One way to perform this data analysis would be to go through all of Steve's stock data manually and use Excel formulas for calculations. But with Visual Basic for Applications, which is typically referred to as "VBA," we can write code that will automate these analyses for us. Often used in the finance industry, VBA provides essentially infinite extensibility to Excel. Using code to automate tasks decreases the chance of errors and reduces the time needed to run analyses, especially if they need to be done repeatedly.

**NOTE**

BASIC (short for Beginner's All-purpose Symbolic Instruction Code) was a programming language invented in the 1960s to help teach programming concepts. It soon gained traction and started to be used as a full-fledged programming language. In the 1990s, Microsoft created a version of BASIC with a visual form builder so that graphical desktop applications could be built, and Visual Basic was born! It lives on today in VBA and VB.NET.

By learning the ins and outs of Excel, you've dipped your toes into the waters of programming; with VBA, you're going to dive all the way in as you learn how to create your own functions and automate tasks in Excel. Adding VBA to Excel is like adding a superpower; but like any great power, it comes with great responsibility.

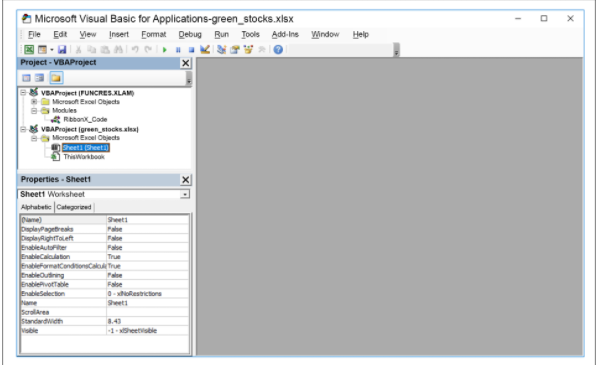
In developer parlance, automated tasks are called **macros**. Originally, macros were created by "recording" a task that you performed in Excel, and VBA would automatically generate code to repeat the task. A task could be something as simple as deleting a comma at the end of a cell's value and then moving to the next cell. The macro could be run repeatedly to quickly perform the task over and over again.

While recording a task seems like it might be a good way to create macros, the reality is that it doesn't work as well as one would hope, and the code is difficult to understand and edit. It's more efficient to write the code from scratch.

Almost all VBA code is written to create macros, which are sometimes called **subroutines**. However, VBA is powerful enough to connect to the internet and run applications in the operating system, which means it can be abused to write malicious code, like viruses and trojan horses. Because of this, VBA is included with Excel, but access to it is disabled by default. To enable VBA in Excel, we need to add the Developer tab to our ribbon.

How you enable VBA depends on whether you are a Windows or macOS user. Watch the video below for step-by-step instructions for your operating system.

On the left side of your screen, you should see a list of VBA Projects, including green\_stocks.xlsx, the data file from Steve. Right-click on the file, and then go to Insert and select Module. All of our VBA code and macros will live inside modules. Double-click on "Module1" to open the editor. Refer to the following screenshot:



We're almost ready to write some code, but first, we need to save the file to a new extension.

Regular Excel workbooks, which have the ".xlsx" extension, can't hold macros. If we try to save the workbook, we'll get a warning telling us that any macros we've written will be deleted.

Therefore, we need to save green\_stocks.xlsx as an Excel Macro-Enabled Workbook, which is an option listed in the Save As menu. This will allow us to save the macros we make. The file extension for a macro-enabled workbook is "xlsm."

It's important to remember that macro-enabled workbooks are very powerful. Some people abuse this power and write malicious code in VBA macros. Fortunately, Microsoft has taken precautions to prevent malicious code from being executed accidentally. Watch the following video to learn more about how to enable macros in Excel, depending on your operating system.

When we send green\_stocks.xlsm back to Steve, he'll also have to enable macros before he can run the code we've embedded.

## Build a Subroutine

A **subroutine** is a key building block of a VBA macro. Subroutines are a collection of steps or instructions. A subroutine is given a name so that the subroutine can be **called**, or run. Technically, a macro can be made up of several subroutines, but in general,  the terms "macro" and "subroutine" are used interchangeably.

In order to check that VBA is working correctly,  we're going to write a subroutine called MacroCheck. Type the following into the editor:

Sub MacroCheck()

End Sub

Here, Sub is a statement that tells VBA to create a subroutine. MacroCheck is what we're telling VBA this subroutine is called.

**IMPORTANT**

**Statements** are words in VBA that have a special meaning—they tell VBA to do something. Statements are part of a broader group of special words in VBA called keywords. **Keywords** are the vocabulary of a programming language.

Now VBA knows that all of the following steps belong to MacroCheck()until it sees End Sub. If we forget to put End Sub, VBA will halt and give us a syntax error.

**IMPORTANT**

The **syntax** of a programming language is the set of rules for how keywords can be arranged. A **syntax error** is when the rules of syntax are not followed.

