Contents

About the Authors

Bill Jelen, Excel MVP and the host of MrExcel.com, has been using spreadsheets since 1985, and he launched the MrExcel.com website in 1998. Bill was a regular guest on *Call for Help* with Leo Laporte and has produced more than 1,900 episodes of his daily video podcast, *Learn Excel from MrExcel*. He is the author of 44 books about Microsoft Excel and writes the monthly Excel column for *Strategic Finance* magazine. Before founding MrExcel.com, Bill Jelen spent 12 years in the trenches—working as a financial analyst for finance, marketing, accounting, and operations departments of a $500 million public company. He lives in Merritt Island, Florida, with his wife, Mary Ellen.

Tracy Syrstad is a Microsoft Excel developer and author of eight Excel books. She has been helping people with Microsoft Office issues since 1997, when she discovered free online forums where anyone could ask and answer questions. Tracy found out she enjoyed teaching others new skills, and when she began working as a developer, she was able to integrate the fun of teaching with one-on-one online desktop sharing sessions. Tracy lives on acreage in eastern South Dakota with her husband, one dog, two cats, one horse (two, hopefully soon), and a variety of wild foxes, squirrels, and rabbits.

Dedications

For Robert K. Jelen
—Bill Jelen

For Marlee Jo Jacobson
—Tracy Syrstad

Acknowledgments

Thanks to Tracy Syrstad for being a great coauthor.

Bob Umlas is the smartest Excel guy I know and is an awesome technical editor. At Pearson, Joan Murray is an excellent acquisitions editor.

Along the way, I've learned a lot about VBA programming from the awesome community at the MrExcel.com message board. VoG, Richard Schollar, and Jon von der Heyden all stand out as having contributed posts that led to ideas in this book. Thanks to Pam Gensel for Excel macro lesson #1. Mala Singh taught me about creating charts in VBA, and Oliver Holloway brought me up to speed with accessing SQL Server. Scott Ruble and Robin Wakefield at Microsoft helped with the charting chapter.

My family was incredibly supportive during this time. Thanks to Mary Ellen Jelen, Robert F. Jelen, and Robert K. Jelen.

—Bill

Juan Pablo Gonzalez Ruiz and Zack Barresse are great programmers, and I really appreciate their time and patience showing me new ways to write better programs. Chris "Smitty" Smith has really helped me sharpen my business acumen.

Thank you to all the moderators at the MrExcel forum who keep the board organized, despite the best efforts of the spammers.

Programming is a constant learning experience, and I really appreciate the clients who have encouraged me to program outside my comfort zone so that my skills and knowledge have expanded.

And last, but not least, thanks to Bill Jelen. His site, MrExcel.com, is a place where thousands come for help. It's also a place where I, and others like me, have an opportunity to learn from and assist others.

—Tracy

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@quepublishing.com

Mail: Que Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at quepublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

INTRODUCTION

As corporate IT departments have found themselves with long backlogs of requests, Excel users have discovered that they can produce the reports needed to run their businesses themselves using the macro language *Visual Basic for Applications* (VBA). VBA enables you to achieve tremendous efficiencies in your day-to-day use of Excel. VBA helps you figure out how to import data and produce reports in Excel so that you don't have to wait for the IT department to help you.

IN THIS INTRODUCTION

What Is in This Book?

You have taken the right step by purchasing this book. We can help you reduce the learning curve so that you can write your own VBA macros and put an end to the burden of generating reports manually.

Reducing the Learning Curve

This Introduction provides a case study about the power of macros. Chapter 1, "Unleashing the Power of Excel with VBA," introduces the tools and confirms what you probably already know: The macro recorder does not work reliably. Chapter 2, "This Sounds Like BASIC, So Why Doesn't It Look Familiar?" helps you understand the crazy syntax of VBA. Chapter 3, "Referring to Ranges," cracks the code on how to work efficiently with ranges and cells.

Chapter 4, "Looping and Flow Control," covers the power of looping using VBA. The case study in this chapter demonstrates creating a program to produce a department report and then wrapping that report routine in a loop to produce 46 reports.

Chapter 5, "R1C1-Style Formulas," covers, obviously, R1C1-style formulas. Chapter 6, "Creating and Manipulate Names in VBA," covers names. Chapter 7, "Event Programming," includes some great tricks that use event programming. Chapters 8, "Arrays,"

and 9, "Creating Classes and Collections," cover arrays, classes, and collections. Chapter 10, "Userforms: An Introduction," introduces custom dialog boxes that you can use to collect information from a human using Excel.

Excel VBA Power

Chapters 11, "Data Mining with Advanced Filter," and 12, "Using VBA to Create Pivot Tables," provide an in-depth look at Filter, Advanced Filter, and pivot tables. Report automation tools rely heavily on these concepts. Chapters 13, "Excel Power," and 14, "Sample User-Defined Functions," include dozens of code samples designed to exhibit the power of Excel VBA and custom functions.

Chapters 15, "Creating Charts," through 20, "Automating Word," handle charting, data visualizations, web queries, sparklines, and automating Word.

