O'REILLY®

Modern Data Analytics in Excel

Using Power Query, Power Pivot and More for Enhanced Data Analytics

Early Release

RAW & UNEDITED

George Mount

Modern Data Analytics in Excel

Transform, Model, and Analyze Data in Spreadsheets

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

George Mount

Beijing • Boston • Farnham • Sebastopol • Tokyo

Modern Data Analytics in Excel

by George Mount

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Michelle Smith and Sara Hunter

Production Editor: Ashley Stussy

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

September 2024: First Edition

Revision History for the Early Release

- 2023-07-27: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098148829 for release details.

978-1-098-14876-8

[FILL IN]

Chapter 1. Tables: The portal to modern analytics

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at shunter@oreilly.com.

Excel offers a wide range of impressive tools for analytics, making it difficult to decide where to begin. However, one crucial starting point is understanding the Excel table. This chapter explores the foundational aspects of Excel tables, which serve as a gateway to Power Query, Power Pivot, and other tools discussed in this book. It also explores the importance of properly arranging data within a table. To follow along with this chapter, head to `ch_01.xlsx` in the `ch_01` folder of the book's companion repository.

Creating and referring to table headers

A dataset without column headers is practically useless as it lacks meaningful context for interpreting what each variable measures. Unfortunately, it's not uncommon to encounter datasets which break this cardinal rule. Excel tables act as a valuable reminder that the quality of a dataset hinges on the presence of clear and informative headers.

In the `start` worksheet of `ch_01.xlsx`, you will come across data in columns `A:F` without corresponding headers, which are currently located in columns `H:L`. This design is far less than optimal. To adjust it, click anywhere within the primary data source and proceed to Insert > Tables > OK, as illustrated in Figure 1-1. Alternatively, you can press `Ctrl + T` from within the data source to launch the same Create Table menu.

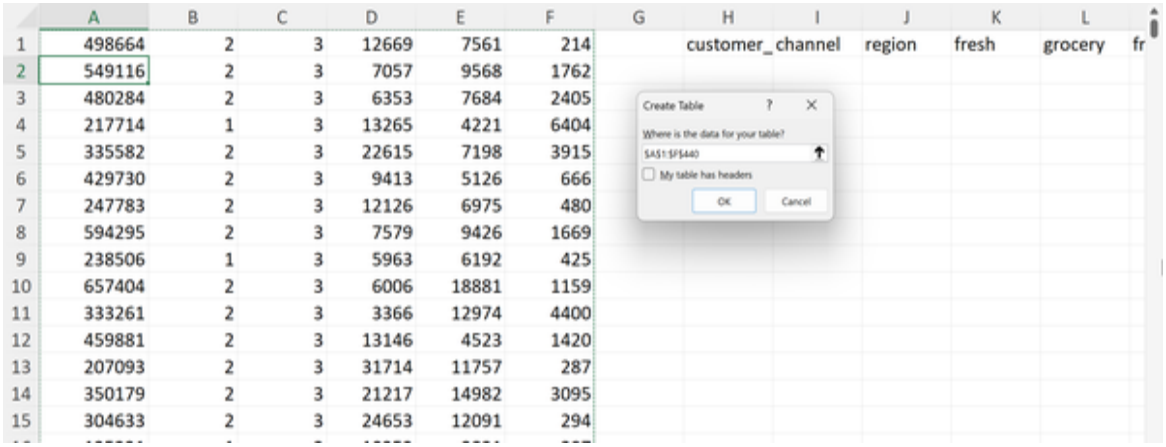


Figure 1-1. Converting the range (?) to a table

Automatically, the Create Table menu prompts you to indicate whether your data has headers. If headers are absent, a series of header columns named `Column1`, `Column2`, and so on is automatically assigned to the dataset.

From here, you can copy and paste the headers from columns `H:M` into the main table to clarify what is being measured in each column, such as in Figure 1-2:

	A	B	C	D	E	F
1	customer_id	channel	region	fresh	grocery	frozen
2	498664	2	3	12669	7561	214
3	549116	2	3	7057	9568	1762
4	480284	2	3	6353	7684	2405
5	217714	1	3	13265	4221	6404
6	335582	2	3	22615	7198	3915
7	429730	2	3	9413	5126	666
8	247783	2	3	12126	6975	480
9	594295	2	3	7579	9426	1669
10	238506	1	3	5963	6192	425
11	657404	2	3	6006	18881	1159
12	333261	2	3	3366	12974	4400

Figure 1-2. Excel table with headers

Header columns in Excel tables occupy a unique role in the dataset. While they are technically part of the table, they function as metadata rather than data itself. Excel tables provide the ability to programmatically distinguish between headers and data, unlike classic Excel formulas.

To see this difference in action, head to a blank cell in your worksheet and enter the equals sign. Point to cells `A1:F1` as your reference, and you'll notice that the formula becomes `Table1[#Headers]`.

Once you've established this reference to `Table1`'s headers, you can utilize it as a downstream reference for other functions. For example, you can use `UPPER()` to dynamically convert the case of all the headers, such as in Figure 1-3:

	A	B	C	D	E	F	G
1							Formula used:
2	customer_id	channel	region	fresh	grocery	frozen	=Table1[#Headers]
3	CUSTOMER_ID	CHANNEL	REGION	FRESH	GROCERY	FROZEN	=UPPER(Table1[#Headers])
4							
5	customer_id	channel	region	fresh	grocery	frozen	
6	498664	2	3	12669	7561	214	
7	549116	2	3	7057	9568	1762	
8	480284	2	3	6353	7684	2405	
9	217714	1	3	13265	4221	6404	
10	335582	2	3	22615	7198	3915	
11	429730	2	3	9413	5126	666	
12	247783	2	3	12126	6975	480	

Figure 1-3. Excel header reference formulas

Viewing the table footers

Just as every story has a beginning, middle, and end, every Excel table comprises headers, data, and footers. However, footers need to be manually enabled. To do this, click anywhere in the table, navigate to Table Design on the ribbon, and select Total Row in the Table Style Options group, such as in Figure 1-4:

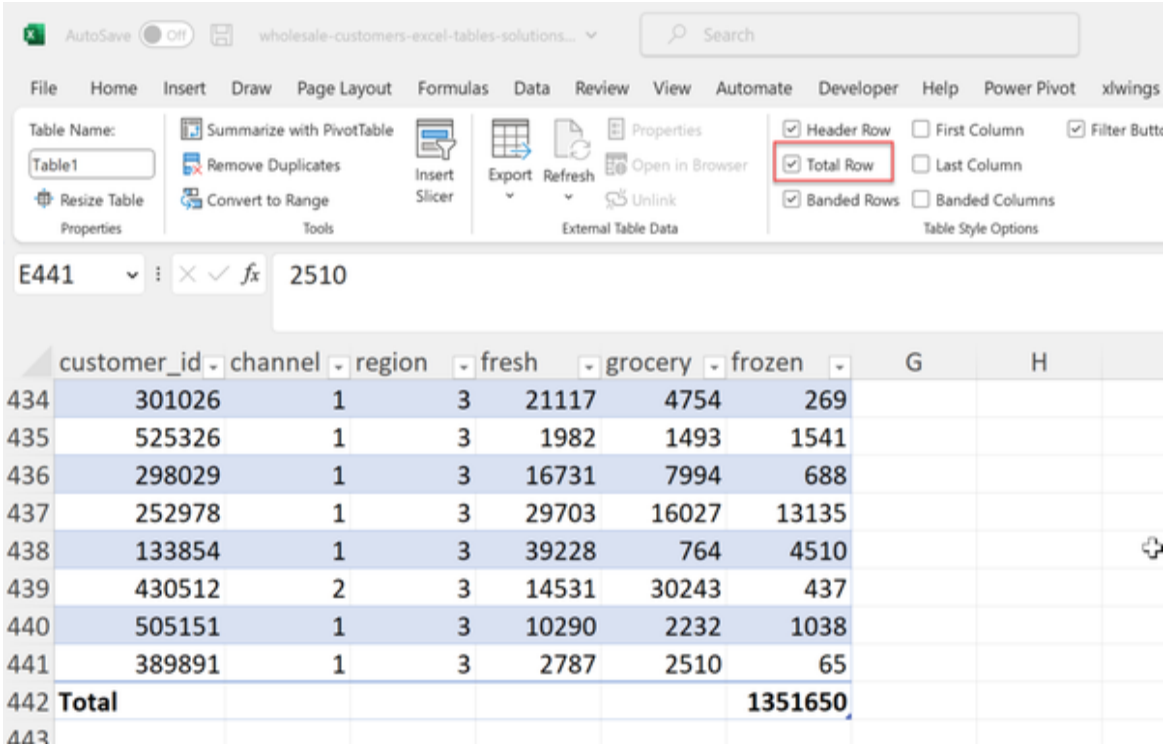


Figure 1-4. Adding footers to a table

By default, the total row in a table will calculate the sum of the last column in your data; in this case, `frozen`. However, you can customize this by clicking the dropdown menu on any column. For instance, you can find the maximum sales amount of the `fresh` category by selecting it from the dropdown menu, such as in Figure 1-5:

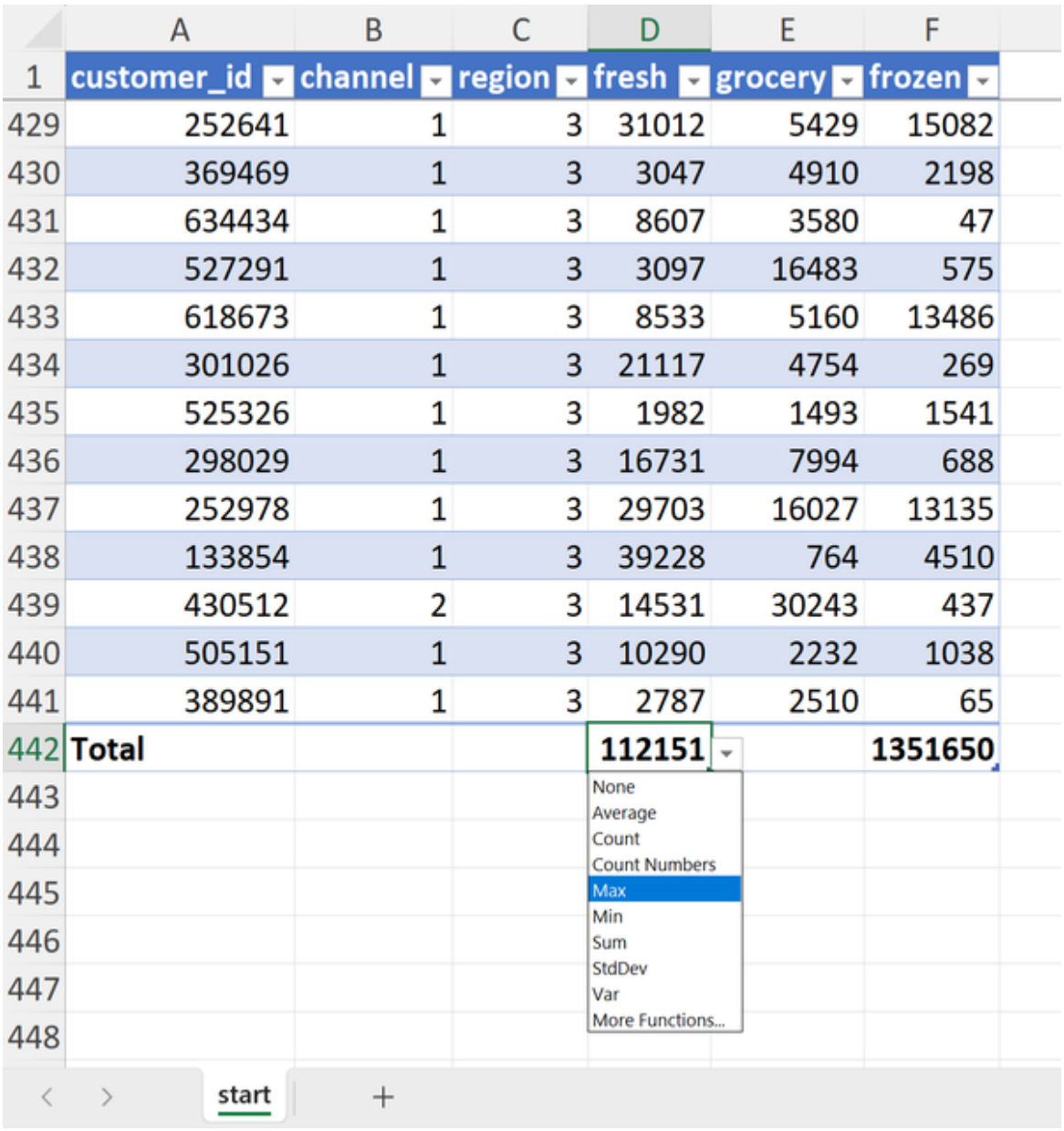


Figure 1-5. Customizing the footers of an Excel table

Table 1-1 summarizes the key formula references for the major components of Excel tables:

Table 1-1. Summary of Excel table formula references

Formula	What it refers to
`=Table1[#Headers]`	Table headers
`=Table1`	Table data
`=Table1[#Totals]`	Table footers
`=Table1[#All]`	Table headers, data and footers

As you progress in your Excel table skills, you'll discover additional helpful formula references that rely on the fundamental structure of headers, body, and footers.

Naming Excel tables

Excel tables offer the advantage of enforcing the use of named ranges, which promotes a more structured and programmatic approach to working with data. Although referring to `Table1` is an improvement over using cell coordinates like `A1:F22`, it is even more advantageous to choose a descriptive name that reflects what the underlying data measures.

To accomplish this, go to the Formulas tab on the home ribbon, select Name Manager, and choose Edit. Change the name to `sales`, then click OK. Figure 1-6 shows what your Name Manager should look like after making this change:

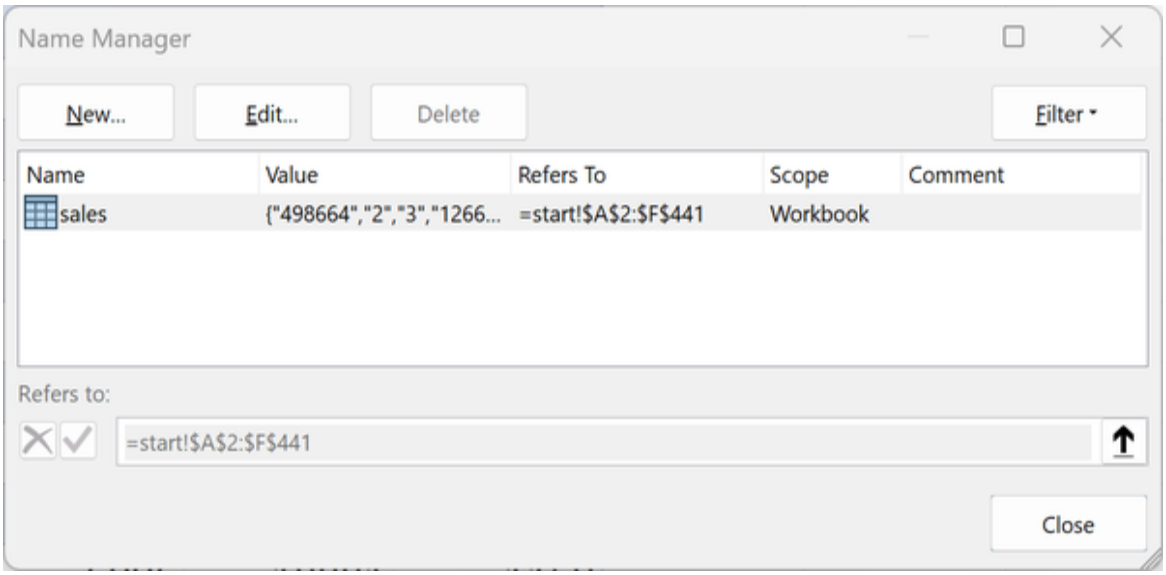


Figure 1-6. Name manager in Excel

Once you exit the Name Manager, you'll notice that all references to `Table1` have been automatically updated to reflect the new name, `sales`.

Formatting Excel tables

As an Excel user, you know the importance of presenting data in an appealing format. Tables can be a game-changer, instantly enhancing the visual appeal of your worksheet. With tables, you can easily add banded rows, colored headers, and more. To customize the look and feel of your table, simply go to Table Design in the ribbon. Take a look at Figure 1-7 for various options, such as changing table colors or toggling banded rows on and off:

Figure 1-7. Table design customization options

Updating table ranges

With Excel tables, the problem of totals breaking when data is added or deleted is a thing of the past. By utilizing structured references, the formula remains intact even when data is modified. Additionally, the total at the bottom of the table remains consistent, and excluding it from external references is straightforward.

Figure 1-8 demonstrates how, even after adding new rows to the data, the sum formula automatically updates without any disruptions. Using precise table references ensures that only the table's data is included in the sum, excluding the header or footer rows.

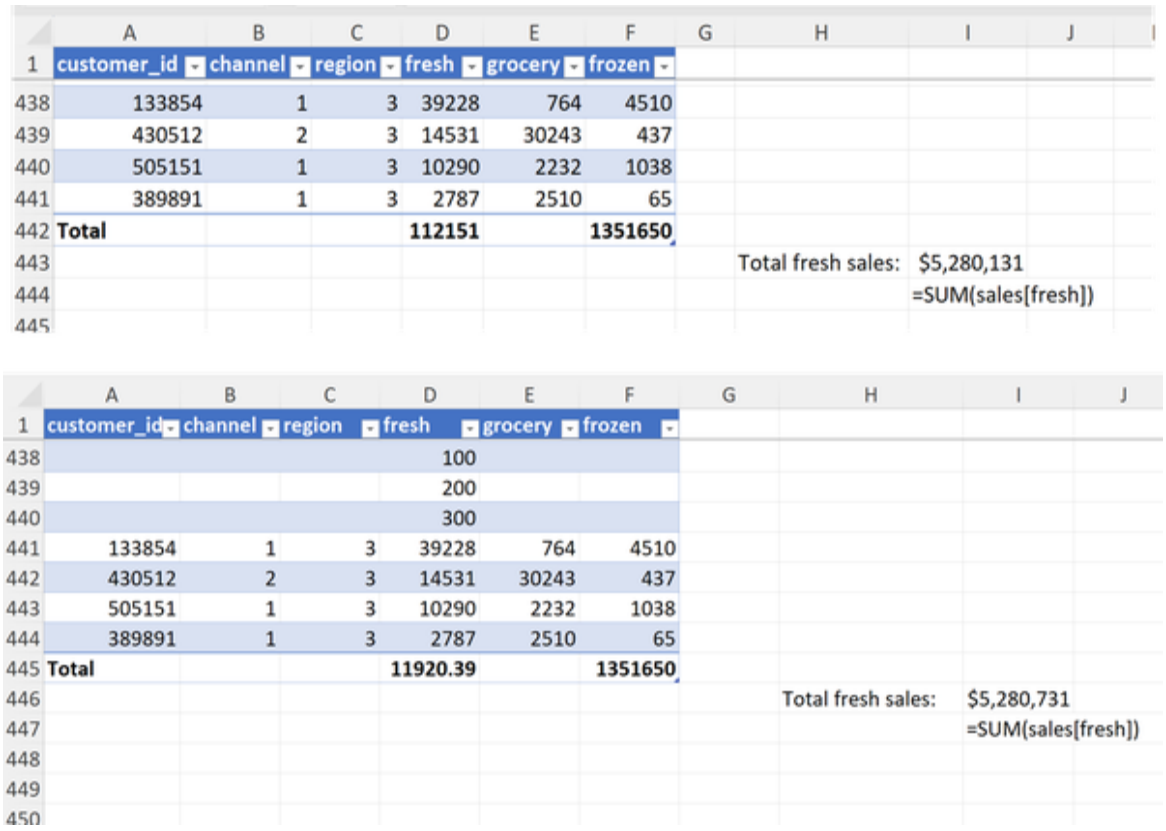


Figure 1-8. Dynamic range updates in Excel tables

Referring to data by object name instead of cell location minimizes potential formula issues arising from changing the table's size and placement. Tables become crucial in preventing problems like missing data in a PivotTable when new rows are added.

Organizing data for analytics

While tables are valuable, an even more significant aspect of ensuring effortless and accurate data analysis lies in storing data in the appropriate shape.

Take a closer look at the `sales` table as an example. When trying to create a PivotTable to calculate total sales by region, the data's storage format poses a challenge. Ideally, the sales information should be in a single column. However, in the current scenario, there is a separate sales column for each department, as shown in Figure 1-9:

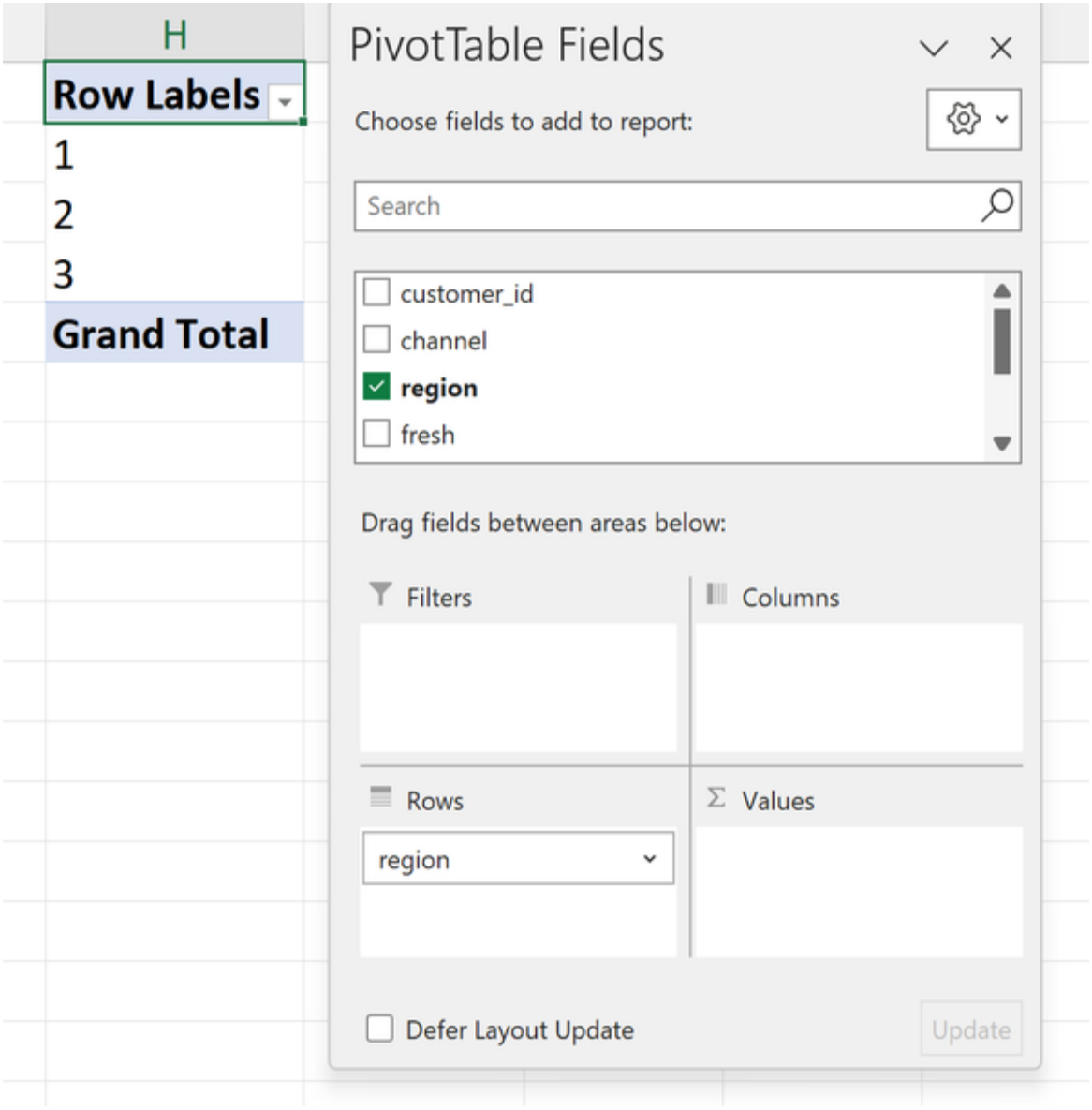


Figure 1-9. Attempting to summarize total sales goes awry

The reason this and many other datasets get difficult to analyze is that they are not stored in a format conducive to analysis. The rules of "tidy data" offer a solution. While Hadley Wickham offers three rules in his 2014 paper by the same name, this book focuses on the first:

Each variable forms a column.

The `sales` dataset violates the rule of tidy data by having multiple entries for the same variable, `field`, across different departments within each row. A helpful rule of thumb is that if multiple columns are measuring the same thing, the data is likely not tidy. By transforming the data into a tidy format, analysis becomes significantly simpler.

In Figure 1-10, you can see a comparison of the dataset before and after the transformation, highlighting the improved tidiness and ease of analysis. Later in this book, you will learn how to perform this fundamental transformation on the dataset with just a few clicks. In the meantime, you can explore the `sales-tidy` worksheet available in `ch01_solutions.xlsx`, which has already been transformed. Take a look to see firsthand how much simpler it is to obtain total sales by region now.

	A	B	C	D	E	F	G	H	I	J	K	L
1	*Before:*							*After:*				
3	customer_id	channel	region	fresh	grocery	frozen		customer_id	channel	region	department	sales
4	498664	2	3	12669	7561	214		498664	2	3	fresh	12669
5	549116	2	3	7057	9568	1762		498664	2	3	grocery	7561
6	480284	2	3	6353	7684	2405		498664	2	3	frozen	214
7	217714	1	3	13265	4221	6404		549116	2	3	fresh	7057
8	335582	2	3	22615	7198	3915		549116	2	3	grocery	9568
9	429730	2	3	9413	5126	666		549116	2	3	frozen	1762
10	247783	2	3	12126	6975	480		480284	2	3	fresh	6353
11	594295	2	3	7579	9426	1669		480284	2	3	grocery	7684
12	238506	1	3	5963	6192	425		480284	2	3	frozen	2405
13	657404	2	3	6006	18881	1159		217714	1	3	fresh	13265
14	333261	2	3	3366	12974	4400		217714	1	3	grocery	4221
15	459881	2	3	13146	4523	1420		217714	1	3	frozen	6404
16	207093	2	3	31714	11757	287		335582	2	3	fresh	22615
17	350179	2	3	21217	14982	3095		335582	2	3	grocery	7198
18	304633	2	3	24653	12091	294		335582	2	3	frozen	3915
19	125231	1	3	10253	3821	397		429730	2	3	fresh	9413
20	126155	2	3	1020	12121	134		429730	2	3	grocery	5126

Figure 1-10. Wholesale customers, before and after tidying

Conclusion

This chapter provided the basics of organizing data neatly and using Excel tables. These principles are essential for any successful data analysis project in Excel. The next chapter, Chapter 2, goes into more detail about using Power Query to make data transformation and preparation easier.

Exercises

To create, analyze, and manipulate data in Excel tables, follow these exercises using the `penguins.xlsx` dataset from the companion repository's `datasets` folder:

1. Convert the data to a table named `penguins`.
2. Utilize a formula reference to capitalize each column header.
3. Generate a new column called `bill_ratio` by dividing `bill_length_mm` by `bill_depth_mm`.
4. Include a total row to calculate the average `body_mass_g`.
5. Remove the banded row styling from the table.

For the solutions, refer to the `ch_01_exercise_solutions.xlsx` file located in the `exercise_solutions` folder of the book's repository.

Chapter 2. Transforming Rows in Power Query

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at shunter@oreilly.com.

Chapter 2 served as an introduction to Power Query's myth-busting capabilities as an ETL tool for Excel. In this and upcoming chapters of Part 1, you'll have the chance to get hands-on practice with common data transformation tasks. The focus of this chapter is on rows.

Data cleaning often requires manipulating rows, including tasks like sorting, filtering, and removing duplicates. While "Classic" Excel does provide menu-driven options for these operations, they can be cumbersome and difficult to reproduce. To solve this challenge, you can use Power Query to establish an auditable and reproducible data cleaning workflow without the need for coding. To follow along with the demonstrations in this chapter, head to `ch_03.xlsx` in the `ch_03` folder of this book's repository.

In the `signups` worksheet of this workbook, your organization's party planning committee has been gathering RSVPs and wants the final list to be sorted alphabetically, with duplicates, blanks, and misprints eliminated. The committee is weary of manually sorting and removing unnecessary rows

whenever new data is added. They desire a workbook that can be easily refreshed and reused as more individuals register or as new parties are scheduled.

Load this data into Power Query, naming the query `signups`. Capture all relevant rows in column `A` and confirm that your table includes headers before proceeding.

