Referring to Ranges

3

IN THIS CHAPTER

A *range* can be a cell, row, column, or a grouping of any of these. The `RANGE` object is probably the most frequently used object in Excel VBA—after all, you're 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 have to 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 learn how to manipulate cells based on the active cell and how to create a new range from overlapping ranges.

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 either a sheet be active or 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 does it. But each of the following is equivalent:

```
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!

Using the Upper-Left and Lower-Right Corners of a Selection to Specify a Range

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

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

In the alternative syntax, you 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").Select
```

For either corner, you can substitute a named range, the `Cells` property, or the `ActiveCell` property. This 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've probably already used named ranges on your sheets and in formulas. You can also use them in VBA.

To refer to the range `"MyRange"` in Sheet1, do this:

```
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, unless you are using the shortcut syntax discussed in the previous section, in which case, quotes are not used.

Shortcut for Referencing Ranges

A shortcut is available when referencing ranges. It uses square brackets, as shown in Table 3.1.

Table 3.1 Shortcuts for Referring to 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 can drastically slow down your code. To avoid this slowdown, you can refer to a sheet that is not active by referencing the Worksheet object first:

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

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

If you need to reference a range in another workbook, include the Workbook object, the Worksheet object, and then the Range object:

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

Be careful if you use the `Range` property as an argument within another `Range` property. You must identify the range fully each time. Suppose, for example, 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? Because `Range(Range("A1"), Range("A7"))` refers to an extra range at the beginning of the code line. Excel does not assume that you want to carry the Worksheet object reference over to the other Range objects. 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 is not only a long line of code, it is difficult to read! Thankfully, there is a simpler way, `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.

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! This makes for code that is very unintuitive. Consider this example:

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

This actually selects cell D7. Think about cell C3. It 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, and this is D7.

Again, I consider this coding style to be very unintuitive. This line of code mentions two addresses, and the actual cell being 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, this line activates the cell three rows down and four columns to the right of the currently active cell:

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

This syntax is mentioned only because the macro recorder uses it. Remember that back in Chapter 1, "Unleash the Power of Excel with VBA!" when we were recording a macro with Relative References on, the following line was recorded:

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

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

Although a worksheet is usually the object of the `Range` property, on occasion, 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
```

The line refers to the original Range object. However, the `Cells` property has a property, `Item`, which 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 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:

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

The ability to use numeric values for parameters proves especially 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 simply 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?

```
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 out the range address, the new code uses the `Cells` and `Resize` properties to find the required cell, based on the active cell.

Using the `Cells` Property in the `Range` Property

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

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

This proves especially useful when you need to specify your variables with a parameter, as in the previous looping example.

Using the `Offset` Property to Refer to a Range

You've already seen a reference to `Offset`; the macro recorder used it when we were recording a relative reference. It enables you to manipulate a cell based off the location of the active cell. In this way, you don't have to know the address of a cell.

The syntax for the `Offset` property is this:

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

To affect cell F5 from cell A1, write this:

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

Or, shorter yet, write this:

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

The count starts at A1 but does not include A1.

But what 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 column parameter. If you need to refer to a cell one column over, use one of these:

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

Both lines mean the same. The choice is yours. Referring to a cell one row up is similar:

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

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

Let's suppose you have a list of produce with totals next to them, and you want to find any total equal to zero and place LOW in the cell next to it. You could do it this way:

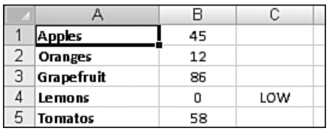
```
Set Rng = Range("B1:B16").Find(What:="0", LookAt:=xlWhole, LookIn:=xlValues)
Rng.Offset(, 1).Value = "LOW"

Sub MyOffset()
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 quickly noted by the program, as shown in Figure 3.1.

Offsetting isn't only for single cells—it can be used 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):

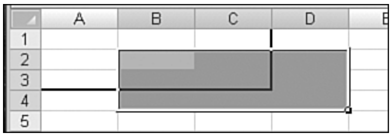
```
Range("A1:C3").Offset(1,1)
```

Figure 3.1
Find the produce with the 0 total.



	A	B	C
1	Apples	45	
2	Oranges	12	
3	Grapefruit	86	
4	Lemons	0	LOW
5	Tomatos	58	

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 you need it.