Techie Stuff Needed to Produce Applications

Chapter 21, "Using Access as a Back End to Enhance Multiuser Access to Data," handles reading and writing to Access databases and SQL Server. The techniques for using Access databases enable you to build an application with the multiuser features of Access while keeping the friendly front end of Excel.

Chapter 22, "Advanced Userform Techniques," shows you how to go further with userforms. Chapter 23, "The Windows Application Programming Interface (API)," teaches some tricky ways to achieve tasks using the Windows API. Chapters 24, "Handling Errors," through 26, "Creating Add-ins," deal with error handling, custom menus, and add-ins. Chapter 27, "An Introduction to Creating Office Add-Ins," provides a brief introduction to building your own JavaScript application within Excel. Chapter 28, "What's New in Excel 2016 and What's Changed," summarizes the changes in Excel 2016.

Does This Book Teach Excel?

Microsoft believes that the ordinary Office user touches only 10% of the features in Office. We realize that everyone reading this book is above average, and MrExcel.com has a pretty smart audience. Even so, a poll of 8,000 MrExcel.com readers showed that only 42% of smarter-than-average users are using any 1 of the top 10 power features in Excel.

I regularly present a Power Excel seminar for accountants. These are hard-core Excelers who use Excel 30 to 40 hours every week. Even so, two things come out in every seminar. First, half of the audience gasps when they see how quickly you can do tasks with a particular feature, such as automatic subtotals or pivot tables. Second, someone in the audience routinely trumps me. For example, someone asks a question, I answer, and someone in the second row raises a hand to give a better answer.

The point? You and I both know a lot about Excel. However, I assume that in any given chapter, maybe 58% of the people have not used pivot tables before and maybe even fewer have used the Top 10 Filter feature of pivot tables. With this in mind, before I show how to

automate something in VBA, I briefly cover how to do the same task in the Excel interface. This book does not teach you how to make pivot tables, but it does alert you when you might need to explore a topic and learn more about it elsewhere.

CASE STUDY: MONTHLY ACCOUNTING REPORTS

This is a true story. Valerie is a business analyst in the accounting department of a medium-size corporation. Her company recently installed an overbudget $16 million enterprise resource planning (ERP) system. As the project ground to a close, there were no resources left in the IT budget to produce the monthly report that this corporation used to summarize each department.

However, Valerie had been close enough to the implementation to think of a way to produce the report herself. She understood that she could export general ledger data from the ERP system to a text file with comma-separated values. Using Excel, Valerie was able to import the general ledger data from the ERP system into Excel.

Creating the report was not easy. As in many other companies, there were exceptions in the data. Valerie knew that certain accounts in one particular cost center needed to be reclassed as expenses. She knew that other accounts needed to be excluded from the report entirely. Working carefully in Excel, Valerie made these adjustments. She created one pivot table to produce the first summary section of the report. She cut the pivot table results and pasted them into a blank worksheet. Then she created a new pivot table report for the second section of the summary. After about three hours, she had imported the data, produced five pivot tables, arranged them in a summary, and neatly formatted the report in color.

Becoming the Hero

Valerie handed the report to her manager. The manager had just heard from the IT department that it would be months before they could get around to producing "that convoluted report." When Valerie created the Excel report, she became the instant hero of the day. In three hours, Valerie had managed to do the impossible. Valerie was on cloud nine after a well-deserved "atta-girl."

More Cheers

The next day, Valerie's manager attended the monthly department meeting. When the department managers started complaining that they could not get the report from the ERP system, this manager pulled out his department's report and placed it on the table. The other managers were amazed. How was he able to produce this report? Everyone was relieved to hear that someone had cracked the code. The company president asked Valerie's manager if he could have the report produced for each department.

Cheers Turn to Dread

You can probably see what's coming. This particular company had 46 departments. That means 46 one-page summaries had to be produced once a month. Each report required importing data from the ERP system, backing out certain accounts, producing five pivot tables, and then formatting the reports in color. It had taken Valerie three hours to produce the first report, but after she got into the swing of things, she could produce the 46 reports in 40 hours. Even after she reduced her time per report, though, this is horrible. Valerie had a job to do before she became responsible for spending 40 hours a month producing these reports in Excel.

VBA to the Rescue

Valerie found my company, MrExcel Consulting, and explained her situation. In the course of about a week, I was able to produce a series of macros in Visual Basic that did all the mundane tasks. For example, the macros imported the data, backed out certain accounts, made five pivot tables, and applied the color formatting. From start to finish, the entire 40-hour manual process was reduced to two button clicks and about 4 minutes.

Right now, either you or someone in your company is probably stuck doing manual tasks in Excel that can be automated with VBA. I am confident that I can walk into any company that has 20 or more Excel users and find a case just as amazing as Valerie's.

The Future of VBA and Windows Versions of Excel

Several years ago, there were many rumblings that Microsoft might stop supporting VBA. There is now plenty of evidence to indicate that VBA will be around in Windows versions of Excel through 2036. When VBA was removed from the Mac version of Excel 2008, a huge outcry from customers led to its being included in the next Mac version of Excel.