Removing the missing values

Unlike basic Excel, Power Query provides a dedicated `null` value to represent missing values. The `signups` data contains three blank values, which may cause confusion. To eliminate them, navigate to the Home tab on the ribbon and select Remove Rows > Remove Blank Rows, such as in Figure 2-1:

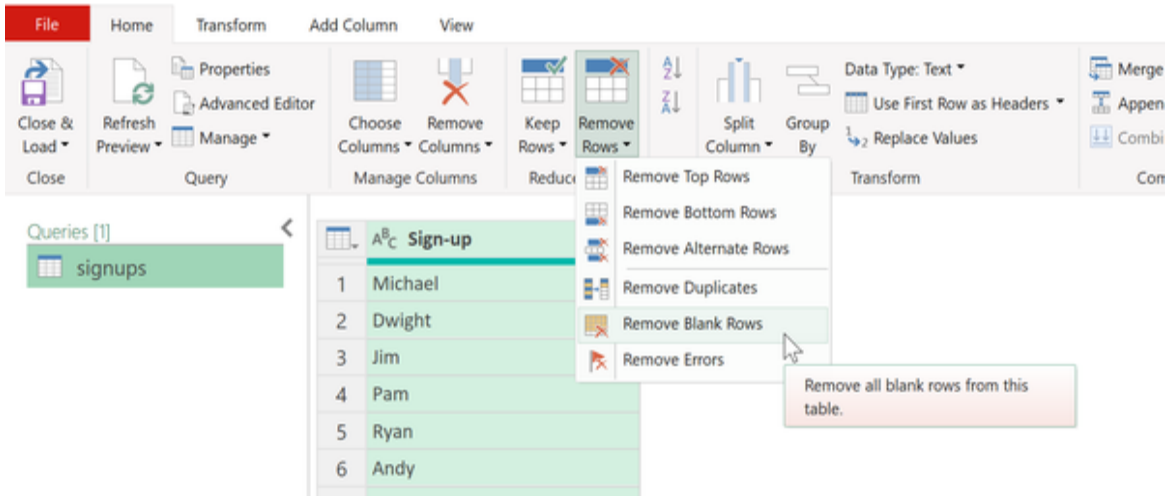


Figure 2-1. Removing blank rows in Power Query

Next, sort the list alphabetically for better legibility and to identify other potential data issues. To do this, click the drop-down button next to the `Sign-up` column, where you'll find the sort and filter options similar to basic Excel, such as in Figure 2-2:

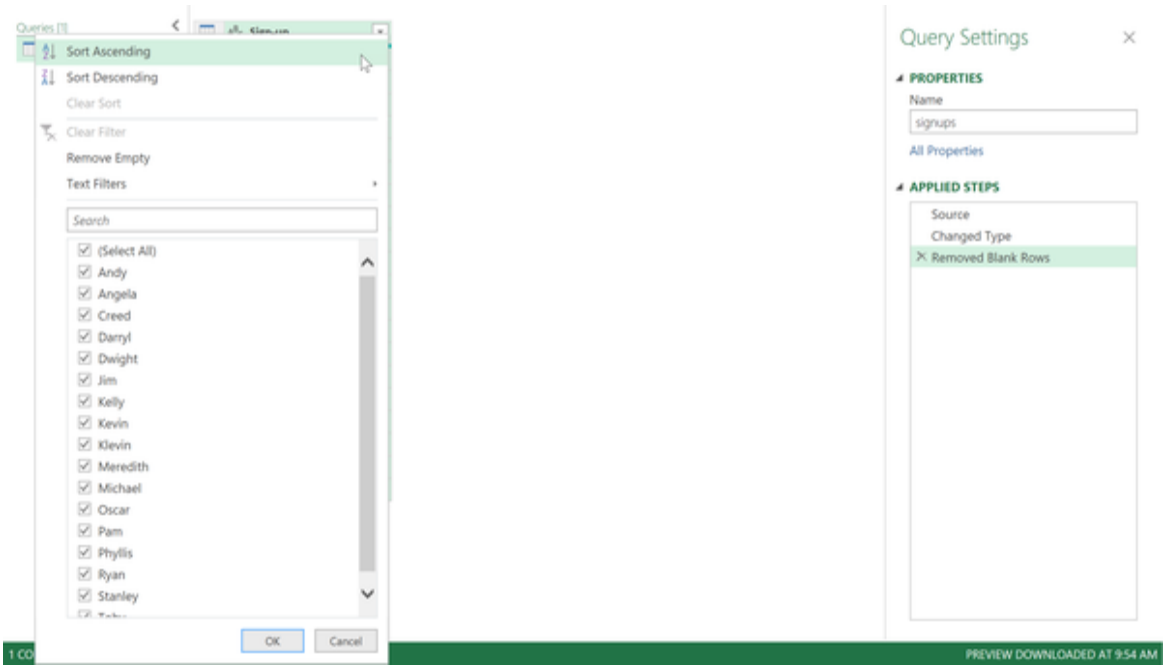


Figure 2-2. Sorting rows in Power Query

You may have noticed that *Phyllis* is entered multiple times in this list. To avoid any duplicates and potential confusion over headcount, remove duplicates by heading back to Home on the ribbon, then Remove Rows > Remove Duplicates.

The list is mostly clean, except for one misprint: “Klevin” in row 8. This situation highlights the importance of having domain knowledge about your data. While Power Query can help with standard data cleaning tasks, there are situations where understanding the subject matter becomes crucial. The final step involves filtering out the typo, as demonstrated in Figure 2-3:

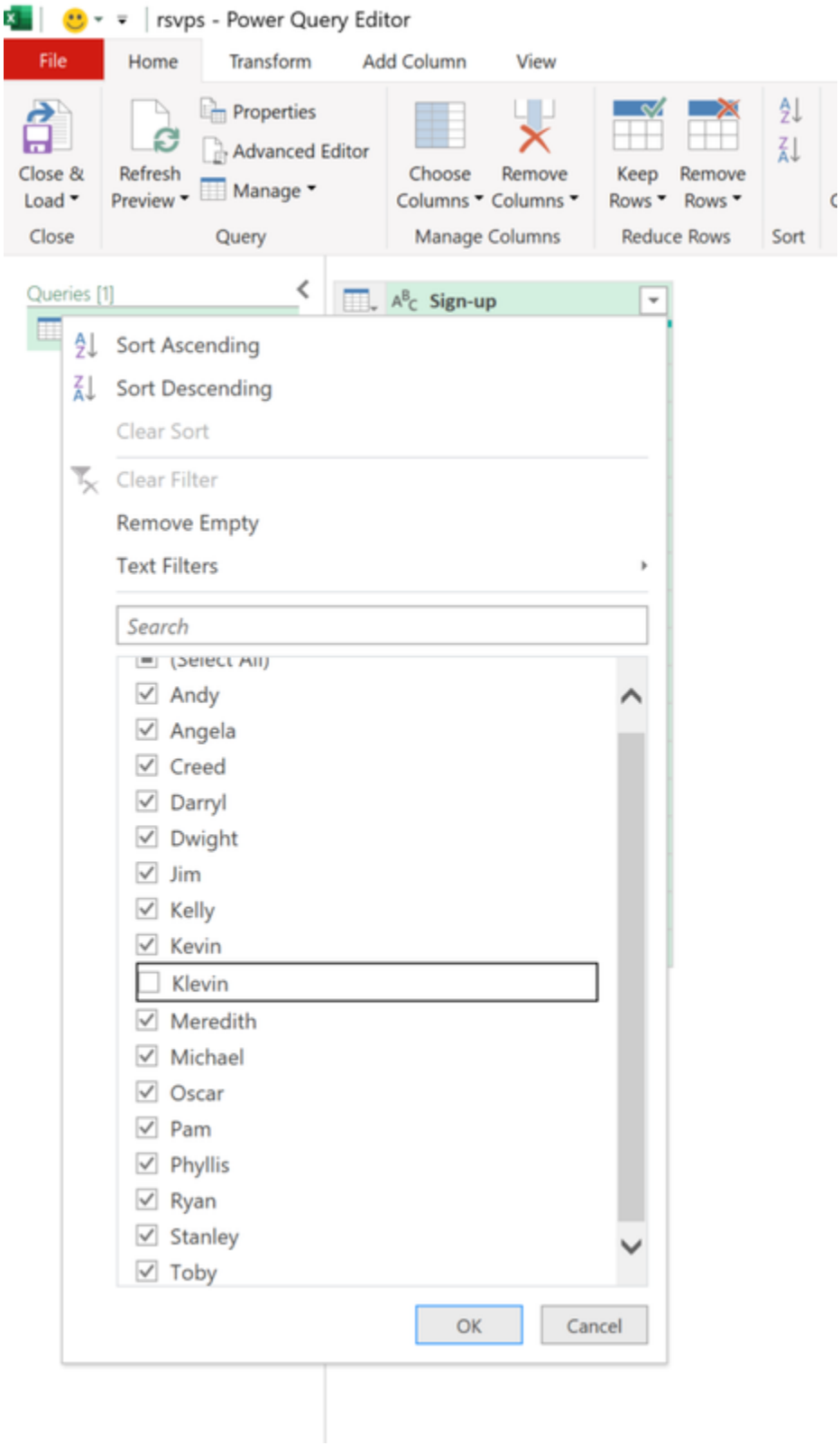


Figure 2-3. Filtering a misprint in Power Query

Data transformation requires collaboration and your guidance. Power Query can assist in making the data clean, but it is essential for you to define what "clean" means in a given situation.

Refreshing the query

Great job following these steps to clean your data. To make the results more accessible, load the cleaned dataset into Excel. From the Home tab, click on "Close & Load."

ORGANIZING EXTRACTS AND LOADS IN THE WORKBOOK

Storing input Excel data in the same workbook as the transformed version has a drawback: it becomes challenging to differentiate between the two. Although Excel attempts to assist by assigning a default blue color to the original table and green to the transformed one, confusion can still arise due to their similar worksheet names. To avoid this confusion, it is advisable to append the Power Query output worksheet with a suffix like `-out` or adhere to another appropriate convention.

Power Query offers more than just an escape from cumbersome Excel menus; its true advantage lies in its ability to refresh your work with a single click. To see this in action, add two rows to your original `signups` table: one blank row and a signup from Nate.

To avoid the manual process of sorting and deleting rows to update the cleaned dataset, you can navigate to the Power Query output table, right-click it, and select Refresh, such as in Figure 2-4:

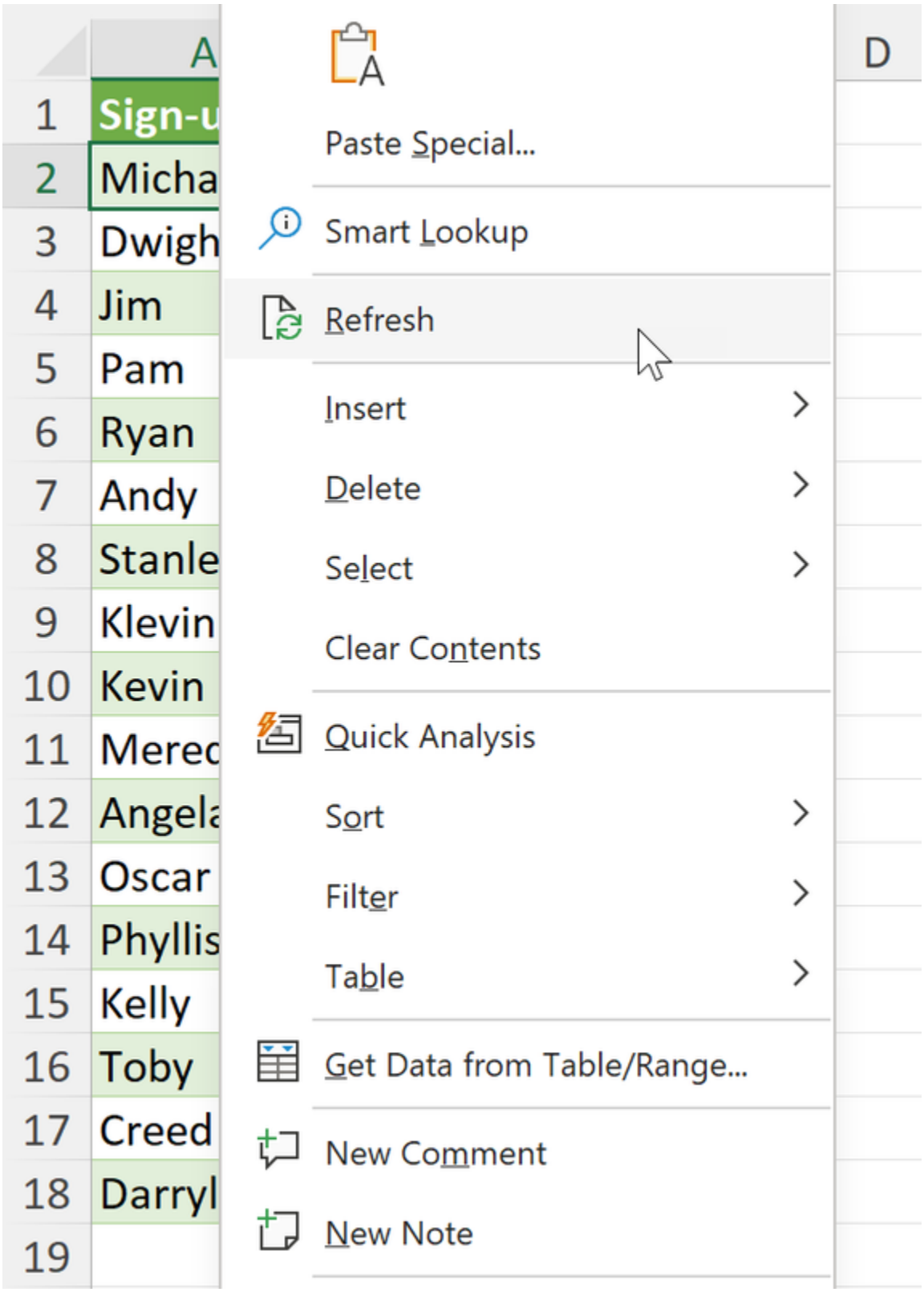


Figure 2-4. Refreshing a query in Power Query

By doing so, the table will be automatically updated, applying all steps to the refreshed data. This workbook now features a one-click, reproducible data cleaning process that can be utilized for any future signup lists.

Splitting data into rows

Have you ever encountered a situation where you have a list of items separated by commas in Excel, and you wished to break them into individual cells? Examine the following example displayed in Figure 2-5. You can find this data in the `roster` worksheet of `ch_03.xlsx`.

	A	B
1	Department	Signups
2	Sales	Jim, Dwight, Phyllis, Andy
3	Accounting	Kevin, Oscar
4	Warehouse	Daryl, Nate
5	Annex	Toby, Ryan, Kelly
6		

Figure 2-5. Cleaning the RSVPs data

This dataset contains signups for an upcoming office party categorized by department and name. Our objective is to conveniently sort and filter this data based on name and department. In classic Excel, one approach would be to utilize the “Text to Columns” feature, resulting in a messy, unsatisfactory outcome like Figure 2-6:

	A	B	C	D	E
1	Department	Signups	Column1	Column2	Column3
2	Sales	Jim	Dwight	Phyllis	Andy
3	Accounting	Kevin	Oscar		
4	Warehouse	Daryl	Nate		
5	Annex	Toby	Ryan	Kelly	
6					

Figure 2-6. RSVPs split into columns using Text to Columns

Power Query provides a convenient solution for splitting the data, offering the advantage of both text-to-columns and text-to-rows functionality.

Splitting Signups by column

To start, import the `roster` data into Power Query and name the query `roster`. In the Power Query editor, go to the Home tab and click on the "Split Column" option. From the dropdown menu, select "By Delimiter," as seen in Figure 2-7, to proceed:

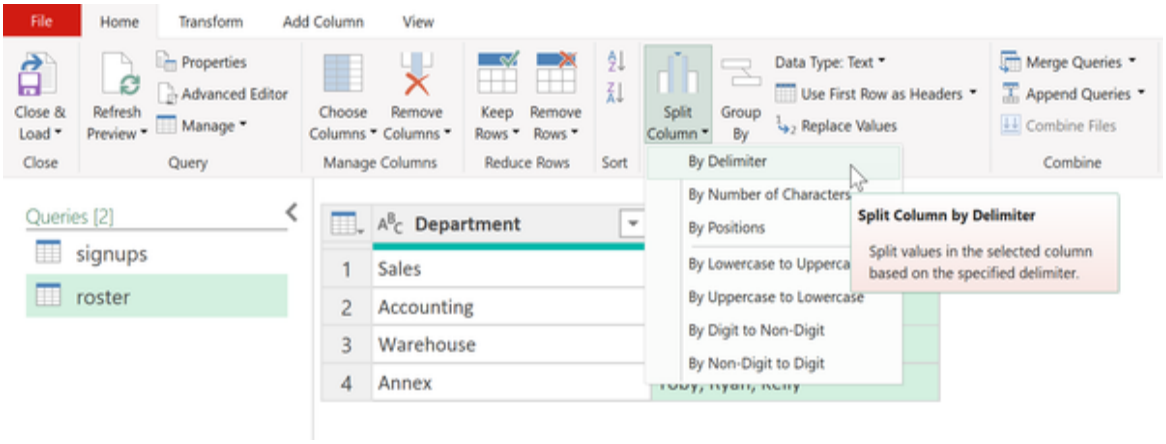


Figure 2-7. Split by delimiter in Power Query

The term "delimiter" refers to the character separating each item in the data. In this case, the delimiter is a comma, and Power Query will likely detect it automatically. If it doesn't, choose "Comma" from the delimiter menu. Next, click on Advanced Options. Here you will find the hidden option to

convert text to rows rather than the standard columns, as seen in Figure 2-8. Click on Rows and then select OK.

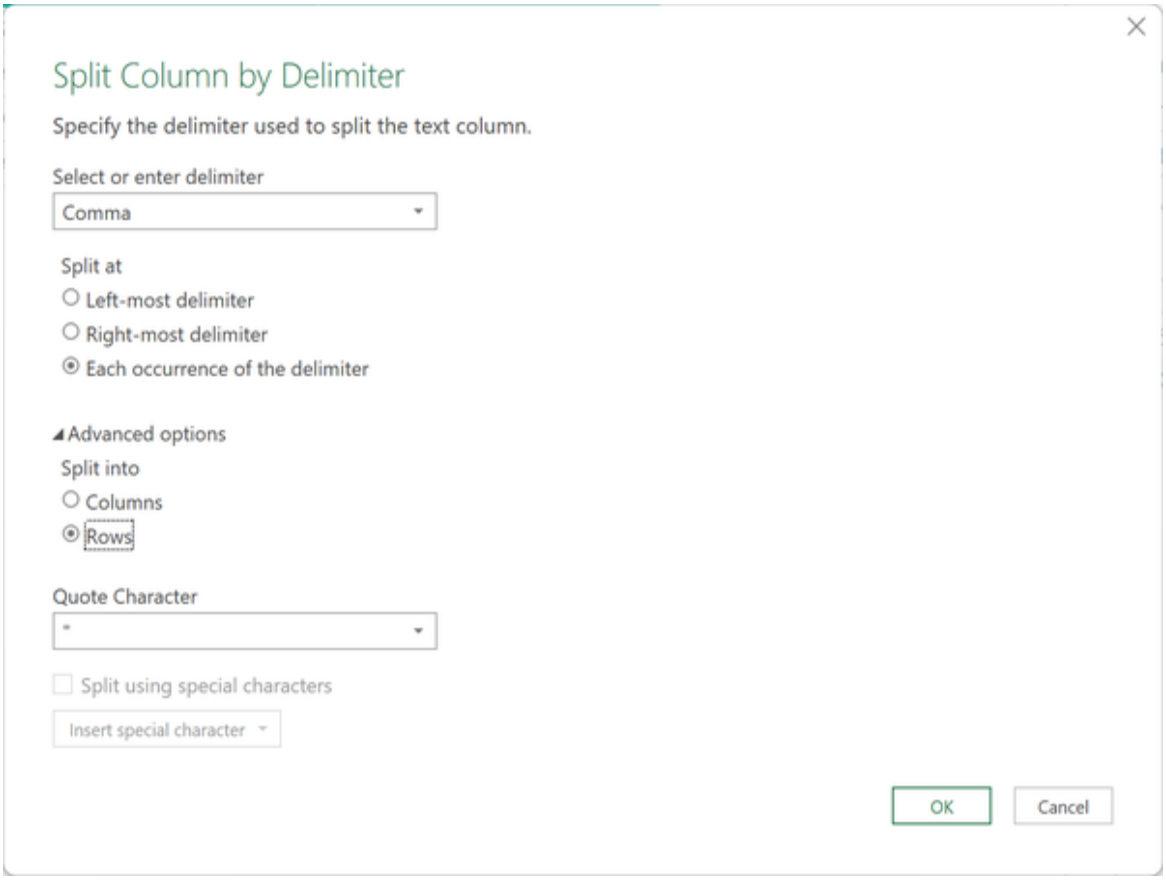


Figure 2-8. Power Query Text to Rows

Stripping the Whitespace

This query is nearly ready to load, but there is one final task to complete. To find out what it is, navigate to the View tab in the Power Query Editor ribbon. Under the Data Preview section, ensure that "Show whitespace" is checked on. For a refresher on Power Query's data previewing and profiling features, refer back to Chapter 2.

Extra spaces, left over from when the roster was delimited by commas, are now seen in the `Signups` column. To remove them, click on the column header, then select Transform > Trim, as seen in Figure 2-9:

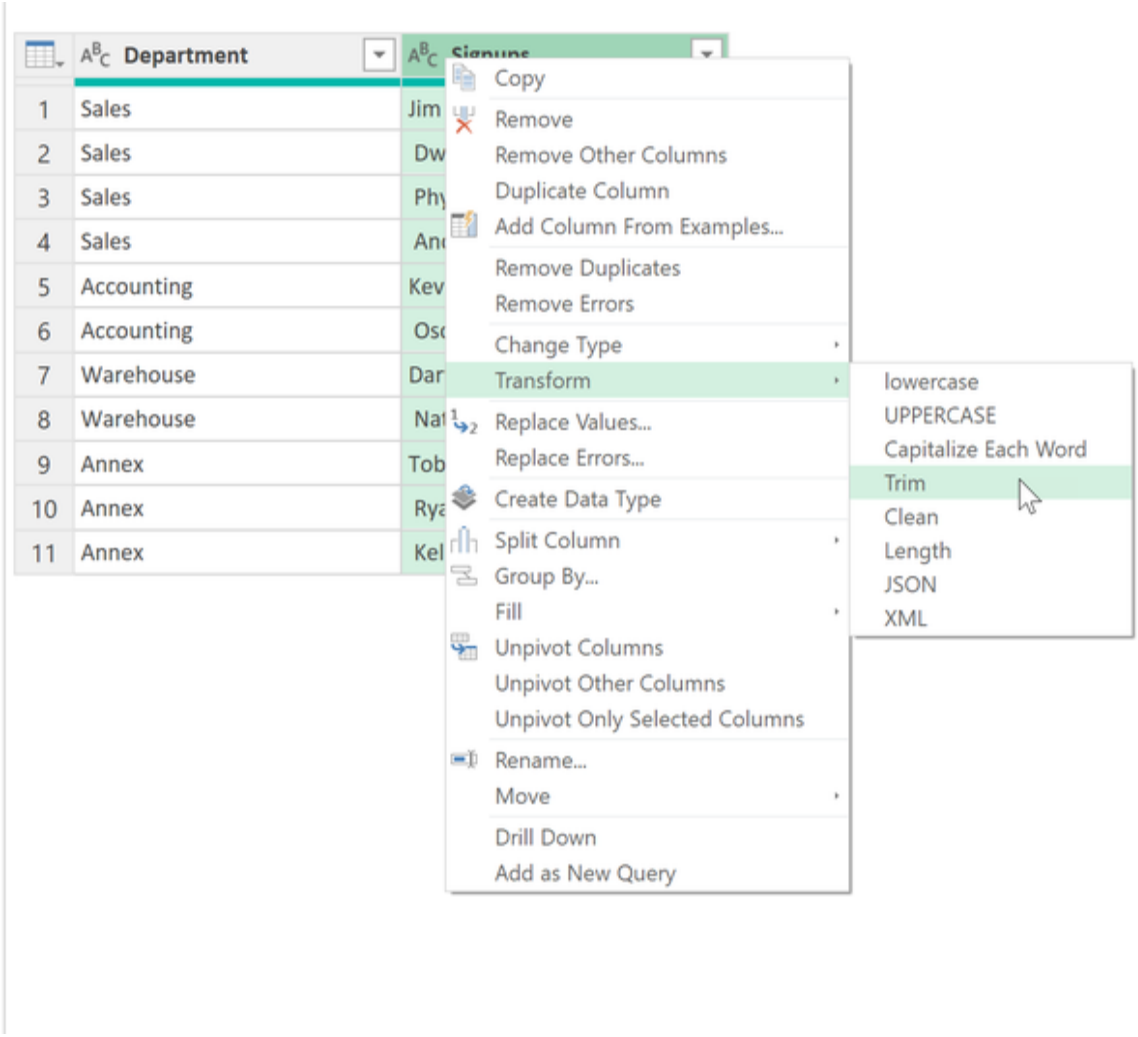


Figure 2-9. Trimming whitespace in Power Query

You can now close & load the results to a table, such as in Figure 2-10:

	A	B
1	Department	Signups
2	Sales	Jim
3	Sales	Dwight
4	Sales	Phyllis
5	Sales	Andy
6	Accounting	Kevin
7	Accounting	Oscar
8	Warehouse	Daryl
9	Warehouse	Nate
10	Annex	Toby
11	Annex	Ryan
12	Annex	Kelly
13		

Figure 2-10. RSVP data split into rows

Filling in headers and cell values

Both Chapter 1 and this chapter have discussed Power Query's handling of missing values as `null`. You may encounter instances where parts of a dataset are erroneously marked as `null` or otherwise missing by mistake. This can be attributed to formatting issues in an external system or poor data storage practices.

This section demonstrates how Power Query can assist in fixing both missing headers and missing values. For this next demonstration, refer to the `sales` worksheet in `ch_03.xlsx`.

Replacing column headers

Some enterprise resource planning extracts contain an additional row filled with irrelevant information. In this particular dataset, the first row is marked with ### in each column, while the actual column headers can be found in row 2, as shown in Figure 2-11. Instead of manually addressing this issue on a weekly basis by deleting the unnecessary row, it is possible to automate the cleanup with Power Query.

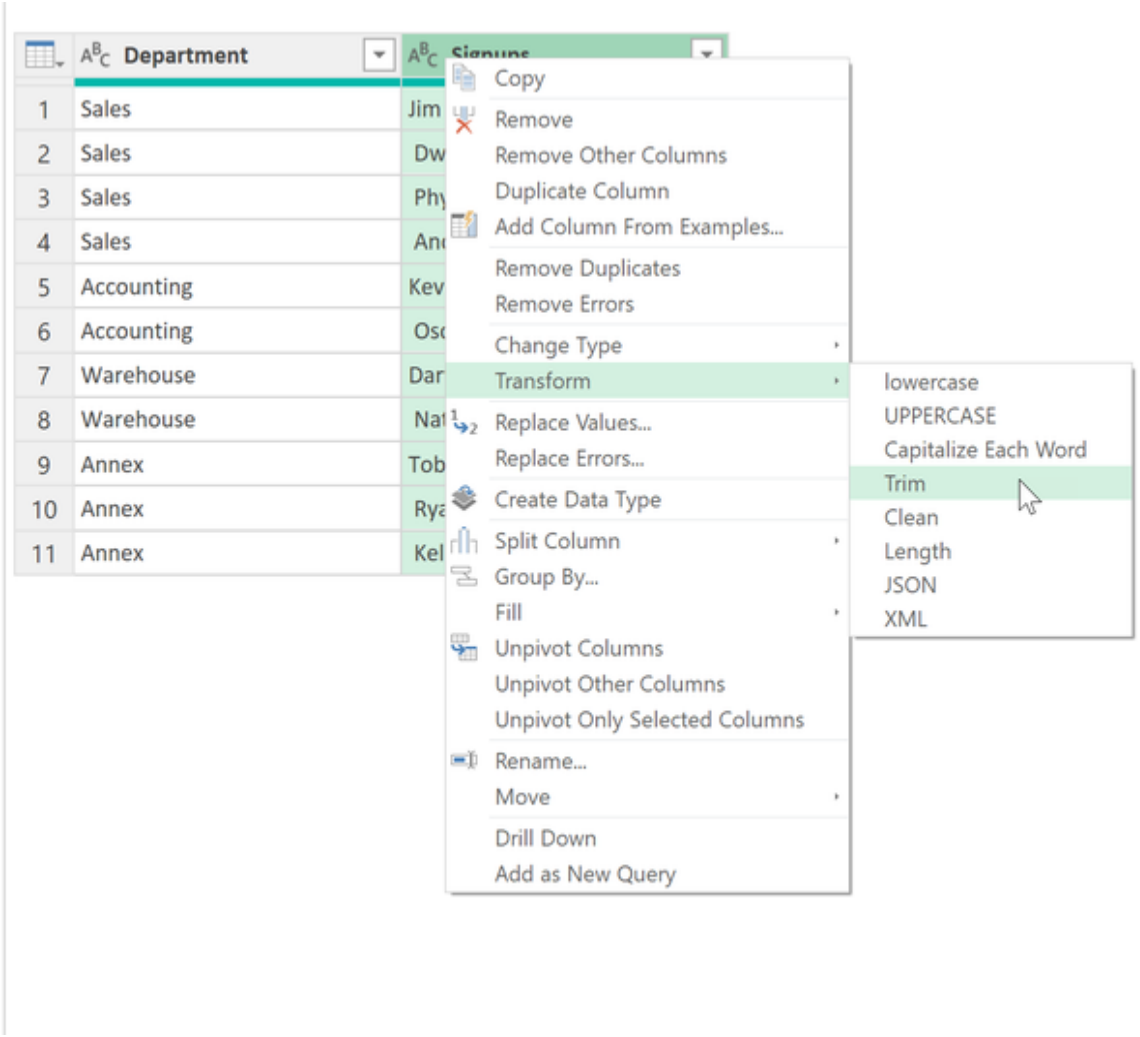


Figure 2-11. An improperly formatted ERP extract

After loading the data into Power Query and naming the query `sales`, navigate to the Home tab and within the Transform group, select Use First Row as Headers such as in Figure 2-12:

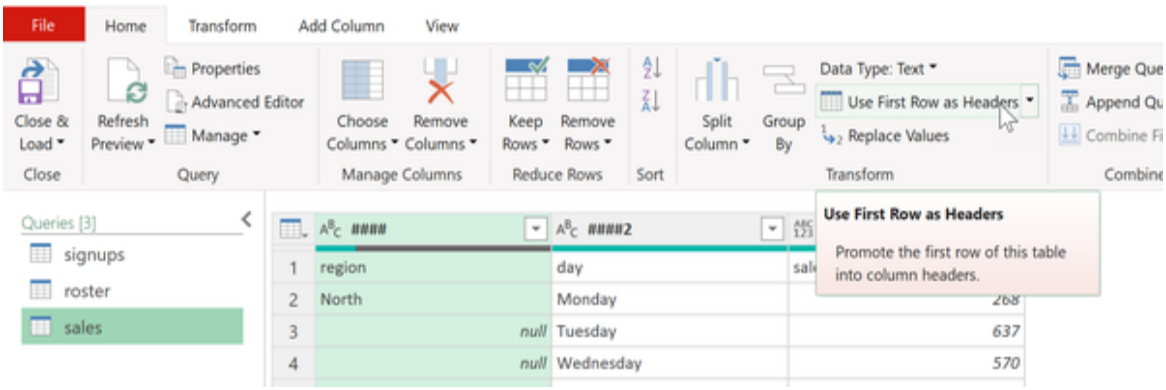


Figure 2-12. Using the first row as headers in Power Query

Now that the column headers have been resolved, it's time to address the issue of mistakenly blank rows. It seems that the ERP system fails to repeat the `region` label across every value for each category. This format is not considered "tidy" and can pose difficulties when utilizing PivotTables or other features to analyze the data. To fix this, head to the Transform tab and within the Any Column group, choose Fill > Down. This can be observed in Figure 2-13:

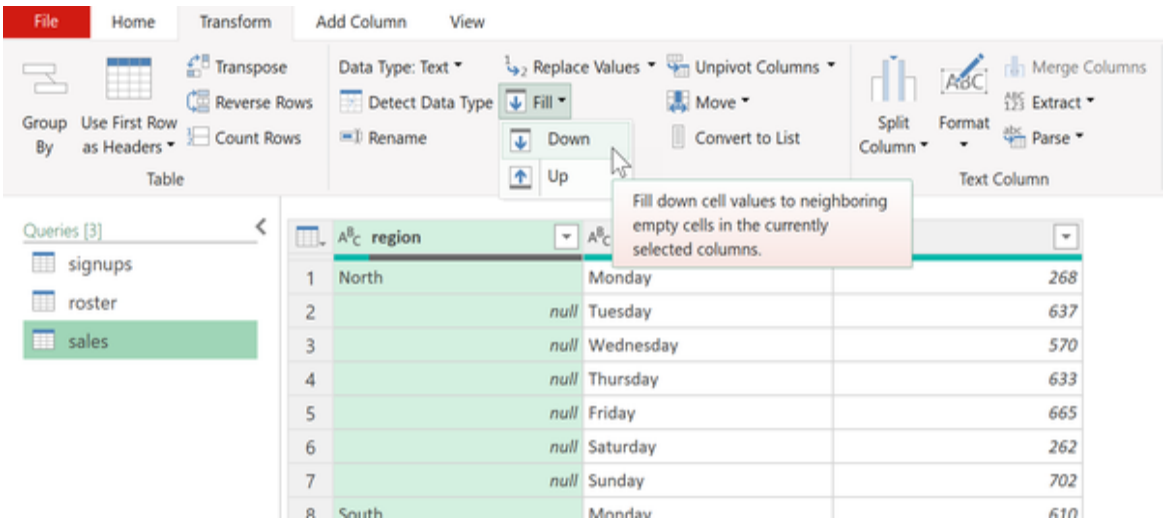


Figure 2-13. Filling down blank values

The data has been successfully organized in a tidier format. You can now proceed to load these results into Excel as a table.