The End Sub keyword tells VBA that we're done with the subroutine—though we'll be adding more code soon—and closes the block of code. The editor may have autocompleted the End Sub code for you.

**NOTE**

Wondering why there are parentheses after the name of the subroutine? The parentheses are automatically added because subroutines can take in inputs, called **arguments**.

The term "arguments" comes from mathematics and refers to the input to a function. In math, a function takes inputs, performs some calculations, and gives an output. In VBA, subroutines can take inputs and run lines of code, but they usually do not return any outputs. Empty parentheses indicate that this subroutine doesn't take any arguments.

Now let's add code to our subroutine to check that the macros are working correctly. We'll write code to display a message that will tell us if our code is running correctly.

**Variables and Data Types**

Before we tell VBA to display a message, we need to give it the exact message we want to display. In programming, all text is just a collection of characters strung together, so they are called **string** data types. This data needs to be stored somewhere with a way for us to reference it. Therefore, we'll tell VBA to clear out some space in the memory for our message, and then give it a name to use in our program. We do this by creating a **variable**.

**Variables** are a fundamental building block of all programming languages. They hold the data in our code. Two important parts of a variable are its **name** and **data type**. Different programming languages have different rules for what you can use for a variable name, but generally speaking, you can name a variable anything that begins with a letter (the first character can't be a number) and isn't already a keyword in the language.

When a variable is created, it takes up space in memory as **bits**, or 1s and 0s. The **data type** of a variable defines how many bits of memory the variable will occupy and how to translate those bits into values.

For example, the **integer** data type stores whole number values in 16 bits, or 2 bytes. One bit is used to store the sign of the value (whether it is positive or negative), and the other 15 bits store the absolute value (or magnitude) in binary.

**NOTE**

Binary numbers, or base-2 numbers, are numbers that are written with only ones and zeros. We normally write numbers in base-10, using the digits 0–9 to represent how many times a power of 10 is included in a number. For example, the number 347 has 3 hundreds (102), 4 tens (101), and 7 ones (100). In binary, we use the digits 0 and 1 to represent how many times a power of 2 is included in a number. So, 100110 in binary has 1 thirty-two (25), 1 four (22), and 1 two (21), which adds up to 38. We say that it takes 6 bits to store the value of 38 because we need 6 digits.

This means there are 215, or 32,768, possible absolute values, and that zero has two representations. Therefore, integer variables can hold values from –32,768 to 32,767. If larger values are needed, the **long** data type uses 32 bits, and can store values from –2,147,483,648 to 2,147,483,647.

**NOTE**

When the value of a variable extends past the range allowed by the data type, this is known as an **overflow**. Nowadays, most programming languages will halt if a line of code is going to cause an overflow. However, older systems would perform the calculation and corrupt the variable's space in memory. This caused bizarre errors. One famous example is the "kill screen" in Pac-Man on the 256th board, where half the screen is filled with garbled, glitched symbols.

**Data Type Examples**

Some common data types are:

* **Integer:** Positive and negative whole numbers between –32,768 and 32,767, stored in 16 bits
* **Long:** Positive and negative whole numbers between –2,147,483,648 and 2,147,483,647, stored in 32 bits
* **Double:** Decimal numbers (i.e., numbers with fractional parts) stored in 64 bits
* **String:** Text
* **Boolean:** True/false values

Why is there a difference between integer, long, and double? Deep down in the architecture of the processor in your computer, the arithmetic for each type of number is handled differently. Also, each data type takes up a different number of bytes of memory: integers are 2 bytes, longs are 4 bytes, and doubles are 8 bytes. This difference can add up if you have a complicated analysis that stores thousands and thousands of variables. (It happens!)

Some examples of variables for each data type are listed below.

**Integer**

* Number of trading days in a year
* How many family members a person has
* The floor number in a skyscraper

**Long**

* Number of views for a video on YouTube
* Population sizes of cities

**Double**

* Latitudes and longitudes
* The constant pi
* Interest rates

**String**

* Employee names
* International postal codes
* Movie titles

**Boolean**

* Whether a door is open or closed
* Whether someone is over 18

### Create a Variable

The keyword to create a variable in VBA is Dim, which is short for "dimension." We have to tell VBA the name of the variable and what kind of data type it will store. Because the message will display text, we need to use String as the data type. So the full line of code is:

Dim testMessage As String

With the variable declared, we can now assign it a value. We do that by referencing the variable by name and setting its value with an equals sign.

**NOTE**

When assigning values, the equals sign is more precisely referred to as an **assignment operator.**

For a string, we put quotes around the text so that VBA knows we're sending it string data, not referencing another variable—a variable that doesn't exist in this case. So the full line of code will look something like this:

testMessage = "Hello World!"

**NOTE**

"Hello World" has a long history in programming as the first program a new coder writes. It's also traditionally used as a sanity test to make sure a newly installed programming language is correctly installed. Now you've also performed this rite of passage. Welcome to programming!