XLM macros were replaced by VBA in 1993, and 23 years later, they are still supported. Microsoft is making strides toward providing a JavaScript alternative to VBA, but it appears that Excel will support VBA for about another 23 years.

Versions of Excel

This fifth edition of *VBA and Macros* is designed to work with Excel 2016. The previous editions of this book covered code for Excel 97 through Excel 2013. In 80% of the chapters, the code for Excel 2016 is identical to the code in previous versions. However, there are exceptions. For example, the new AutoGroup functionality in pivot tables adds new options that were not available in Excel 2013.

Differences for Mac Users

Although Excel for Windows and Excel for the Mac are similar in terms of user interface, there are a number of differences when you compare the VBA environment. Certainly, nothing in Chapter 23 that uses the Windows API will work on the Mac. That said, the overall concepts discussed in this book apply to the Mac. You can find a general list of differences as they apply to the Mac at http://www.mrexcel.com/macvba.html. Development in VBA for Mac Excel 2016 is far more difficult than in Windows, with only rudimentary VBA editing tools. Microsoft actually recommends that you write all of your VBA in Excel 2016 for Windows and then use that VBA on the Mac.

Special Elements and Typographical Conventions

The following typographical conventions are used in this book:

- *Italic*—Indicates new terms when they are defined, special emphasis, non-English words or phrases, and letters or words used as words.
- `Monospace`—Indicates parts of VBA code, such as object or method names.
- **`Bold monospace`**—Indicates user input.

In addition to these typographical conventions, there are several special elements. Each chapter has at least one case study that presents a real-world solution to common problems. The case study also demonstrates practical applications of topics discussed in the chapter.

In addition to the case studies, you will see Notes, Tips, and Cautions.

> **NOTE**
>
> Notes provide additional information outside the main thread of the chapter discussion that might be useful for you to know.

> **TIP**
>
> Tips provide quick workarounds and time-saving techniques to help you work more efficiently.

> **CAUTION**
>
> Cautions warn about potential pitfalls you might encounter. Pay attention to the Cautions; they alert you to problems that might otherwise cause you hours of frustration.

Code Files

As a thank-you for buying this book, we have put together a set of 50 Excel workbooks that demonstrate the concepts included in this book. This set of files includes all the code from the book, sample data, additional notes from the authors, and 25 bonus macros. To download the code files, visit this book's web page at http://www.quepublishing.com or http://www.mrexcel.com/getcode2016.html.

Next Steps

Chapter 1 introduces the editing tools of the Visual Basic environment and shows why using the macro recorder is not an effective way to write VBA macro code.

This page intentionally left blank

This page intentionally left blank

Referring to Ranges

3

A *range* can be a cell, a row, a column, or a grouping of any of these. The `RANGE` object is probably the most frequently used object in Excel VBA; after all, you are manipulating data on a sheet. Although a range can refer to any grouping of cells on a sheet, it can refer to only one sheet at a time. If you want to refer to ranges on multiple sheets, you must refer to each sheet separately.

This chapter shows you different ways of referring to ranges, such as specifying a row or column. You'll also find out how to manipulate cells based on the active cell and how to create a new range from overlapping ranges.

IN THIS CHAPTER

The `Range` Object

The following is the Excel object hierarchy:

```
Application > Workbook > Worksheet >
Range
```

The `Range` object is a property of the `Worksheet` object. This means it requires that a sheet be active or else it must reference a worksheet. Both of the following lines mean the same thing if `Worksheets(1)` is the active sheet:

```
Range("A1")
Worksheets(1).Range("A1")
```

There are several ways to refer to a `Range` object. `Range("A1")` is the most identifiable because that is how the macro recorder refers to it. However, all the following are equivalent when referring to a range:

```
Range("D5")
[D5]
Range("B3").Range("C3")
Cells(5,4)
Range("A1").Offset(4,3)
Range("MyRange") 'assuming that D5 has a
'Name of MyRange
```

Which format you use depends on your needs. Keep reading....It will all make sense soon!

Syntax for Specifying a Range

The `Range` property has two acceptable syntaxes. To specify a rectangular range in the first syntax, specify the complete range reference just as you would in a formula in Excel:

```
Range("A1:B5")
```

In the alternative syntax, specify the upper-left corner and lower-right corner of the desired rectangular range. In this syntax, the equivalent statement might be this:

```
Range("A1", "B5")
```

For either corner, you can substitute a named range, the `Cells` property, or the `ActiveCell` property. The following line of code selects the rectangular range from A1 to the active cell:

```
Range("A1", ActiveCell).Select
```


The following statement selects from the active cell to five rows below the active cell and two columns to the right:

```
Range(ActiveCell, ActiveCell.Offset(5, 2)).Select
```

Named Ranges

You probably have already used named ranges on your worksheets and in formulas. You can also use them in VBA.