Conclusion

Power Query proves to be an excellent tool for cleaning up rows in data. It simplifies and streamlines tasks such as sorting, filtering, removing duplicates, and handling missing values. Power Query's user-friendly interface enables auditable and reproducible data cleaning without coding. Chapter 3 continues this theme, focusing on transforming not rows but columns.

Exercises

The `datasets` folder of the book's companion repository contains a `state-populations.xlsx` workbook with two worksheets. Use Power Query to perform the following:

`states` worksheet:

1. Remove the `United States` row from the data.
2. Fill down blanks on the `region` and `division` columns.
3. Sort by `population` from high to low.
4. Load results into a PivotTable.

`midwest_cities` worksheet:

1. Load this data into a table where each city is in its own row.

You can find a completed version of this file in `ch_03_exercise_solutions.xlsx` in the `exercise_solutions` folder of the book's companion repository.

Chapter 3. Transforming Columns in Power Query

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at shunter@oreilly.com.

After getting familiar with operating on rows in the Chapter 2, the focus now shifts to columns. This chapter includes various techniques like transforming string case, reformatting columns, creating calculated fields, and more. To follow the demonstrations in this chapter, refer to `ch_04.xlsx` in the `ch_04` folder of the book's repository. Go ahead and load the `rentals` table into Power Query.

Changing column case

Power Query simplifies the conversion of text columns from lowercase to uppercase, uppercase to proper case, and so forth. To demonstrate this, hold down the `Ctrl` key and select both the `Title` and `Artist Name` columns. Then, right-click on either column and choose Transform > Capitalize Each Word, such as in Figure 3-1:

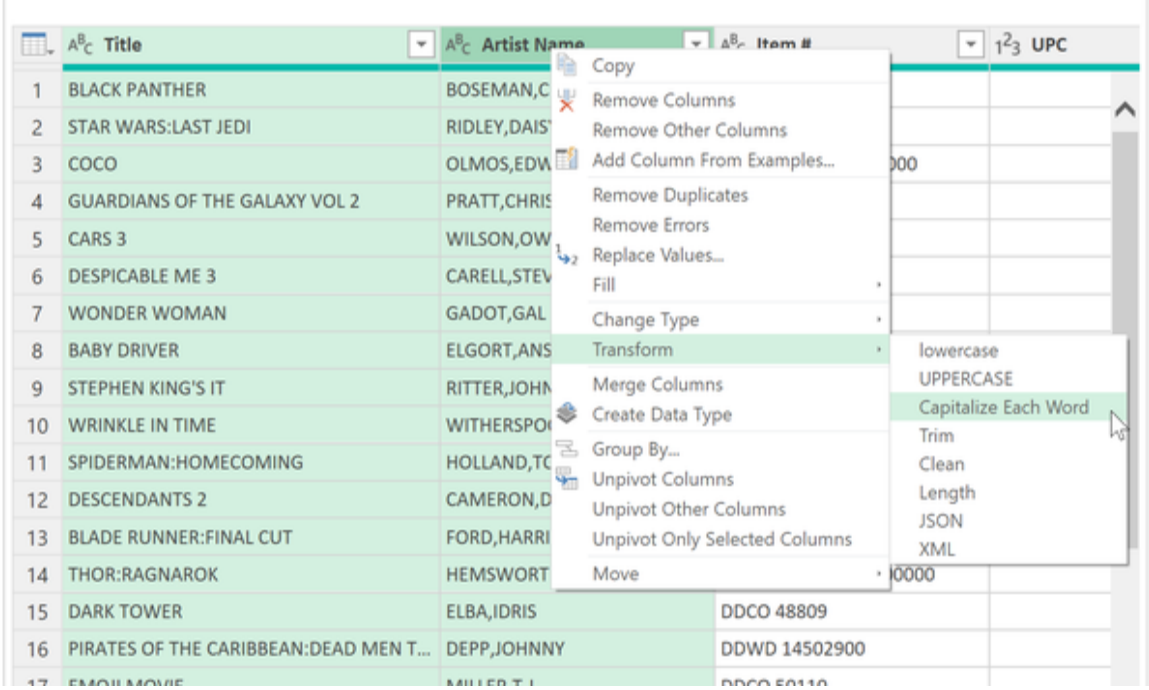


Figure 3-1. Changing Text Case in Power Query

Notice that both `Title` and `Artist Name` lack spaces after colons or commas. To address this, right-click on either column, choose Replace Values, and in the Replace Values menu, search for `:`. Replace it with `: ` and an additional space, such as in Figure 3-2:

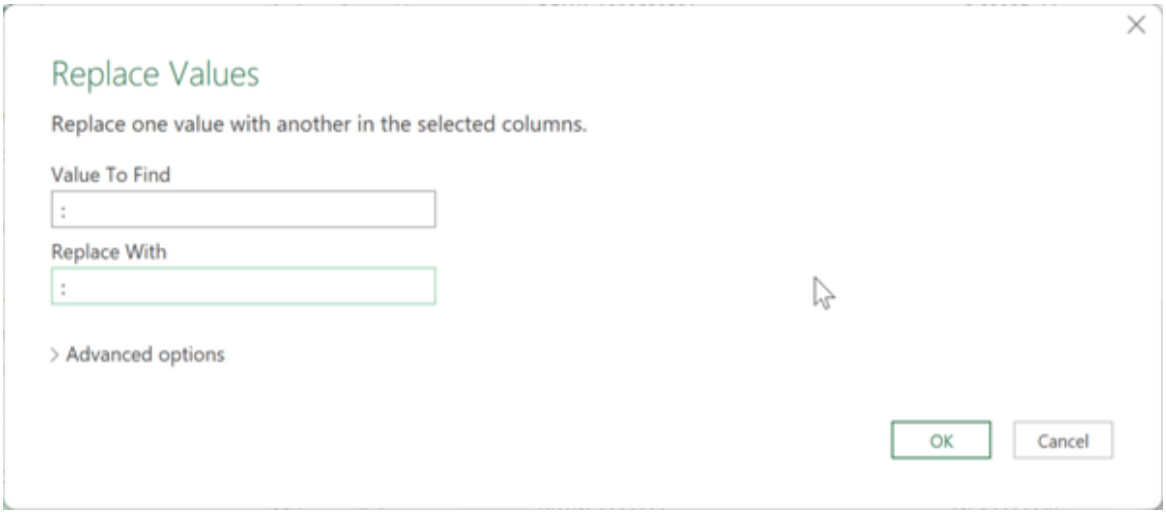


Figure 3-2. Replacing values in Power Query

Next, apply the same process to commas: replace them with a comma followed by an extra space.

As demonstrated in *Chapter 3*, Power Query captures every step you perform on the data in the Applied Steps menu. This feature greatly facilitates the auditing of text changes compared to the conventional Find and Replace menu.

Delimiting by column

In Chapter 2, you learned how to delimit text by rows. Now it's time to do the same with columns. Right-click on the `Item #` column and split it into two by selecting Split Column > By Delimiter > Space. Once again, this process offers improved user-friendliness and a broader range of features compared to the traditional Text to Columns menu.

Changing data types

In Power Query, each column is assigned a specific data type, which defines the operations that can be performed on it. When importing a dataset, Power Query automatically tries to determine the most appropriate data type for each column. However, there are situations where this automatic detection can be enhanced or adjusted.

For example, take the `UPC` column. By default, it is assigned the Whole Number data type. However, since we don't anticipate conducting significant mathematical operations on this column, it's more suitable to store it as text. To do this, click the number icon next to the `UPC` column and change its data type to Text, as seen in Figure 3-3:

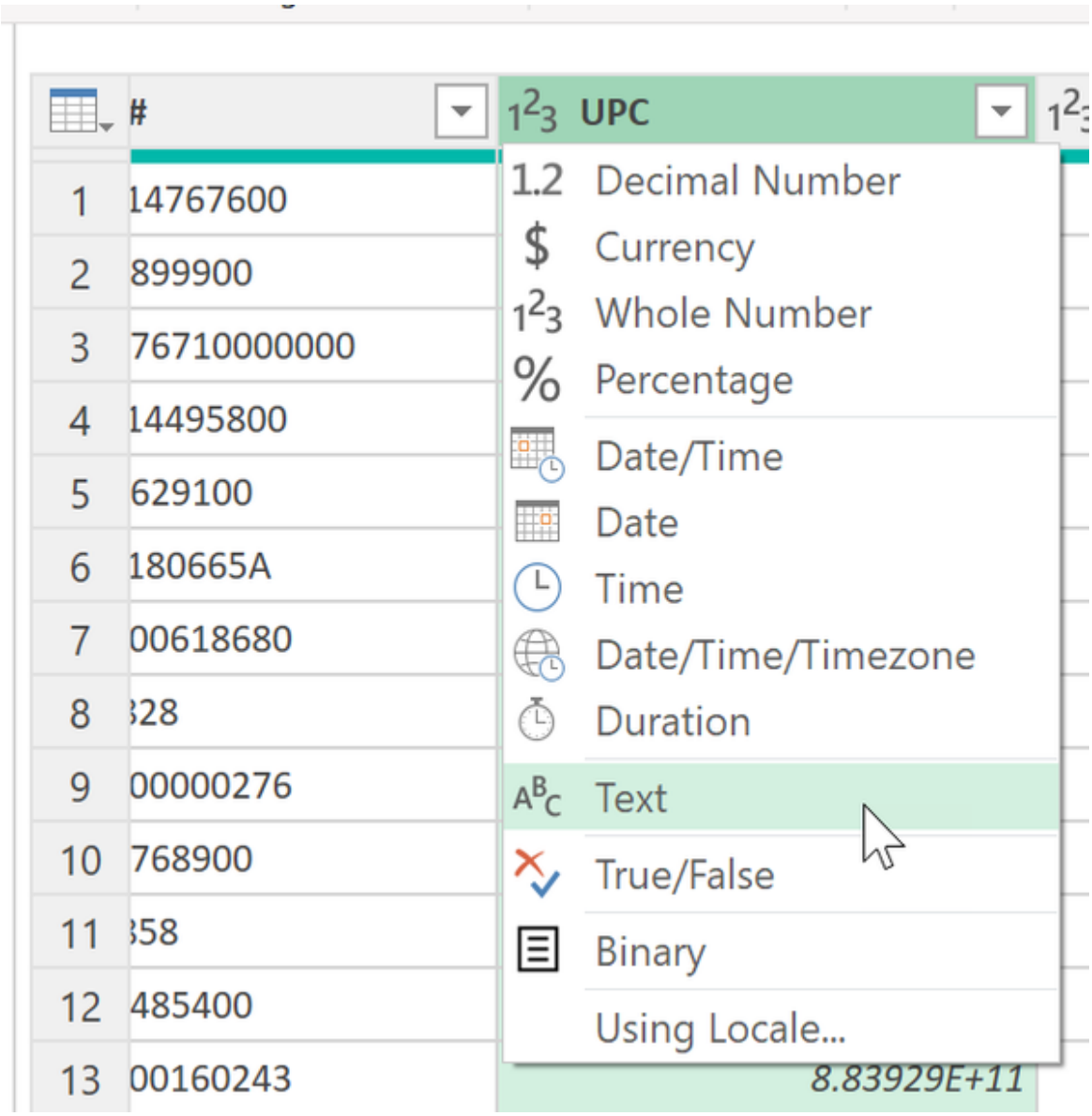


Figure 3-3. Changing column data types in Power Query

Proceed with the following data type changes:

- Convert the `ISBN 13` column to Text.
- Convert the `Retail` column to Currency.

Deleting columns

Removing unnecessary columns from a dataset makes it easier to process and analyze. Select the `BTkey` column and press `Delete` to remove it from your query. If you decide to include this column later, you can easily retrieve it through the Applied Steps menu, as explained in Chapter 2.

Reformatting data

To modify the display of a column without altering its data type, you can adjust its formatting. This is often done for date fields such as `Release Date`. To achieve this, create distinct columns for the year, month, and day values derived from the original data. Start by duplicating the column through a right-click and selecting “Duplicate Column.” Repeat this action twice, resulting in a total of three date columns.

Choose the initial `Release Date` column and navigate to Transform > Year > Year. The column will then be reformatted to display only the year number instead of the complete date.

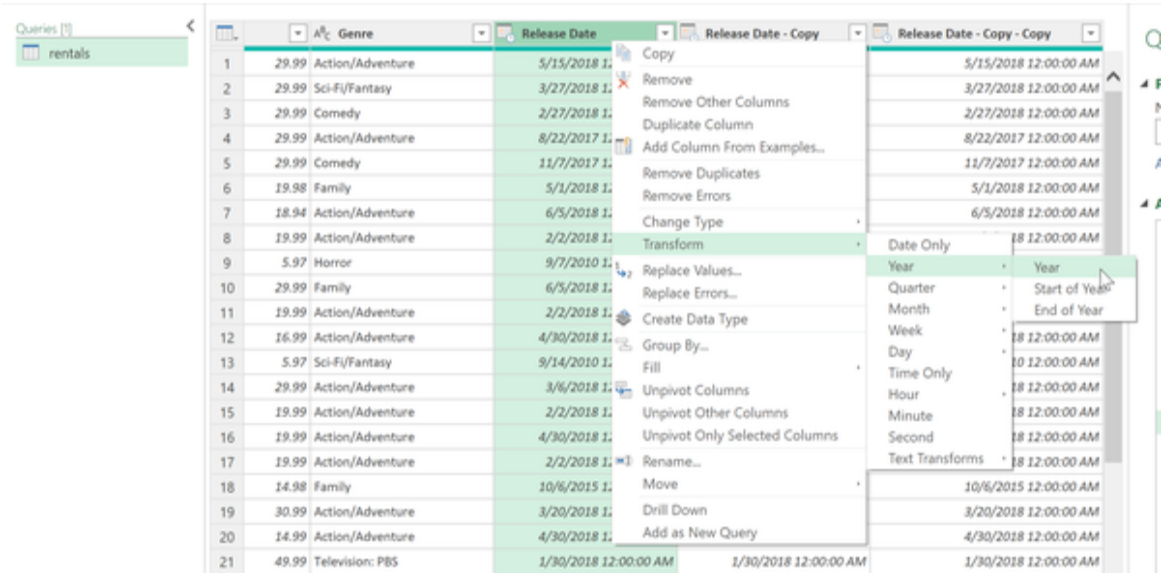


Figure 3-4. Transforming date columns in Power Query

Extract the month and day numbers from the next two columns. Double-click the column headers and rename them as `Year`, `Month`, and `Day`, respectively, to reflect the reformatted data. Close and load your results to an Excel table.

Creating custom columns

Adding a calculated column is a common task in data cleaning. Whether it's a profit margin, date duration, or something else, Power Query handles this process through its M programming language.

For this next demonstration, head to the `teams` table in the `teams` worksheet of `ch_04.xlsx`. This dataset includes season records for every Major League Baseball team since 2000. Our objective is to create a new column that calculates the win record for each team during the season. This calculation is done by dividing the number of wins by the total number of wins and losses combined.

The first step, of course, is to load the data into Power Query. Then from the ribbon of the Editor, head to Add Column > Custom Column. Name your custom column `Wpct` and define it using the following formula:

```
[W] + ([W] + [L])
```

The M programming language of Power Query follows a syntax resembling Excel tables, where column references are enclosed in single square brackets. Take advantage of Microsoft's IntelliSense to hit tab and automatically complete the code as you type these references. Additionally, you have the option to click on the desired columns from the "Available columns" list, which will insert them into the formula area.

If everything is correct, a green checkmark will appear at the bottom of the menu, indicating that no syntax errors have been detected, such as in Figure 3-5:

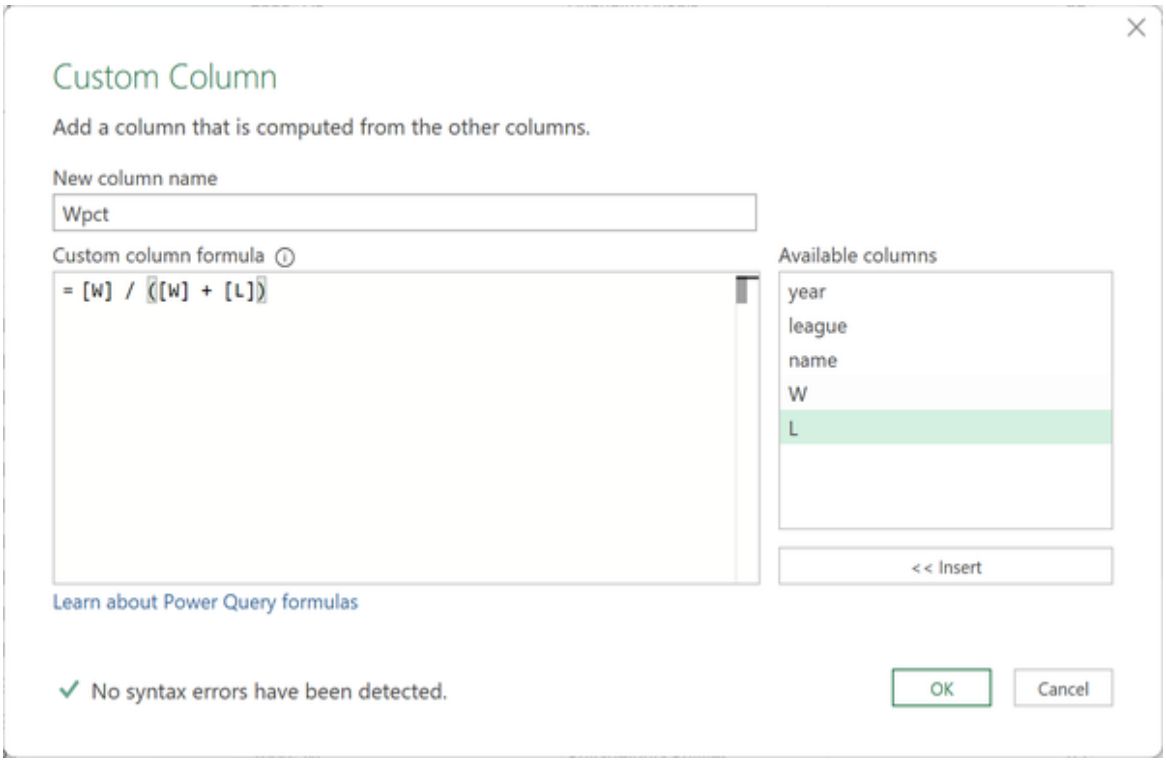
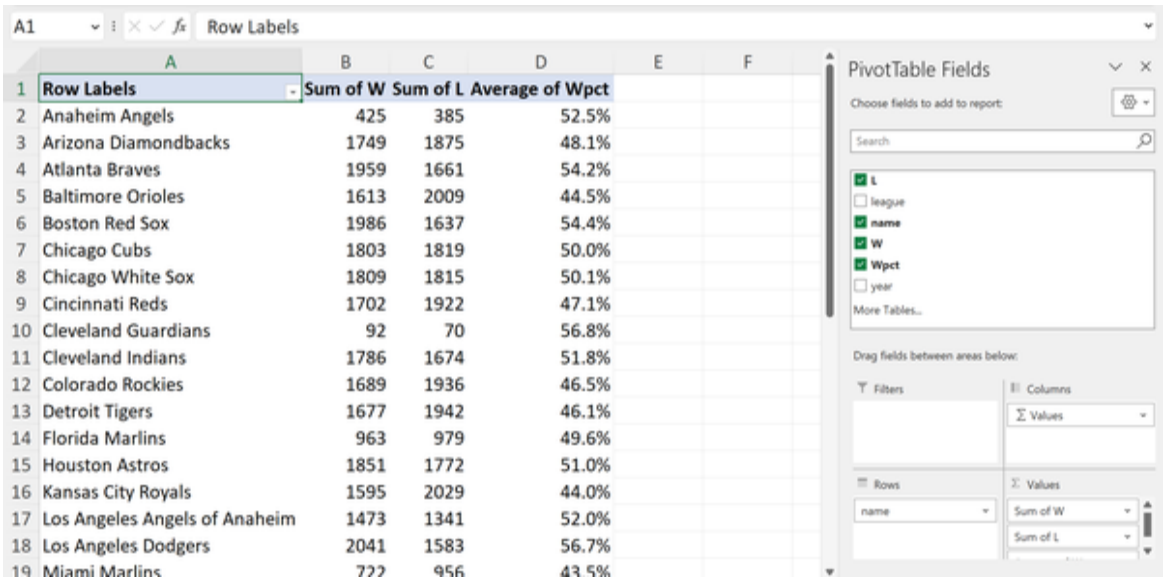


Figure 3-5. Creating winning percentage calculation

Loading & inspecting the data

Our new column is calculated and ready to work with. On the Power Query Editor ribbon, head to Home > Close & Load > Close & Load To, then select PivotTable Report. From there, you can analyze the data, such as calculating the average `Wpct` for each team name, such as in Figure 3-6:



Row Labels	Sum of W	Sum of L	Average of Wpct
Anaheim Angels	425	385	52.5%
Arizona Diamondbacks	1749	1875	48.1%
Atlanta Braves	1959	1661	54.2%
Baltimore Orioles	1613	2009	44.5%
Boston Red Sox	1986	1637	54.4%
Chicago Cubs	1803	1819	50.0%
Chicago White Sox	1809	1815	50.1%
Cincinnati Reds	1702	1922	47.1%
Cleveland Guardians	92	70	56.8%
Cleveland Indians	1786	1674	51.8%
Colorado Rockies	1689	1936	46.5%
Detroit Tigers	1677	1942	46.1%
Florida Marlins	963	979	49.6%
Houston Astros	1851	1772	51.0%
Kansas City Royals	1595	2029	44.0%
Los Angeles Angels of Anaheim	1473	1341	52.0%
Los Angeles Dodgers	2041	1583	56.7%
Miami Marlins	722	956	43.5%

Figure 3-6. Summarizing the results in a PivotTable

Calculated columns versus measures

It is important to note that the average `Wpct` displayed in the PivotTable is a simple, unweighted average of the season winning percentages. This means that seasons with a lower number of games, such as the pandemic-affected 2020 season, have a disproportionate impact on the calculation. To verify this, we can compare the `Average of Wpct` value in the PivotTable with our own Excel calculation, such as in Figure 3-7:

	A	B	C	D	E	F	G
1	**Row Labels**	**Sum of W**	**Sum of L**	**Average of Wpct**	Average Wpct (Measure)		
2	Anaheim Angels	425	385	52.47%	52.47%	=B2 / (B2 + C2)	
3	Arizona Diamondbacks	1749	1875	48.08%	48.26%		
4	Atlanta Braves	1959	1661	54.23%	54.12%		
5	Baltimore Orioles	1613	2009	44.45%	44.53%		
6	Boston Red Sox	1986	1637	54.41%	54.82%		
7	Chicago Cubs	1803	1819	49.97%	49.78%		
8	Chicago White Sox	1809	1815	50.14%	49.92%		
9	Cincinnati Reds	1702	1922	47.09%	46.96%		
10	Cleveland Guardians	92	70	56.79%	56.79%		
11	Cleveland Indians	1786	1674	51.81%	51.62%		
12	Colorado Rockies	1689	1936	46.50%	46.59%		
13	Detroit Tigers	1677	1942	46.14%	46.34%		
14	Florida Marlins	963	979	49.59%	49.59%		
15	Houston Astros	1851	1772	51.02%	51.09%		
16	Kansas City Royals	1595	2029	43.99%	44.01%		
17	Los Angeles Angels of Anaheim	1473	1341	52.03%	52.35%		
18	Los Angeles Dodgers	2041	1583	56.74%	56.32%		

Figure 3-7. Apparent PivotTable miscalculation

To address this issue, one approach is to utilize dynamic measures that perform real-time aggregation and calculations based on the specific context of the analysis. This is achieved through tools like Power Pivot and the DAX language, which Part 2 of this book discusses.

This doesn't mean that calculated columns in Power Query should be avoided altogether. They are simple to create and computationally efficient. Nevertheless, if there is a possibility that these columns might lead to misleading aggregations, it is advisable to opt for a DAX measure instead.

Reshaping data

In Chapter 1, you were introduced to the concept of "tidy" data, where every variable is stored in one and only one column. You may recall the data in the `sales` worksheet of `ch_03.xlsx` as an example of untidy data. Fortunately, Power Query resolves this critical data storage problem effortlessly using Power Query. To begin, navigate to the familiar `sales` table in the `sales` worksheet of the `ch_04.xlsx` workbook. Load this table into Power Query to initiate the data transformation process.

The goal is to “unpivot” or “melt” the all sales into one column `sales`, along with the labels for those sales in one column called `department`. To do so, hold down the `Ctrl` key and select the first three variables: `customer_id`, `channel` and `region`. Right-click and select Unpivot Other Columns, such as in Figure 3-8:

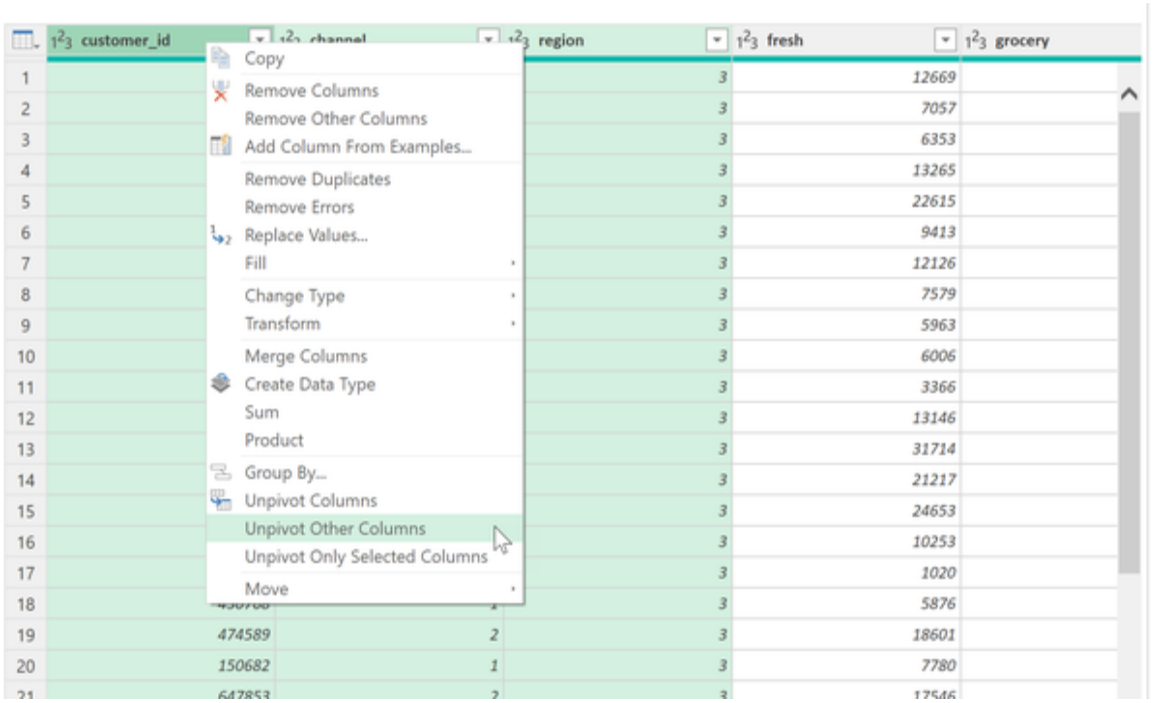


Figure 3-8. Unpivoting a dataset in Power Query

By default, your two resulting unpivoted columns will be called `Attribute` and `Value`. Rename them to the desired `department` and `sales`. You can now load the query to a PivotTable and easily analyze sales by channel and region.

Conclusion

This chapter explored different ways to manipulate columns in Power Query. Chapter 5 takes a step further by working with multiple datasets in a single query. You’ll learn how to merge and append data sources, as well as how to connect to external sources like *.csv* files.

Exercises

Practice transforming columns in Power Query using the `work_orders.xlsx` dataset from the `datasets` folder in the book's companion repository. Perform the following transformations to this dataset:

1. Transform the `date` column to a month format, such as changing *1/1/2023* to *January*.
2. Transform the `owner` column to proper case.
3. Split the `location` column into two separate columns: `zip` and `state`.
4. Reshape the dataset so that `subscription_cost`, `support_cost`, and `services_cost` are consolidated into two columns: `category` and `cost`.
5. Introduce a new column named `tax` that calculates 7% of the values in the `cost` column.
6. Convert the `zip` variable to the Text data type, and update both `cost` and `tax` variables to Currency.
7. Load the results to a table.