## Display a Message

The keyword to create a message box is MsgBox. MsgBox takes in an **argument**, which, in our case, is what we want our pop-up box to display. For now, let's set testMessage as the argument. Make sure your code looks like the following, and then click the Play button located in the top toolbar or press F5 to run the code.

Sub MacroCheck()

Dim testMessage As String

testMessage = "Hello World!"

MsgBox (testMessage)

End Sub

If the code works correctly, you should see a message box pop up in the green\_stocks.xlsmwindow. Watch the following demo to see this process in action. Choose the video for your operating system.

**Work with Cells**

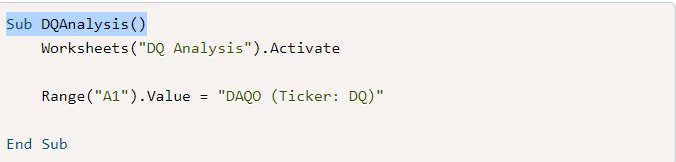
Since Excel holds its data in cells, we want to be able to access them in VBA. There are two ways to access cells in VBA: the Range() method and the Cells() method. For our project, we're going to use both.

**IMPORTANT**

Everything we interact with in Excel—for instance, cells, ranges, charts, and worksheets—are **objects** in VBA. VBA objects have properties that we read and methods that we call. **Properties** are like predefined variables that hold values about the object. A **method** is like a subroutine: a collection of instructions that can be called. Methods often take in arguments and can return values.

In this case, the Range() method belongs to the Worksheet object that we activated.

First, we'll use the Range() method, which selects cells with the same range format that Excel formulas use. The Range() method can also select a range of only one cell, which is what we are going to use here. We'll set the value of cell A1 to "DAQO (Ticker: DQ)" with the code Range("A1").Value = "DAQO (Ticker: DQ)", as shown below:



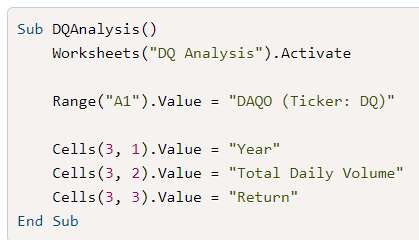
Next, we'll use the Cells() method. It works similarly to the Range() method, but it takes two arguments:

* how many rows down from the top the target cell is
* how many columns over from the left the target cell is

For example, to put "Year" in the cell A3, we would use Cells(3, 1).Value = "Year".

Let's use Cells() to create a header for cells A3 through C3 with column names Year, Total Daily Volume, and Return.

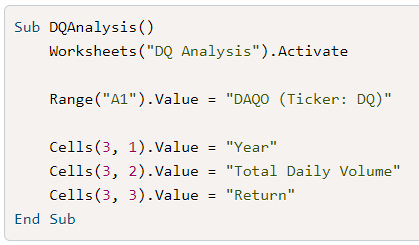
In this example, we could use Range() to accomplish the same goal, but Cells() will be more flexible as we move to automated code because individual numbers are easier to work with than strings of cell coordinates. When filling in the table below the header, use the same pattern of code but specify the row value using a variable instead.



VBA doesn't care much what your code looks like, as long as everything is typed correctly and in the right order. In general, however, code is read more often than it is written, so it helps to document and format your code to improve its overall **readability**. Documenting code is done by adding **comments**. Formatting code involves the use of **whitespace**.

## Comments

Take a look at the code we just wrote:



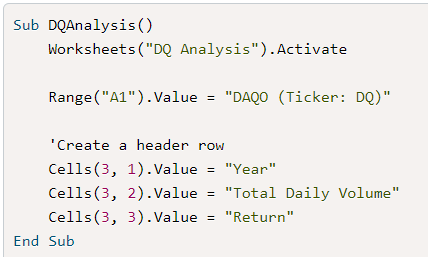
This code works great, but it isn't self-explanatory. Also, as the code becomes more complicated (which it will), it will be useful to insert comments to explain what the code is doing. Comments can also be used to mark sections of code in order to make them easier to find.

**IMPORTANT**

Every programming language has some way of adding **comments**. If there's a tricky line of code that is difficult to understand, adding comments can help the next person reading your code understand what your code is doing—and that person might be you. Adding comments to our code helps us remember our thought process, especially after taking a break from a project for any length of time.

In VBA, comments start with a single quote. Anything you write after the single quote gets ignored, which allows us to add notes to our code. Let's explain with a comment what the three Cellslines are doing.

Above the first Cells line, insert a single quote followed by the text "Create a header row."



You just wrote your first comment—nice work!

**Whitespace**

Readable code is also well organized. Enter **whitespace,** which refers to the use of spaces, tabs, and line breaks. Whitespace is used to organize code.