Use the following code to refer to the range `"MyRange"` in Sheet1:

```
Worksheets("Sheet1").Range("MyRange")
```

Notice that the name of the range is in quotes—unlike the use of named ranges in formulas on the sheet itself. If you forget to put the name in quotes, Excel thinks you are referring to a variable in the program. One exception is if you use the shortcut syntax discussed in the next section. In that case, quotes are not used.

Shortcut for Referencing Ranges

A shortcut is available when referencing ranges. The shortcut involves using square brackets, as shown in Table 3.1.

Table 3.1 ***Shortcuts*** **for Referencing Ranges**

Standard Method	Shortcut
`Range("D5")`	`[D5]`
`Range("A1:D5")`	`[A1:D5]`
`Range("A1:D5, G6:I17")`	`[A1:D5, G6:I17]`
`Range("MyRange")`	`[MyRange]`

Referencing Ranges in Other Sheets

Switching between sheets by activating the needed sheet slows down your code. To avoid this, refer to a sheet that is not active by first referencing the `Worksheet` object:

```
Worksheets("Sheet1").Range("A1")
```

This line of code references Sheet1 of the active workbook even if Sheet2 is the active sheet.

To reference a range in another workbook, include the `Workbook` object, the `Worksheet` object, and then the `Range` object:

```
Workbooks("InvoiceData.xlsx").Worksheets("Sheet1").Range("A1")
```

To use the `Range` property as an argument within another `Range` property, identify the range fully each time. For example, suppose that Sheet1 is your active sheet and you need to total data from Sheet2:

```
WorksheetFunction.Sum(Worksheets("Sheet2").Range(Range("A1"), _
    Range("A7")))
```


This line does not work. Why not? Although `Range("A1"), Range("A7")` is meant to refer to the sheet at the beginning of the code line (Sheet2), Excel does not assume that you want to carry the `Worksheet` object reference over to these other `Range` objects and assumes that they refer to the active sheet, Sheet1. So what do you do? Well, you could write this:

```
WorksheetFunction.Sum(Worksheets("Sheet2").Range(Worksheets("Sheet2"). _
    Range("A1"), Worksheets("Sheet2").Range("A7")))
```

But this not only is a long line of code but also difficult to read! Thankfully, there is a simpler way, using `With...End With`:

```
With Worksheets("Sheet2")
    WorksheetFunction.Sum(.Range(.Range("A1"), .Range("A7")))
End With
```

Notice now that there is a `.Range` in your code, but without the preceding object reference. That's because `With Worksheets("Sheet2")` implies that the object of the range is the worksheet. Whenever Excel sees a period without an object reference directly to the left of it, it looks up the code for the closest `With` statement and uses that as the object reference.

Referencing a Range Relative to Another Range

Typically, the `Range` object is a property of a worksheet. It is also possible to have `Range` be the property of another range. In this case, the `Range` property is relative to the original range, which makes for unintuitive code. Consider this example:

```
Range("B5").Range("C3").Select
```

This code actually selects cell D7. Think about cell C3, which is located two rows below and two columns to the right of cell A1. The preceding line of code starts at cell B5. If we assume that B5 is in the A1 position, VBA finds the cell that would be in the C3 position relative to B5. In other words, VBA finds the cell that is two rows below and two columns to the right of B5, which is D7.

Again, I consider this coding style to be very unintuitive. This line of code mentions two addresses, and the actual cell selected is neither of these addresses! It seems misleading when you are trying to read this code.

You might consider using this syntax to refer to a cell relative to the active cell. For example, the following line of code activates the cell three rows down and four columns to the right of the currently active cell:

```
Selection.Range("E4").Select
```

I mention this syntax only because the macro recorder uses it. Recall that when you recorded a macro in Chapter 1, "Unleashing the Power of Excel with VBA," with relative references on, the following line was recorded:

```
ActiveCell.Offset(0, 4).Range("A2").Select
```

This line found the cell four columns to the right of the active cell, and from there it selected the cell that would correspond to A2. This is not the easiest way to write code, but it is the way the macro recorder does it.

Although a worksheet is usually the object of the `Range` property, occasionally, such as during recording, a range may be the property of a range.

Using the `Cells` Property to Select a Range

The `Cells` property refers to all the cells of the specified `Range` object, which can be a worksheet or a range of cells. For example, this line selects all the cells of the active sheet:

```
Cells.Select
```

Using the `Cells` property with the `Range` object might seem redundant:

```
Range("A1:D5").Cells
```

This line refers to the original `Range` object. However, the `Cells` property has an `Item` property that makes the `Cells` property very useful. The `Item` property enables you to refer to a specific cell relative to the `Range` object.

The syntax for using the `Item` property with the `Cells` property is as follows:

```
Cells.Item(Row,Column)
```

You must use a numeric value for `Row`, but you may use the numeric value or string value for `Column`. Both of the following lines refer to cell C5:

```
Cells.Item(5,"C")
Cells.Item(5,3)
```

Because the `Item` property is the default property of the `Range` object, you can shorten these lines as follows:

```
Cells(5,"C")
Cells(5,3)
```

The ability to use numeric values for parameters is particularly useful if you need to loop through rows or columns. The macro recorder usually uses something like `Range("A1")`.

