Automate Repetitive Database Tasks

The more you use Excel, the greater the chance that you will be performing the same sets of actions over and over again. For instance, you may frequently change the visual format of a selected set of worksheet cells to a similar font style, row height, and border style. Or perhaps you frequently sort data records that relate to sales transactions by transaction date. In this chapter, you will learn how to use Excel's macro recorder and the Excel programmatic object model with Visual Basic for Applications to automate these types of actions. By recording your Excel interactions and then customizing these recordings to automate Excel to get the desired results, you can save time and reduce errors when you want to do similar actions in the future.

7.1 Use the Macro Recorder

Excel's macro recorder works very similar to a digital music recorder. Just as a digital music recorder records music that you can play back, in Excel you can name a *macro* (a set of instructions to perform an action); the recorder starts, you perform a set of actions in Excel, and then you stop recording. Later, you can play the recorded macro as needed to perform the desired action. For example, you could record a set of actions to apply conditional formatting to a group of worksheet cells. You could then modify that set of actions and reuse them to quickly apply conditional formatting to another group of worksheet cells.

Quick Start

To use the macro recorder, do the following:

1. In Excel 2007, click Developer ➤ (Code) Record Macro. In Excel 2003, click Tools ➤ Macro ➤ Record New Macro.
2. Complete the options in the Record Macro dialog box.
3. In Excel, perform the actions that you want to automate.
4. When finished, in Excel 2007, click Developer ➤ (Code) Stop Recording. In Excel 2003, on the Stop Recording toolbar, click the Stop Recording button.

IF YOU DO NOT SEE THE DEVELOPER TAB IN EXCEL 2007

To display the Developer tab in Excel 2007, do the following:

1. Click Office Button ➤ Excel Options.
2. On the Popular tab, in the Top Options for Working with Excel area, select the Show Developer Tab In the Ribbon check box, and then click OK.

To run the macro, in Excel 2007, click Developer ➤ (Code) Macros, select the desired macro, and then click Run. In Excel 2003, click Tools ➤ Macros ➤ Macros, select the desired macro, and then click Run.

How To

To use the macro recorder, do the following:

1. In Excel 2007, click Developer ➤ (Code) Record Macro. In Excel 2003, click Tools ➤ Macro ➤ Record New Macro.
2. Complete the options in the Record Macro dialog box:
   - In the Macro Name box, you can type a name that you can easily remember later for reference.
   - In the Shortcut Key box, you can type a keyboard character that you want to use to associate with running this macro. For example, if you type the letter **J**, you can use the Ctrl+J shortcut key combination to run this macro later.
   - In the Store Macro In list, you can select the location in which you want to store this macro.
   - In the Description box, you can type a description for the macro for future reference.
3. In Excel, perform the actions that you want to automate.
4. When finished, in Excel 2007, click Developer ➤ (Code) Stop Recording. In Excel 2003, on the Stop Recording toolbar, click the Stop Recording button.

To run the macro, in Excel 2007, click Developer ➤ (Code) Macros, select the desired macro, and then click Run. In Excel 2003, click Tools ➤ Macros ➤ Macros, select the desired macro, and then click Run.

RECORDING AND RUNNING MACROS WITH EXCEL SECURITY SETTINGS

Depending on Excel's macro security level setting, you may not be able to record or run macros. To change your macro security level, do one of the following:

- In Excel 2007, click Developer ➤ (Code) Macro Security, select the desired macro security level on the Macro Settings tab, click OK, and then quit and restart Excel.
- In Excel 2003, click Tools ➤ Macro ➤ Security, select the desired macro security level on the Security Level tab, click OK, and then quit and restart Excel.

In Excel 2007, the macro security levels for documents that are opened from untrusted locations are the following:

- *Disable All Macros Without Notification*: Excel will not allow you to run macros in the workbook. Excel will not inform you that macros cannot be run, and Excel will not allow you to run macros in the workbook until you reopen the workbook and choose the Disable All Macros with Notification option or the Enable All Macros option.
- *Disable All Macros with Notification*: Excel will not allow you to run macros in the workbook. Excel will inform you that macros have been disabled in the workbook, and Excel will present you with an Options button to enable you to run macros in the workbook.
- *Disable All Macros Except Digitally Signed Macros*: If the workbook is opened from an untrusted location, and the macros are not digitally signed, Excel will not allow you to run macros in the workbook until the macros are digitally signed or you reopen the workbook and choose the Disable All Macros with Notification option or the Enable All Macros option.
- *Enable All Macros (Not Recommended; Potentially Dangerous Code Can Run)*: Excel will enable all macros in any workbook to be run, regardless of where the workbook originates or is stored. Because this setting can leave your computer very vulnerable to dangerous code, you should consider setting your macro security level to one of the other macro security levels.

In Excel 2003, the macro security levels are the following:

- *Very High*: Macros will run only if they are in a workbook stored in a trusted location.
- *High*: Macros will run only if they are in a workbook originating from a trusted source.
- *Medium*: If the workbook does not originate from a trusted source, Excel will ask if you want to enable the macros in the workbook to be run.
- *Low*: Excel will enable all macros in any workbook to be run, regardless of where the workbook originates or is stored. Because this setting can leave your computer very vulnerable to dangerous code, you should consider setting your macro security level to Medium or higher.