Whitespace is "invisible" to VBA. When VBA runs the subroutine you've written, it translates all your keywords and statements into instructions your computer can digest and completely ignores whitespace—with the exception of spaces between keywords and line breaks. Therefore, you can add as many spaces, tabs, and line breaks as you want, and your macro will still run the same way.

**NOTE**

Discussions about how to format code can get very heated—seriously! Programmers still argue about whether tabs or spaces should be used to indent code. The correct answer is, quite simply, to be consistent. Code is easiest to read and understand when it follows consistent formatting.

We'll be using whitespace to organize our code. For example,  in our macro, all the code inside the subroutine is indented. This way, it's easier to see that the code belongs to this particular subroutine. Later, when we get into more complicated program flow, we'll have code blocks within code blocks, so we'll indent multiple times to keep the code organized. As programs get longer and more complicated, well-formatted code makes life so much easier.

Go ahead and test your code as often as you'd like. It's often helpful to run code as you build it and check for errors as you go. If you need to reset the worksheet after running your macro, resetting it is pretty easy, too.

First, make sure you save your macro in a text file (a text editor such as Notepad or even Google Docs or Microsoft Word will work, as long as you keep a copy of your code).

Then, close your Excel workbook without saving, then open it again. The data is once again fresh and ready for analysis. The code you saved in a text editor can now be copied back into the VBA code editor if needed.

**Loops**

We're going to calculate the total daily volume in 2018 using **loops**.

**IMPORTANT**

**Loops** tell a computer to repeat lines of code over and over (and over, and over) again. for loops tell the computer to repeat the lines of code a specific number of times. You can think of a for loop like telling the computer to "run this code *for* as many loops as I tell you to." There are a few different kinds of loops, but for loops are the workhorse of loops. It's entirely possible that you'll never need to use any other kind of loop.

To find the total daily volume, we'll loop through every row in our stock data worksheet and check if the ticker for that row is DQ. If it is, we'll add its daily volume to our total volume.

In VBA, the syntax for a for loop has a beginning, middle, and end. The beginning is one line that tells VBA that we're opening a for loop; the middle is the block of code to be repeated; and the end is one line that closes the block of code.

The opening line of a for loop uses the keyword For and an **iterator**. Iterators are named variables that change values over the course of the for loop, usually increasing by 1, thus holding the number of times the loop has repeated. For example, if we wanted a for loop that would loop exactly 3 times, the opening line would look like this:

For i = 1 to 3

Inside the code block, we can use the iterator like any other variable. So, if we want to display 3 message boxes in a row, showing the number of times the loop has repeated, our code block would look like this:



Now we need to close the for loop with our last line. To tell VBA that the for loop has ended, we use the keyword Next and the iterator we used: i.

**NOTE**

We need to specify the iterator because there can be more than one for loop going on. These are **nested** for loops, which we'll get to later.

So our full for loop looks like this:



Make a new macro and name it DQAnalysis, then run the for loop to make sure it works.

Because the iterator is treated like any other variable inside of the code block, we can use it to interact with our data. To see this, change the for loop to display each of the stock data column names in a MsgBox. There are 8 columns, so change the iterator to go from 1 to 8, and then change the MsgBox to display Cells(1, i).



Make sure the 2018 sheet is active. There are two ways to do this. The first is to select the correct sheet from the file structure display to the left of your window, then run the macro. The other option is to update your code by adding Worksheets("2018").Activate before the for loop. Once ready, execute your macro.

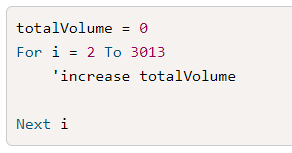
**NOTE**

Using Worksheets("2018").Activate in this macro will ensure that the analysis is always performed on the correct sheet.

You should get 8 message boxes in a row, each displaying one of the column headers.

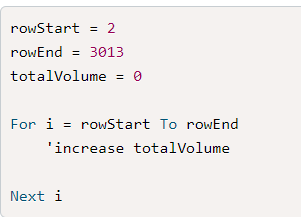
We can also use variables that are defined outside of the for loop inside our block of code. If we create a totalVolume variable and set it equal to zero, we can add to it inside of the loop; when the loop is done, totalVolume will hold the sum of all the volume.

There are 3,013 rows in the 2018 worksheet. We don't want to include the header row, so our iterator will go from 2 to 3013. We could write a loop where we hard-code the numbers for the iterator, like this:



Hard-coded values like these are known as **magic numbers**: if you didn't write this code, it would seem like the numbers were created by magic. Magic numbers should be avoided as much as possible. A better way to use these numbers is to create variables with informative names and assign them the values we want to use.