Select for a single cell and Range("A1:C5").Select for a range of cells. If you are learning to code only from the recorder, you might be tempted to write code like this:

```
FinalRow = Cells(Rows.Count, 1).End(xlUp).Row
For i = 1 to FinalRow
    Range("A" & i & ":E" & i).Font.Bold = True
Next i
```

This little piece of code, which loops through rows and bolds the cells in columns A through E, is awkward to read and write. But how else can you do it? Like this:

```
FinalRow = Cells(Rows.Count, 1).End(xlUp).Row
For i = 1 to FinalRow
        Cells(i,"A").Resize(,5).Font.Bold = True
Next i
```

Instead of trying to type the range address, the new code uses the Cells and Resize properties to find the required cell, based on the active cell. See the "Using the Resize Property to Change the Size of a Range" section later in this chapter, for more information on the Resize property.

You can use the Cells properties for parameters in the Range property. The following refers to the range A1:E5:

```
Range(Cells(1,1),Cells(5,5))
```

This is particularly useful when you need to specify variables with a parameter, as in the previous looping example.

Using the Offset Property to Refer to a Range

You have already seen a reference to Offset when you recorded a relative reference. Offset enables you to manipulate a cell based on the location of another cell, such as the active cell. Therefore, you do not need to know the address of the cell you want to manipulate.

The syntax for the Offset property is as follows:

```
Range.Offset(RowOffset, ColumnOffset)
```

For example, the following code affects cell F5 from cell A1:

```
Range("A1").Offset(RowOffset:=4, ColumnOffset:=5)
```

Or, shorter yet, you can write this:

```
Range("A1").Offset(4,5)
```

The count of the rows and columns starts at A1 but does not include A1.

If you need to go over only a row or a column, but not both, you don't have to enter both the row and the column parameters. To refer to a cell one column over, use one of these lines:

```
Range("A1").Offset(ColumnOffset:=1)
Range("A1").Offset(,1)
```

Both of these lines mean the same, so the choice is yours. If you use the second line, make sure to include the comma so Excel knows that the `1` refers to the `ColumnOffset` argument. Referring to a cell one row up is similar:

```
Range("B2").Offset(RowOffset:=-1)
Range("B2").Offset(-1)
```

Once again, you can choose which one to use. It is a matter of readability of the code.

Suppose you have a list of produce in column A, with totals next to the produce items in column B. If you want to find any total equal to zero and place `LOW` in the cell next to it, do this:

```
Set Rng = Range("B1:B16").Find(What:="0", LookAt:=xlWhole, _
    LookIn:=xlValues)
Rng.Offset(, 1).Value = "LOW"
```

When used in a `Sub` and looping through a data set, it would look like this:

```
Sub FindLow()
With Range("B1:B16")
    Set Rng = .Find(What:="0", LookAt:=xlWhole, LookIn:=xlValues)
    If Not Rng Is Nothing Then
        firstAddress = Rng.Address
        Do
            Rng.Offset(, 1).Value = "LOW"
            Set Rng = .FindNext(Rng)
        Loop While Not Rng Is Nothing And Rng.Address <> firstAddress
    End If
End With
End Sub
```

The `LOW` totals are noted by the program, as shown in Figure 3.1.

Figure 3.1
Find the produce with zero totals.

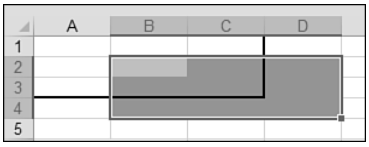
	A	B	C
1	Apples	45	
2	Oranges	12	
3	Grapefruit	86	
4	Lemons	0	LOW

> **NOTE** Refer to the section "Object Variables" in Chapter 4, "Looping and Flow Control," for more information on the `Set` statement.

Offsetting isn't only for single cells; you can use it with ranges. You can shift the focus of a range over in the same way you can shift the active cell. The following line refers to B2:D4 (see Figure 3.2):

```
Range("A1:C3").Offset(1,1)
```

Figure 3.2
Offsetting a range: `Range("A1:C3").Offset(1,1).Select`.



Using the `Resize` Property to Change the Size of a Range

The `Resize` property enables you to change the size of a range based on the location of the active cell. You can create a new range as needed. This is the syntax for the `Resize` property:

```
Range.Resize(RowSize, ColumnSize)
```

To create the range B3:D13, use the following:

```
Range("B3").Resize(RowSize:=11, ColumnSize:=3)
```

Here's a simpler way to create this range:

```
Range("B3").Resize(11, 3)
```

But what if you need to resize by only a row or a column—not both? You don't have to enter both the row and the column parameters.

To expand by two columns, use either of the following:

```
Range("B3").Resize(ColumnSize:=2)
```

or

```
Range("B3").Resize(,2)
```

Both lines mean the same thing. The choice is yours. If you use the second line, make sure to include the comma so Excel knows the `2` refers to the `ColumnSize` argument. Resizing just the rows is similar. You can use either of the following:

```
Range("B3").Resize(RowSize:=2)
```

or

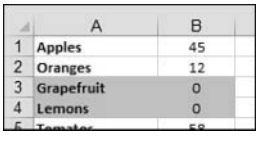
```
Range("B3").Resize(2)
```

Once again, the choice is yours. It is a matter of readability of the code.