Be aware that changing Excel's macro security level can cause potentially unexpected and unwanted results in some cases. For example, in Excel 2003, setting your macro security level to Very High will prevent all of your macros from running unless they are in a workbook stored in a trusted location. Similarly, setting your macro security level to Low in Excel 2003 can unwittingly expose your computer to unwanted viruses hiding in macros created by others with malicious intent. In some organizations, your IT department may restrict or prevent your ability to change macro security levels such as Disable All Macros Without Notification or Very High.

Try It

In this exercise, you will practice using Excel's macro recorder to change the visual formatting of selected worksheet cells.

Open the workbook:

1. Click Office Button ➤ Open (in Excel 2007) or File ➤ Open (in Excel 2003).
2. Browse to and select the ExcelDB_Ch07_01-12.xls file, and then click Open.

Record the macro:

1. On the RecordMacros worksheet, select cells A2 through A9.
2. Do one of these:
   - In Excel 2007, click Developer ➤ (Code) Record Macro.
   - In Excel 2003, click Tools ➤ Macro ➤ Record New Macro.
3. In the Macro Name box, type **FormatSelectedCells**.
4. With the Store Macro in List showing This Workbook, click OK.
5. In Excel 2007, click Home ➤ (Cells) Format ➤ Format Cells. In Excel 2003, click Format ➤ Cells.
6. On the Font tab, in the Font Style list, select Bold Italic.
7. In the Color list, select the red box.
8. On the Fill tab (in Excel 2007) or the Patterns tab (in Excel 2003), in the Background Color area (in Excel 2007) or the Cell Shading area (in Excel 2003), click the yellow box, then click OK.
9. In Excel 2007, click Developer ➤ (Code) Stop Recording. In Excel 2003, on the Stop Recording toolbar, click Stop Recording.

Play back the macro:

1. Select cells C2 through C9.
2. In Excel 2007, click Developer ➤ (Code) Macros. In Excel 2003, click Tools ➤ Macro ➤ Macros.
3. In the list, select FormatSelectedCells, and click Run. The selected cells match the formatting in cells A2 through A9.

7.2 Understand Excel Visual Basic for Applications

If you're familiar with Microsoft Visual Basic, then Excel's Visual Basic for Applications (VBA) programming environment will also be very familiar. Even if you're not a Visual Basic programmer, using Excel's built-in programming environment is very straightforward.

When you want to write code to automate repetitive database tasks, you attach Excel VBA code to the desired workbook by using the Excel Visual Basic Editor (VBE). In the VBE, the

Project Explorer window displays the code attached to each open Excel workbook. To facilitate code reuse among Excel workbooks, you can also create modules that can be shared with other workbooks. For example, you could create a module that applies specific visual formatting to a group of worksheet cells. Then you could copy that module to other worksheets or share that module with your coworkers.

Quick Start

- To access the VBE, in Excel 2007, click Developer ➤ (Code) Visual Basic. In Excel 2003, click Tools ➤ Macro ➤ Visual Basic Editor.

Note If you do not see the Developer tab in Excel 2007, see the sidebar in section 7.1 to learn how to display it.

- To create a module, right-click any node associated with the target workbook, click Insert, and then click the desired module type.
- To share a module, right-click the module, click Export File, and complete the Export File dialog box.
- To attach a module to a workbook, right-click any node associated with the target workbook, click Import File, and complete the Import File dialog box.

How To

To access the Visual Basic Editor (VBE) in Excel 2007, click Developer ➤ (Code) Visual Basic. In Excel 2003, click Tools ➤ Macro ➤ Visual Basic Editor.

To access the VBE's Project Explorer window, click View ➤ Project Explorer. Each Excel workbook has a corresponding node in the Project Explorer window, representing code attached to the workbook itself. To access the workbook's code, expand the workbook's node, and then double-click the ThisWorkbook icon. Each worksheet in a workbook has its own associated code as well, typically accessible only to that worksheet. To access a worksheet's code, double-click the corresponding worksheet icon.

To facilitate code reuse among Excel workbooks, you can also create three types of modules (*code modules*, *class modules*, and *UserForms*) that can be shared with other workbooks:

- *Code modules* are typically used to create reusable subroutines and functions.
- *Class modules* are used to create custom programmatic objects and classes that are best leveraged in object-oriented code solutions.
- *UserForms* are used to create interactive dialog boxes.

To create a code module, right-click in the Project Explorer any node associated with a workbook or worksheet, and then click Insert ➤ Module, Insert ➤ Class Module, or Insert ➤ UserForm. You can then attach code to the newly created code module by double-clicking that module in the Project Explorer window.

To share a code module, right-click in the Project Explorer the code module, click Export File, and complete the Export File dialog box to save a copy of the code. (Note that this copied code is a snapshot of the code module; future changes that you make to the original code module are not reflected in this saved copy of the code.) To attach an exported module to a workbook, right-click in the Project Explorer any node associated with the target workbook or worksheet, click Import File, and complete the Import File dialog box.

Tip To learn more about how to use VBE in Excel 2007, in the VBE, click Help ➤ Microsoft Visual Basic Help, type phrases such as **Visual Basic Editor**, **Project Explorer**, **Code Window**, or **Toolbars** in the Type Words to Search For box, and click the Search Developer Reference button.

To learn more about how to use the VBE in Excel 2003, in the VBE, click Help ➤ Microsoft Visual Basic Help, and in the Table of Contents list, expand the Microsoft Visual Basic Documentation book, and expand the Visual Basic User Interface Help book. Read Help topics such as Project Explorer Window and Code Window (in the Windows book) and Standard Toolbar (in the Toolbars book), and the various Help topics in the Menus book.