Make a rowStart and a rowEnd variable with the values 2 and 3013, respectively, and use those in the opening line of your for loop:

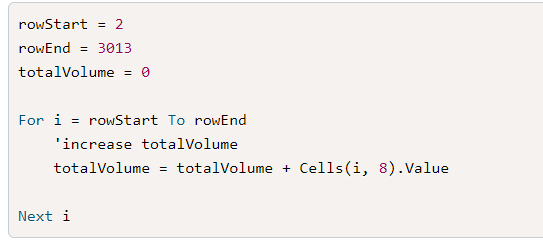


**NOTE**

totalVolume, rowStart, and rowEnd are all new variables, so you might be asking yourself why we didn't need to initialize them with the Dim keyword. In this case, because we're setting values as we initialize them, VBA can infer the data type from the value being assigned. Here, all the values are integers, so VBA will create integer variables for us in the background.

Using named variables here allows us to make our code more flexible, because if we update the data, we only have to change the value of the variables to match. However, an even better solution is to use code to find the value dynamically based on the data. We'll explain how to do that in VBA when we're done writing our loop.

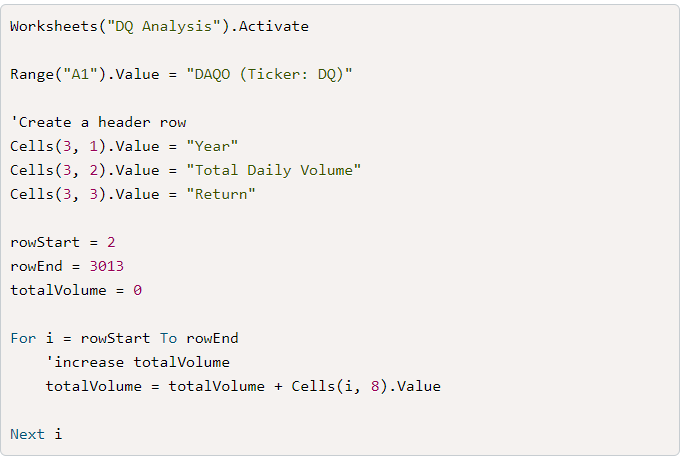
Volume is in the 8th column, so we can increase the totalVolume by the value in Cells(i, 8):



To check that totalVolume increased, display the value with MsgBox(). To do this, adjust your code like so:

MsgBox(totalVolume)

This is the total volume for all the stocks; we want just the total volume for DQ. To do this, we need another tool in our toolbelt: conditionals. Before we jump into that, let's make sure your code is accurate. So far, your code should look like this:



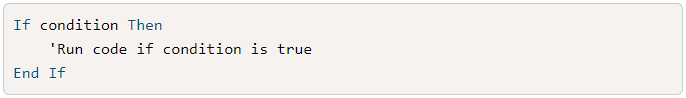
**Conditionals**

**IMPORTANT**

**Conditionals** tell the computer that certain lines of code should run only under certain conditions. The workhorse of conditionals is an **if-then statement**. The if-then statement checks if a condition is true. If it is, then a block of code will be run.

In VBA, the syntax of an if-then statement also has a beginning, middle, and end. The beginning is one line to open the if-then statement; the middle is a block of code to run if the condition is true; the end is a line that closes the if-then statement.

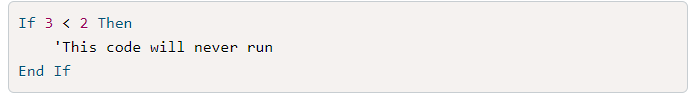
The opening of an if-then statement uses the keywords if and then, and a condition. The code block can be any number of lines of code. To close the if-then statement in VBA, add the line End If.



The condition is a logical expression that is either true or false. For example, if we check that 2 is less than 3, it evaluates to true, so the code will run.

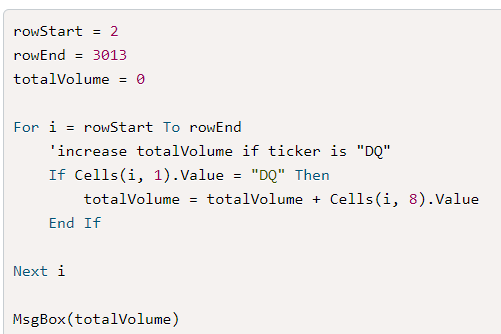


However, if we check that 3 is less than 2, it will evaluate to false, so the code block will not run.

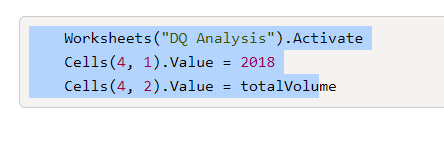


Let's hop back into DQAnalysis. In the for loop we wrote in the previous section, if we add a conditional to check whether the ticker is "DQ," we can then only increase totalVolume for DQ's volume.