From the list of produce, say that you want to find the zero totals and color the cells of the total and corresponding produce (see Figure 3.3). Here's what you do:

```
Set Rng = Range("B1:B16").Find(What:="0", LookAt:=xlWhole, _
    LookIn:=xlValues)
Rng.Offset(, -1).Resize(, 2).Interior.ColorIndex = 15
```

Figure 3.3
Resizing a range to extend the selection.



Notice that the `Offset` property first moves the active cell over to the produce column. When you are resizing, the upper-left-corner cell must remain the same.

Resizing isn't only for single cells; you can use it to resize an existing range. For example, if you have a named range but need it and the column next to it, use this:

```
Range("Produce").Resize(,2)
```

Remember, the number you resize by is the total number of rows/columns you want to include.

Using the `Columns` and `Rows` Properties to Specify a Range

The `Columns` and `Rows` properties refer to the columns and rows of a specified `Range` object, which can be a worksheet or a range of cells. They return a `Range` object referencing the rows or columns of the specified object.

You have seen the following line used, but what is it doing?

```
FinalRow = Cells(Rows.Count, 1).End(xlUp).Row
```

This line of code finds the last row in a sheet in which column A has a value and places the row number of that `Range` object into the variable called `FinalRow`. This can be useful when you need to loop through a sheet row by row; you will know exactly how many rows you need to go through.

> **NOTE** Some properties of columns and rows require contiguous rows and columns in order to work properly. For example, if you were to use the following line of code, `9` would be the answer because only the first range would be evaluated:
>
> ```
> Range("A1:B9, C10:D19").Rows.Count
> ```
>
> However, if the ranges were grouped separately, the answer would be `19`. Excel takes the top, left-most cell address, `A1`, and the bottom, rightmost cell address, `D19`, and counts the cells in the range `A1:D19`:
>
> ```
> Range("A1:B9", "C10:D19").Rows.Count
> ```

Using the `Union` Method to Join Multiple Ranges

The `Union` method enables you to join two or more noncontiguous ranges. It creates a temporary object of the multiple ranges, which enables you to affect them together:

```
Application.Union(argument1, argument2, etc.)
```

The expression `Application` is not required. The following code joins two named ranges on the sheet, inserts the `=RAND()` formula, and bolds them:

```
Set UnionRange = Union(Range("Range1"), Range("Range2"))
With UnionRange
    .Formula = "=RAND()"
    .Font.Bold = True
End With
```

Using the Intersect Method to Create a New Range from Overlapping Ranges

The Intersect method returns the cells that overlap between two or more ranges. If there is no overlap, an error will be returned:

```
Application.Intersect(argument1, argument2, etc.)
```

The expression Application is not required. The following code colors the overlapping cells of the two ranges:

```
Set IntersectRange = Intersect(Range("Range1"), Range("Range2"))
IntersectRange.Interior.ColorIndex = 6
```


Using the IsEmpty Function to Check Whether a Cell Is Empty

The IsEmpty function returns a Boolean value that indicates whether a single cell is empty: True if empty, False if not. The cell must truly be empty for the function to return True. If it contains even just a space that you cannot see, Excel does not consider the cell to be empty:

```
IsEmpty(Cell)
```

Say that you have several groups of data separated by a blank row. You want to make the separations a little more obvious. The following code goes down the data in column A. When it finds an empty cell in column A, it colors in the first four cells of that row (see Figure 3.4):

```
LastRow = Cells(Rows.Count, 1).End(xlUp).Row
For i = 1 To LastRow
    If IsEmpty(Cells(i, 1)) Then
        Cells(i, 1).Resize(1, 4).Interior.ColorIndex = 1
    End If
Next i
```

Figure 3.4
Colored rows separating data.

	A	B	C	D
1	Apples	Oranges	Grapefruit	Lemons
2	45	12	86	15
3	71%	53%	82%	52%
4				
5	Tomatoes	Cabbage	Lettuce	Green Peppers
6	58	24	31	0
7	30%	43%	68%	1%
8				
9	Potatoes	Yams	Onions	Garlic
10	10	61	26	29
11	18%	19%	22%	82%

Using the `CurrentRegion` Property to Select a Data Range

`CurrentRegion` returns a `Range` object that represents a set of contiguous data. As long as the data is surrounded by one empty row and one empty column, you can select the data set by using `CurrentRegion`:

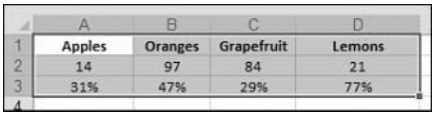
```
RangeObject.CurrentRegion
```

The following line selects A1:D3 because this is the contiguous range of cells around cell A1 (see Figure 3.5):

```
Range("A1").CurrentRegion.Select
```

This is useful if you have a data set whose size is in constant flux.