If you've worked with other programming languages, you should be able to program in VBA with little difficulty. Here is a brief summary of the most common VBA language keywords:

- You begin a procedure or method with either the `Sub` keyword (for procedures and methods that don't return a value) or the `Function` keyword (for procedures and methods that can return a value), and you end the procedure or method declaration with `End Sub` or `End Function`—for example, `Sub ChangeCellColor()` and then `End Sub`.
- VBA defines data types such as `Array`, `Boolean`, `Byte`, `Currency`, `Date`, `Decimal`, `Double`, `Integer`, `Long`, `Object`, `Single`, `String`, and `Variant`, in addition to Excel-specific data types such as `Workbook`, `Worksheet`, and `PivotTable`. You declare a variable with the `Dim` keyword and specify the variable's data type. If the variable is an object data type, you then initialize the variable with the `Set` keyword—for example, `Dim intNumberOfCells As Integer` and then `intNumberOfCells = 10` or `Dim wbk As Workbook` and then `Set wbk = ThisWorkbook`.
- VBA defines control flow constructs such as `Do...Loop`, `For...Next`, `If...Then...Else`, and `Select Case`—for example, `For i = 1 to 10`, then `MsgBox i`, and then `Next` to display the numbers 1 to 10.

Note To learn more about how to use the VBA programming language in Excel 2007, in the VBE included with Excel 2007, click Help ➤ Microsoft Visual Basic Help, type phrases such as **writing declaration statements**, **data types keyword summary**, **control flow keyword summary**, or **keywords by task** in the Type Words to Search For box, and click the Search Developer Reference button.

To learn how to use the VBA programming language in Excel 2003, in the VBE included with Excel 2003, click Help ➤ Microsoft Visual Basic Help, and in the Table of Contents list, expand the Microsoft Visual Basic Documentation book, and expand the Visual Basic Conceptual Topics book. Read Help topics such as Writing a Sub Procedure; Declaring Variables; Creating Object Variables; Understanding Objects, Properties, Methods, and Events; and Looping Through Code.

Try It

In this exercise, you will practice attaching a module to an existing workbook. You will then export the module, attach the exported module to a new workbook, and run the module's code from the new workbook.

1. Start Excel.
2. Create two new Excel workbook files. Save the first Excel workbook with the file name Before.xlsm (in Excel 2007) or Before.xls (in Excel 2003) and the second Excel workbook with the file name After.xlsm (in Excel 2007) or After.xls (in Excel 2003). Both workbooks should be open in Excel at this point.

Note In Excel 2007, click Office Button ➤ Save As ➤ Excel Macro-Enabled Workbook to save the workbook with the extension .xlsm.

---

3. Open the VBE:
   - In Excel 2007, click Developer ➤ (Code) Visual Basic.

Note If you do not see the Developer tab, see the sidebar in section 7.1 to learn how to display it.

   - In Excel 2003, click Tools ➤ Macro ➤ Visual Basic Editor.
4. In the Project Explorer window, right-click the VBAProject (Before.xlsm in Excel 2007) or the VBAProject (Before.xls in Excel 2003) node, and then click Insert ➤ Module.
5. In the Code window, type this code:

```
Public Sub HelloWorld()

    MsgBox "Hello from the " & ThisWorkbook.Name & " workbook!"

End Sub
```

6. Click anywhere between the first and last lines of code, and then click Run ➤ Run Sub/User Form. The message "Hello from the Before.xlsm workbook!" (in Excel 2007) or "Hello from the Before.xls workbook!" (in Excel 2003) appears. Click OK.
7. Right-click the Module1 subnode in the VBAProject (Before.xlsm in Excel 2007) or the VBAProject (Before.xls in Excel 2003) node, and click Export File. Save the Module1.bas file to a convenient location.
8. In the Project Explorer window, right-click the VBAProject (After.xlsm in Excel 2007) or the VBAProject (After.xls in Excel 2003) node, and then click Import File.
9. Browse to and select the Module1.bas file, and then click Open.

10. In the Project Explorer window, expand the Modules subnode in the VBAProject (After.xlsm in Excel 2007) or the VBAProject (After.xls in Excel 2003) node, and double-click the Module1 node.

11. Click anywhere between the first and last lines of code, and then click Run ➤ Run Sub/User Form. The message "Hello from the After.xlsm workbook!" (in Excel 2007) or "Hello from the After.xls workbook!" (in Excel 2003) appears. Click OK.

12. Add one line of code (as shown in bold) to the body of the Module1 code module in the VBAProject (After.xlsm in Excel 2007) or the VBAProject (After.xls in Excel 2003) node:

```
Public Sub HelloWorld()

    MsgBox "Hello from the " & ThisWorkbook.Name & " workbook!"
    MsgBox "This code will not appear in the Before.xls(m) workbook."

End Sub
```

13. Click File ➤ Save After.xlsm (in Excel 2007) or File ➤ Save After.xls (in Excel 2003).

14. In the Project Explorer window, double-click the Module1 node in the VBAProject (Before.xlsm in Excel 2007) or the VBAProject (Before.xls in Excel 2003) node. Notice that the code you added to the Module1 subnode in the VBAProject (After.xlsm in Excel 2007) or the VBAProject (After.xls in Excel 2003) node does not appear in this code. The code in the Before.xlsm (in Excel 2007) or Before.xls (in Excel 2003) Module1 code module is now separate from the code in the After.xlsm (in Excel 2007) or After.xls (for Excel 2003) Module1 code module.

