4)}.

\textit{Listing 6. TestMax accepts two arguments and returns the larger}

\begin{verbatim}
Function TestMax(x, y)
    If x >= y Then
        TestMax = x
    Else
        TestMax = y
    End If
End Function
\end{verbatim}

\textbf{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, \texttt{=PositiveSum(A3)} passes the value of cell A3, and \textit{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.

\textbf{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 for experienced programmers and is beyond the scope of this guide.

16 | Writing macros that act like built-in functions
Accessing cells directly

You can access the LibreOffice 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 automatically set to reference the current document when the macro starts. A Calc document contains sheets and the macro accesses these via a call to `ThisComponent.getSheets()`. Use `getCellByPosition(col, row)` to return a cell at a specific row and column.

**Listing 7. SumCellsAllSheets adds the values in cell A2 of every sheet**

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

    TheSum = 0
    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. SumCellsAllSheets adds the values in cells A2:C5 of every sheet**

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

    TheSum = 0
    oSheets = ThisComponent.getSheets()
    For i = 0 To oSheets.getCount() - 1
        oSheet = oSheets.getByIndex(i)
        oCells = oSheet.getCellRangeByName("A2:C5")
        REM The getDataArray() method returns strings and numbers
        REM but is not used in this function.
```

---

Accessing cells directly | 17
REM The getData() method returns only numbers and is applicable
REM to this function.

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
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, except the value of the cell that contains the
function.

Sorting

Consider sorting the data shown in Figure 19. First, sort on column B descending and then on
column A ascending.

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

Listing 9. SortRange sorts cells A1:C5 of 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
REM 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
Dim oSortDesc(1) As New com.sun.star.beans.PropertyValue

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

Figure 19: Sort column B descending and column A ascending
Overview of BeanShell, JavaScript, and Python macros

Introduction

Many programmers may not be familiar with LibreOffice Basic and so Calc supports macros written in three other languages that may be more familiar. These are BeanShell, JavaScript, and Python.

The primary macro scripting language for Calc is LibreOffice Basic and the standard LibreOffice installation provides a powerful integrated development environment (IDE) together with more options for this language.

Macros are organized in the same way for all four scripting languages. The LibreOffice Macros container holds all the macros that are supplied in the LibreOffice installation. The My Macros library container holds your macros that are available to any of your LibreOffice documents. Each document can also contain your macros that are not available to any other document.

When you use the macro recording facility, Calc creates the macro in LibreOffice Basic. To use the other available scripting languages you must write the code yourself.

When you select to run a macro using Tools > Macros > Run Macro from the Menu bar, Calc displays the Macro Selector dialog. This dialog enables selection and running of any available macro, coded in any of the available languages (Figure 20).

When you select to edit a macro using Tools > Macros > Edit Macros from the Menu bar, Calc displays the LibreOffice Basic IDE. This dialog enables selection and editing of any available LibreOffice Basic macro, but not macros in other languages.
The component model used in LibreOffice is known as Universal Network Objects or UNO. LibreOffice macros in any scripting language use a UNO runtime application programming interface (API). The XSCRIPTCONTEXT interface is provided to macro scripts in all four languages, and provides a means of access to the various interfaces which they might need to perform some action on a document.

**BeanShell macros**

BeanShell is a Java-like scripting language that was first released in 1999.

When you select **Tools > Macros > Organize Macros > BeanShell** from the Menu bar, Calc displays the BeanShell Macros dialog (Figure 21).
Click the **Edit** button on the BeanShell Macros dialog to access the BeanShell Debug Window (Figure 22).
Listing 10 is an example of a BeanShell macro that inserts the text “Hello World from BeanShell” in cell A1 of the active Calc spreadsheet.

**Listing 10. Sample BeanShell macro**