The syntax for the Resize property is this:

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

To create a range B3:D13, use this:

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

Or, simpler, use this:

```
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 column parameters. If you need to expand by two columns, use one of these:

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

or

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

Both lines mean the same. The choice is yours. Resizing just the rows is similar:

```
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, find the zero total and color the cells of the total and corresponding produce (see Figure 3.3):

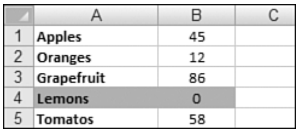
```
Set Rng = Range("B1:B16").Find(What:="0", LookAt:=xlWhole, LookIn:=xlValues)
Rng.Offset(, -1).Resize(, 2).Interior.ColorIndex = 15
```

Notice that the Offset property was used first to move the active cell over; when you are resizing, the upper-left corner cell must remain the same.

Resizing isn't only for single cells—it can be used to resize an existing range. For example, if you have a named range but need it and the two columns 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.



	A	B	C
1	Apples	45	
2	Oranges	12	
3	Grapefruit	86	
4	Lemons	0	
5	Tomatos	58	

Figure 3.3
Resizing a range to extend the selection.

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

`Columns` and `Rows` 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've 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 `FinalRow`. This can be very useful when you need to loop through a sheet row by row—you'll know exactly how many rows you need to go through.

CAUTION

Some properties of columns and rows require contiguous rows and columns 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
```

But if the ranges are grouped separately,

```
Range("A1:B9", "C10:D19").Rows.Count
```

the answer would be 19.

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, allowing you to affect them together:

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

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:

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

The following code colors the overlapping cells of the two ranges.

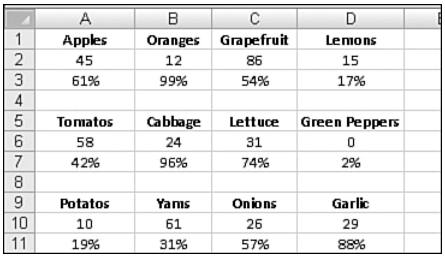
```
Set IntersectRange = Intersect(Range("Range1"), Range("Range2"))
IntersectRange.Interior.ColorIndex = 6
```

Using the `ISEMPTY` Function to Check Whether a Cell Is Empty

The `ISEMPTY` function returns a Boolean value of whether a single cell is empty or not; `True` if empty, `False` if not. The cell must truly be empty. Even if it has a space in it, which you cannot see, Excel does not consider it empty:

```
IsEmpty(Cell)
```

Look at Figure 3.4. You have several groups of data separated by a blank row. You want to make the separations a little more obvious.

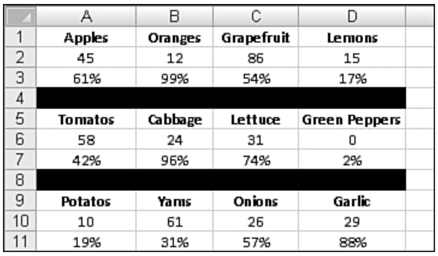


	A	B	C	D
1	Apples	Oranges	Grapefruit	Lemons
2	45	12	86	15
3	61%	99%	54%	17%
4				
5	Tomatos	Cabbage	Lettuce	Green Peppers
6	58	24	31	0
7	42%	96%	74%	2%
8				
9	Potatos	Yams	Onions	Garlic
10	10	61	26	29
11	19%	31%	57%	88%

Figure 3.4
Blank empty rows separating data.

The following code goes down the data in Column A; where it finds an empty cell, it colors in the first four cells for that row (see Figure 3.5):

```
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
```



	A	B	C	D
1	Apples	Oranges	Grapefruit	Lemons
2	45	12	86	15
3	61%	99%	54%	17%
4				
5	Tomatos	Cabbage	Lettuce	Green Peppers
6	58	24	31	0
7	42%	96%	74%	2%
8				
9	Potatos	Yams	Onions	Garlic
10	10	61	26	29
11	19%	31%	57%	88%

Figure 3.5
Colored rows separating data.

Using the `CurrentRegion` Property to Quickly Select a Data Range

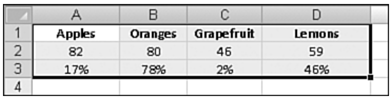
`CurrentRegion` returns a Range object representing a set of contiguous data. As long as the data is surrounded by one empty row and one empty column, you can select the table with `CurrentRegion`:

```
RangeObject.CurrentRegion
```

Look at Figure 3.6. The following line selects A1:D3 because this is the contiguous range of cells around cell A1:

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

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



	A	B	C	D
1	Apples	Oranges	Grapefruit	Lemons
2	82	80	46	59
3	17%	78%	2%	46%
4				

Figure 3.6
Use `CurrentRegion` to quickly select a range of contiguous data around the active cell.

CASE STUDY

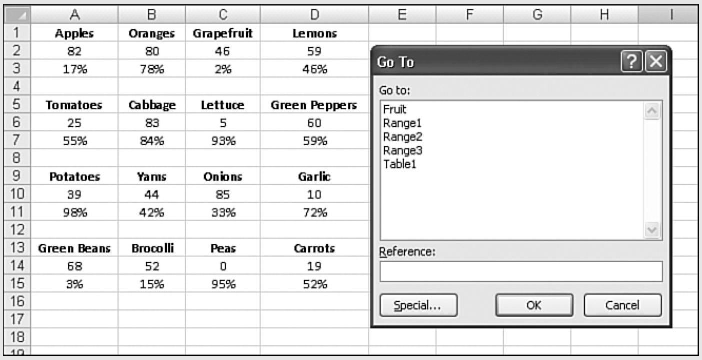
Using the `SpecialCells` Method to Select Specific Cells

Even Excel power users may never 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 3.7). In the lower-left corner of this dialog is a button labeled Special. Click that button to get to the super-powerful Go To Special dialog (see Figure 3.8).

In the Excel interface, the Go To Special dialog enables you to select only cells with formulas, or only blank cells, or only the visible cells. Selecting visible cells only 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 a certain criteria:

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



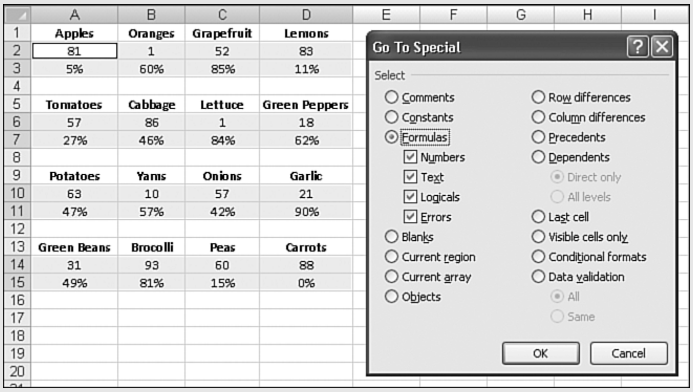
	A	B	C	D
1	Apples	Oranges	Grapefruit	Lemons
2	82	80	46	59
3	17%	78%	2%	46%
4				
5	Tomatoes	Cabbage	Lettuce	Green Peppers
6	25	83	5	60
7	55%	84%	93%	59%
8				
9	Potatoes	Yams	Onions	Garlic
10	39	44	85	10
11	98%	42%	33%	72%
12				
13	Green Beans	Brocolli	Peas	Carrots
14	68	52	0	19
15	3%	15%	95%	52%

Go To

Go to:
Fruit
Range1
Range2
Range3
Table1

Reference:

Figure 3.7
Although the Go To dialog doesn't seem very useful, click the Special button in the lower-left corner.



	A	B	C	D
1	Apples	Oranges	Grapefruit	Lemons
2	81	1	52	83
3	5%	60%	85%	11%
4				
5	Tomatoes	Cabbage	Lettuce	Green Peppers
6	57	86	1	18
7	27%	46%	84%	62%
8				
9	Potatoes	Yams	Onions	Garlic
10	63	10	57	21
11	47%	57%	42%	90%
12				
13	Green Beans	Brocolli	Peas	Carrots
14	31	93	60	88
15	49%	81%	15%	0%

Go To Special

Select
- Comments
- Constants
- [✓ Formulas]
  - [✓ Numbers]
  - [✓ Text]
  - [✓ Logicals]
  - [✓ Errors]
- Blanks
- Current region
- Current array
- Objects
- Row differences
- Column differences
- Precedents
- Dependents
  - [✓ Direct only]
  - All levels
- Last cell
- Visible cells only
- Conditional formats
- Data validation
  - [✓ All]
  - Same

Figure 3.8
The Go To Special dialog has many incredibly useful selection tools.

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

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

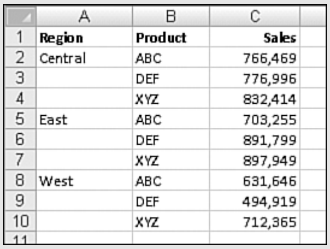
`Value` is optional and can be one of the following:

`xlErrors`	`xlNumbers`
`xlLogical`	`xlTextValues`

The following code returns all the ranges that have conditional formatting set up. It produces an error if there are no conditional formats. It puts 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 consider that the data shown in Figure 3.9 looks neat. They enter the Region field only once for each region. This might look aesthetically pleasing, but it is impossible to sort. Even Excel's pivot table routinely returns data in this annoying format.



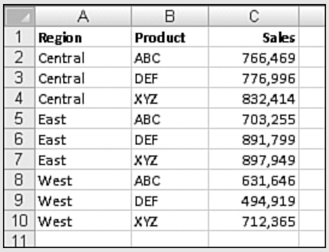
	A	B	C
1	**Region**	**Product**	**Sales**
2	Central	ABC	766,469
3		DEF	776,996
4		XYZ	832,414
5	East	ABC	703,255
6		DEF	891,799
7		XYZ	897,949
8	West	ABC	631,646
9		DEF	494,919
10		XYZ	712,365
11			

Figure 3.9
The blank cells in the region column make data tables such as this very difficult to sort.

Using the `SpecialCells` method to select all the blanks in this range is one way to quickly fill in all the blank region cells 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. Although you can read more about R1C1 style formulas in Chapter 6, "R1C1-Style Formulas," this particular formula fills in all the blank cells with a formula that points to the cell above the blank cell. The second line of code is a fast way to simulate doing a Copy and then Paste Special Values. Figure 3.10 shows the results.



	A	B	C
1	**Region**	**Product**	**Sales**
2	Central	ABC	766,469
3	Central	DEF	776,996
4	Central	XYZ	832,414
5	East	ABC	703,255
6	East	DEF	891,799
7	East	XYZ	897,949
8	West	ABC	631,646
9	West	DEF	494,919
10	West	XYZ	712,365
11			

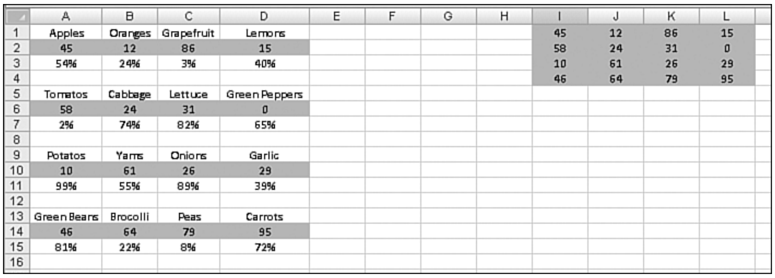
Figure 3.10
After the macro runs, the blank cells in the Region column have been filled in with data.

Using the Areas Collection to Return a Noncontiguous Range

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

You might be tempted to loop through the sheet, copy a row, and paste it to another section. But there's an easier way (see Figure 3.11):