7.3 Understand the Excel Programming Model

Understanding Excel's collection of programmatic objects is key to being able to successfully automate Excel using VBA code. There are about 200 programmatic objects and collections of objects in Excel 2003 VBA, and close to 250 programmatic objects and collections of objects in Excel 2007 VBA. Fortunately you don't have to know them all to begin with. If you learn how to automate the five most important Excel objects—`Application`, `Workbooks`, `Workbook`, `Worksheet`, and `Range`—you can leverage them to automate the rest of the objects. For example, once you have access to a `Workbook` object, you can automate the workbook's individual worksheets, and in turn you can automate each worksheet's individual cells.

Quick Start

- To access the Excel application, use Excel VBA's `Application` object.
- To access a set of Excel workbooks, use Excel VBA's `Workbooks` collection.
- To access an Excel workbook, use Excel VBA's `Workbook` object.
- To access an Excel worksheet, use Excel VBA's `Worksheet` object.
- To access one or more Excel cells, use Excel VBA's `Range` object.

How To

To access the Excel application, call the `Application` object, as in this example:

```
Excel.Application.Speech.Speak Text:="Hello, World!"
```

To access a set of Excel workbooks, use the `Workbooks` collection, as in this example:

```
Dim wkbs As Excel.Workbooks

Set wkbs = Excel.Application.Workbooks

MsgBox Prompt:=wkbs.Count
```

To access an Excel workbook, use the `Workbook` object (or the `ThisWorkbook` object to access the current workbook), as in this example:

```
Dim wkb As Excel.Workbook

' And one of the following sets of code:

Set wkb = Excel.Application.Workbooks.Item _
    (Index:="ExcelDB_Ch07_01-12.xls")
MsgBox Prompt:=wkb.FullName

Set wkb = Excel.Application.ThisWorkbook
MsgBox Prompt:=wkb.FullName
```

To access an Excel worksheet, use the `Worksheet` object (or the `ActiveSheet` property to access the current worksheet), as in this example:

```
' One of the following lines of code:

MsgBox Prompt:=Excel.Application.Workbooks.Item _
    (Index:="ExcelDB_Ch07_01-12.xls"). _
    Worksheets.Item(Index:="Sample Data").Name

MsgBox Prompt:=Excel.Application.ThisWorkbook.ActiveSheet.Name

' Or:

Dim wks As Excel.Worksheet

' And one of the following sets of code:

Set wks = Excel.Application.Workbooks.Item _
    (Index:="ExcelDB_Ch07_01-12.xls"). _
    Worksheets.Item(Index:="Sample Data")
MsgBox Prompt:=wks.Name

Set wks = Excel.Application.ThisWorkbook.ActiveSheet
MsgBox Prompt:=wks.Name
```

EXCEL PROGRAMMING MODEL CODE SHORTCUTS

Excel VBA lets you omit the Excel library qualifier and the `Application` property to return the `Application` object, as long as you are not referring to any other `Application` objects defined in other libraries at the same time. For example, the following lines of code are equivalent to referring to an Excel workbook:

```
MsgBox Prompt:=Excel.Application.Workbooks.Item _
    (Index:="ExcelDB_Ch07_01-12.xls"). _
    Worksheets.Item(Index:="Sample Data").Name

MsgBox Prompt:=Excel.Workbooks.Item _
    (Index:="ExcelDB_Ch07_01-12.xls"). _
    Worksheets.Item(Index:="Sample Data").Name

MsgBox Prompt:=Application.Workbooks.Item _
    (Index:="ExcelDB_Ch07_01-12.xls"). _
    Worksheets.Item(Index:="Sample Data").Name

MsgBox Prompt:=Workbooks.Item _
    (Index:="ExcelDB_Ch07_01-12.xls"). _
    Worksheets.Item(Index:="Sample Data").Name
```

Similarly, the following lines of code are equivalent to accessing the current workbook by using the `ThisWorkbook` property to return the `ThisWorkbook` object:

```
MsgBox Prompt:=Excel.Application.ThisWorkbook.Worksheets.Item _
    (Index:="Sample Data").Name

MsgBox Prompt:=Excel.ThisWorkbook.Worksheets.Item _
    (Index:="Sample Data").Name

MsgBox Prompt:=Application.ThisWorkbook.Worksheets.Item _
    (Index:="Sample Data").Name

MsgBox Prompt:=ThisWorkbook.Worksheets.Item _
    (Index:="Sample Data").Name
```

To access one or more Excel cells, use the `Range` object, as in this example:

```
Dim rng As Excel.Range

Set rng = Excel.Application.Workbooks.Item(Index:="ExcelDB_Ch07_01-12.xls"). _
    Worksheets.Item(Index:="Sample Data").Range(Cell1:="A1")

MsgBox prompt:=rng.Value2
```

USING THE RANGE PROPERTY'S CELL1 AND CELL2 ARGUMENTS

You can access Excel cells with the `Range` property. To access a single cell, use code similar to `.Range(Cell1:="B5")`. To access a group of cells, do one of the following:

- Use a cell reference for the `Cell1` argument with code similar to `.Range(Cell1:="B4:D11")`.
- Use a starting cell reference for the `Cell1` argument and an ending cell reference for the `Cell2` argument with code similar to `.Range(Cell1:="B4", Cell2:="D11")`.
- Use a named cell group for the `Cell1` argument with code similar to `.Range(Cell1:="StoreData")`.