For the solutions to these transformations, refer to the `ch_04_solutions.xlsx` file in the `exercise_solutions` folder of the book's companion repository.

Chapter 4. Introducing Dynamic Array Functions

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at shunter@oreilly.com.

Thus far, this book has covered building measures for Power Pivot with DAX and touched on M code in Power Query to a lesser extent. However, it hasn't examined the traditional workbook formulas and functions that have long been the foundation of Excel. Even this area of the program, which may have seemed neglected in favor of other flashy developments like Power Pivot and Power Query, has undergone significant improvements, becoming more powerful and capable.

This chapter introduces dynamic array functions, shedding light on their capabilities. You will learn how to sort, filter, and join datasets, among other tasks, using the familiar environment of the Excel formula bar.

FORMULAS VERSUS FUNCTIONS IN EXCEL

In Excel, formulas and functions have distinct meanings. A formula is a mathematical expression that manipulates data using operators, cell references, and constants; for example, `=B1 * B2`. In contrast, a function is a predefined formula providing extensive data analysis and manipulation capabilities; for example, `TRIMMEAN(B1:B3)`.

Formulas can include functions, blurring the distinction. This chapter aims to respect these as separate concepts while recognizing their interdependence.

Dynamic array functions explained

Dynamic array functions have impressive capabilities, and it's tempting to start experimenting with them right away. However, it's crucial to understand what makes these functions special and how they differ from traditional Excel approaches. The following section explores the path from arrays to array references and finally to dynamic array functions.

What is an array in Excel?

To follow this demonstration, open `ch_10.xlsx` in the `ch_10` folder of the book's resources. Go to the `array-references` worksheet.

First and foremost, an array in Excel refers to a collection of values. For instance, a basic array could consist of the numbers 3, 4, and 7 placed in cells `A2:C2`, as shown in Figure 4-1:

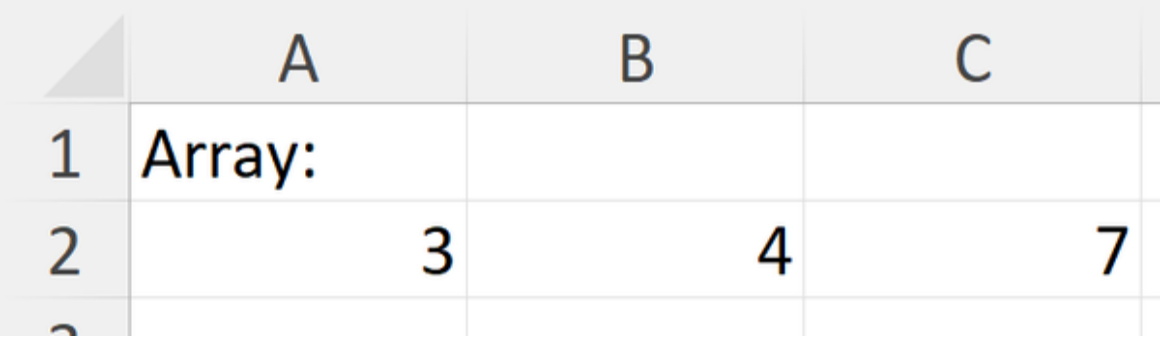


Figure 4-1. A basic Excel array

> **ARRAYS VERSUS RANGES IN EXCEL**
>
> `A2:C2` is an example of a range, yet we're referring to it as an array. What's the difference? The range is the set of cells and their physical location, whereas the array is the data in those cells. You can read more about the difference here.

Array references

With an understanding of what constitutes an array in Excel, this next section explores various approaches to build array references.

Static array references

To create a tradititional Excel array reference, enter `=A2:C2` into cell `E2` and press `Ctrl + Shift + Enter` to indicate that you are referencing an array of values rather than a single value, as in Figure 4-2:

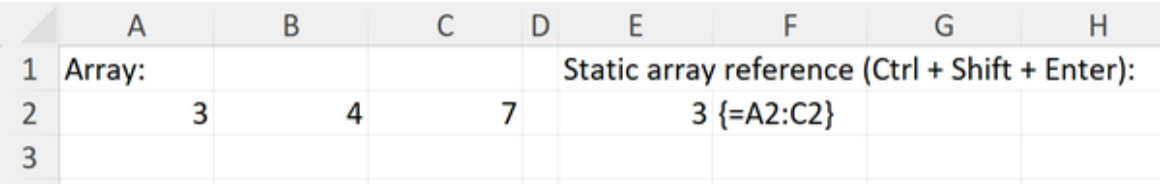


Figure 4-2. A basic Excel array reference

You will see the resulting formula enclosed in curly-cue brackets `{}`, but you will not see all three values in the array. Why is this?

In Excel, each cell typically holds only one data point — not three, as you are trying to do here. To populate multiple cells with data from an array, highlight `E2:G2` and enter the same reference, as in Figure 4-3:

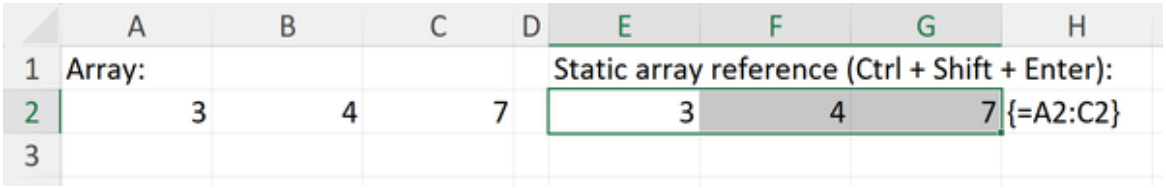


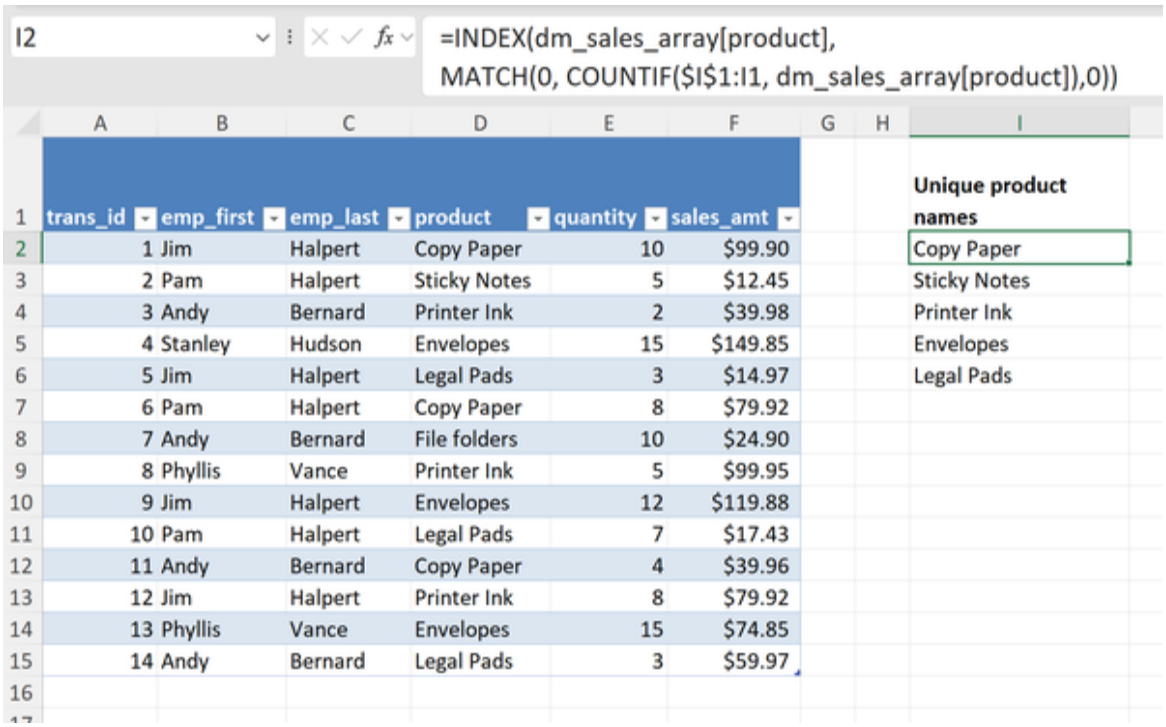
Figure 4-3. An improved Excel array reference

Excel's conventional approach to handling arrays comes with limitations. The process of writing and managing references using `Ctrl + Shift + Enter` can be arduous and lacks automatic adjustment.

Consider a scenario where cells are inserted or deleted between `A2:C2` — in such cases, the array reference fails to resize itself. Consequently, these array references could be described as "static" because they do not dynamically adapt to changes in the spreadsheet's structure or cell count.

Dynamic array references

Modern Excel addresses the limitations of static arrays with dynamic arrays. Now, to refer to `A2:C2`, you simply need to type `=A2:C2` and press Enter, such as in cell `E5` of Figure 4-4:



trans_id	emp_first	emp_last	product	quantity	sales_amt
1	Jim	Halpert	Copy Paper	10	$99.90
2	Pam	Halpert	Sticky Notes	5	$12.45
3	Andy	Bernard	Printer Ink	2	$39.98
4	Stanley	Hudson	Envelopes	15	$149.85
5	Jim	Halpert	Legal Pads	3	$14.97
6	Pam	Halpert	Copy Paper	8	$79.92
7	Andy	Bernard	File folders	10	$24.90
8	Phyllis	Vance	Printer Ink	5	$99.95
9	Jim	Halpert	Envelopes	12	$119.88
10	Pam	Halpert	Legal Pads	7	$17.43
11	Andy	Bernard	Copy Paper	4	$39.96
12	Jim	Halpert	Printer Ink	8	$79.92
13	Phyllis	Vance	Envelopes	15	$74.85
14	Andy	Bernard	Legal Pads	3	$59.97

Figure 4-4. A dynamic Excel array reference

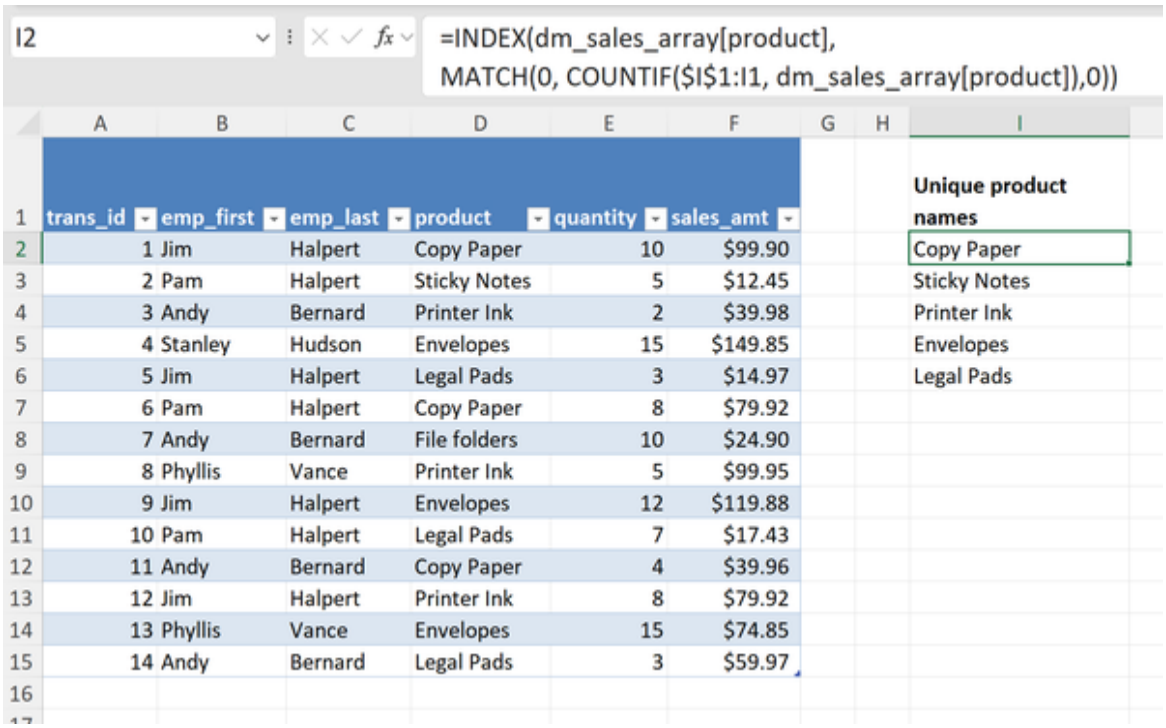
Using this reference, Excel intelligently recognizes the number of cells in the array. This means that if you insert or delete cells between `A2:C2`, the dynamic array reference will automatically resize to accommodate the changes. This dynamic adjustment saves time and effort by removing the need to manually update or modify the references whenever the data layout is altered.

Array formulas

Having contrasted the behavior of array references in classic and modern Excel, the next section delves into the impact on functions utilizing these references.

Static array formulas

Consider the following example which uses a static array formula to list the unique products sold in a transactions dataset, as seen in Figure 4-5. You can follow along with this example in the `array-functions` worksheet of `ch_10.xlsx`.



	A	B	C	D	E	F	G	H	I
1	trans_id	emp_first	emp_last	product	quantity	sales_amt			Unique product names
2	1	Jim	Halpert	Copy Paper	10	$99.90			Copy Paper
3	2	Pam	Halpert	Sticky Notes	5	$12.45			Sticky Notes
4	3	Andy	Bernard	Printer Ink	2	$39.98			Printer Ink
5	4	Stanley	Hudson	Envelopes	15	$149.85			Envelopes
6	5	Jim	Halpert	Legal Pads	3	$14.97			Legal Pads
7	6	Pam	Halpert	Copy Paper	8	$79.92			
8	7	Andy	Bernard	File folders	10	$24.90			
9	8	Phyllis	Vance	Printer Ink	5	$99.95			
10	9	Jim	Halpert	Envelopes	12	$119.88			
11	10	Pam	Halpert	Legal Pads	7	$17.43			
12	11	Andy	Bernard	Copy Paper	4	$39.96			
13	12	Jim	Halpert	Printer Ink	8	$79.92			
14	13	Phyllis	Vance	Envelopes	15	$74.85			
15	14	Andy	Bernard	Legal Pads	3	$59.97			
16									

Figure 4-5. A static array formula

Don’t worry too much about the formula’s inner workings; we’ll explore a more sensible alternative shortly. For now, observe how static arrays can be challenging when it comes to determining the number of output pieces it should return.

Similar to array references, this approach lacks automatic adjustment to list the correct number of unique values, making it cumbersome and unintuitive. For example, if more transactions are added to the `dm_sales` table, any additional unique values, such as *Rubber bands* in this case, are not reflected in the array function, as seen in Figure 4-6:

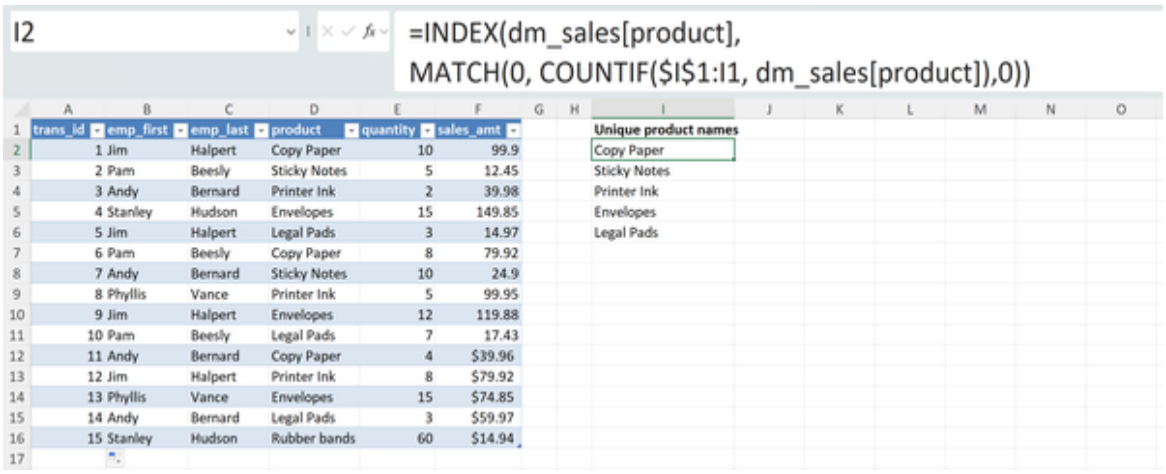


Figure 4-6. Rubber bands (row 16) are not listed as a unique product

To see that extra value, you'll need to extend your `Ctrl + Shift + Enter` formula down one more row.

Dynamic array functions

Dynamic array functions, on the other hand, perform exceptionally well in this scenario. There is even a `UNIQUE()` function specifically designed to handle this task, as seen in Figure 4-7:

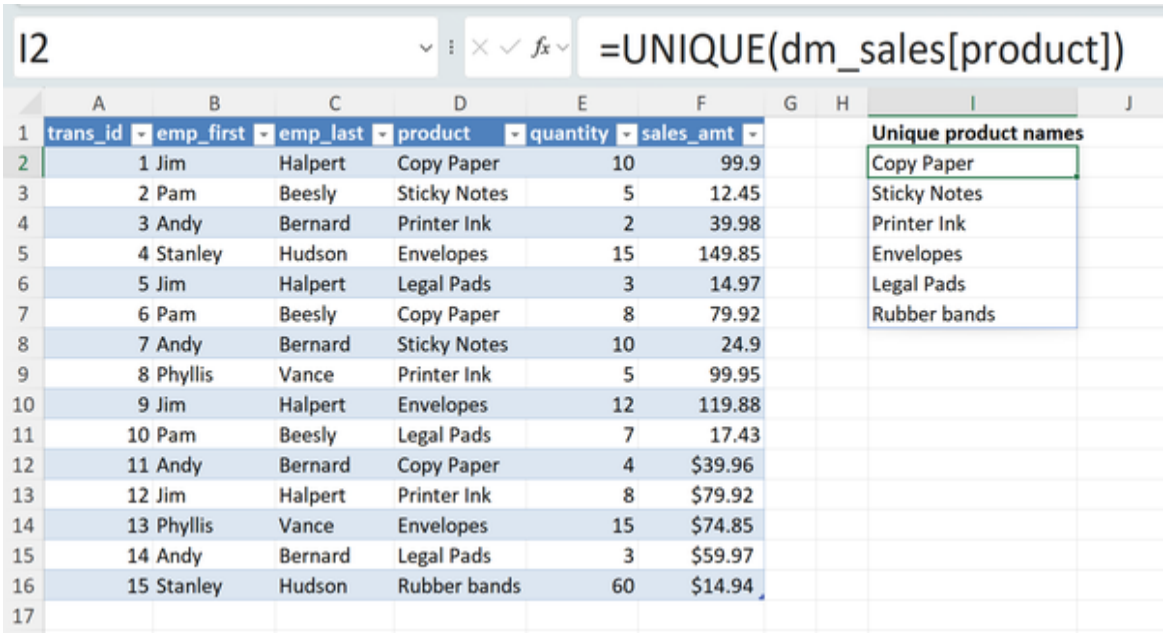


Figure 4-7. Finding unique values with the UNIQUE() function

> **NOTE**
>
> If you see a `#SPILL` error after using the `UNIQUE()` function, make sure you have empty cells below it. This error happens when the results of the function spill over to non-empty adjacent cells.

Dynamic array functions offer a significant advancement over traditional array formulas by instantly updating output cells in response to changes in inputs. This dynamic behavior eliminates the need for manual recalculations or refreshing formulas, providing a seamless and efficient workflow.

An overview of dynamic array functions

With the capabilities of dynamic array functions in mind, it's time to explore a few examples. The first will expand on the `UNIQUE()` function discussed earlier.

Finding Distinct and Unique Values with `UNIQUE()`

In the previous example, the `UNIQUE()` dynamic array function was used to generate a list of unique product names. To explore this function and dataset further, continue using the `dm_sales` table in the `ch_10.xlsx` workbook.

The `UNIQUE()` function parameters

The `UNIQUE()` function has three parameters, two of which are optional. To understand how they work, refer to Table 4-1:

Table 4-1. `UNIQUE()` parameters

Parameter	Description
`range`	Required argument that specifies the data range or array from which you want to extract unique values.
`[by_col]`	Optional argument that determines whether the unique values should be extracted by column or row. By default, unique values are extracted by column.
`[exactly_once]`	Optional argument that specifies whether only values that appear exactly once should be considered unique. By default, all unique values, regardless of their frequency, are extracted. If you set this parameter to `TRUE`, only the values that appear exactly once will be extracted.

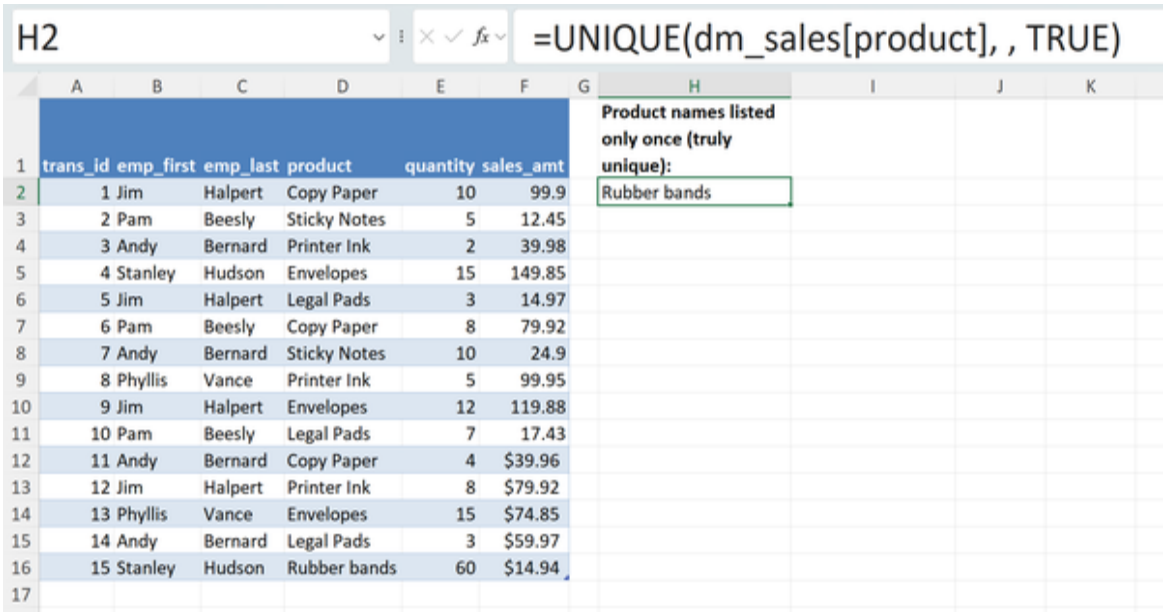
PARAMETERS AND ARGUMENTS IN EXCEL

Parameters and arguments in Excel are often used interchangeably, but they have distinct meanings. Parameters are variables within a function that act as placeholders for values, while arguments are the actual values or references used in a formula. For instance, the `ABS()` function takes a parameter called `number`, and when using the function like `ABS(-10)`, `-10` is the argument.

Finding unique versus distinct values

Chapter 2 covered data profiling in Power Query, emphasizing the difference between distinct and unique values. As a refresher, unique values technically refer to those that appear only once in a range, which make the

`UNIQUE()` function's name somewhat misleading. By default, the function lists *distinct* values rather than strictly unique ones. However, it's possible to obtain genuinely unique values by setting the third argument of the function to `TRUE`, as shown in Figure 4-8:



trans_id	emp_first	emp_last	product	quantity	sales_amt
1	Jim	Halpert	Copy Paper	10	99.9
2	Pam	Beesly	Sticky Notes	5	12.45
3	Andy	Bernard	Printer Ink	2	39.98
4	Stanley	Hudson	Envelopes	15	149.85
5	Jim	Halpert	Legal Pads	3	14.97
6	Pam	Beesly	Copy Paper	8	79.92
7	Andy	Bernard	Sticky Notes	10	24.9
8	Phyllis	Vance	Printer Ink	5	99.95
9	Jim	Halpert	Envelopes	12	119.88
10	Pam	Beesly	Legal Pads	7	17.43
11	Andy	Bernard	Copy Paper	4	$39.96
12	Jim	Halpert	Printer Ink	8	$79.92
13	Phyllis	Vance	Envelopes	15	$74.85
14	Andy	Bernard	Legal Pads	3	$59.97
15	Stanley	Hudson	Rubber bands	60	$14.94

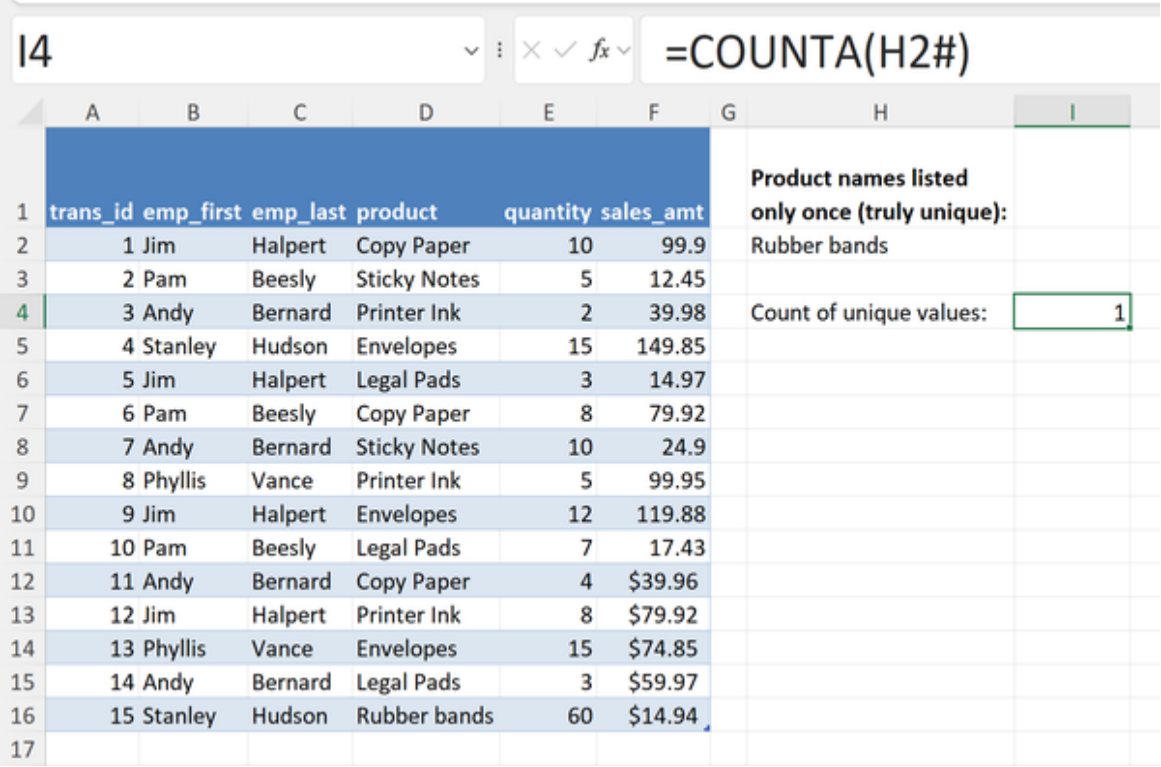
Figure 4-8. Finding the truly unique values with `UNIQUE()`

Using the spill operator

In Excel, it is common to create additional calculations on top of existing ones, such as aggregating the results of a calculated column. The spill operator, denoted by the hash symbol (#), streamlines the aggregation process for dynamic array functions. Like dynamic arrays themselves, it automatically expands the output range to accommodate the data, eliminating the manual entry of array formulas and resizing of ranges. This feature enhances efficiency and conciseness when constructing formulas for aggregating data in Excel.

The number of unique values generated by the formula displayed in Figure 4-8 can be determined using the `COUNTA()` function. The choice of `COUNTA()` over `COUNT()` is due to its inclusion of text-based cells, which is applicable in this case. When selecting the range from `D2:D7`, Excel

automatically refers to it using the spill range indicated by the # operator, such as in Figure 4-9:



	A	B	C	D	E	F	G	H	I
1	trans_id	emp_first	emp_last	product	quantity	sales_amt		Product names listed only once (truly unique):	
2	1	Jim	Halpert	Copy Paper	10	99.9		Rubber bands	
3	2	Pam	Beesly	Sticky Notes	5	12.45			
4	3	Andy	Bernard	Printer Ink	2	39.98		Count of unique values:	1
5	4	Stanley	Hudson	Envelopes	15	149.85			
6	5	Jim	Halpert	Legal Pads	3	14.97			
7	6	Pam	Beesly	Copy Paper	8	79.92			
8	7	Andy	Bernard	Sticky Notes	10	24.9			
9	8	Phyllis	Vance	Printer Ink	5	99.95			
10	9	Jim	Halpert	Envelopes	12	119.88			
11	10	Pam	Beesly	Legal Pads	7	17.43			
12	11	Andy	Bernard	Copy Paper	4	$39.96			
13	12	Jim	Halpert	Printer Ink	8	$79.92			
14	13	Phyllis	Vance	Envelopes	15	$74.85			
15	14	Andy	Bernard	Legal Pads	3	$59.97			
16	15	Stanley	Hudson	Rubber bands	60	$14.94			
17									

Figure 4-9. Aggregating a dynamic array with the spill operator

Although not extensively covered in this chapter, spill operators provide significant benefits for constructing dependent dropdown lists, dynamic charts, and various other functionalities.

With a firm grasp of `UNIQUE()` and spill ranges, the following sections explore additional dynamic array functions.