```java
import com.sun.star.uno.UnoRuntime;
import com.sun.star.sheet.XSpreadsheetView;
import com.sun.star.text.XText;
model = XSCRIPTCONTEXT.getDocument();
controller = model.getCurrentController();
view = UnoRuntime.queryInterface(XSpreadsheetView.class, controller);
sheet = view.getActiveSheet();
cell = sheet.getCellByPosition(0, 0);
cellText = UnoRuntime.queryInterface(XText.class, cell);
textCursor = cellText.createTextCursor();
cellText.insertString(textCursor, "Hello World from BeanShell", true);
return 0;
```

**JavaScript macros**

JavaScript is a high-level scripting language that was first released in 1995.

When you select **Tools > Macros > Organize Macros > JavaScript** from the Menu bar, Calc displays the JavaScript Macros dialog (Figure 23).

![JavaScript Macros dialog](image)

**Figure 23: JavaScript Macros dialog**
Click the **Edit** button on the JavaScript Macros dialog to access the Rhino JavaScript Debugger (Figure 24). Detailed instructions for using this tool can be found on Mozilla’s website at https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/Debugger.

Listing 11 is an example of a JavaScript macro that inserts the text “Hello World from JavaScript” in cell A1 of the first sheet in a Calc spreadsheet.

Listing 11

```
// Hello World in JavaScript
importClass(Packages.com.sun.star.uno.UnoRuntime);
importClass(Packages.com.sun.star.text.XTextDocument);
importClass(Packages.com.sun.star.text.XText);
importClass(Packages.com.sun.star.text.XTextRange);

//get the document from the scripting context
oDoc = XSCRIPTCONTEXT.getDocument();
//get the XTextDocument Interface
xTextDoc = UnoRuntime.queryInterface(XTextDocument, oDoc);
//get the XText Interface
xText = xTextDoc.getText();
//get an (empty) XTextRange interface at the end of the text
```
Listing 11. Sample JavaScript macro

importClass(Packages.com.sun.star.uno.UnoRuntime);
importClass(Packages.com.sun.star.sheet.XSpreadsheetDocument);
importClass(Packages.com.sun.star.container.XIndexAccess);
importClass(Packages.com.sun.star.sheet.XCellRange);
importClass(Packages.com.sun.star.table.XCell);

documentRef = XSCRIPTCONTEXT.getDocument();

spreadsheetInterface = UnoRuntime.queryInterface(XSpreadsheetDocument, documentRef);

allSheets = UnoRuntime.queryInterface(XIndexAccess, spreadsheetInterface.getSheets());

thesheet = allSheets.getByIndex(0);

Cells = UnoRuntime.queryInterface(XCellRange, thesheet);

cellA1 = Cells.getCellByPosition(0,0);

theCell = UnoRuntime.queryInterface(XCell,cellA1);

theCell.setFormula("Hello World from JavaScript");

**Python macros**

Python is a high-level, general-purpose programming language that was first released in 1991.

When you select **Tools > Macros > Organize Macros > Python** from the Menu bar, Calc displays the Python Macros dialog (Figure 25).

![Python Macros dialog](image)

*Figure 25: Python Macros dialog*
Facilities to edit and debug Python scripts are not currently integrated into the standard LibreOffice user interface. However you can edit Python scripts with your preferred text editor or an external IDE. The Alternative Python Script Organizer (APSO) extension eases the editing of Python scripts, in particular when embedded in a document. Using APSO you can configure your preferred source code editor, start the integrated Python shell and debug Python scripts. For more information search for Python in the LibreOffice Help system and visit the Designing & Developing Python Applications section of The Document Foundation’s wiki (https://wiki.documentfoundation.org/Macros/Python_Design_Guide).

Listing 12 is an example of a Python macro that sets cell A1 of the first sheet in a Calc spreadsheet to the text “Hello World from Python”.

Listing 12. Sample Python macro

```python
import uno

def HelloWorld():
    doc = XSCRIPTCONTEXT.getDocument()
    cell = doc.Sheets[0]['A1']
    cell.setString('Hello World from Python')
    return
```

**Conclusion**

This chapter provides an overview of how to create libraries and modules, using the macro recorder, using macros as Calc functions, and writing your own macros without the macro recorder. Each topic deserves at least one chapter, and writing your own macros for Calc could easily fill an entire book. In other words, this is just the beginning of what you can learn.

Additional detail about Calc’s macro facilities can be obtained from the Help system, The Document Foundation’s wiki pages (for example https://wiki.documentfoundation.org/Macros) and other Internet sources (for example the http://ask.libreoffice.org/ Q&A site).