Try It

To experiment with writing code to work with the Excel programming model, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the following subroutines in the SampleCode code module to help you get started writing your own code: AccessExcelApplicationExample, AccessWorkbooksExample, AccessWorkbookExample, AccessWorkbookByNameExample, AccessWorksheetsExample, AccessWorksheetExample, AccessWorksheetByNameExample, and AccessCellRangeExample. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications."

7.4 Automate Sorting Data

Just as you can use Excel's menu commands to manually sort data values, you can use Excel's Sort method to automate sorting data values with Excel VBA code. For example, you may want to automatically change how data values are sorted based on how the user types in or imports certain data values.

Quick Start

To automate sorting data with code, use the `Range` object's `Sort` method.

How To

To automate sorting data values with Excel VBA code, do this:

1. Get an instance of a `Range` object that contains the data to be sorted. For example, consider this code:

```
Dim wks As Excel.Worksheet
Dim rng As Excel.Range

Set wks = ThisWorkbook.Worksheets.Item(Index:="Sample Data")
Set rng = wks.Range(Cell1:="StoreData")
```

   This code accesses the cell group named StoreData on the Sample Data worksheet in the active Excel workbook.

2. Call the Range object's Sort method. For example, consider this code:

```
rng.Sort Key1:="Employees", Order1:=xlAscending, Header:=xlYes
```

   This code instructs Excel to sort the cell group named StoreData in ascending order, based on the values in the Employees column. The code Header:=xlYes instructs Excel to find the Employees column by searching the field names in the StoreData cell group's first row.

Try It

To experiment with writing code to automate sorting data, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's SortingDataExample subroutine and the ExcelHelpers code module's SortData subroutine. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications," and section "7.3: Understand the Excel Programming Model."

7.5 Automate Filtering Data

Just as you can use worksheet cells' AutoFilter buttons to display only those rows and columns that meet certain criteria, you can use the AutoFilter method to automate filtering data with Excel VBA code. For example, you may want to automatically change how data values are filtered depending on how certain data values are typed in or imported by the user.

Quick Start

To automate filtering data with code, use the Range object's AutoFilter method.

How To

To automate sorting data with Excel VBA code, do this:

1. Get an instance of a Range object that contains the data to be filtered. For example, consider this code:

```
Dim wks As Excel.Worksheet
Dim rng As Excel.Range

Set wks = ThisWorkbook.Worksheets.Item(Index:="Sample Data")
Set rng = wks.Range(Cell1:="StoreData")
```

   This code accesses the cell group named StoreData on the Sample Data worksheet in the active Excel workbook.

2. Call the Range object's Filter method. For example, consider this code:

```
rng.AutoFilter Field:=4, Criteria1:=">500", VisibleDropDown:=True
```

   This code instructs Excel to filter the cell group named StoreData based on the values in the fourth field from the left edge of the StoreData cell group. Only rows with a value of more than 500 in the fourth column are displayed. The code VisibleDropDown:=True instructs Excel to also display drop-down arrows in the cell group's first row.

Try It

To experiment with writing code to automate filtering data, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's FilteringDataExample subroutine and the ExcelHelpers code module's FilterData subroutine. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications," and section "7.3: Understand the Excel Programming Model."

7.6 Automate Subtotaling Data

Just as you can use Excel's menu commands to manually subtotal data values, you can use Excel's `Subtotal` method to automate subtotaling data values with Excel VBA code. For example, you may want to automatically change how data values are subtotaled based on how the data values are typed in or imported by the user.

Quick Start

To automate filtering data with code, use the `Range` object's `Subtotal` method.

How To

To automate subtotaling data with Excel VBA code, do this:

1. Get an instance of a `Range` object that contains the data to be sorted. For example, consider this code:

   ```
   Dim wks As Excel.Worksheet
   Dim rng As Excel.Range

   Set wks = ThisWorkbook.Worksheets.Item(Index:="Sample Data")
   Set rng = wks.Range(Cell1:="StoreData")
   ```

   This code accesses the cell group named StoreData on the Sample Data worksheet in the active Excel workbook.

2. Create an array of `Variant` objects to hold the column numbers by which to subtotal, and then add the field numbers to the array. For example, consider this code:

   ```
   Dim fieldNumbers() As Variant
   fieldNumbers = Array(4)
   ```

   This code stores the number 4, which will be used in the `Subtotal` method to refer to the fourth column number. If you want to add more field numbers, you could substitute code such as `fieldNumbers = Array(3, 4)` to specify the third and fourth column numbers.

3. Call the `Range` object's `Subtotal` method. For example, consider this code:

   ```
   rng.Subtotal GroupBy:=2, Function:=xlSum, TotalList:=fieldNumbers
   ```

   This code instructs Excel to subtotal the cell group named StoreData. It groups the subtotals based on the values in the second column from the left edge of the StoreData cell group. Excel then adds up the values in the fourth column from the left edge of the StoreData cell group and displays the sum for each subtotal.

Try It

To experiment with writing code to automate subtotaling data, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's SubtotalingDataExample subroutine and the ExcelHelpers code module's SubtotalData subroutine. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications" and section "7.3: Understand the Excel Programming Model."

7.7 Automate Calculating a Worksheet Function

Just as you can manually type a worksheet function in a cell or use the Insert Function dialog box to assist you in inserting a worksheet function into a cell, you can use the `Formula` property to automate inserting a worksheet function in a cell. For example, you may want to insert a specific worksheet function based on how the data values in a certain column are typed in or imported by a user.