Figure 3.5
Use `CurrentRegion` to select a range of contiguous data around the active cell.

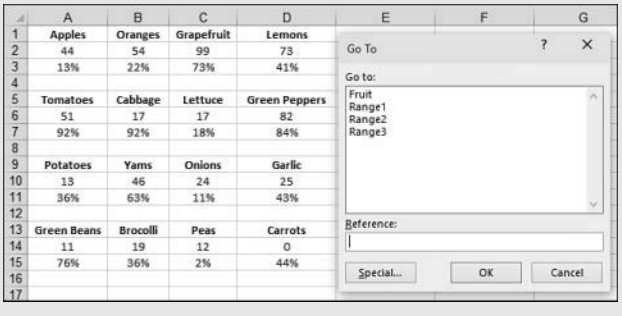


	A	B	C	D
1	Apples	Oranges	Grapefruit	Lemons
2	14	97	84	21
3	31%	47%	29%	77%
4				

CASE STUDY: USING THE SPECIALCELLS METHOD TO SELECT SPECIFIC CELLS

Even Excel power users might not have encountered the Go To Special dialog box. If you press the F5 key in an Excel worksheet, you get the normal Go To dialog box (see Figure 3.6). In the lower-left corner of this dialog is a button labeled Special. Click this button to get to the super-powerful Go To Special dialog box (see Figure 3.7).

Figure 3.6
Although the Go To dialog doesn't seem useful, click the Special button in the lower-left corner.



	A	B	C	D
1	Apples	Oranges	Grapefruit	Lemons
2	44	54	99	73
3	13%	22%	73%	41%
4				
5	Tomatoes	Cabbage	Lettuce	Green Peppers
6	51	17	17	82
7	92%	92%	18%	84%
8				
9	Potatoes	Yams	Onions	Garlic
10	13	46	24	25
11	36%	63%	11%	43%
12				
13	Green Beans	Brocolli	Peas	Carrots
14	11	19	12	0
15	76%	36%	2%	44%
16				
17				

In the Excel interface, the Go To Special dialog enables you to select only cells with formulas, only blank cells, or only the visible cells. Selecting only visible cells is excellent for grabbing the visible results of AutoFiltered data.

To simulate the Go To Special dialog in VBA, use the `SpecialCells` method. This enables you to act on cells that meet certain criteria, like this:

```
RangeObject.SpecialCells(Type, Value)
```

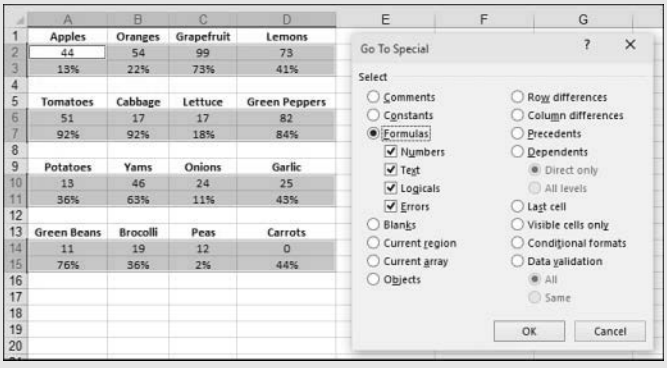


Figure 3.7
The Go To Special dialog has many incredibly useful selection tools, such as one for selecting only the formulas on a sheet.

This method has two parameters: `Type` and `Value`. `Type` is one of the `xlCellType` constants:

```
xlCellTypeAllFormatConditions
xlCellTypeAllValidation
xlCellTypeBlanks
xlCellTypeComments
xlCellTypeConstants
xlCellTypeFormulas
xlCellTypeLastCell
xlCellTypeSameFormatConditions
xlCellTypeSameValidation
xlCellTypeVisible
```

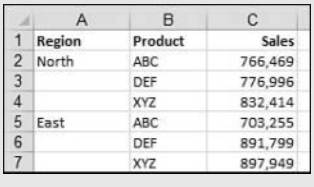
`Value` is optional and can be one of the following:

```
xlErrors
xlLogical
xlNumbers
xlTextValues
```

The following code returns all the ranges that have conditional formatting set up. It produces an error if there are no conditional formats and adds a border around each contiguous section it finds:

```
Set rngCond = ActiveSheet.Cells.SpecialCells(xlCellTypeAllFormatConditions)
If Not rngCond Is Nothing Then
    rngCond.BorderAround xlContinuous
End If
```

Have you ever had someone send you a worksheet without all the labels filled in? Some people think that the data shown in Figure 3.8 looks neat. They enter the Region field only once for each region. This might look aesthetically pleasing, but it is impossible to sort.



	A	B	C
1	Region	Product	Sales
2	North	ABC	766,469
3		DEF	776,996
4		XYZ	832,414
5	East	ABC	703,255
6		DEF	891,799
7		XYZ	897,949

Figure 3.8
The blank cells in the Region column make it difficult to sort data sets such as this.