Filtering records with `FILTER()`

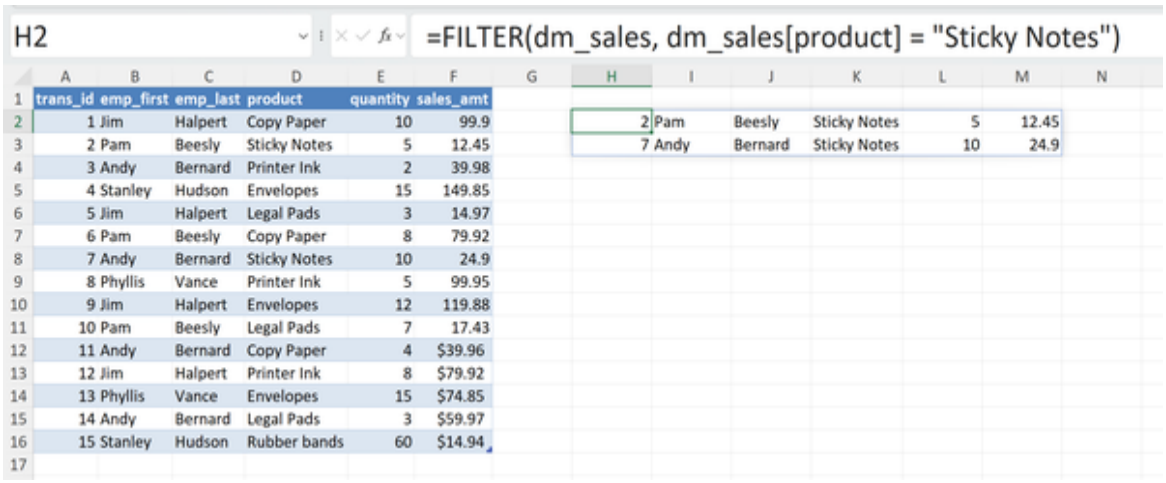
Excel's traditional visual dropdowns for filtering data are intuitive but have limitations. For instance, once they are applied, it's no longer possible to view the raw data in its entirety. It would be better to create a separate copy of the data and then apply filters to that copy, similar how Power Query approaches data cleaning. Additionally, defining complex filter logic rules across multiple columns can be tedious and repetitive.

To address these limitations, Excel introduced the `FILTER()` dynamic array function. `FILTER()` has three parameters, which are explained in detail in Table 4-2:

Table 4-2. The three parameters of `FILTER()`

Parameter	Description
`array`	A required argument specifying the range or array of data that you want to filter.
`include`	A required argument specifying the filtering criteria or condition. It defines the values that should be included in the filtered result. This can be a logical expression, a value to match, or a formula that evaluates to `TRUE` or `FALSE` for each element in the array.
`[if_empty]`	An optional argument specifying the value to return if the filtered result is empty. By default, if no values meet the filtering criteria, the function returns an array with `#N/A` error values.

To achieve the desired result, the `dm_sales` table needs to be filtered so that only the records with `product` set to *Sticky Notes* are returned, such as in Figure 4-10:



trans_id	emp_first	emp_last	product	quantity	sales_amt
1	Jim	Halpert	Copy Paper	10	99.9
2	Pam	Beesly	Sticky Notes	5	12.45
3	Andy	Bernard	Printer Ink	2	39.98
4	Stanley	Hudson	Envelopes	15	149.85
5	Jim	Halpert	Legal Pads	3	14.97
6	Pam	Beesly	Copy Paper	8	79.92
7	Andy	Bernard	Sticky Notes	10	24.9
8	Phyllis	Vance	Printer Ink	5	99.95
9	Jim	Halpert	Envelopes	12	119.88
10	Pam	Beesly	Legal Pads	7	17.43
11	Andy	Bernard	Copy Paper	4	$39.96
12	Jim	Halpert	Printer Ink	8	$79.92
13	Phyllis	Vance	Envelopes	15	$74.85
14	Andy	Bernard	Legal Pads	3	$59.97
15	Stanley	Hudson	Rubber bands	60	$14.94

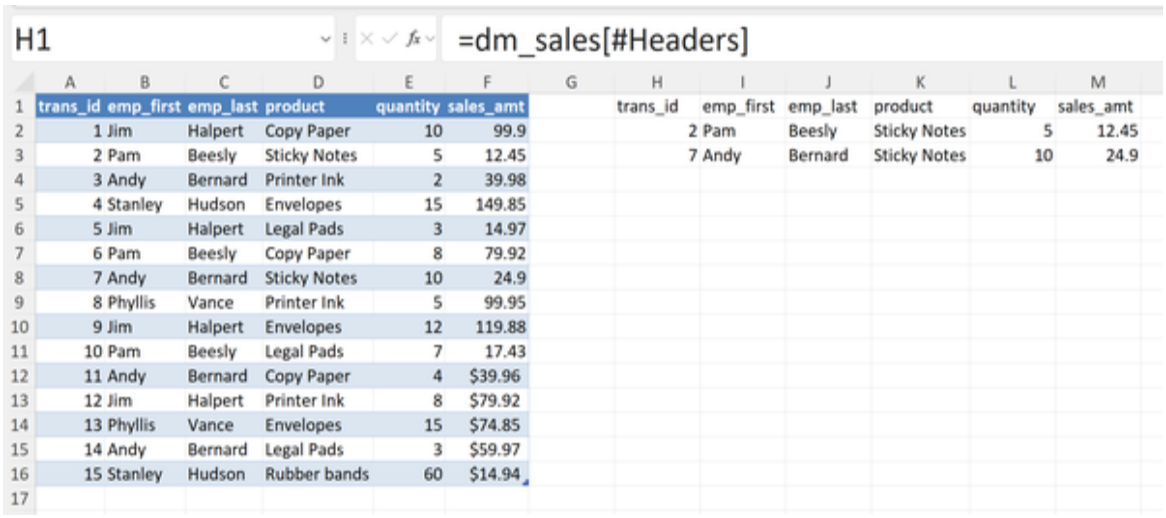
2	Pam	Beesly	Sticky Notes	5	12.45
7	Andy	Bernard	Sticky Notes	10	24.9

Figure 4-10. A basic `FILTER()` function

> **NOTE**
>
> The `FILTER()` function is not case sensitive by default. In the previous example, both “Sticky Notes” and “sticky notes” would yield the same result. To perform a case-sensitive filter, combine the `FILTER()` function with `EXACT()`. Here’s an example: `=FILTER(dm_sales, EXACT(dm_sales[product], "Sticky Notes"))`.

Adding a header column

The `FILTER()` function is already useful, but it lacks one crucial feature — it only returns matching rows, not the data’s header columns. To include these headers, it’s necessary to use a dynamic table header reference. For a quick recap on structured table references, refer to Chapter 1. Include this reference above your filter output for dynamic header labels, such as in Figure 4-11:



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	trans_id	emp_first	emp_last	product	quantity	sales_amt		trans_id	emp_first	emp_last	product	quantity	sales_amt
2	1	Jim	Halpert	Copy Paper	10	99.9		2	Pam	Beesly	Sticky Notes	5	12.45
3	2	Pam	Beesly	Sticky Notes	5	12.45		7	Andy	Bernard	Sticky Notes	10	24.9
4	3	Andy	Bernard	Printer Ink	2	39.98							
5	4	Stanley	Hudson	Envelopes	15	149.85							
6	5	Jim	Halpert	Legal Pads	3	14.97							
7	6	Pam	Beesly	Copy Paper	8	79.92							
8	7	Andy	Bernard	Sticky Notes	10	24.9							
9	8	Phyllis	Vance	Printer Ink	5	99.95							
10	9	Jim	Halpert	Envelopes	12	119.88							
11	10	Pam	Beesly	Legal Pads	7	17.43							
12	11	Andy	Bernard	Copy Paper	4	$39.96							
13	12	Jim	Halpert	Printer Ink	8	$79.92							
14	13	Phyllis	Vance	Envelopes	15	$74.85							
15	14	Andy	Bernard	Legal Pads	3	$59.97							
16	15	Stanley	Hudson	Rubber bands	60	$14.94							
17													

Figure 4-11. `FILTER()` results with header labels

> **WARNING**
>
> Applying the `FILTER()` function or other dynamic array functions to an Excel table does not include the table headers in the results.

Filtering by multiple criteria

What sets the `FILTER()` function apart from the traditional menu-driven approach is its capability to employ formulas for data filtering. This opens up powerful possibilities while maintaining an intuitive and straightforward process for constructing and comprehending the criteria.

Filtering by multiple criteria

To incorporate multiple criteria into the `FILTER()` function, employ the `*` symbol for “and” statements and the `+` symbol for “or” statements.

`AND` criteria

To search for records where both the `product` is *Copy Paper* and the quantity is greater than 5, you can combine the criteria by multiplying them within separate sets of parentheses:

```
=FILTER(dm_sales, (dm_sales[product] = "Copy Paper") *
(dm_sales[quantity] > 5))
```

`OR` criteria

If you'd rather find records that meet *either* of this conditions, replace the `*` symbol with the `+` symbol to create an `OR` statement:

```
=FILTER(dm_sales, (dm_sales[product] = "Copy Paper") +
(dm_sales[quantity] > 5))
```

Nested `AND`/`OR` criteria

To create a filter function with nested `AND` or `OR` statements, group the statements using parentheses. This filter function includes records where the `sales_amt` is at least $100, or both `quantity` is at least 10 and `product` is "Envelopes":

```
=FILTER(dm_sales,
(dm_sales[sales_amt] >= 100) +
((dm_sales[quantity] >= 10) * (dm_sales[product] =
"Envelopes")))
```

You can continue adding and adjusting multiple criteria to `FILTER()` functions by following these guidelines.

Sorting records with `SORT()` and `SORTBY()`

The `SORT()` and `SORTBY()` functions, similar to `FILTER()`, provide a scalable and non-destructive approach to sorting data. The following section explains their functionality and compare them.

Sorting by one criterion with `SORT()`

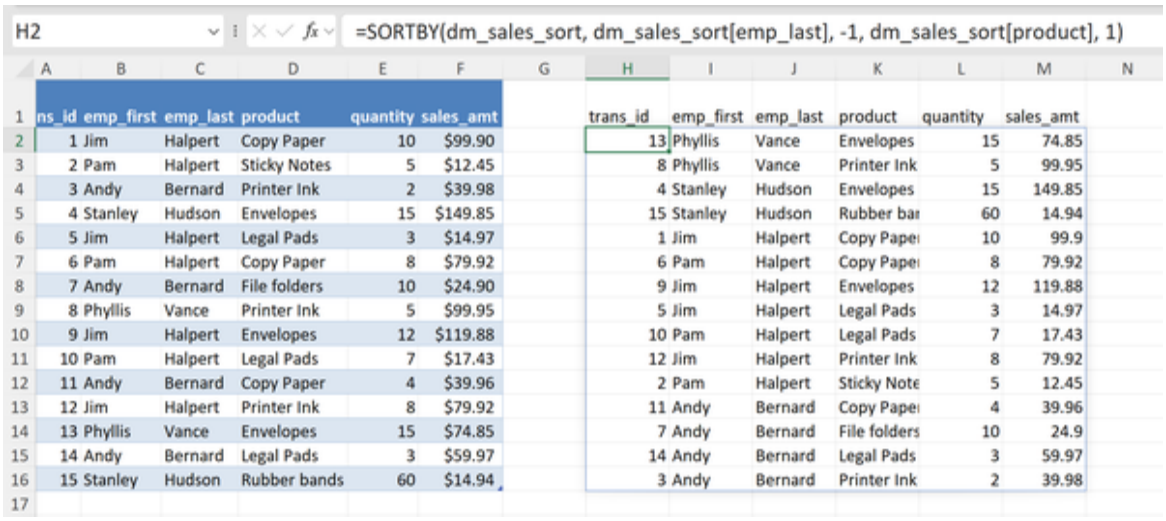
The `SORT()` function arranges a specified range or array of data in ascending or descending order based on one or more sort criteria. Table 4-3

explains the parameters:

Table 4-3. Parameters of the `SORT()` function

Parameter	Description
`array`	A required argument specifying the range or array to be sorted.
`sort_index`	An optional argument specifying the column or columns by which the array should be sorted. It can be specified as a number or a range of numbers.
`sort_order`	An optional argument specifying the sort order for each `sort_index` column. Use `1` for ascending order and `-1` for descending order.
`by_col`	An optional argument specifying whether the sorting should be done by column (`TRUE`) or by row (`FALSE`).

The second argument of `SORT()` determines the column index number to sort by, similar to the column index number in `VLOOKUP()`. To prevent hard-coding the position of the `quantity` column and minimize formula breakage, I'll utilize the `INDEX()` function to dynamically retrieve its column position. Setting the third argument to `-1` will sort `dm_sales` by `quantity` in descending order, as shown in Figure 4-12:



ns_id	emp_first	emp_last	product	quantity	sales_amt
1	Jim	Halpert	Copy Paper	10	$99.90
2	Pam	Halpert	Sticky Notes	5	$12.45
3	Andy	Bernard	Printer Ink	2	$39.98
4	Stanley	Hudson	Envelopes	15	$149.85
5	Jim	Halpert	Legal Pads	3	$14.97
6	Pam	Halpert	Copy Paper	8	$79.92
7	Andy	Bernard	File folders	10	$24.90
8	Phyllis	Vance	Printer Ink	5	$99.95
9	Jim	Halpert	Envelopes	12	$119.88
10	Pam	Halpert	Legal Pads	7	$17.43
11	Andy	Bernard	Copy Paper	4	$39.96
12	Jim	Halpert	Printer Ink	8	$79.92
13	Phyllis	Vance	Envelopes	15	$74.85
14	Andy	Bernard	Legal Pads	3	$59.97
15	Stanley	Hudson	Rubber bands	60	$14.94

trans_id	emp_first	emp_last	product	quantity	sales_amt
13	Phyllis	Vance	Envelopes	15	74.85
8	Phyllis	Vance	Printer Ink	5	99.95
4	Stanley	Hudson	Envelopes	15	149.85
15	Stanley	Hudson	Rubber ba	60	14.94
1	Jim	Halpert	Copy Pape	10	99.9
6	Pam	Halpert	Copy Pape	8	79.92
9	Jim	Halpert	Envelopes	12	119.88
5	Jim	Halpert	Legal Pads	3	14.97
10	Pam	Halpert	Legal Pads	7	17.43
12	Jim	Halpert	Printer Ink	8	79.92
2	Pam	Halpert	Sticky Note	5	12.45
11	Andy	Bernard	Copy Pape	4	39.96
7	Andy	Bernard	File folders	10	24.9
14	Andy	Bernard	Legal Pads	3	59.97
3	Andy	Bernard	Printer Ink	2	39.98

Figure 4-12. Sorting one criteria with SORT()

`SORTBY()` orders an array by another array

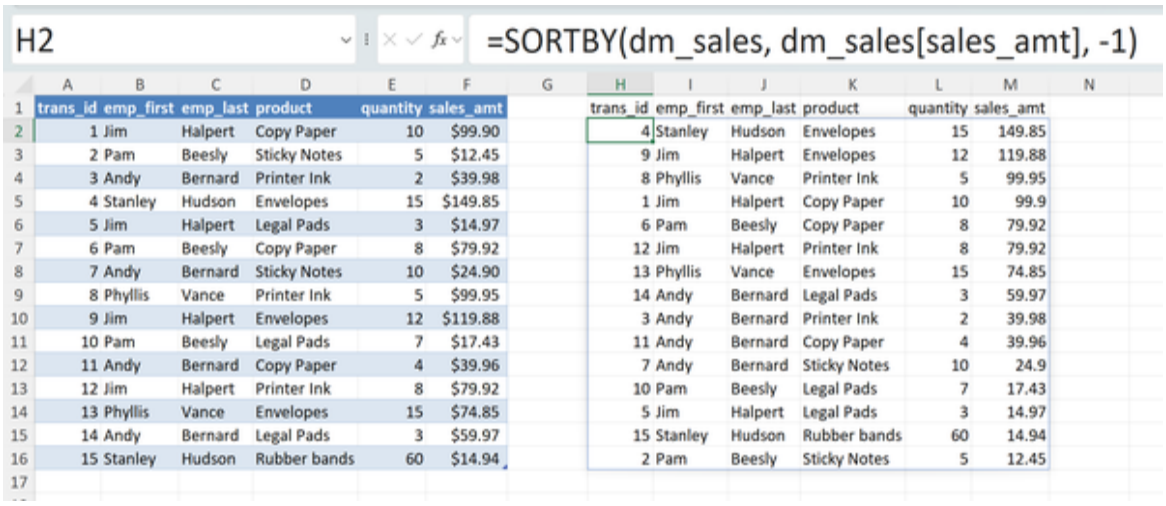
`SORTBY()` is a dynamic and user-friendly function providing an often easier alternative to `SORT()`. Its syntax resembles `SUMIFS()` and allows you to handle multiple criteria simultaneously. Table 4-4 walks through the parameters:

Table 4-4. The parameters of `SORTBY()`

Argument	Description
`array`	A required argument specifying the array or range to sort.
`by_array1`	A required argument specifying the array or range to sort on.
`sort_order1`	An optional argument specifying how to sort the results. 1 sorts ascendingly, -1 sorts descendingly. The default is ascending.
`by_array2`	An optional argument specifying the array or range to sort on.
`sort_order2`	An optional argument specifying how to sort the results of `by_array2`.

Similar to `SUMIFS()`, `SORTBY()` supports a significant number of optional parameter pairs.

For example, the dataset can be sorted in descending order based on `sales_amt` using the `SORTBY()` function, as shown in Figure 4-13:



trans_id	emp_first	emp_last	product	quantity	sales_amt
1	Jim	Halpert	Copy Paper	10	$99.90
2	Pam	Beesly	Sticky Notes	5	$12.45
3	Andy	Bernard	Printer Ink	2	$39.98
4	Stanley	Hudson	Envelopes	15	$149.85
5	Jim	Halpert	Legal Pads	3	$14.97
6	Pam	Beesly	Copy Paper	8	$79.92
7	Andy	Bernard	Sticky Notes	10	$24.90
8	Phyllis	Vance	Printer Ink	5	$99.95
9	Jim	Halpert	Envelopes	12	$119.88
10	Pam	Beesly	Legal Pads	7	$17.43
11	Andy	Bernard	Copy Paper	4	$39.96
12	Jim	Halpert	Printer Ink	8	$79.92
13	Phyllis	Vance	Envelopes	15	$74.85
14	Andy	Bernard	Legal Pads	3	$59.97
15	Stanley	Hudson	Rubber bands	60	$14.94

trans_id	emp_first	emp_last	product	quantity	sales_amt
4	Stanley	Hudson	Envelopes	15	149.85
9	Jim	Halpert	Envelopes	12	119.88
8	Phyllis	Vance	Printer Ink	5	99.95
1	Jim	Halpert	Copy Paper	10	99.9
6	Pam	Beesly	Copy Paper	8	79.92
12	Jim	Halpert	Printer Ink	8	79.92
13	Phyllis	Vance	Envelopes	15	74.85
14	Andy	Bernard	Legal Pads	3	59.97
3	Andy	Bernard	Printer Ink	2	39.98
11	Andy	Bernard	Copy Paper	4	39.96
7	Andy	Bernard	Sticky Notes	10	24.9
10	Pam	Beesly	Legal Pads	7	17.43
5	Jim	Halpert	Legal Pads	3	14.97
15	Stanley	Hudson	Rubber bands	60	14.94
2	Pam	Beesly	Sticky Notes	5	12.45

Figure 4-13. Sorting an Excel table with `SORTBY()`

`SORTBY()` produces the same results as `SORT()` without requiring complex indexing.

Sorting by multiple criteria

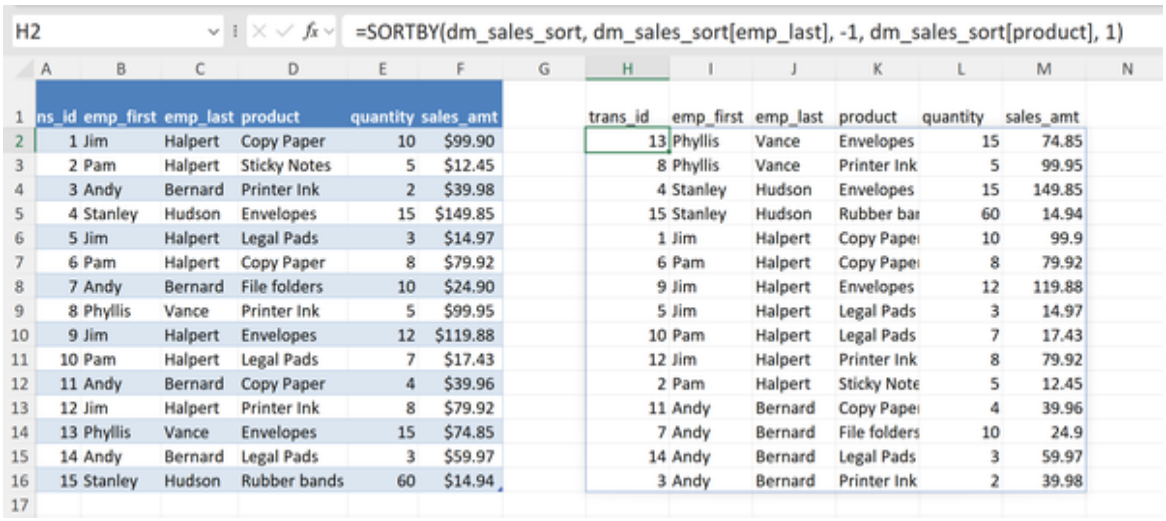
Moreover, `SORTBY()` enables sorting the data based on multiple criteria, with the flexibility to specify an ascending or descending order for each. For example, the data can be sorted in descending order by `emp_last` and in ascending order by `product`:

```
=SORTBY(dm_sales, dm_sales[emp_last], -1,
dm_sales[product], 1)
```

Sorting by another column without printing it

`SORTBY()` can even sort a range by another range, even without including the original sort range. Take this example: the goal is to obtain a list of transaction IDs sorted in descending order by sales.

Instead of using the entire `dm_sales` table as the first argument, only select `trans_id`. The subsequent steps should be familiar. The result will consist of a single column, as shown in Figure 4-14:



ns_id	emp_first	emp_last	product	quantity	sales_amt
1	Jim	Halpert	Copy Paper	10	$99.90
2	Pam	Halpert	Sticky Notes	5	$12.45
3	Andy	Bernard	Printer Ink	2	$39.98
4	Stanley	Hudson	Envelopes	15	$149.85
5	Jim	Halpert	Legal Pads	3	$14.97
6	Pam	Halpert	Copy Paper	8	$79.92
7	Andy	Bernard	File folders	10	$24.90
8	Phyllis	Vance	Printer Ink	5	$99.95
9	Jim	Halpert	Envelopes	12	$119.88
10	Pam	Halpert	Legal Pads	7	$17.43
11	Andy	Bernard	Copy Paper	4	$39.96
12	Jim	Halpert	Printer Ink	8	$79.92
13	Phyllis	Vance	Envelopes	15	$74.85
14	Andy	Bernard	Legal Pads	3	$59.97
15	Stanley	Hudson	Rubber bands	60	$14.94

trans_id	emp_first	emp_last	product	quantity	sales_amt
13	Phyllis	Vance	Envelopes	15	74.85
8	Phyllis	Vance	Printer Ink	5	99.95
4	Stanley	Hudson	Envelopes	15	149.85
15	Stanley	Hudson	Rubber ba	60	14.94
1	Jim	Halpert	Copy Pape	10	99.9
6	Pam	Halpert	Copy Pape	8	79.92
9	Jim	Halpert	Envelopes	12	119.88
5	Jim	Halpert	Legal Pads	3	14.97
10	Pam	Halpert	Legal Pads	7	17.43
12	Jim	Halpert	Printer Ink	8	79.92
2	Pam	Halpert	Sticky Note	5	12.45
11	Andy	Bernard	Copy Pape	4	39.96
7	Andy	Bernard	File folders	10	24.9
14	Andy	Bernard	Legal Pads	3	59.97
3	Andy	Bernard	Printer Ink	2	39.98

Figure 4-14. SORTBY() results in one column

`SORT()` and `SORTBY()` serve similar purposes but with distinct focuses. `SORT()` is designed specifically for sorting a dataset based on a single index number. On the other hand, `SORTBY()` allows sorting by multiple criteria while excluding those criteria columns from the resulting output.

Creating modern lookups with `XLOOKUP()`

Until now, the dataset used for showcasing dynamic array functions has been a flat sales table. However, it is common for data to come from multiple tables, requiring consolidation. The `XLOOKUP()` function offers a more flexible alternative to the traditional `VLOOKUP()` function by leveraging dynamic arrays.

To explore this further, refer to the `xlookup` worksheet in `ch_10.xlsx`. This worksheet contains three distinct tables associated with office supply sales.

`XLOOKUP()` versus `VLOOKUP()`

`XLOOKUP()` offers a familiar experience for users accustomed to `VLOOKUP()` for retrieving data from one table and transferring it to another based on a shared lookup value. However, it introduces a range of additional search methods that are more versatile and sophisticated. For an

overview of the key differences between `VLOOKUP()` and `XLOOKUP()`, refer to Table 4-5:

Table 4-5. `VLOOKUP()` versus `XLOOKUP()`

Feature	`VLOOKUP()`	`XLOOKUP()`
Search direction	Can only search vertically	Can search both vertically and horizontally
Return direction	Can only return values to the right of lookup value	Can return values from columns to the left and right of lookup
Error handling	Returns #N/A if the value is not found	Can specify default value for unmatched items and handle errors

`XLOOKUP()` introduces six parameters in total, as reviewed in Table 4-6:

Table 4-6. Parameters of `XLOOKUP()` explained

Parameter	Description
`lookup_value`	Required argument specifying value to search for in the `lookup_array`.
`lookup_array`	Required argument specifying the range or array to search for the `lookup_value`.
`return_array`	Required argument specifying the range or array from which to retrieve the data.
`if_not_found`	Optional argument specifying the value to return if the lookup_value is not found.
`match_mode`	Optional argument specifying the method for matching the `lookup_value`.
`search_mode`	Optional argument specifying the search behavior for finding the lookup_value.

This demonstration highlights the first four parameters of `XLOOKUP()`. For a more in-depth overview, check out Chapter 12 of Alan Murray's *Advanced Excel Formulas: Unleashing Brilliance with Excel Formulas* (Apress, 2022).

A basic `XLOOKUP()`

Begin with a straightforward example: the `transactions` table includes a `product_id` that requires matching with its corresponding

`product_name`. Here, the `product_id` serves as the lookup array and `product_name` serves as the return array, as shown in Figure 4-15:

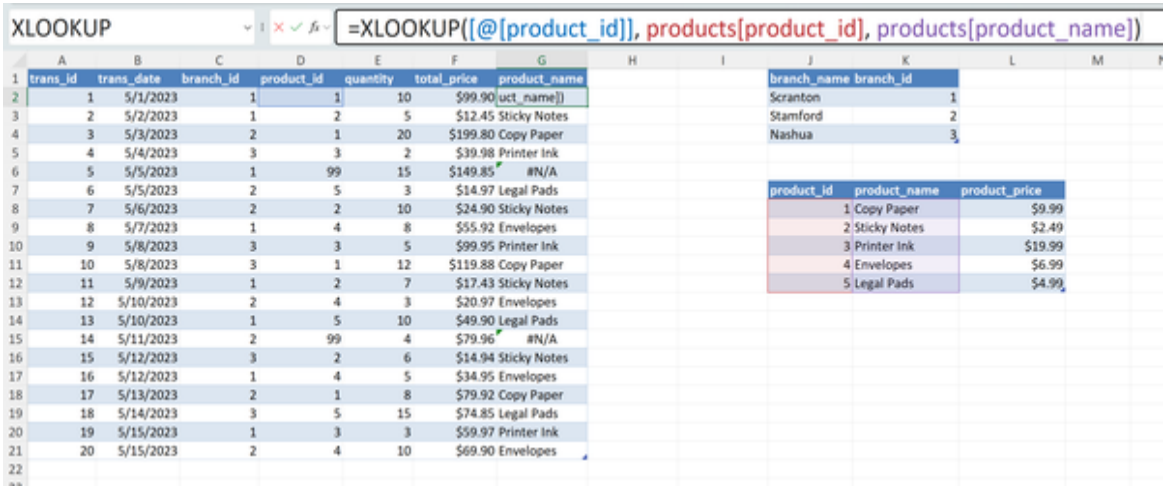


Figure 4-15. A basic `XLOOKUP()`

`XLOOKUP()` and error handling

Lookups for the `product_id` of *99* generate an error. Using `#N/A` as the result for a missing match in `VLOOKUP()` is problematic. It can introduce calculation errors and create confusion for users who may not understand why the `#N/A` is being returned.

To customize the error message in the `XLOOKUP()` statement, specify the fourth optional parameter. In this specific case, you discovered that products assigned to the number *99* should be labeled as `"Other"`. The results are demonstrated in Figure 4-16:

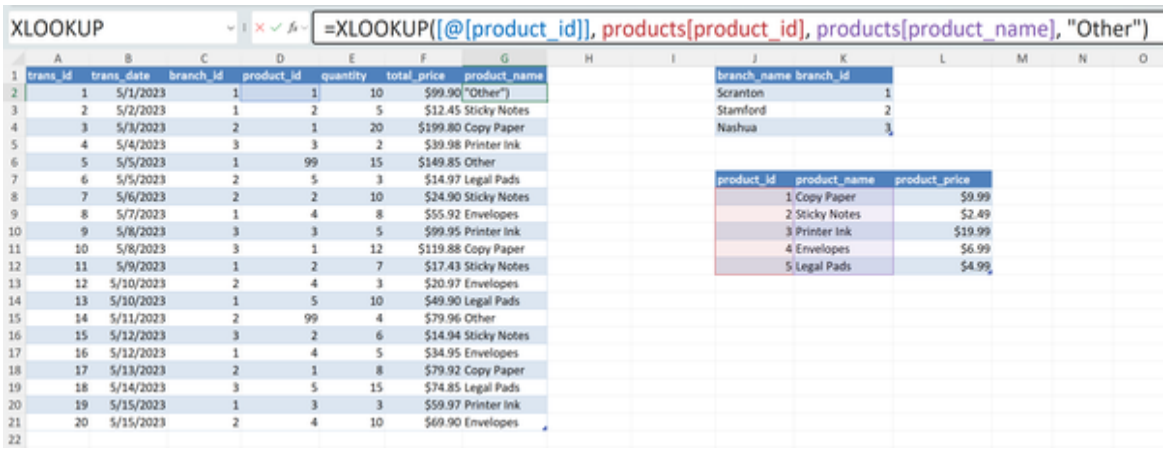


Figure 4-16. `XLOOKUP()` with error handling

After looking up product names into the table, it's time to do the same for branch names.