Quick Start

To automate calculating a worksheet function, use the `Range` object's `Formula` property.

How To

To automate calculating a worksheet function with Excel VBA code, do this:

1. Get an instance of a `Range` object that contains the cell into which the worksheet function will be inserted and then calculated. For example, consider this code:

   ```
   Dim wks As Excel.Worksheet
   Dim rng As Excel.Range

   Set wks = ThisWorkbook.Worksheets.Item(Index:="Sample Data")
   Set rng = wks.Range(Cell1:="D18")
   ```

   This code accesses cell D18 on the Sample Data worksheet in the active Excel workbook.

2. Call the `Range` object's `Formula` property. For example, consider this code:

   ```
   rng.Formula = "=SUM(D2:D17)"
   ```

   This code instructs Excel to insert the SUM worksheet function into cell D18. This function displays the sum of the values in cells D2 through D17.

Try It

To experiment with writing code to insert worksheet functions, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's CalculatingWorksheetFunctionExample subroutine and the ExcelHelpers code module's CalculateWorksheetFunction subroutine. For more information on how to work with Excel VBA

code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications" and section "7.3: Understand the Excel Programming Model."

7.8 Automate Offsets

Just as you can use the OFFSET worksheet function to locate a cell that is a given distance from another cell, you can automate locating a cell with the `Offset` method using Excel VBA code. For example, you could automatically find the location of a given cell as rows or columns are inserted or deleted around that cell.

Quick Start

To automate calculating a cell offset, use the `Range` object's `Offset` method.

How To

To automate calculating a cell offset with Excel VBA code, do this:

1. Get an instance of a `Range` object from which to calculate the offset. For example, consider this code:

   ```
   Dim wks As Excel.Worksheet
   Dim rng As Excel.Range

   Set wks = ThisWorkbook.Worksheets.Item(Index:="Sample Data")
   Set rng = wks.Range(Cell1:="A16")
   ```

   This code accesses cell A16 on the Sample Data worksheet in the active Excel workbook.

2. Get an instance of a `Range` object that will reference the offset cell. For example

   ```
   Dim rngOffset As Excel.Range
   ```

3. Calculate the offset using the `Offset` method. For example, consider this code:

   ```
   Set rngOffset = rng.Offset(rowOffset:=0, columnOffset:=3)

   MsgBox Prompt:="After calculating the offset from cell " & _
       rng.Address & ", the value of cell " & rngOffset.Address & _
       " is " & rngOffset.Value2 & "."
   ```

   This code returns the cell offset that is three columns to the right of cell A16 (which is cell D16), and displays the offset cell's value in a message box.

Tip You could also insert the result of an offset calculation by using the `Range` object's `Formula` property with code similar to this:

```
Dim rngValue As Excel.Range

Set rng = wks.Range(Cell1:="A16")

rngValue.Formula = "=OFFSET(A16, 0, 3)"
```

Try It

To experiment with writing code to calculate cell offsets, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's Calculating-OffsetExample subroutine and the ExcelHelpers code module's CalculateOffset function. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications" and section "7.3: Understand the Excel Programming Model."

7.9 Automate HLOOKUP and VLOOKUP

Just as you can use the HLOOKUP and VLOOKUP worksheet functions to locate a given cell in a group of cells, you can automate locating cells with the `HLookup` and `VLookup` methods using Excel VBA code. For example, you could automatically find the location of given cells as rows or columns are inserted or deleted around those cells.

Quick Start

To calculate the result of an HLOOKUP worksheet function, call the `Worksheet` object's `HLookup` method.

To calculate the result of a VLOOKUP worksheet function, call the `Worksheet` object's `VLookup` method.

How To

To calculate the result of an HLOOKUP worksheet function with Excel VBA code, do this:

1. Get an instance of a `Range` object from which to calculate the horizontal lookup. For example, consider this code:

   ```
   Dim wks As Excel.Worksheet
   Dim rng As Excel.Range

   Set wks = ThisWorkbook.Worksheets.Item(Index:="Sample Data")
   Set rng = wks.Range(Cell1:="D19")
   ```

   This code accesses cell D19 on the Sample Data worksheet in the active Excel workbook.

2. Calculate the lookup using the `HLookup` method. For example consider this code:

```
rng.Value2 = Excel.Application.WorksheetFunction.HLookup(Arg1:="Store Number", _
    Arg2:=Range(Cell1:="StoreData"), Arg3:=7, Arg4:=False)
```

   In this code, the `HLookup` method takes the following arguments:

   - `Arg1` is the field in the cells' first row to look up—in this case, the Store Number field.
   - `Arg2` is the cells to reference—in this case, the group of cells named StoreData.
   - `Arg3` is the row to reference—in this case, the seventh row.
   - `Arg4` is an approximate match (`True`) or an exact match (`False`, as in this case).

To calculate the result of a VLOOKUP worksheet function with Excel VBA code, do this:

1. Get an instance of a `Range` object from which to calculate the vertical lookup. For example, consider this code:

```
Dim wks As Excel.Worksheet
Dim rng As Excel.Range

Set wks = ThisWorkbook.Worksheets.Item(Index:="Sample Data")
Set rng = wks.Range(Cell1:="D20")
```

   This code accesses cell D20 on the Sample Data worksheet in the active Excel workbook.