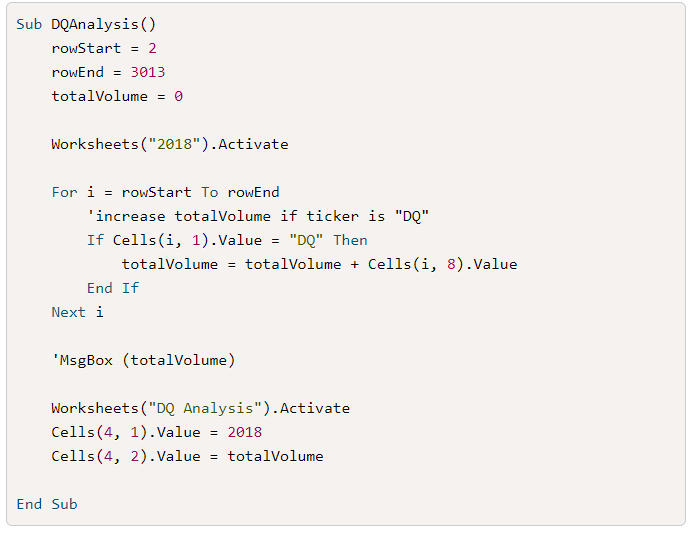
The condition we're checking is that the value of the cell in the first column is "DQ," so our code will look like this:



Now we just need to set the value in our output worksheet to show the total volume. To do this, activate the output worksheet, and then use Cells().value or Range().value to set the value:



Let's do a quick code check. Compare your code to the code below to check for errors.



## Patterns

The code that we used to calculate DQ's yearly volume is a simple **design pattern**. Recognizing and utilizing design patterns is crucial for writing good code.

**IMPORTANT**

In programming, **design patterns** (or just **patterns)** are templates to solve similar problems. Patterns aren't necessarily code but rather reusable structures to help us write our code.

The pattern we used follows this general structure:

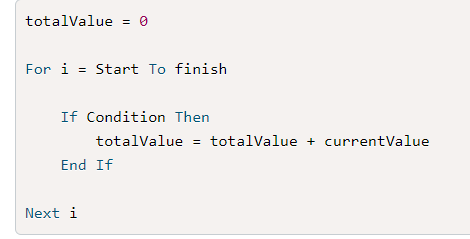
1. Initialize a variable to hold a sum.
2. Set the variable to zero.
3. Start a for loop.
4. Use a conditional to increase the sum variable by a value.
5. End the loop.

**IMPORTANT**

Design patterns are larger than a single programming language. They offer a way to organize a process so that it can be put into code.

Programmers use **pseudocode** to break down algorithms and processes into a design pattern without being tied to a specific language. Pseudocode can range from writing a list of tasks in natural language to formatting code with indentation and adding simple keywords. The main purpose of pseudocode is to think through the logic of the code before actually writing it.

In pseudocode, the pattern looks like this:



You should always be on the lookout for new patterns to use. Every time you solve a single problem, consider whether you can use that same pattern to solve another problem.

The opposite of a good design pattern is an anti-pattern. **Anti-patterns** are common responses to problems, but they may be ineffective, too specialized, or generally counterproductive. Quick and dirty workarounds, called **kludges**, often use anti-patterns.

**NOTE**

Anti-patterns and kludges are part of a broader idea of **code smells**, where the code works and solves the problem it's supposed to, but something about the code indicates that there is probably a more elegant and productive solution to the problem.

We've already avoided one anti-pattern: an unnamed magic number for the row count. Let's take it even further and, instead of giving our magic number a name, we'll have VBA figure out the value for us. This way, if the data changes, our code will still work.

## Research Practice

To calculate DQ's total daily volume, we need to loop through all of the stocks, so we've typed the number of rows into the code itself. What would be even better, though, is to use VBA to find the number of rows to loop over. Unfortunately, VBA doesn't have a nice function or method to figure that out. But we can't be the first person to have this problem; someone must have found a solution. We just need to find it.

Enter Google! Programmers and analysts use Google all the time to find solutions to on-the-job problems and tasks—there's no shame in the googling game. Let's get some hands-on experience with this real-world skill and find a way to get the number of rows with VBA.

Search for something like "VBA get the number of rows with data." Note that searching for the right keywords is way more important than being grammatically correct. Also be sure to include the programming language that you're using—in this case, VBA.

A ton of search results will come up, but most of them will fall into four categories:

1. Official documentation
2. Stack Overflow
3. Quora
4. Expert blog posts

### Official Documentation

For VBA questions, the official documentation is published by Microsoft. In general, official documentation is a reference guide published by the creator of the language or software. You might think this is the logical place to check first, but that's not always the case. Because official documentation is meant to be an exhaustive reference, it's often extremely dense and difficult to understand, especially when you're first learning a programming language. In the worst cases, the documentation isn't even up-to-date and therefore contains incorrect information. It's often better to get explanations from people who are actually using the code.

### Stack Overflow

[Stack Overflow (Links to an external site.)](https://stackoverflow.com/) is almost always the best resource for getting practical solutions. Programmers will half-joke that their job is just googling problems and copying and pasting Stack Overflow code. Stack Overflow works like this: someone asks a question about a problem they're running into and experts will provide answers, almost always with sample code. The expertise level is top-notch; sometimes the person who invented the program will be the one giving the answer! Answers are voted on, so you can get a sense of which answers are more authoritative than others.