`XLOOKUP()` and looking up to the left

A common criticism of `VLOOKUP()` is its inability to search to the left of the lookup column, unless helper functions are used. On the other hand, `XLOOKUP()` can search through values in any Excel range, including a table column to the left of the lookup value. An example of this is shown in Figure 4-17:

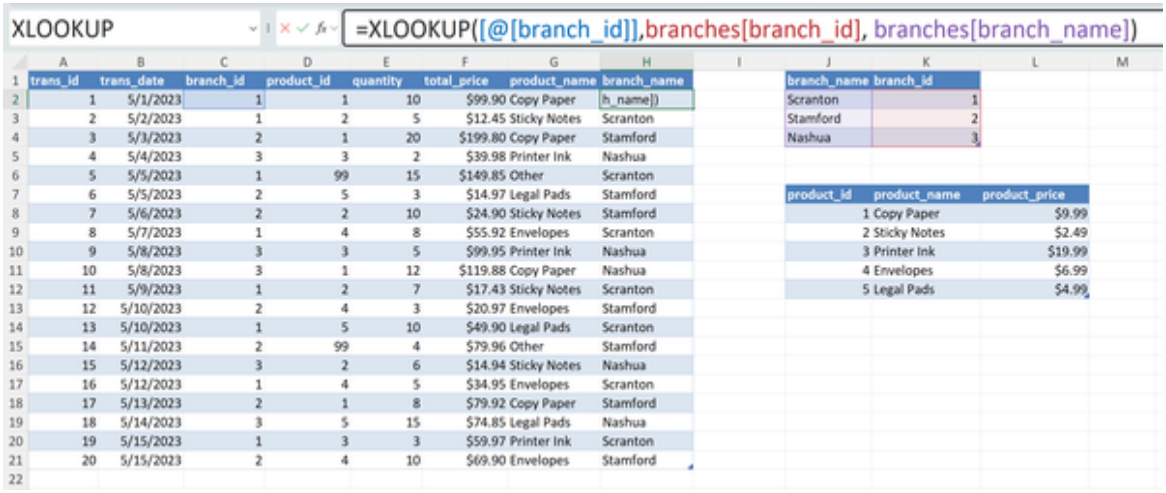


Figure 4-17. `XLOOKUP()` with lefthand lookup

Thanks to its versatility in searching both vertically and horizontally, retrieving values from columns on both sides of the lookup value, and

handling errors within its formula, `XLOOKUP()` has emerged as the preferred formula for data retrieval in Excel.

Other dynamic array functions

The dynamic array functions showcased here were among Excel's initial offerings, with subsequent additions expanding the range of capabilities. Many other dynamic array functions are geared towards text manipulation, including `VSTACK()` for vertical array combination and `TEXTSPLIT()` for splitting text using a specified delimiter. To explore a comprehensive list of dynamic array functions and access tutorials, visit Exceljet.com's article on the topic.

Dynamic arrays and modern Excel

Dynamic array functions may appear to be a step backward in Excel, given the availability of tools like Power Query and Power Pivot. Why willingly return to the days of delicate, formula-driven workbooks when these advanced features exist? This attitude overlooks the value that dynamic arrays bring to the modern Excel analytics stack. Here's why they are an important component.

Simplicity

Dynamic array functions streamline data manipulation and analysis by enabling calculations within a single formula, enhancing comprehension and maintainability. This stands in contrast to the complex and multi-step process involved in constructing and managing data cleaning tasks in Power Query or Data Models in Power Pivot.

Familiarity

Dynamic array functions distinguish themselves from many of the other tools discussed in this book through their integration within the familiar Excel environment. Unlike add-ins that require installation or separate, hard-to-access editors, dynamic array formulas are readily available within

Excel itself. They are easily accessible, making adoption significantly smoother for the typical user.

Real-time updates

Dynamic array functions offer the advantage of automatic result updates whenever the underlying data changes. This eliminates the need for manual formula recalculations or connection refreshes, enabling real-time analysis and insights. This functionality proves especially beneficial in dynamic scenarios where data is continuously evolving, such as real-time dashboards or financial models.

Conclusion

The chapter introduced dynamic array functions, revitalizing traditional, sometimes-clunky Excel references and formulas. These features now hold a vital place alongside Power Query and Power Pivot in the Excel analytics toolkit.

While dynamic array functions offer simplicity and low overhead, the subsequent chapters in Part 3 delve into more advanced tools. These tools require additional setup but offer intricate analytics insights that surpass what can be achieved through formulas alone. Throughout these chapters, you'll learn how to incorporate artificial intelligence, machine learning techniques, and advanced automation features using Python into your Excel workflow.

Exercises

To practice dynamic array functions, locate the `fueleconomy.xlsx` workbook in the `datasets` folder of the book's companion repository. This workbook includes two datasets: `vehicles` and `common`. Complete the following exercises:

1. Find the distinct and truly unique values in the `make` column of the `vehicles` dataset. How many are there of each?

2. Display only the vehicles with city mileage greater than 30.
3. Display only the vehicles where either the city mileage is greater than 30, or where both cylinders are less than 6 and fuel is Regular.
4. Sort the `vehicles` dataset in descending order based on the highway mileage.
5. Sort just the `model` column of the `common` dataset based on the `years` column, descending.
6. Add the `years` column from the `common` dataset to the `vehicles` dataset. Return `Not reported` if a match is not found.

The solutions can be found in the `ch_10_solutions.xlsx` file, located in the `exercise_solutions` folder of the book's companion repository.

Chapter 5. Augmented Analytics and the Future of Excel

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at shunter@oreilly.com.

As the world of data analytics grows vaster and more complex, what role will Excel serve? Will it become obsolete in an artificial intelligence (AI)-powered data ecosystem? This chapter explores the emergence of augmented analytics and the role Excel will play in this transformation, along with some current use cases.

The growing complexity of data and analytics

In 2017, the International Data Corporation (IDC), a market intelligence firm, predicted a remarkable tenfold increase in the amount of data in existence from 2016 to 2025, totaling 163 zettabytes or a trillion gigabytes.

With the overall quantity of data on the rise, it's also expanding in variety. According to AI services provider Taiger, 80% of digital data was unstructured by 2020 — a figure that has likely escalated with the emergence of generative natural language processing products like ChatGPT. In addition, real-time data has gained significant importance,

with IDC estimating that streaming data will comprise 30% of all data by 2025.

The explosion of data, characterized by research advisory Gartner as volume, velocity, and variety, has necessitated the use of advanced analytics methods. Data science has helped organizations uncover relationships and insights in the data using a variety of computational and statistical methods, while machine learning and artificial intelligence (AI) enable computers to learn and simulate human intelligence. These techniques have allow businesses to automate decision-making processes, identify trends in real-time, and create personalized experiences.

This revolution is here to stay: 94% of business leaders responded to business advisory Deloitte that AI is critical to success over the next five years, and the Bureau of Labor Statistics has projected a 36% increase in the number of data scientists employed over a decade, from 113,000 in 2021.

Excel and the legacy of self-service BI

Self-service business intelligence (BI), enabled by tools like Excel, has revolutionized decision-making for businesses. It allows individual users to access and analyze data independently, without relying on IT. However, self-service BI is limited in scope and sophistication. Data must be structured to work with Excel, restricting analysis to descriptive and diagnostic analytics, and lacking the ability to perform advanced algorithms and machine learning models for predictive or prescriptive analytics.

To make more strategic decisions, businesses need to complement self-service BI with more advanced analytics tools and techniques like data mining, machine learning, and artificial intelligence. Combining these tools provides a more comprehensive view of data, enabling informed decisions in today’s rapidly evolving business environment.

Excel for augmented analytics

So how does this play out for the future of Excel? With augmented analytics, it can serve as a bridge between self-service BI and data science/AI, enabling organizations to make better use of their data and stay competitive in a rapidly evolving business landscape.

In the remainder of this chapter, you'll get a hands-on look at some of the use cases for augmented analytics in Excel as they exist today. First, you'll learn how to make the most of Analyze Data for AI-powered insights. Then, you'll build a basic predictive model using XLMiner. Finally, you'll use optical character recognition and Azure Machine Learning to perform sentiment analysis. These examples aim to expand your perception of Excel's capabilities and solidify its promising future in the realm of converged analytics.

Using Analyze Data for AI-powered insights

The Analyze Data feature in Excel is an augmented analytics product that uses artificial intelligence (AI) to derive meaningful insights more efficiently. That said, AI is not a complete substitute for genuine expertise. To fully leverage Excel's potential with AI, properly structured data is essential.

The starter file for this demo can be found in the `ch_11` folder of the book's resources as `analyze-data-start.xlsx`. We will use the Wholesale Customers dataset from the UC Irvine Machine Learning repository.

Analyze Data is ready to use in Excel with no downloads required. Just place your mouse anywhere in the `wholesale_customers` table and head to Home > Analyze Data to get started. Instantly, you'll be presented with a range of intriguing AI-generated insights. Click on any of them to have them inserted into your workbook.

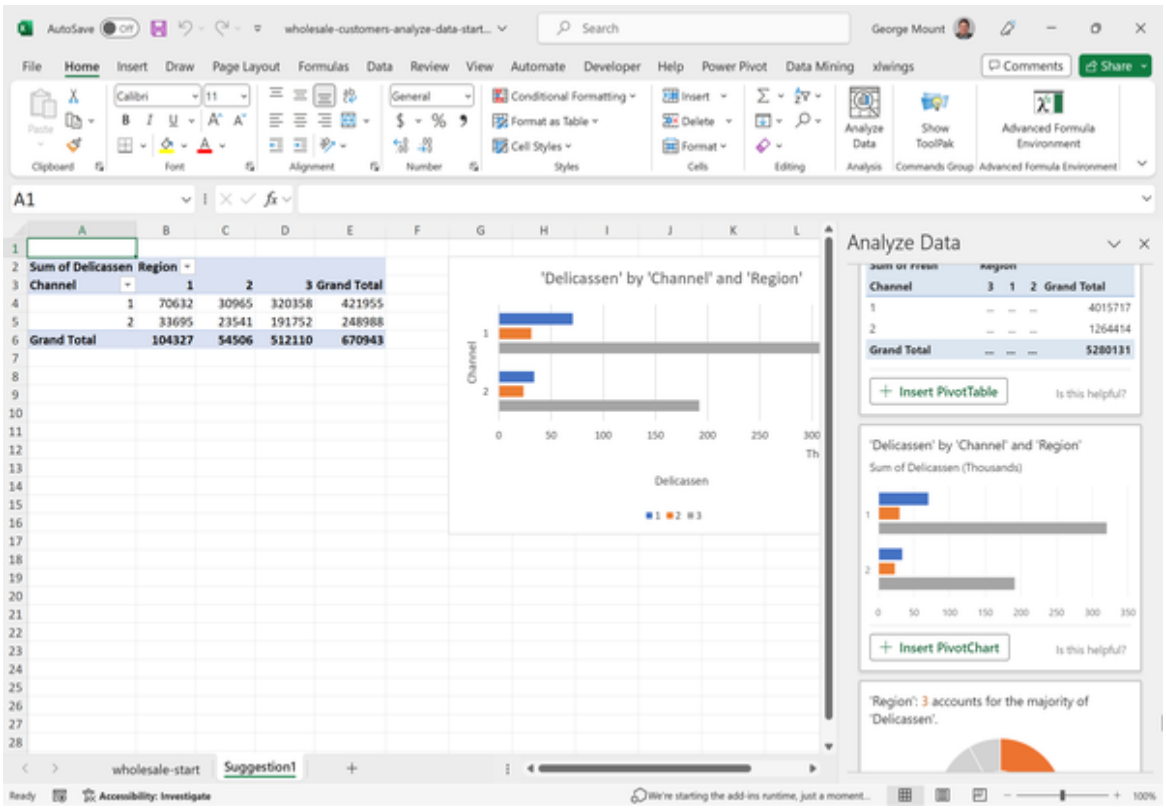


Figure 5-1. Inserting an Analyze Data insight

The power of Analyze Data becomes more apparent with its natural language querying. For instance, imagine you're in a meeting with colleagues and need to swiftly retrieve the total sales for the Grocery department. Instead of spending time manually calculating the answer, you can directly pose the question to Analyze Data and obtain the desired information right away:

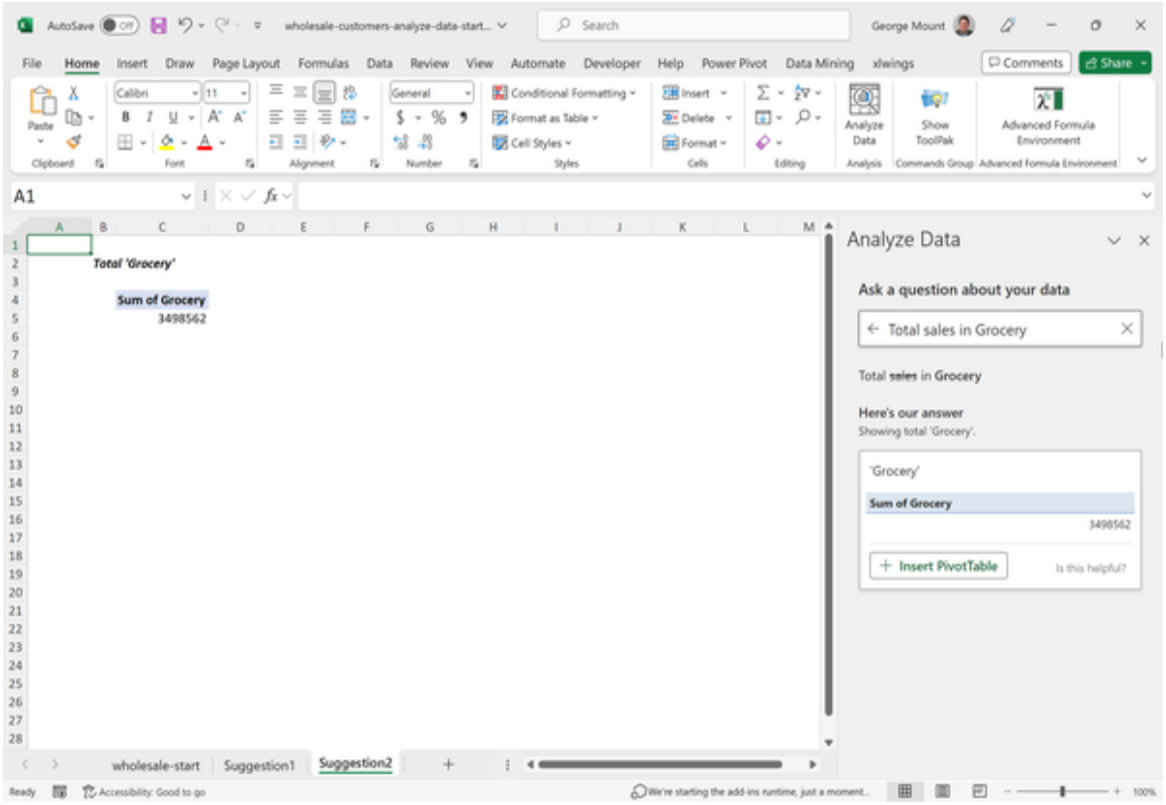


Figure 5-2. Natural language querying in Analyze Data

While querying this dataset is undoubtedly impressive, it does have certain limitations, primarily related to the data's layout. For instance, if you attempt to ask Analyze Data for the total sales by region, you will instead receive the sum of all region numbers:

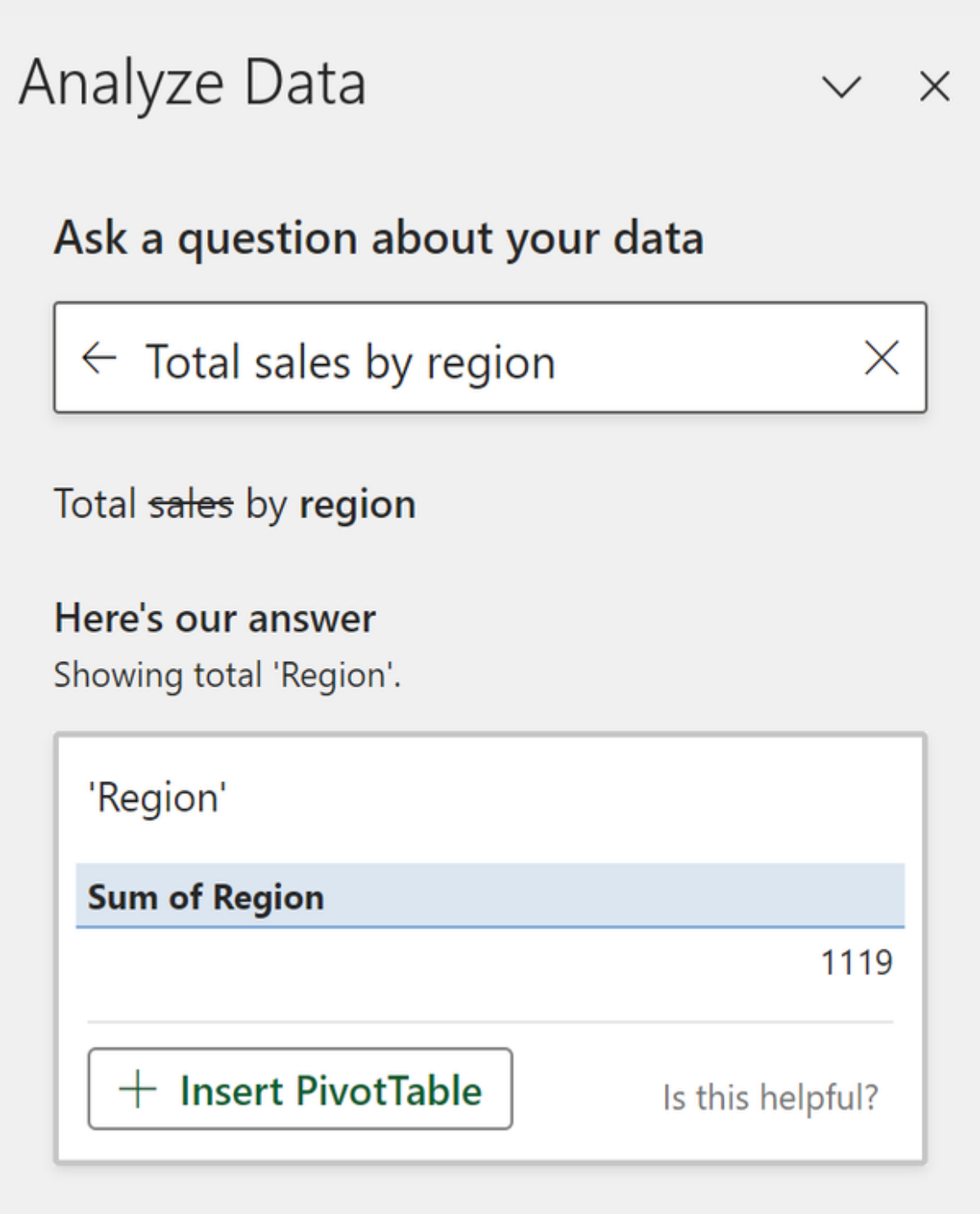
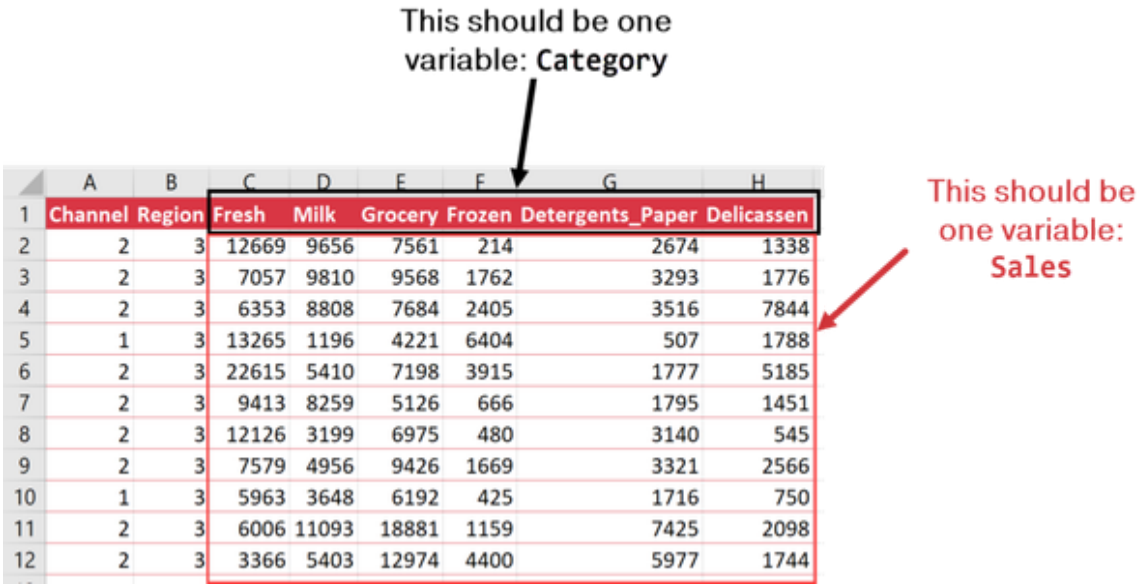


Figure 5-3. Analyze Data struggles with improperly constructed data

Analyze Data isn't sure what to do because the data is presented in an improper format. Instead of consolidating all sales figures in one column, they are spread across multiple columns. As a result, Analyze Data cannot

determine which columns contain the relevant sales figures that need to be summed up. The gist of this formatting error is illustrated in Figure 5-4:



	A	B	C	D	E	F	G	H
1	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
2	2	3	12669	9656	7561	214	2674	1338
3	2	3	7057	9810	9568	1762	3293	1776
4	2	3	6353	8808	7684	2405	3516	7844
5	1	3	13265	1196	4221	6404	507	1788
6	2	3	22615	5410	7198	3915	1777	5185
7	2	3	9413	8259	5126	666	1795	1451
8	2	3	12126	3199	6975	480	3140	545
9	2	3	7579	4956	9426	1669	3321	2566
10	1	3	5963	3648	6192	425	1716	750
11	2	3	6006	11093	18881	1159	7425	2098
12	2	3	3366	5403	12974	4400	5977	1744

Figure 5-4. How to "tidy" this dataset for better insights

Storing data in an unclean or "untidy" format is a significant obstacle in analytics. You may have encountered this issue in your own work without putting your finger on exactly what was wrong. Developing a conceptual understanding of dirty data allows you to identify issues early on in your project, saving significant amounts of time later on. To delve deeper into the theory of "tidy data" and learn how to handle it effectively, refer back to Chapter 1.

For AI to unleash its full potential in uncovering insights, data must be in a machine-readable, "tidy" format where each variable resides in a single column. To address this, we will use Power Query to unpivot the columns from `Fresh` to `Delicassen` and rename them `Department` and `Sales`. Make sure to load the results of your query into an Excel table. For a refresher on how to unpivot and load this dataset in Power Query, refer back to Chapter 4.

With the data in a "tidy" format, querying the data to find total sales by region is a breeze:

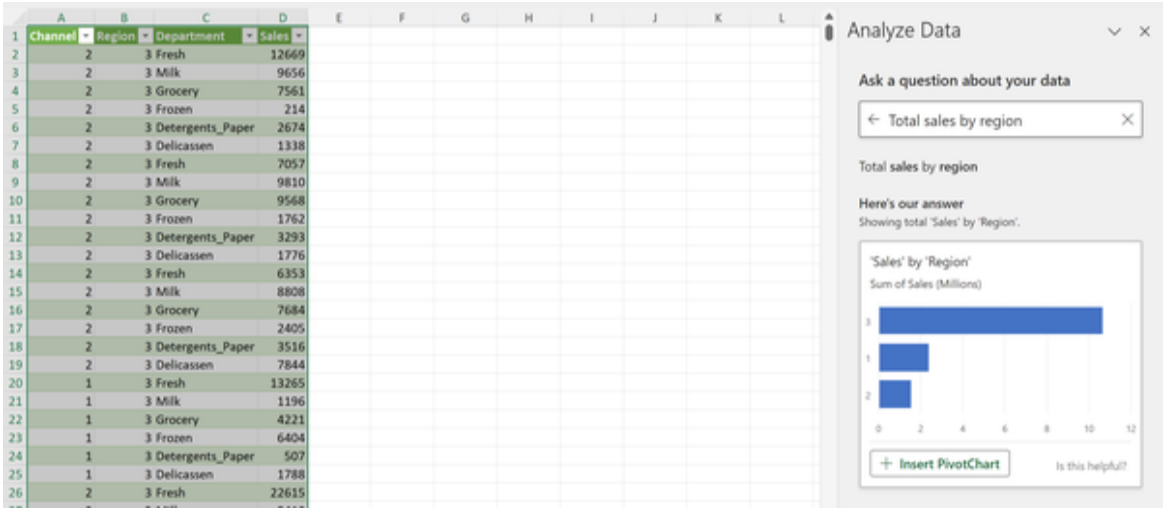


Figure 5-5. Finding total sales by region in Analyze Data

What other insights can you glean with the power of Analyze Data? To access the cleaned dataset, download the `analyze-data-solutions.xlsx` file from the `solutions` subfolder in `ch_11` of the book's resources.

Analyze Data is a powerful augmented analytics tool that utilizes AI to efficiently derive valuable insights. However, it is crucial to have properly structured data for optimal results. By understanding the concept of tidy data and resolving formatting issues, users can fully leverage the potential of AI in uncovering meaningful insights.

Building statistical models with XLMiner

The XLMiner add-in in Excel contributes to augmented analytics by providing essential tools for data analysis and modeling. It enables users to leverage advanced analytical capabilities within the familiar Excel environment, enhancing the overall experience of augmented analytics. The starter file for this demo can be found in the `ch_11` folder of the book's resources as `xlminer-start.xlsx`.

XLMiner is a free add-in for statistical analysis that works on most versions of Excel, including the web. To get started, head to the ribbon then Insert > Add-ins > Get Add-ins. From the Office Add-ins menu, search for XLMiner and click Add:

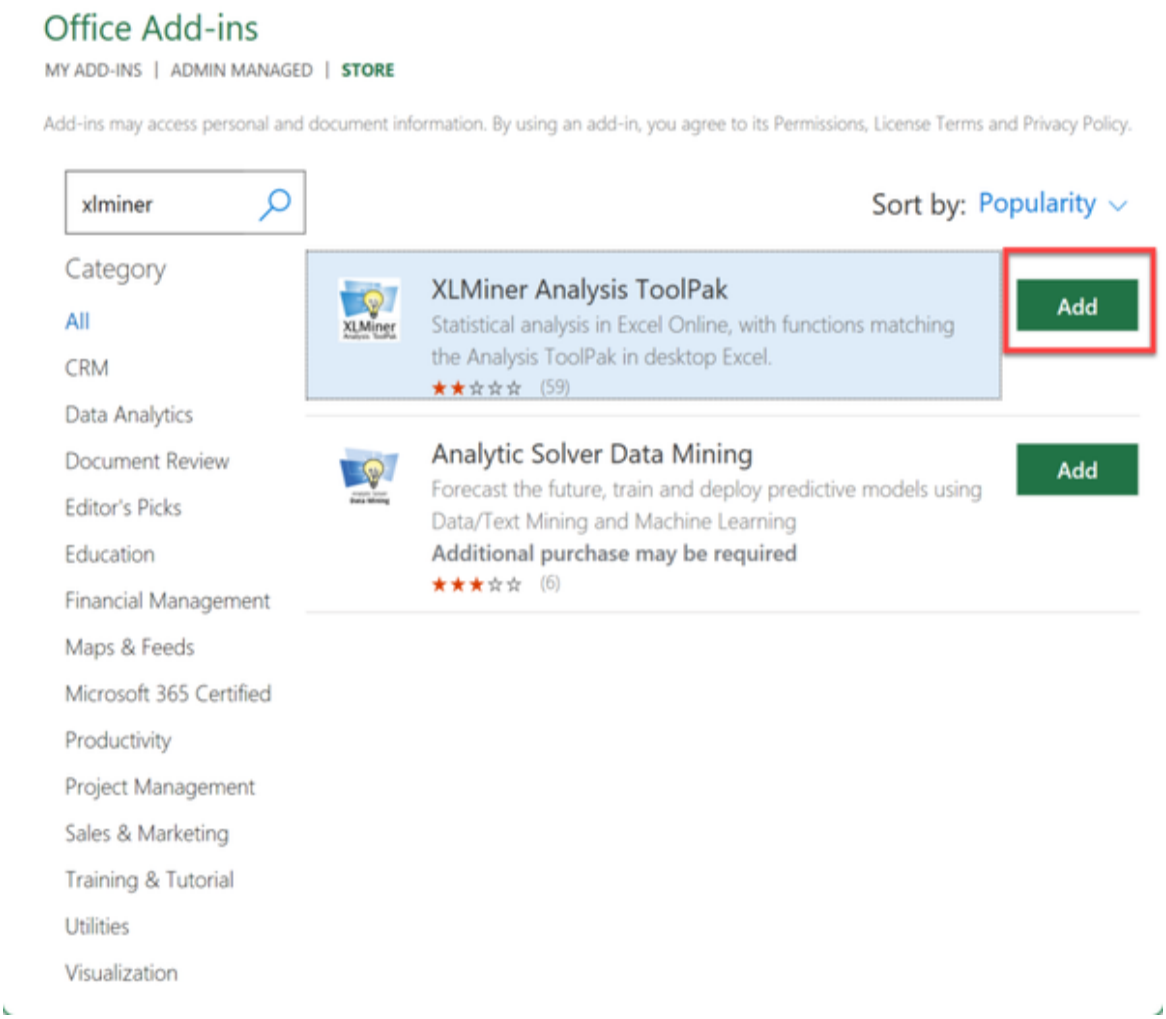


Figure 5-6. Getting the XLMiner Add-in

Agree to the Terms & Conditions, click OK and you should see an XLMiner menu on the sidebar of your screen. As you're seeing, XLMiner comes with plenty of statistical tools and techniques. Let's focus on the "mother of all models," linear regression.