2. Calculate the lookup using the `VLookup` method. For example consider this code:

```
rng.Value2 = Excel.Application.WorksheetFunction.VLookup(Arg1:="South", _
    Arg2:=Range(Cell1:="StoreData"), Arg3:=4, Arg4:=False)
```

   In this code, the `VLookup` method takes the following arguments:

   - `Arg1` is the value in the first column to look up—in this case, the value South.
   - `Arg2` is the cells to reference—in this case, the group of cells named StoreData.
   - `Arg3` is the column to reference—in this case, the fourth column.
   - `Arg4` is an approximate match (`True`) or an exact match (`False`, as in this case).

Tip You could also insert the results of the HLOOKUP and VLOOKUP worksheet functions by using the `Range` object's `Formula` property with code similar to this:

```
rng.Formula = "=HLOOKUP(""Store Number"", StoreData, 7, False)"
rng.Formula = "=VLOOKUP(""South"", StoreData, 4, False)"
```

Try It

To experiment with writing code to calculate HLOOKUP and VLOOKUP worksheet functions, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's CalculatingHLOOKUPExample and CalculatingVLOOKUPExample subroutines and the ExcelHelpers code module's CalculateHLOOKUP and CalculateVLOOKUP subroutines. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications" and section "7.3: Understand the Excel Programming Model."

7.10 Automate Creating a PivotTable and PivotChart

Just as you can use the PivotTable and PivotChart Wizard to manually create a PivotTable and a PivotChart, you can automate creating a PivotTable and a PivotChart with Excel VBA code. For example, you may want to automatically create a PivotTable and a PivotChart based on data values that a user types in or references or imports from an external data source.

Quick Start

To create a PivotTable and a PivotChart, do this:

1. Create a `PivotCache` object, a `PivotTable` object, and a `Chart` object.
2. Call the `Add` method of the desired workbook's `PivotCaches` collection to set the `PivotCache` object to the desired workbook's `PivotCaches` collection.
3. Call the `PivotCache` object's `CreatePivotTable` method to create a PivotTable and set the PivotTable object to the result of the `CreatePivotTable` method call.
4. Add a `Chart` object to the desired workbook.
5. Call the `Chart` object's `SetSourceData` method, passing to the method the value of the `PivotTable` object's `TableRange2` property.

How To

To create a PivotTable and a PivotChart using Excel VBA code, do this:

1. Create a `Workbook` object, a `PivotCache` object, a `PivotTable` object, a `Worksheet` object, and a `Chart` object. For example, consider this code:

```
Dim wkb As Excel.Workbook
Dim pvc As Excel.PivotCache
Dim pvt As Excel.PivotTable
Dim wks As Excel.Worksheet
Dim cht As Excel.Chart
```

```
Set wkb = Excel.Application.ThisWorkbook
    Set wks = wkb.Worksheets.Add

wks.Name = "PivotTable Example"
```

This code references the active workbook, adds a new blank worksheet to the workbook, and names the new worksheet PivotTable Example.

2. Call the `Add` method of the desired workbook's `PivotCaches` collection to set the `PivotCache` object to the desired workbook's `PivotCaches` collection. For example, consider this code:

```
Set pvc = wkb.PivotCaches.Add(SourceType:=xlDatabase, _
    SourceData:=Range(Cell1:="StoreData"))
```

This code initializes a `PivotCache` object and sets its data source to the cell group with the name StoreData in the workbook.

3. Call the `PivotCache` object's `CreatePivotTable` method to create a PivotTable and set the `PivotTable` object to the result of the `CreatePivotTable` method call. For example, consider this code:

```
Set pvt = pvc.CreatePivotTable(TableDestination:=wks.Range(Cell1:="A3"), _
    TableName:="StoreDataPivotTable")
```

This code creates a PivotTable, inserts it at cell A3 in the PivotTable Example worksheet, and gives the PivotTable the name StoreDataPivotTable.

4. Add a `Chart` object to the desired workbook, with code similar to this:

```
Set cht = wkb.Charts.Add
```

5. Set the chart's display format; give the chart a name; and then call the `Chart` object's `SetSourceData` method, passing to the method the value of the `PivotTable` object's `TableRange2` property. For example, consider this code:

```
With cht

    .ChartType = xl3DColumn
    .Name = "PivotChart Example"
    .SetSourceData pvt.TableRange2

End With
```

This code sets the chart's display format to 3-D Column, names the chart PivotChart Example, and sets the chart's data source to the data source of the PivotTable named StoreDataPivotTable.

Tip You can also create a PivotTable by calling the `Worksheet` object's `PivotTableWizard` method. Although this is a quicker approach, it provides less flexibility and can't be used with OLE DB-based data sources. For example, using this approach, you could write code similar to this:

```
Dim wkb As Excel.Workbook
Dim wks As Excel.Worksheet

Set wkb = Excel.Application.ThisWorkbook
Set wks = wkb.Worksheets.Add

wks.Name = "PivotTable Example"
wks.PivotTableWizard SourceType:=xlDatabase, _
    SourceData:=Range(Cell1:="StoreData"), _
    TableDestination:=wks.Range(Cell1:="A3"), _
    TableName:="StoreDataPivotTable"
```

Try It

To experiment with writing code to create PivotTables and PivotCharts, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's CreatingPivotTableAndPivotChartExample subroutine and the ExcelHelpers code module's CreatePivotTableAndPivotChart subroutine. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications" and section "7.3: Understand the Excel Programming Model."