Using the `SpecialCells` method to select all the blanks in this range is one way to fill in all the blank region cells quickly with the region found above them:

```
Sub FillIn()
On Error Resume Next 'Need this because if there aren't any blank
'cells, the code will error
Range("A1").CurrentRegion.SpecialCells(xlCellTypeBlanks).FormulaR1C1 _
     = "=R[-1]C"
Range("A1").CurrentRegion.Value = Range("A1").CurrentRegion.Value
End Sub
```

In this code, `Range("A1").CurrentRegion` refers to the contiguous range of data in the report. The `SpecialCells` method returns just the blank cells in that range. This particular formula fills in all the blank cells with a formula that points to the cell above the blank cell. (You can read more about R1C1-style formulas in Chapter 5, "R1C1-Style Formulas.") The second line of code is a fast way to simulate doing a Copy and then Paste Special Values. Figure 3.9 shows the results.

Figure 3.9
After the macro runs, the blank cells in the Region column have been filled in with data.

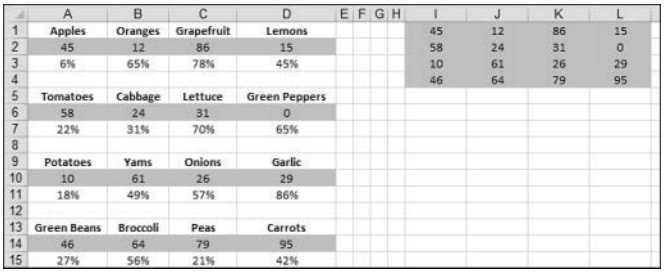
	A	B	C
1	Region	Product	Sales
2	North	ABC	766,469
3	North	DEF	776,996
4	North	XYZ	832,414
5	East	ABC	703,255
6	East	DEF	891,799
7	East	XYZ	897,949

Using the `Areas` Collection to Return a Noncontiguous Range

The `Areas` collection is a collection of noncontiguous ranges within a selection. It consists of individual `Range` objects representing contiguous ranges of cells within the selection. If a selection contains only one area, the `Areas` collection contains a single `Range` object that corresponds to that selection.

You might be tempted to loop through the rows in a sheet and check the properties of a cell in a row, such as its formatting (for example, font or fill) or whether the cell contains a formula or value. Then you could copy the row and paste it to another section. However, there is an easier way. In Figure 3.10, the user enters the values below each fruit and vegetable. The percentages are formulas. The following line of code selects the cells with numeric constants and copies them to another area:

```
Range("A:D").SpecialCells(xlCellTypeConstants, xlNumbers).Copy _
    Range("I1")
```



	A	B	C	D	E	F	G	H	I	J	K	L
1	Apples	Oranges	Grapefruit	Lemons					45	12	86	15
2	45	12	86	15					58	24	31	0
3	6%	65%	78%	45%					10	61	26	29
4									46	64	79	95
5	Tomatoes	Cabbage	Lettuce	Green Peppers								
6	58	24	31	0								
7	22%	31%	70%	65%								
8												
9	Potatoes	Yams	Onions	Garlic								
10	10	61	26	29								
11	18%	49%	57%	86%								
12												
13	Green Beans	Broccoli	Peas	Carrots								
14	46	64	79	95								
15	27%	56%	21%	42%								

Figure 3.10
The `Areas` collection makes it easier to manipulate noncontiguous ranges.

Referencing Tables

A table is a special type of range that offers the convenience of referencing named ranges. However, tables are not created in the same manner as other ranges. For more information on how to create a named table, see Chapter 6, "Creating and Manipulating Names in VBA."

Although you can reference a table by using `Worksheets(1).Range("Table1")`, you have access to more of the properties and methods that are unique to tables if you use the `ListObjects` object, like this:

```
Worksheets(1).ListObjects("Table1")
```

This opens the properties and methods of a table, but you can't use that line to select the table. To do that, you have to specify the part of the table you want to work with. To select the entire table, including the header and total rows, specify the `Range` property:

```
Worksheets(1).ListObjects("Table1").Range.Select
```

The table part properties include the following:

- `Range`—Returns the entire table.
- `DataBodyRange`—Returns the data part only.
- `HeaderRowRange`—Returns the header row only.
- `TotalRowRange`—Returns the total row only.

What I really like about coding with tables is the ease of referencing specific columns of a table. You don't have to know how many columns to move in from a starting position or the letter/number of the column, and you don't have to use a `FIND` function. Instead, you can use the header name of the column. For example, to select the data of the Qty column of the table, but not the header or total rows, do this:

```
Worksheets(1).ListObjects("Table1").ListColumns("Qty")_
    .DataBodyRange.Select
```

NOTE

For more details on coding with tables, check out *Excel Tables: A Complete Guide for Creating, Using, and Automating Lists and Tables* by Zack Barresse and Kevin Jones (ISBN: 978-1615470280).

Next Steps

Chapter 4 describes a fundamental component of any programming language: loops. If you have taken a programming class, you will be familiar with basic loop structures. VBA supports all the usual loops. That chapter also describes a special loop, `For Each...Next`, which is unique to object-oriented programming such as VBA.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y-Z