We will use *price* as the dependent variable and *lotsize*, *airco* and *prefarea* as the independent variables. Head to the Linear Regression section of the XLMiner menu, fill it out like so, then click OK:

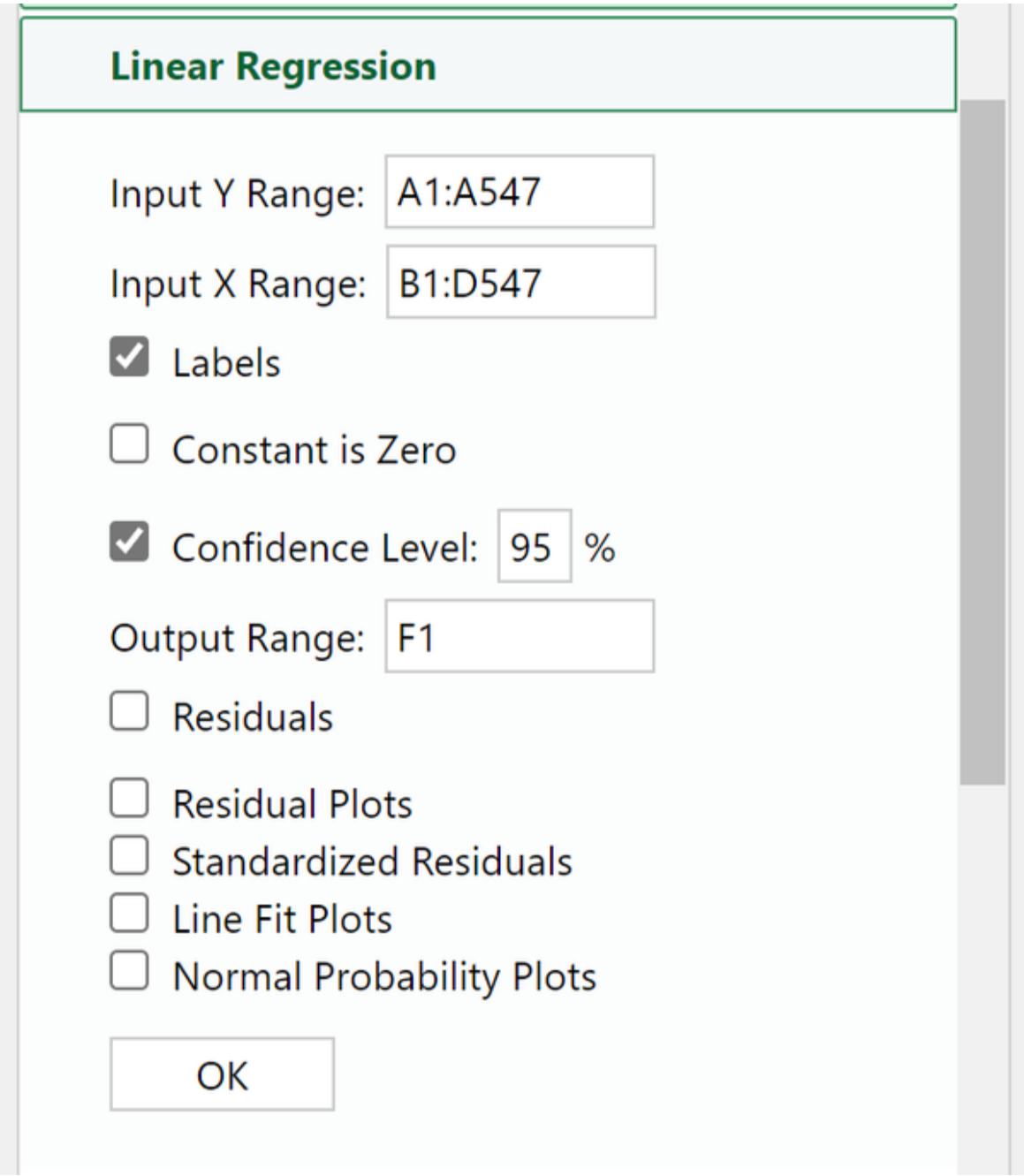


Figure 5-7. Setting up a Linear Regression in XLMiner

Unfortunately, it can be difficult to use the drag-and-drop feature in XLMiner to name an input range, so you may need to physically type in the cell locations.

It can be tempting to jump right into building models and making predictions as we did, but in practice it's necessary to explore the data and check whether it meets the assumptions of whatever model you're using.

To be fair, Python and R provide much more robust environments for these regression checks, but if you'd like that hands-on approach that only Excel can provide, check out my book *Advancing into Analytics: From Excel to Python and R*. XLMiner can also provide additional outputs to help in checking these assumptions.

You should see the output in Figure 5-8 from XLMiner after running the regression:

	E	F	G	H	I	J	K	L	M	N	O
6		Adjusted R Square	0.436467707								
7		Standard Error	20045.37142								
8		Observations	546								
9											
10		ANOVA									
11			df	SS	MS	F	Significance F				
12		Regression	3	1.70818E+11	56939339259	141.7046846	0				
13		Residual	542	2.17785E+11	401816915.2						
14		Total	545	3.88603E+11							
15											
16			Coefficients	Standard Error	t Stat	P-value	Lower 95%	Upper 95%	Lower 95%	Upper 95%	
17		Intercept	32770.49675	2216.879895	14.78226079	8.65278E-42	28415.76783	37125.22568	28415.76783	37125.22568	
18		lotsize	5.116814473	0.415952537	12.3014383	7.6259E-31	4.299737937	5.933891009	4.299737937	5.933891009	
19		airco	19437.2651	1895.231911	10.25587686	1.12388E-22	15714.36553	23160.16468	15714.36553	23160.16468	
20		prefarea	12112.04184	2087.892931	5.80108379	1.12101E-08	8010.688509	16213.39516	8010.688509	16213.39516	
21											

Figure 5-8. Results of linear regression in XLMiner

Here you have typical regression diagnostics such as coefficient p-values, R-square and more. If you'd like to learn more about interpreting these, check out *Advancing into Analytics*. To access the solutions for this demo, download the `xlminer-solutions.xlsx` file from the `solutions` subfolder in `ch_11` of the book's resources.

XLMiner is a moderately user-friendly tool for statistical modeling in Excel, but lacks the advanced capabilities of R or Python. It enhances Excel's analytical capabilities and integrates seamlessly within the environment. However, it falls short of being a comprehensive augmented analytics tool due to limited AI integration and narrower functionalities.

The upcoming examples will demonstrate how Excel leverages external artificial intelligence features, making them more robust use cases of augmented analytics in Excel.

Reading data from camera

Analysts frequently encounter situations where they must analyze data that exists solely in printouts. To avoid the slow and error-prone manual process, Excel offers a feature to directly convert text from images into a workbook.

Converting scanned paper documents into editable computer text files, known as optical character recognition (OCR), is not a novel concept. OCR technology has been in existence since the 1970s and has undergone significant advancements since then. Today, it is widely available in various programs, including Excel.

For this demonstration, we have customer reviews that exist only as printed copies. Our goal is to import them into Excel for sentiment analysis. You can find the file named `scanned_reviews.png` in the `ch_11` folder of the book's resources.

To get started, open a new Excel workbook and from the ribbon select Data > Get & Transform Data > From Picture > Picture from File. From here, you can navigate to and select the `scanned_reviews.png` exercise file. Import the file and you should see a Data from Picture menu to the right of your workbook:

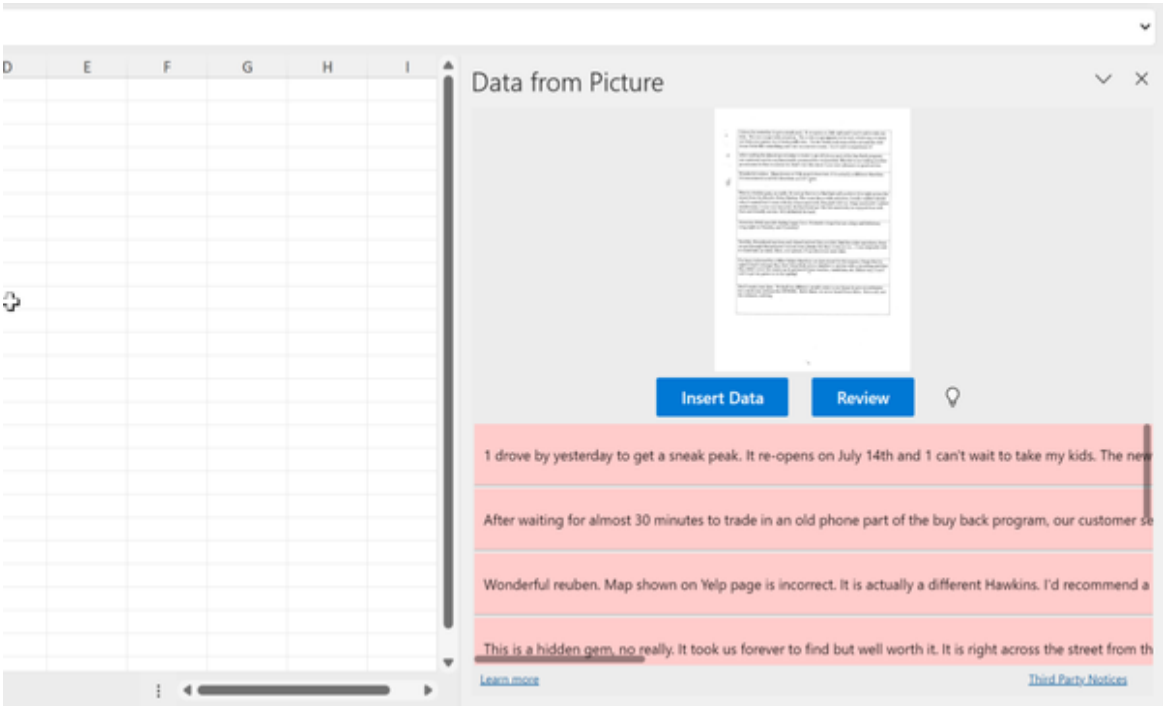


Figure 5-9. Data from Picture Warnings

Excel’s OCR converts images to text—seemingly magical, but it does make mistakes. Leveraging AI features, Excel can also predict where these mistakes might occur.

In this case, Excel has flagged all but one record as likely containing an error. You can click Review to scan and double-check each of them, then make any adjustments to the data. For example, the first entry starts with the letter 1 when it should be the pronoun I:

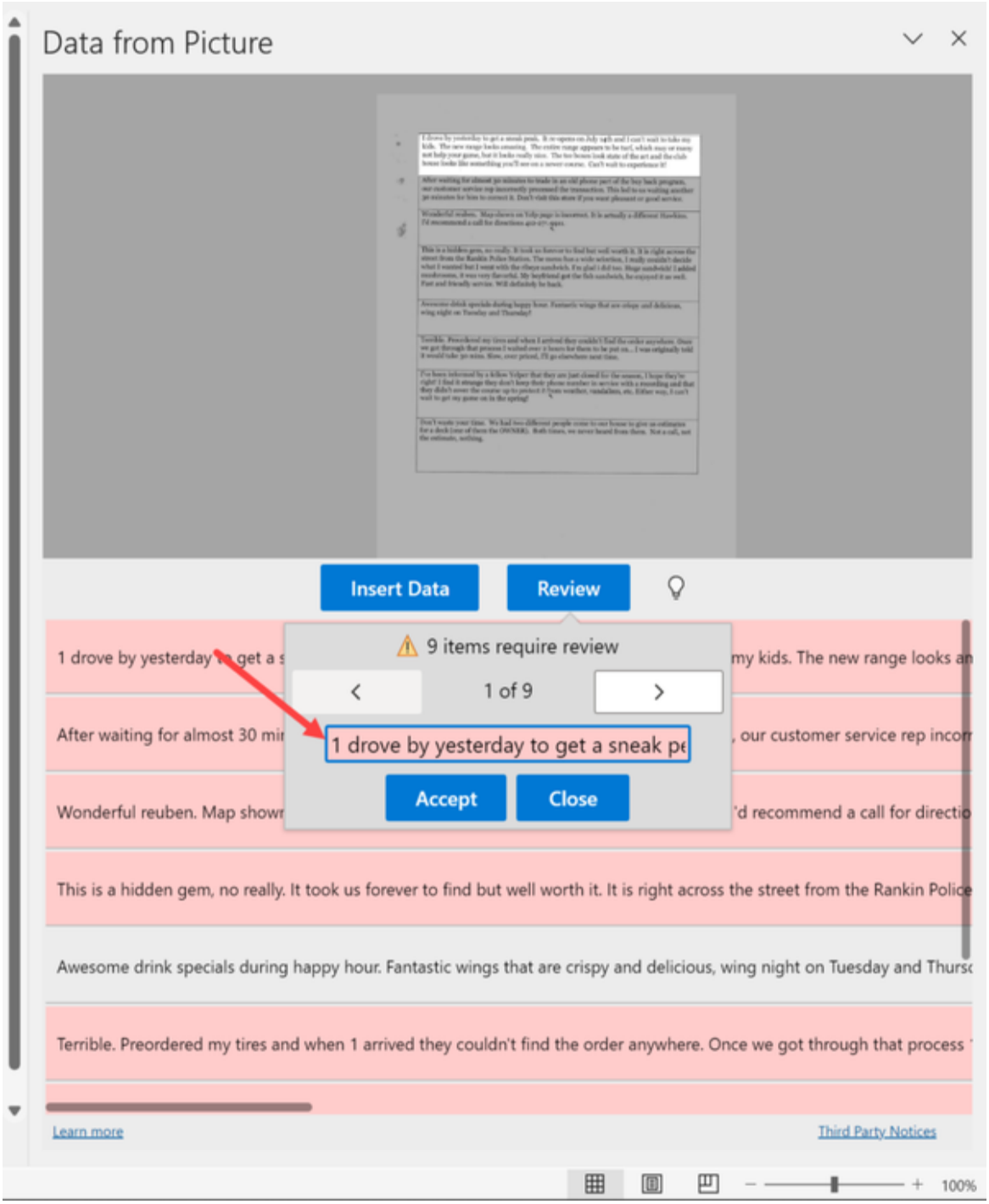


Figure 5-10. AI-detected OCR transcription error

After reviewing and identifying any potential erroneous entries, click “Insert Data” to transfer the results to Excel.

Excel’s AI does a decent job predicting when text is likely to contain an error, but it’s not perfect. For example, it could detect an error in one entry

when none exist — known as a false positive in statistics. On the flipside, it could approve an entry that actually does have errors — a false negative.

The balance between flagging potential false positives and false negatives is one of the biggest topics in statistics and machine learning. For now, we'll trust Excel to get it right, but as you continue in your analytics journey there may be places where it's best for you to decide yourself.

Inserting unstructured data like text into Excel may pose certain challenges, as Excel is not inherently designed for such data. To maintain organization, it is recommended to allocate a separate cell for each review and manually adjust accordingly. For instance, rows 6 and 7 can be combined to form a single review.

Traditionally, OCR has primarily been used for text data rather than Excel's core functionalities. However, with the emergence of tools like sentiment analysis, this landscape is evolving. While Excel has long incorporated OCR technology, its convenience is unparalleled, especially when working with financial statements or similar numerical applications that exist solely as printed records or smartphone photos.

Our findings from this demonstration will serve as a foundation for the next one.

Sentiment analysis with Azure Machine Learning

While Excel has traditionally been regarded as a tool suitable for working with small, structured datasets, the introduction of features related to artificial intelligence and machine learning (AI/ML) has blurred these conventional limitations. This highlights the immense potential of augmented analytics within Excel. A prime example of this is the ability to employ Excel for sentiment analysis, enabling the assessment of sentiment in a collection of text reviews.

Continuing with the series of reviews imported from the image in the previous section, our objective now is to classify the sentiment of each review as positive, negative, or neutral.

Sentiment analysis is a data analysis tool that uses machine learning algorithms to decipher emotions and opinions in unstructured data. It often categorizes text as positive, negative, or neutral, enabling businesses to improve customer satisfaction and address concerns based on overall sentiment towards a brand, product, or service.

Manually assessing a few reviews isn't a problem, but it becomes challenging with thousands or more. To automate this task, we will leverage Azure's text analytics features.

The first step is to load the Azure Machine Learning add-in to your workbook. From the ribbon, head to Insert > Add-ins > Get Add-ins, then search for Azure Machine Learning. Click Add, then Continue. You should now see the Azure Machine Learning add-in appear to the right of your Excel session. Click the second option in this menu, Text Sentiment Analysis (Excel Add-in Solver), to continue.

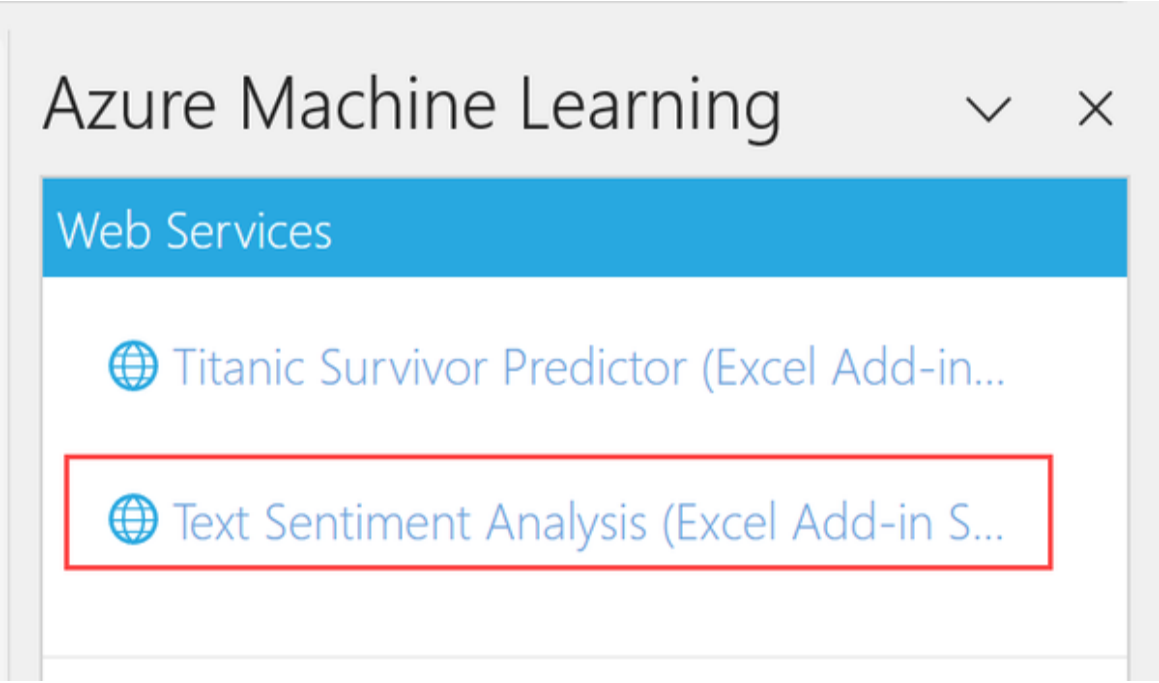


Figure 5-11. Azure sentiment analysis selection

Azure requires the input data for sentiment analysis to adhere to a specific format or schema. Here is just example where structuring data in a machine-friendly format is crucial for AI to work effectively

For this particular task, we need to create three column headers in the workbook: *tweet_text*, *Sentiment*, and *Score*. You can find these under the View Schema section of the sentiment analysis menu.

The first column header, *tweet_text*, is where we input the restaurant reviews. Despite its name, this column header can handle full reviews, not just tweets. Azure's sentiment analysis add-in will still work effectively with this column. The *Sentiment* and *Score* columns will be populated by Azure's sentiment analysis add-in:

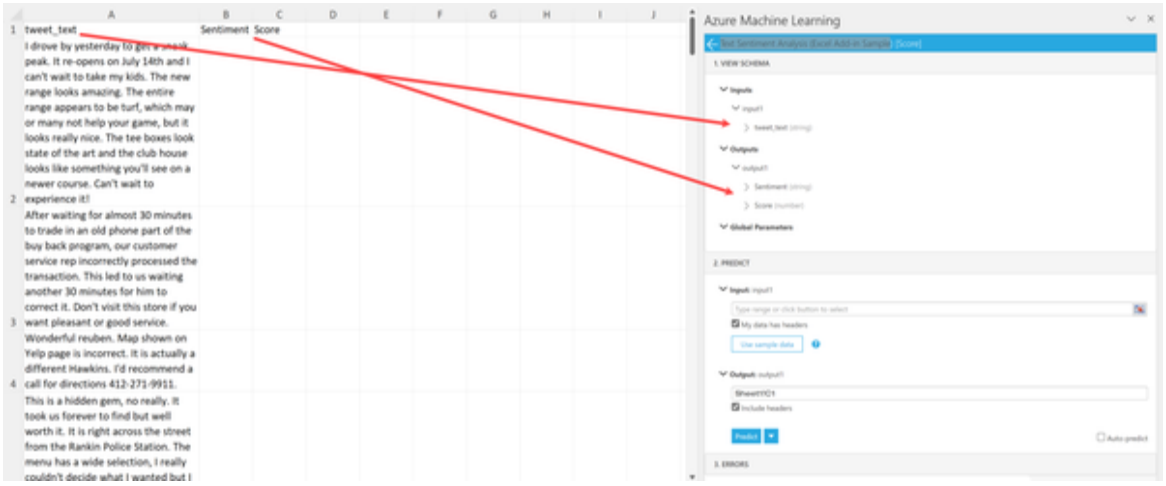


Figure 5-12. Creating the schema for sentiment analysis

To configure the input for sentiment analysis, head to the "Predict" section of the add-in and define the input area. This should cover cells `A1:A9`, including the header. Make sure to check on the "My data has headers" option.

For the output area, the results of the sentiment analysis will be displayed starting from cell `B1`. Set this cell to confirm that the results will be populated in that location.

After clicking on the Predict button, you will see columns `B` and `C` populated with results. Unfortunately, this process can be glitchy at times. If you encounter any issues, double-check the schema and inputs or try restarting Excel.

As anticipated, Azure has successfully classified each review as either positive, negative, or neutral, and the results are recorded in the *Sentiment*

column. The *Score* column contains a numerical value ranging from 0 to 1, representing the sentiment score generated by Azure. A higher score indicates a more positive sentiment. These scores are subsequently categorized into negative, neutral, and positive groups.

If you're wondering how Azure generates these scores, it's through a complex machine learning model that only Azure can explain. Automated tools like this are convenient, but they often lack transparency and explainability. Although undeniably powerful, these tools are not infallible. For instance, in the sentiment analysis results seen in Figure 5-13, rows 3 and 5 were labeled as neutral and negative, respectively, despite being negative and positive reviews upon reading.

	A	B	C
1	tweet_text	Sentiment	Score
2	I drove by yesterday to get a sneak peak. It re-opens on July 14th and I can't wait to take my kids. The new range looks amazing. The entire range appears to be turf, which may or many not help your game, but it looks really nice. The tee boxes look state of the art and the club house looks like something you'll see on a newer course. Can't wait to experience it!	positive	9.05E-01
3	After waiting for almost 30 minutes to trade in an old phone part of the buy back program, our customer service rep incorrectly processed the transaction. This led to us waiting another 30 minutes for him to correct it. Don't visit this store if you want pleasant or good service.	neutral	0.57314
4	Wonderful reuben. Map shown on Yelp page is incorrect. It is actually a different Hawkins. I'd recommend a call for directions 412-271-9911.	positive	8.55E-01
5	This is a hidden gem, no really. It took us forever to find but well worth it. It is right across the street from the Rankin Police Station. The menu has a wide selection, I really couldn't decide what I wanted but I went with the ribeye sandwich. I'm glad i did too. Huge sandwich! I added mushrooms, it was very flavorful. My boyfriend got the fish sandwich, he enjoyed it as well. Fast and friendly service. Will definitely be back.	negative	2.97E-02
6	Awesome drink specials during happy hour. Fantastic wings that are crispy and delicious, wing night on Tuesday and Thursday!	positive	0.953895

Figure 5-13. Mislabeled reviews from sentiment analysis

The moral of the story is to leverage the full potential of AI, but also to exercise critical thinking and not rely solely on it. While AI possesses artificial intelligence, you possess the power of genuine human judgment and intuition. By combining the strengths of both, you can make more informed decisions and harness the true potential of AI technology. The solutions for this sentiment analysis can be found in the `sentiment-`

`analysis-solutions.xlsx` file from the `solutions` subfolder in `ch_11` of the book's resources.

Unstructured data is widely recognized as challenging to work with, but AI is well-suited to handle such data. Despite Excel's primary focus on structured data, there is a growing trend towards utilizing it for unstructured data, including text and images.

Sentiment analysis is only the beginning. The upcoming integration of GPT-powered language modeling in Excel, along with tools like Copilot, represents a significant leap forward in this regard. This integration promises to enhance Excel's capabilities and enable users to leverage powerful language modeling capabilities within their Excel workflows

Converged Analytics and the Future of Excel

In conclusion, predictive analytics and AI are powerful tools that can help you gain deeper insights into your data and predict future outcomes. Excel has evolved to incorporate these tools, allowing users to leverage AI and predictive analytics for better decision-making and forecasting. By using Analyze Data for AI-powered insights, building predictive models with XLMiner, and integrating Excel with Azure Machine Learning, you can unlock the full potential of predictive analytics and AI in Excel.

With the future of converged analytics in Excel, we can expect to see even more advancements in AI and predictive analytics that make it easier for users to gain insights from their data. By keeping up with these advancements, you can stay ahead of the curve and make more informed decisions that drive success.

Exercises

Practice Excel's augmented analytics and AI features with the following exercises using the datasets in the `datasets` folder of the book's repository:

1. Conduct sentiment analysis on the `imdb.xlsx` dataset using the Azure Machine Learning add-in. Then, use the XLMiner add-in to derive descriptive statistics for the resulting scores.
2. Import the `life_expectancy.png` image into Excel. Utilize the Analyze Data feature to create a line chart that visualizes the average life expectancy over time. Reshaping the data may be necessary to accomplish this.

The solutions to these exercises can be found in `ch_11_solutions.xlsx` within the `exercise_solutions` folder of the book's companion repository.

Chapter 6. Python with Excel

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at shunter@oreilly.com.

Thus far in the book we've been looking at all the things Excel can do. Power Query and Power Pivot aren't the only big new things. There is growing interoperability with Python. Let's take a look at that in this chapter.

Reader prerequisites

Although this chapter can be completed without prior knowledge of Python, familiarity with concepts such as lists, indexing, and the `pandas` and `seaborn` packages will greatly enhance your understanding.

If you'd like to learn more about Python before reading, I recommend starting with my book, *Advancing into Analytics: From Excel to Python and R* (O'Reilly, 2021), for a basic introduction to Python for Excel users. To delve even deeper into the subject, check out Felix Zumstein's *Python for Excel: A Modern Environment for Automation and Data Analysis* (O'Reilly, 2021).

This chapter includes a hands-on Python coding demonstration. To fully benefit from it, I encourage you to follow along on your own computer. All you need is a completely free version of Python. I recommend downloading the Anaconda distribution, which is available at [anaconda.com] (*https://www.anaconda.com/*).

The Role of Python in Modern Excel

Between Power Query and Power Pivot alone, "modern" Excel boasts so many features that it's impossible to learn them all. Add to that Office Scripts, Power BI, `LAMBDA()` and more and it's easy to get overwhelmed.

Prospective learners often find Python, an Excel-adjacent tool, intimidating. Many Excel users believe it should be the last tool on their learning list since it's not a Microsoft product and requires acquiring a new language proficiency.

Python may not be the best choice for every Excel user, but it's worth serious consideration for those looking to build complex automations, version-controlled projects and other advanced development products. Let's explore the role of Python in modern analytics and its relationship with modern Excel.

A growing stack requires glue

When I started as an analyst, my choices started and ended with Excel. All data, reporting, dashboards, everything was under that one green-and-white roof.

Fast forward a few years and there's now Power BI, Office Scripts, Jupyter Notebooks and more. This broadening of the analyst's tech stack is in line with wider trends in tech: a move away from one single, monolithic application to a loosely connected suite of specialized products.

To make this architecture work, a "conductor" or "glue" application is needed. Whether it's reading a dataset in from one source and visualizing it another, or deploying a machine learning model from the cloud to an end-user's dashboard, Python is a great choice for that glue. It's one of the rare

languages used for simple amateur-written scripts to enterprise-level applications alike, and it can work smoothly with a variety of operating systems and other programming languages.

Microsoft has acknowledged and celebrated Python's role as a "glue" language, and it's already available to use for a variety of purposes in Azure, Power BI, SQL Server and more. Although Python is not currently officially supported for Excel, it still offers significant advantages in Excel-related tasks, as you will discover in this chapter. Additionally, Python has gained widespread adoption among developers and organizations, making it a language with a thriving community and extensive resources.

Network effects mean faster development time

"Everybody's doing it" isn't usually a good reason to engage in something, but in the case of programming languages it may hold merit

Network effects, the concept that the value of something increases with its user base, apply to programming languages. As more programmers join, code sharing expands, providing a larger codebase for usage and further development, creating a virtuous cycle.

Python's versatility as a neutral "glue" language has led to its adoption in various professions, including database management, web development, data analytics, and more. This implies that regardless of the direction your Excel project takes or the tools you require, there's a high likelihood of finding collaborators who "speak" Python.

For example, you might develop an inventory tracker or other application using Excel. The program gets too unwieldy for a workbook, or it becomes so popular that your organization wants it to become a standalone web program. The turnaround time for this project will be much faster if the existing code is already in Python.

It's unfortunate that data analysts and tech professionals often use different tools, leading to slow and cumbersome collaboration. However, by embracing a shared language such as Python, the network effects amplify, resulting in accelerated development and reduced time requirements.

Bring modern development to Excel

The job title of “software developer” has gained popularity in recent years, often being seen as desirable as the role of a “data scientist.” However, the effective methods and best practices developed by software developers have not been widely adopted by BI or VBA developers. This situation raises concerns at the organizational level, as having two technology professionals labeled as “developers” but following distinct methodologies can lead to complications.

Python enables modern Excel developers to implement best practices, such as:

Unit testing

Many programming languages provide automated unit tests to ensure code functions correctly. However, Excel lacks this feature. Although there are some workaround tools available, Python is a strong choice for unit testing due to its network effects and abundance of packages. Automated unit testing enhances reliability and reduces the likelihood of errors, which is especially valuable for Excel workbooks accessed by users with diverse technical backgrounds.

Version control

Unlike most modern development tools, Excel lacks a version control system. This system tracks changes in a repository, enabling users to view contributions, revert to previous versions, and more. If you’ve ever struggled to differentiate between multiple workbooks like *budget-model-final.xlsx* and *budget-model-FINAL-final.xlsx*, you can appreciate the usefulness of version control.

Although Excel offers limited version control features like viewing version history in OneDrive and using the Spreadsheet Compare add-in, it falls short compared to the extensive features available when transitioning code production to Python.

Package development and distribution

If you’re seeking a more immediate reason to learn Python for your daily analysis tasks, then let me introduce you to one word: packages.