7.11 Automate Changing the View of a PivotTable and PivotChart

Just as you can use the PivotTable Field List to manually change the view of a PivotTable and its associated PivotChart, you can automate these view changes using Excel VBA code. For example, you could automatically change the view of a PivotTable and its associated PivotChart based on how a user adds data to or deletes data from the underlying data source.

Quick Start

To change the view of a PivotTable and PivotChart, call the `PivotTable` object's `AddFields` and `AddDataField` methods. The linked PivotChart's view will change to synchronize with the PivotTable's view.

How To

To change the view of a PivotTable and PivotChart using Excel VBA code, do this:

1. Create a `Workbook` object, a `Worksheet` object, and a `PivotTable` object, and then initialize the `Workbook`, `Worksheet`, and `PivotTable` objects. For example, consider this code:

```
Dim wkb As Excel.Workbook
Dim wks As Excel.Worksheet
Dim pvt As Excel.PivotTable

Set wkb = Excel.Application.ThisWorkbook
Set wks = wkb.Worksheets(Index:="PivotTable Example")
Set pvt = wks.PivotTables(Index:="StoreDataPivotTable")
```

   This code references a PivotTable with the name StoreDataPivotTable on a worksheet named PivotTable Example in the current workbook.

2. Call the `PivotTable` object's `AddFields` method to add fields to the row area or page area, and call the `PivotTable` object's `AddDataField` method to add a field to the data area. For example, consider this code:

```
With pvt

    .AddFields RowFields:=Array("Region", "State"), _
        PageFields:="Store Number"
    .AddDataField Field:=pvt.PivotFields(Index:="Employees"), _
        Function:=xlSum

End With
```

   The `AddFields` method call adds the two fields, a Region field and a State field, to the row area. The `AddDataField` method call adds an Employees field to the data area, and the field displays the sum of the employees by region and by state.

Try It

To experiment with writing code to change PivotTable and PivotChart views, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's ChangingPivotTableAndPivotChartViewsExample subroutine. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications" and section "7.3: Understand the Excel Programming Model."

7.12 Automate Connecting to External Data

Just as you can manually connect to external data using Excel's menu commands and their associated tools and wizards, you can automate connecting to external data using Excel VBA code. For example, you may want to automatically change an external data source connection or its associated connection behavior based on how the user adds data to or removes data from the external data source.

Quick Start

To automate connecting to external data, use the `QueryTable` object, which represents a cell group that displays the external data in a worksheet.

How To

To automate connecting to external data with Excel VBA code, such as connecting to data in an external text file, do this:

1. Create a `Workbook` object and a `Worksheet` object, and then initialize the `Workbook` and `Worksheet` objects. For example, consider this code:

   ```
   Dim wkb As Excel.Workbook
   Dim wks As Excel.Worksheet

   Set wkb = xlApp.ThisWorkbook
   Set wks = wkb.Worksheets(Index:="Sample Data Connection")
   ```

   This code references a worksheet named Sample Data Connection in the current workbook. This worksheet will be used to display the data from the external text file.

2. Create and initialize a `String` object representing a connection string that Excel will use to locate the external text file. For example, consider this code:

   ```
   Dim strConn As String

   strConn = "TEXT;C:\My Sample Excel Files\ExcelDB_Ch01_02.txt"
   ```

   The connection string instructs Excel to find a text file named ExcelDB_Ch01_02.txt in the C:\My Sample Excel Files folder.

3. Create a `QueryTable` object, and then use the `QueryTables` collection's `Add` method to add a `QueryTable` object to the `QueryTables` collection. Then, for the `QueryTable` object, specify the connection string, where to begin displaying the data from the external text file, how to import the data, and what to name the cell group that displays the external data. For example, consider this code:

   ```
   Dim qyt As Excel.QueryTable

   Set qyt = wks.QueryTables.Add(Connection:=strConn, _
       Destination:=Range(Cell1:="A1"))

   With qyt

      .TextFileCommaDelimiter = True
      .Refresh
      .Name = "ExcelDB_Ch01_02 Query Table"

   End With
   ```

This code uses the QueryTables collection's Add method to set the QueryTable object's connection string and where to begin displaying the data from the external text file (beginning at cell A1 in the Sample Data Connection worksheet). The code then uses the QueryTable object's TextFileCommaDelimiter property to instruct Excel to use the comma characters in the external text file to determine how to display the data in the worksheet. The QueryTable object's Refresh method displays the external text file's data in the worksheet, and the QueryTable object's Name property instructs Excel to give the name ExcelDB_Ch01_02 Query Table to the cell group displaying the external text file's data.

Tip

The format of the Connection argument in the QueryTables collection's Add method depends on the external data source type. Here are some examples:

- For an OLE DB or ODBC external data source, specify a string containing an OLE DB or ODBC connection string. The ODBC connection string has the form "ODBC;<connection string>".
- For an ADO or DAO recordset, create and initialize an ADO or DAO Recordset object, and then provide the name of the ADO or DAO Recordset object.
- For a Web-based external data source, specify a string in the form "URL;<url>".

For additional examples, see the Add method documentation in Excel VBA Help.

Try It

To experiment with writing code to connect to external data, you can start with the sample code that is provided in the ExcelDB_Ch07_01-12.xls file. See the SampleCode code module's ConnectingToExternalTextFileExample subroutine. For more information on how to work with Excel VBA code in the VBE, see section "7.2: Understand Excel Visual Basic for Applications" and section "7.3: Understand the Excel Programming Model."