### Quora

[Quora (Links to an external site.)](https://www.quora.com/) has a similar format to Stack Overflow, but the questions and answers tend to be more theoretical. The expertise level is similarly high, but the emphasis is more on understanding concepts rather than just getting some code that works. If you're struggling to do something, go to Stack Overflow; if you're struggling to understand something, go to Quora.

### Expert Blog Posts

On the internet, anyone can claim to be an expert, so blog posts vary wildly in quality—especially because there is no mechanism for people to give feedback on the quality of a blog's explanations. However, expert blog posts tend to blend sample code and theory, so a good post will help a new concept click.

## Narrow Your Search

There's no reason to feel awkward about using search engines to figure out problems. Welcome to being a programmer! A big part of any programming job is googling solutions and spending a lot of time on Stack Overflow. Even experienced programmers do this on a daily basis.

Our search "VBA get number of rows with data" gives a number of results, and many look promising. There's a Stack Overflow link, a link to the official documentation, and a handful of blogs and forums. Click on a few links and skim the information to get a feel for how different sites answer questions like this. Here are two curated, potential solutions. (Don't worry about finding the exact links to these. And remember that you might find even better solutions!)

## Potential Solution #1: Stack Overflow

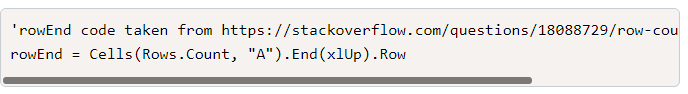
An answer on Stack Overflow provides the following code for finding the number of rows with data:

lastRow = Cells(Rows.Count, "A").End(xlUp).Row

This is fairly complicated, so let's break it down:

* Cells(Rows.Count, "A") goes to the bottom cell in column A, which may extend past the last row of data in the sheet.
* .End(xlUp) is the same as pressing END and then the up arrow in Excel, which will go to the last cell with data in column A. We use this to move from the bottom of the sheet to the last row of data.
* .Row returns the row number.

If your eyes glazed over during the explanation of how that line of code works, don't worry. In a perfect world, we would have time to research everything and understand it completely, but sometimes we just need to get code that works, even if we can't explain exactly how it works. It's a good idea to add a comment that explains where the code came from, especially if we're not entirely sure why it works. Linking back to where the code came from can help anyone reading the code understand why it is being used.



## Potential Solution #2: Blog Post from an Excel Expert

The previous Stack Overflow solution works (and spoiler alert: it's what we're going to use in our code), but here's another possible solution to demonstrate that there are usually multiple ways to solve a problem. This one is from an Excel expert. It's more involved than the previous solution, so don't worry if there are some things that don't make sense yet.

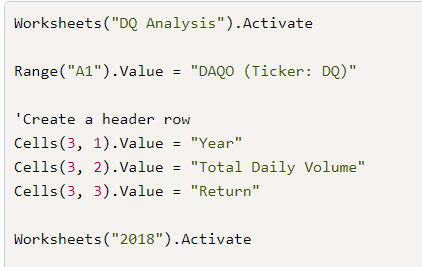
LastRow = Cells.Find("\*", searchorder:=xlByRows, searchdirection:=xlPrevious).Row

As you probably noticed, we haven't used the Find method before. It has many options, so figuring out how it works can be confusing. Let's dissect what's happening here.

* The first argument, "\*", says to look for anything. Asterisks are often used as a wildcard that means to "match anything." In this case, it means to match anything except a blank cell.
* searchorder says to search by rows, not columns.
* searchdirection set to xlPrevious means to start from the end of the worksheet range and work backward until the Find method finds something that matches, which will be in the last row.
* Finally, the Find() method returns a single cell. That cell has a row property, which is the number of the row that it is in. This is the number that we want, so we access it with the .Row property.

Keep in mind that these solutions are not the only ones out there; you may have found a different way to find the number of rows based on your search results. That's perfectly fine! Just be sure to test it out to make sure it works in your workbook. If it doesn't, go back to your Google results and find another solution.

Let's take a moment to do another code check. So far, your code should look like the code below. If not, go ahead and update it to match.



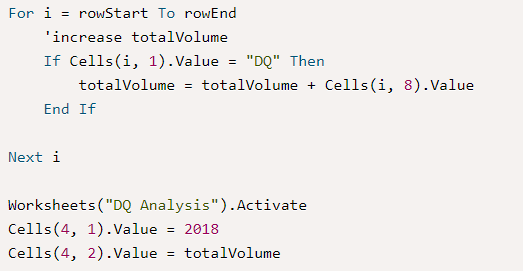
rowStart = 2

'DELETE: rowEnd = 3013

'rowEnd code taken from https://stackoverflow.com/questions/18088729/row-count-where-data-exists

rowEnd = Cells(Rows.Count, "A").End(xlUp).Row

totalVolume = 0



Let's calculate the yearly return of DQ's stock. To do this calculation, we need DQ's first closing price and last closing price.