```
Range("A:D").SpecialCells(xlCellTypeConstants, 1).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	54%	24%	3%	40%					10	61	26	29
4									46	64	79	95
5	Tomatos	Cabbage	Lettuce	Green Peppers								
6	58	24	31	0								
7	2%	74%	82%	65%								
8												
9	Potatos	Yams	Onions	Garlic								
10	10	61	26	29								
11	99%	55%	89%	39%								
12												
13	Green Beans	Brocolli	Peas	Carrots								
14	46	64	79	95								
15	81%	22%	8%	72%								
16												

Figure 3.11
The Areas collection makes it easy to manipulate noncontiguous ranges.

Referencing Tables

NEW With Excel 2007, we're introduced to a new way of interacting with ranges of data: tables. These special ranges offer the convenience of referencing named ranges, but are not created in the same manner. For more information on how to create a named table, see Chapter 8, "Create and Manipulate Names in VBA."

The table itself is referenced using the standard method of referring to a ranged name. To refer to the data in table Table1 in Sheet1, do this:

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

This references just the data part of the table; it does not include the header or total row. To include the header and total row, do this:

```
Worksheets(1).Range("Table1[#All]")
```

What I really like about this new feature is the ease of referencing specific columns of a table. You don't have to know how many columns in from a starting position or the letter/number of the column, and you don't have to use a `FIND` function. You can just use the header name of the column. To reference the Qty column of the table, for example, do this:

```
Worksheets(1).Range("Table1[Qty]")
```

Next Steps

Now that you’re getting an idea of how Excel works, it’s time to apply it to useful situations. The next chapter looks at user-defined functions, uses the skills you’ve learned so far, and introduces other programming methods that you will learn more about throughout this book.