While I enjoy developing my own tolls, I believe in leveraging existing solutions when they fit my needs. Python's powerful features for building and sharing packages, particularly through the Python Package Index, unlock a world of tools that are challenging to replicate with Excel add-ins or VBA modules.

Whether you want to gather data from an application programming interface (API), analyze images, or simply generate descriptive statistics, the availability of Python packages justifies the investment in learning Python. Some of these packages are even designed to seamlessly integrate with Excel.

Python and the future of Excel

In an AI-driven world, learning to code may appear to be the wrong choice for staying relevant. Ironically, as data continues to grow in various formats, including AI-powered ones, the importance of coding becomes more relevant than ever.

Python's integration with Excel is becoming increasingly expansive, aligning with the modern analytics stack and the incorporation of AI-driven features within Excel.

Using Python and Excel together with `pandas` and `openpyxl`

With Python's role in modern Excel in mind, let's explore how the two can work together. Two key packages to facilitate this integration are `pandas` and `openpyxl`. Let's consider the two in turn.

Why `pandas` for Excel?

If you're working with any kind of tabular data in Python, you won't get far without `pandas`. This package lets you, among other operations:

- Sort and filter rows

- Add, remove and transform columns
- Aggregate and reshape a table
- Merge or append multiple tables

This is Python's equivalent to Power Query, enabling you to create reusable data cleaning and transformation workflows. And just like Power Query, `pandas` can effortlessly import data from diverse sources, including Excel, and even export the results of your analysis back to Excel.

The limitations of working with pandas for Excel

That said, pandas has limited features for deeply interacting with Excel workbooks. For example, it cannot help with the following tasks:

- Advanced formatting options for cells, such as applying specific styles or conditional formatting
- Support for executing Excel macros or VBA code within workbooks
- Direct access to Excel-specific features like data validation, charts, pivot tables, or formulas
- The ability to manipulate worksheets, such as renaming, adding, or deleting sheets
- Fine-grained control over workbook properties, such as password protection or worksheet visibility
- Handling of Excel-specific file formats like .xlsb or .xlsm
- Integration with Excel add-ins or plugins

Fortunately, several packages exist to provide these more advanced Python/Excel features, most notably openpyxl.

What `openpyxl` contributes

`openpyxl` (pronounced *open pie Excel*) is a Python package providing functionality for working with Excel files, specifically the newer *.xlsx* file format. It allows users to read, write, and modify Excel spreadsheets programmatically. openpyxl integrates smoothly with pandas, allowing users to clean data using pandas and add additional functionality to the workbook using openpyxl.

Specifically, openpyxl can help with the following tasks where `pandas` cannot:

- Advanced formatting options for cells, such as applying specific styles or conditional formatting
- The ability to manipulate worksheets, such as renaming, adding, or deleting sheets
- Fine-grained control over workbook properties, such as password protection or worksheet visibility
- Working with named ranges and tables
- Adding images, shapes, and charts to Excel files
- Handling print settings, page layout, and page breaks
- Working with formulas and formula-related functionality in Excel

Although `openpyxl` has limitations and cannot cover every use case, such as some mentioned earlier, it remains the most powerful and accessible Python package for automating Excel tasks.

How to use `openpyxl` with `pandas`

Let’s take a typical use case for automating a routine Excel business report where an analyst needs to generate monthly sales reports from multiple Excel worksheets. The analyst might read the data from each worksheet into `pandas` DataFrames, then continue to use `pandas` to clean and analyze the data. Finally, `openpyxl` is used to generate a consolidated report in a new Excel workbook, which contains conditional formatting,

charts, and more. The analyst could then use other Python tools to automate the distribution of the report.

For these and other tasks, the basic workflow for using pandas with openpyxl is like so:

1. Read the data: Use pandas to extract data from a variety of sources into tabular DataFrames
2. Clean and analyze the data: Use pandas to clean and manipulate the data, perform calculations, apply filters, handle missing values, and derive relevant insights.
3. Generate the report: Use openpyxl to create a new Excel workbook or select an existing one. Populate the workbook with the consolidated data, applying conditional formatting, creating charts, and incorporating any required visual elements.
4. Save the report: Save the updated Excel workbook using openpyxl, specifying the desired filename and location.
5. Distribute and automate the report: Send the generated report to the intended recipients through email, file sharing platforms, or any preferred method.

Other Python packages for Excel

Powerful as it is for Excel tasks, especially when combined with `pandas`, `openpyxl` has limitations. Thankfully, other packages are available to handle specific use cases. Some other packages to be aware of:

- `xlsxwriter`: Similar to `openpyxl`, `xlsxwriter` can be used to write data, formatting, and charts to Excel files in the *.xlsx* format. This package is optimized for performance, particularly when working with large datasets. It also offers more advanced cell formatting options compared to openpyxl. That said, as the name implies, `xlsxwriter` can only handle writing data to Excel, while `openpyxl` can both read and write.

- `xlwings`: This package enables the automation of Excel tasks, including interacting with Excel workbooks, running VBA macros, and accessing Excel's COM (Component Object Model) API on Windows. It provides complete two-way communication between Excel and Python in a way that `openpyxl` does not. On the other hand, this package requires a much more complex development environment, with many features only available on Windows.
- `pyxll`: This is a paid library that enables users to write Excel add-ins using Python. Instead of automating Excel workbooks, `pyxll` allows developers to build standalone applications for data science, financial trading, and other purposes.

Many other Python packages exist for Excel-related tasks, each with unique strengths and weaknesses. Considering Python's vast and active user community, the possibility of even more Excel-focused packages emerging is likely.

Demonstration of Excel automation with `pandas` and `openpyxl`

Let's stop discussing and start building. In this section we'll automate production of a small report from Python using `pandas`, `openpyxl` and more.

First, we will use `pandas` to perform complex data cleaning and analysis tasks that are difficult to achieve in Excel. After that, we will create an overview worksheet consisting of summary statistics and two charts, one from native Excel and one from Python. Finally, we'll load the entire dataset to a new worksheet and format the results.

A completed version of this script is available as `ch_12.ipynb` in the `ch_12` folder of the book's companion repository. If you're not sure how to open, navigate or interact with this file, check out Part 3 of *Advancing into Analytics: From Excel to Python and R* (O'Reilly, 2021) as a primer to Python and Jupyter Notebooks.

Let’s import the relevant modules and dataset to get started:

```
In [1]: import pandas as pd
        import seaborn as sns
        from openpyxl import Workbook
        from openpyxl.chart import BarChart, Reference
        from openpyxl.drawing.image import Image
        from openpyxl.utils.dataframe import
dataframe_to_rows
        from openpyxl.utils import get_column_letter
        from openpyxl.utils.dataframe import
dataframe_to_rows
        from openpyxl.worksheet.table import Table,
TableStyleInfo
        from openpyxl.styles import PatternFill
```

The `pandas` library can import data from various formats, such as Excel workbooks using the `read_excel()` function. Let’s read the `contestants.xlsx` file as `contestants`.

```
In [2]: contestants =
pd.read_excel('data/contestants.xlsx')
```

Cleaning up the data in pandas

A `pandas` DataFrame may have thousands of rows or more, so printing them all is impractical. However, it is important to visually inspect the data — a benefit Excel users are familiar with. To get a glimpse of the data and ensure it meets our expectations, we can use the `head()` method to display the first five rows.

```
In [3]: contestants.head()

Out[3]:

 EMAIL  COHORT  PRE     POST    AGE     SEX
EDUCATION       SATISFACTION    STUDY_HOURS
 0      smehaffey0@creativecommons.org  4       485
```

```
494     32.0    Male    Bachelor's      2       36.6
 1      dbateman1@hao12@.com    4       462     458
33.0    Female  Bachelor's      8       22.4
 2      bbenham2@xrea.com       3       477     483
NaN     Female  Bachelor's      1       19.8
 3      mwison@@g.co    2       480     488     31.0
Female  Bachelor's      10      33.1
 4      jagostini4@wordpress.org        1       495
494     38.0    Female  NaN     9       32.5
```

Based on this data preview, we identified there are a few issues that need to be addressed. First, it appears that some of the emails contain an invalid format. We also have some values in the `AGE` and `EDUCATION` columns called NaN which don't seem to belong. We can address these and other issues in the dataset in ways that would be difficult or impossible to do with Excel's features.

Working with the metadata

A good data analysis and transformation program should be equally proficient in handling both data and metadata. In this regard, `pandas` stands out as a particularly suitable tool.

Currently, our DataFrame has column names in uppercase. To make typing column names easier, I prefer using lowercase names. Fortunately, in `pandas`, we can accomplish this with a single line of code:

```
In [4]: contestants.columns =
contestants.columns.str.lower()
        contestants.head()

Out[4]:

email   cohort  pre     post    age     sex
education       satisfaction    study_hours
0       smehaffey0@creativecommons.org  4       485
494     32.0    Male    Bachelor's      2       36.6
1       dbateman1@hao12@.com    4       462     458
33.0    Female  Bachelor's      8       22.4
2       bbenham2@xrea.com       3       477     483
```

```
NaN     Female  Bachelor's      1       19.8
3       mwison@@g.co    2       480     488     31.0
Female  Bachelor's      10      33.1
4       jagostini4@wordpress.org        1       495
494     38.0    Female  NaN     9       32.5
```

Pattern matching/regular expressions

The `email` column of this DataFrame contains email addresses for each contest participant. Our task is to remove any rows from this column that have invalid email addresses.

Text pattern matching like this is accomplished using a set of tools known as regular expressions. While Power Query does offer basic text manipulation capabilities, such as case conversions, it lacks the ability to search for specific patterns of text, a feature available in Python.

Regular expressions can be challenging to create and validate, but there are online resources available to assist with that. Here is the regular expression we will be using:

```
In [5]: # Define a regular expression pattern for valid
email addresses
        email_pattern = r'^[a-z0-9]+[\._]?[a-z0-9]+
[@]\w+[.]\w{2,3}$'
```

Next, we can use the `str.contains()` method to keep only the records that match the pattern.

```
In [6]: full_emails =
contestants[contestants['email'].str.contains(email_patt
ern)]
```

To confirm how many rows have been filtered out, we can compare the `shape` attribute of the two DataFrames:

```
In [7]: # Dimensions of original DataFrame
        contestants.shape
```

```
Out[7]: (100, 9)
In [8]: # Dimensions of DataFrame with valid emails ONLY
        full_emails.shape

Out[8]: (82, 9)
```

Analyzing missing values

The `info()` method offers a comprehensive overview of the DataFrame's dimensions and additional properties:

```
In [9]: full_emails.info()

    <class 'pandas.core.frame.DataFrame'>
    Int64Index: 82 entries, 0 to 99
    Data columns (total 9 columns):
    #   Column        Non-Null Count  Dtype
    ---  ------        --------------  -----
    0   email         82 non-null     object
    1   cohort        82 non-null     int64
    2   pre           82 non-null     int64
    3   post          82 non-null     int64
    4   age           81 non-null     float64
    5   sex           82 non-null     object
    6   education     81 non-null     object
    7   satisfaction  82 non-null     int64
    8   study_hours   82 non-null     float64
    dtypes: float64(2), int64(4), object(3)
    memory usage: 6.4+ KB
```

In Python and other programming languages, `null` refers to a missing or undefined value. In pandas DataFrames, this is typically denoted as `NaN`, which stands for "Not a Number."

While basic Excel lacks an exact equivalent to `null`, Power Query does provide this value, which greatly aids in data management and inspection. However, it can be difficult to programatically work with these missing values in Power Query. pandas makes this easier.

For example I might want to see what columns have the most percentage of missing values. I can do this easily with `pandas`:

```
In [10]:
full_emails.isnull().mean().sort_values(ascending=False)
```

```
Out[10]:

        age              0.012195
        education        0.012195
        email            0.000000
        cohort           0.000000
        pre              0.000000
        post             0.000000
        sex              0.000000
        satisfaction     0.000000
        study_hours      0.000000
        dtype: float64
```

Because there are so few missing values, we will simply drop any row that has a missing observation in any column:

```
In [11]: complete_cases = full_emails.dropna()
```

To confirm that all missing observations have been cleared from the DataFrame, we can use the `info()` method again:

```
In [12]: complete_cases.info()
```

```
        <class 'pandas.core.frame.DataFrame'>
        Int64Index: 80 entries, 0 to 99
        Data columns (total 9 columns):
        #   Column         Non-Null Count  Dtype
        ---  ------         --------------  -----
        0   email          80 non-null     object
        1   cohort         80 non-null     int64
        2   pre            80 non-null     int64
        3   post           80 non-null     int64
        4   age            80 non-null     float64
```

```
5   sex           80 non-null     object
6   education     80 non-null     object
7   satisfaction  80 non-null     int64
8   study_hours   80 non-null     float64
dtypes: float64(2), int64(4), object(3)
memory usage: 6.2+ KB
```

Creating a percentile

Using `pandas`, we'll create a percentile rank for the `post` column and confirm its validity by running descriptive statistics with `describe()`:

```
In [13]: complete_cases['post_pct'] =
complete_cases['post'].rank(pct=True)
         complete_cases['post_pct'].describe()

Out[13]:

         count    80.000000
         mean      0.506250
         std       0.290375
         min       0.012500
         25%       0.264062
         50%       0.506250
         75%       0.756250
         max       1.000000
         Name: post_pct, dtype: float64
```

Creating a percentile column in Excel is straightforward, but validating results is easier in `pandas`. It offers statistical functions, methods for handling missing values, and more.

Our dataset has been cleaned and transformed using `pandas`:

```
In [14]: complete_cases.describe()

Out[14]:

             cohort      pre     post    age
satisfaction    study_hours     post_pct
```

```
        count   80.000000       80.000000
80.000000       80.000000       80.000000
80.000000       80.000000
        mean    2.475000        480.550000
480.987500      30.162500       5.237500
28.316250       0.506250
        std     1.147093        20.752856
23.181995       6.042976        2.869498        5.228245
0.290375
        min     1.000000        409.000000
398.000000      20.000000       1.000000
14.500000       0.012500
        25%     1.000000        470.000000
466.500000      25.000000       3.000000
25.950000       0.264062
        50%     2.500000        484.000000
483.000000      30.500000       5.000000
28.050000       0.506250
        75%     3.250000        494.000000
497.000000      35.000000       7.000000
32.225000       0.756250
        max     4.000000        521.000000
540.000000      40.000000       10.000000
40.900000       1.000000
```

Now let's use `openpyxl` to create a styled summary report.

Summarize findings with `openpyxl`

Creating a summary worksheet

To get started building an Excel workbook with `openpyxl`, we'll declare variables representing workbook and worksheet objects:

```
In [14]: # Create a new workbook and select the
worksheet
         wb = Workbook()

         # Assign the active worksheet to ws
         ws = wb.active
```

From there, we can populate any cell of the active sheet using its alphanumeric reference. I am going to insert and label the average pre and post scores in cells `A1:B2`:

```
In [16]: ws['A1'] = "Average pre score"
         ws['B1'] = round(complete_cases['pre'].mean(),
2)  # Round output to two decimals
         ws['A2'] = "Average post score"
         ws['B2'] = round(complete_cases['post'].mean(),
2)
```

Inserting data into the workbook this way is just a raw data dump; it does not affect the appearance of the data in Excel. Given my experience formatting data, I suspect the labels in column `A` will require more width. I'll adjust that now via the `width` property of the worksheet:

```
In [17]: ws.column_dimensions['A'].width = 16
```

Later in this chapter, we'll cover achieving AutoFit-like adjustments for column widths. But for now, let's shift our focus to adding charts to this summary.

Inserting charts

Option A: Create a native Excel plot

Excel's data visualization features are popular because they are easy to use and effective. Let's explore how to create native Excel charts from Python using `openpyxl`.

To begin, we need to specify the type of Excel chart we want to create and identify the location of the data for the chart within the worksheet:

```
In [18]: # Create a bar chart object
         chart = BarChart()

         # Define the data range
         data = Reference(ws, min_col=2, min_row=1,
max_col=2, max_row=2)
```

Next, we'll add this data source to the chart and label the chart's title and axes:

```
In [19]: # Add data to the chart
         chart.add_data(data)

         # Set chart title, axis labels
         chart.title = "Score Comparison"
         chart.x_axis.title = "Score Type"
         chart.y_axis.title = "Score Value"
```

Let's further customize this chart. We'll set category labels to reflect the data in the first column and also eliminate the chart legend.

```
In [20]: # Set category names
         categories = Reference(ws, min_col=1,
min_row=1, max_row=2)
         chart.set_categories(categories)

         # Remove the legend
         chart.legend = None
```

With the chart fully defined and styled, it's time to insert it into the worksheet.

```
In [21]: # Add the chart to a specific location on the
worksheet
         ws.add_chart(chart, "D1")
```

Option B: Insert a Python image

Python offers advantages in data visualization compared to Excel, as it provides more diverse visualization options and allows for easier customization of plots. For example, Excel lacks a built-in solution for analyzing relationships between multiple variables simultaneously. However, the `seaborn` data visualization package offers the `pairplot()` function, which provides a quick and convenient way to explore such relationships.

The following block visualizes these relationships across the selected variables of contestants:

```
In [22]: sns.pairplot(contestants[['pre', 'post', 'age',
'study_hours']])
```

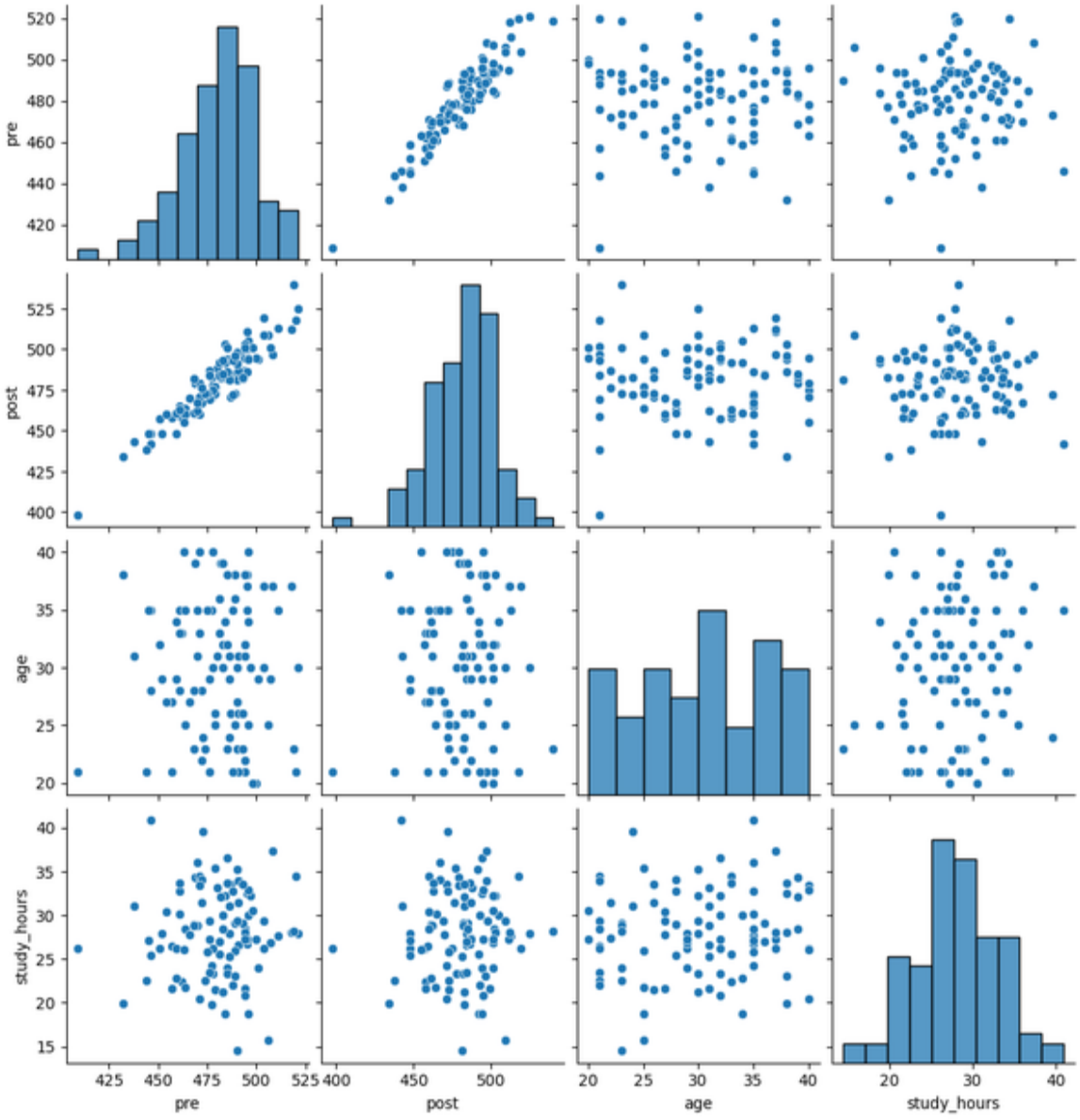


Figure 6-1. Pairplot

Not only does Python include a number of chart types that are difficult to build in Excel, they are easy to customize as well. For example, I'd like to see this visualization broken down by `sex`, which can be done by passing it

to the `hue` parameter. I'm going to save the results of this plot as `sns_plot` so I can refer to it later.

```
In [23]: sns_plot = sns.pairplot(contestants[['pre',
'post', 'age', 'study_hours', 'sex']],
             hue='sex')
```

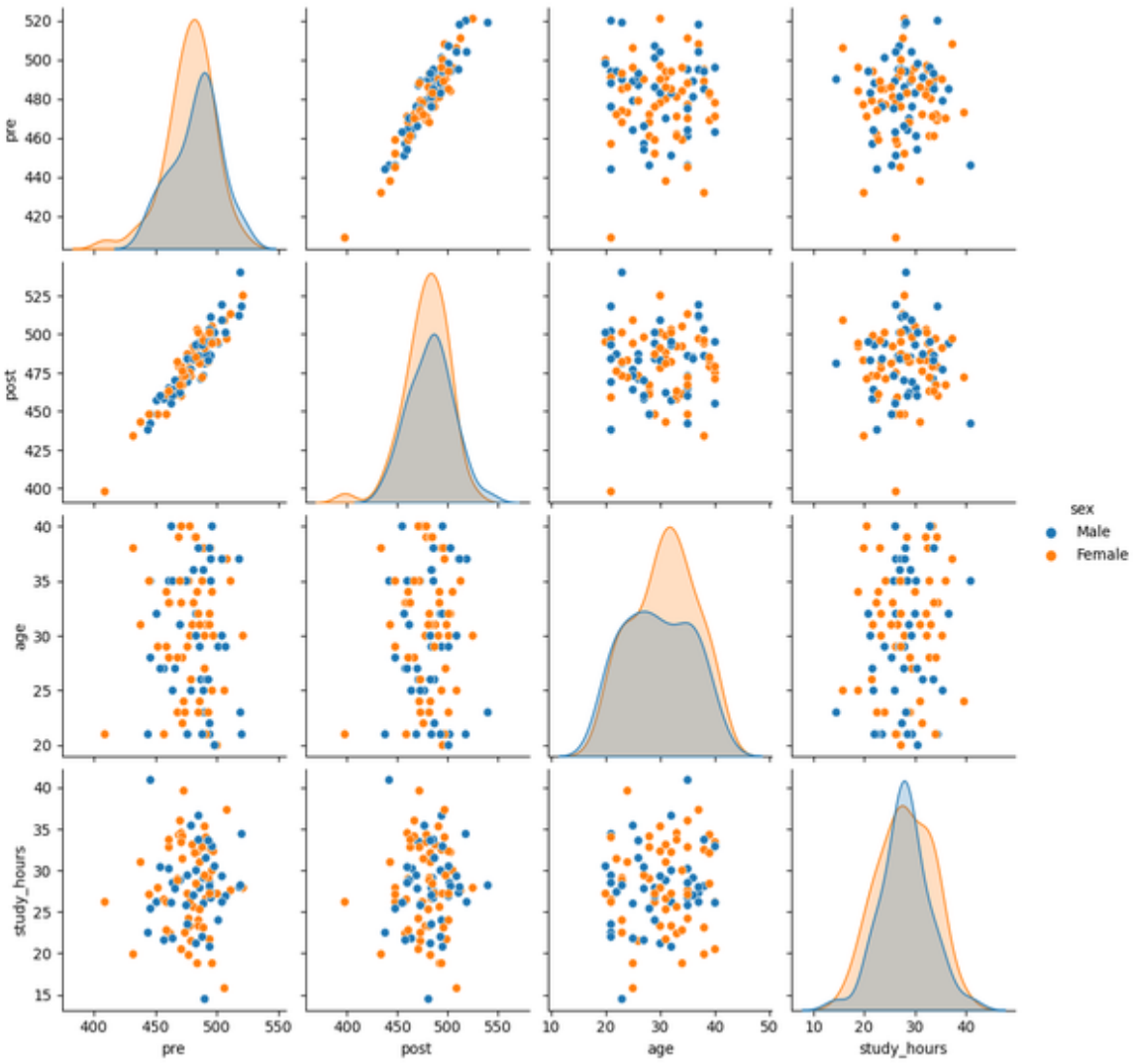


Figure 6-2. Pairplot by sex

Next, let's place a static image of this pairplot into the workbook. This requires first saving the image to disk, then specifying where to place it into the workbook:

```
In [21]: # Save pairplot to disk as an image
         sns_plot.savefig('pairplot.png')

         # Load saved image into the worksheet
         image = Image('pairplot.png')
         ws.add_image(image, 'A20')
```

Excel versus Python charts

A summary of the pros and cons of these two methods follows in Table 6-1:

Table 6-1. Pros and cons of Python versus Excel charts

	Pros	Cons
Building a native Excel plot	- Plot will update with changes in Excel data User can interact with and customize plot Plot can integrate with other Excel features like formulas and PivotTables	- Limited number of Excel plot types exist Some complex calculations or statistical functions may be easier to build in Python
Inserting an image of a Python plot	- Access to several powerful plotting libraries such as matplotlib and seaborn Plot is easily audited and reproduced through the source code	- The plot is a static image and lacks interactivity Updating or refreshing the chart from Excel is not possible

The choice between methods depends on different factors, such data refresh needs and the availability of specific chart types in Excel. That said, the flexibility and range of options available in itself highlight Python's powerful capabilities for working with Excel.

Adding a styled data source

Now that our summary worksheet has been created, we will create a second, styled worksheet consisting of the `complete_cases` DataFrame. The first step is to define this new worksheet:

```
In [25]: ws2 = wb.create_sheet(title='data')
```

Next, we'll loop over each row of `complete_cases` and insert them into the worksheet:

```
In [26]: for row in dataframe_to_row(complete_cases,
index=False, header=True):
    ws2.append(row)
```

Inserting the DataFrame into the worksheet is a start, but the resulting data may be challenging for users to read and manipulate. Let's make a few improvements.

Formatting percentages

By default, the `post_pct` column will be formatted in Excel as decimals instead of more readable percentages. To address this, we need to specify the location of this column in the worksheet and re-format it.

I will use the `get_loc()` method to find the index position of this column in the DataFrame, adding 1 to the results to account for Excel's 1-based indexing. The `get_column_letter()` function will convert this index number into Excel's alphabetical column referencing.

```
In [27]: post_pct_loc = complete_cases.columns.
get_loc('post_pct') + 1
         post_pct_col = get_column_letter(post_pct_loc)
         post_pct_col

Out[27]: 'J'
```

With the proper column identified, I will apply the desired number formatting to each row:

```
In [28]: number_format = '0.0%'

         for cell in ws2[post_pct_col]:
             cell.number_format = number_format
```

Converting to a table

As discussed in the Preface of this book, tables hold a number of benefits for data storage and analysis. We can convert this dataset into an Excel table with the following code:

```
In [29]: # Specify desired table formatting
         style =
TableStyleInfo(name='TableStyleMedium9',
showRowStripes=True)

         # Name and identify range of table
         table = Table(displayName='contestants',
                       ref='A1:' +
get_column_letter(ws2.max_column) + str(ws2.max_row))

         # Apply styling and insert in worksheet
         table.tableStyleInfo = style
         ws2.add_table(table)
```

Applying conditional formatting

To enhance readability for end users, we can apply conditional formatting to the worksheet. The following code will apply a green background fill to participants above the 90th percentile and a yellow background fill to participants above the 70th percentile:

```
In [30]: # Define conditional formatting style
         green_fill = PatternFill(start_color="B9E8A2",
end_color="B9E8A2", fill_type="solid")
         yellow_fill = PatternFill(start_color="FFF9D4",
```

```
end_color="FFF9D4", fill_type="solid")

        # Loop through data table and conditonally
apply formatting
        for row in ws2.iter_rows(min_row=2, min_col=1,
max_col=len(complete_cases.columns)):
            post_pct = row[post_pct_loc - 1].value #
Convert index to 0-based indexing
            if post_pct > .9:
                for cell in row:
                    cell.fill = green_fill
            elif post_pct > .7:
                for cell in row:
                    cell.fill = yellow_fill
```

Auto-fitting column widths

Although `openpyxl` lacks an AutoFit feature to automatically resize worksheet columns, we can achieve a similar outcome with the following code. It finds the widest row in each column of the worksheet, then adds enough padding to adjust the column width accordingly:

```
In [31]: for column in ws2.columns:
        max_length = 0
        column_letter = column[0].column_letter
        for cell in column:
            try:
               if len(str(cell.value)) > max_length:
               max_length = len(cell.value)
           except:
               pass
        adjusted_width = (max_length + 2) * 1.2
        ws2.column_dimensions[column_letter].width =
adjusted_width
```

After completing the workbook, we can save the results to `ch12-output.xlsx`.

```
In [32]: wb.save('ch12-output.xlsx')
```

Conclusion

This chapter explored Python's role in modern Excel, highlighting its versatility as a "glue" language for development and its ability to enhance Excel's capabilities. Through a hands-on demo, it demonstrated how Python automates Excel, adding features difficult or impossible to achieve within the spreadsheet program alone. As Microsoft further integrates Python into its data analytics stack, the Python and Excel landscape will evolve. However, this chapter provides a solid framework for effectively using Python and Excel together, maximizing their potential.

Exercises

For this chapter, create a concise summary report of the `websites.xlsx` file located in the `datasets` folder of the book's companion repository. Use the provided `ch_12_starter_script.ipynb` file as a starting point for the project. Fill in the missing sections of the Jupyter notebook to achieve the desired solution, which can be found in the `exercise_solutions` folder of the repository. Refer to the chapter's examples for guidance in writing the code accurately, and feel free to incorporate additional automated features into your work.

To be completed later