To find the first closing price of DQ's data, we'll need to do the following:

1. Loop through all the rows.
2. Check if the current row is the first row of DQ's data.
3. If so, set the starting price to the closing price in the current row.

We already have a loop going through all the rows, so we don't need to make another loop. This means Step 1 is done. But how do we do Step 2? We need to check for two conditions:

1. If if the ticker in the current row is DQ
2. If the ticker in the previous row is *not* DQ

We can check both conditions at once using **logical operators**.

**IMPORTANT**

**Logical operators,** also called Boolean operators, link more than one condition together, which allows for more complicated conditional arguments. The logical operators in VBA are And, Or, and Not. That is:

* condition1 And condition2 will only evaluate as true if **both** condition1 and condition2 are true.
* condition1 Or condition2 will evaluate as true if **either** condition1 or condition2 are True.
* Not condition will give the **opposite** value of whatever condition is.

In addition, there is a "not equal to" **comparison operator** that checks whether two values are not equal to each other. In VBA, the "not equal to" operator is two angle brackets: **<>**

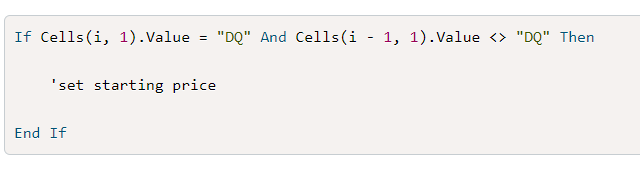
The condition to check if the current row's ticker is DQ is

Cells(i, 1).Value = "DQ"

And the condition to check if the previous row's ticker is not DQ is

Cells(i - 1, 1).Value <> "DQ"

Since we want both conditions to be true, we'll join them together with the And operator in our if statement. This new code will be inserted in the for loop directly after the if block that computes totalVolume.



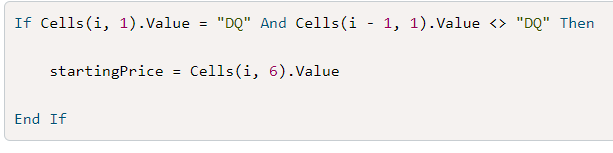
Before the for loop,  create a variable for starting price. Since the prices have decimal values, we'll use the Double data type.

Dim startingPrice As Double

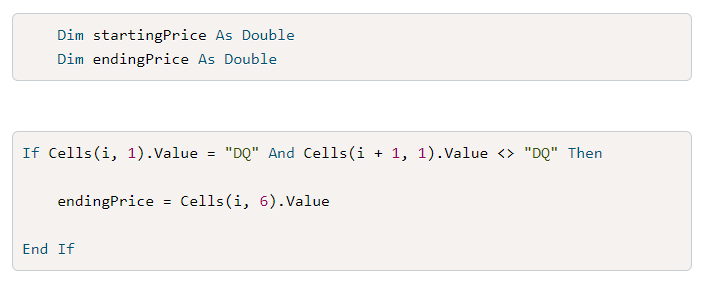
Price data is in the sixth column, so the value in Cells(i, 6) has the starting price.

startingPrice = Cells(i, 6).Value

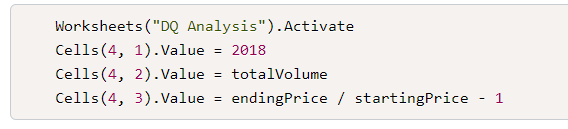
Put together, the code should look like this:



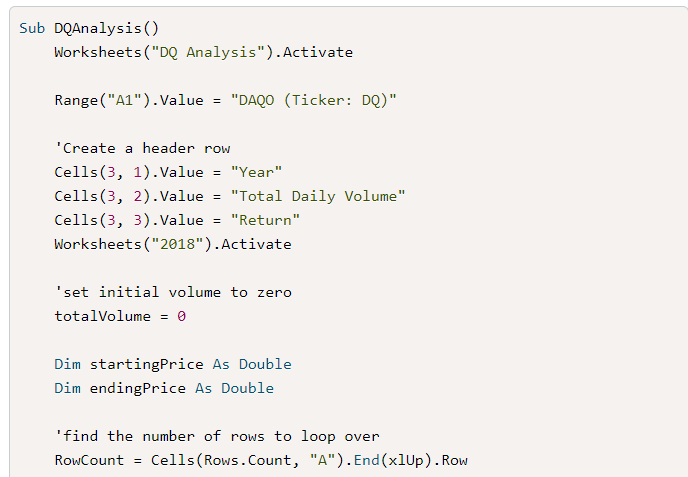
Give it a try. Your code should look like this:

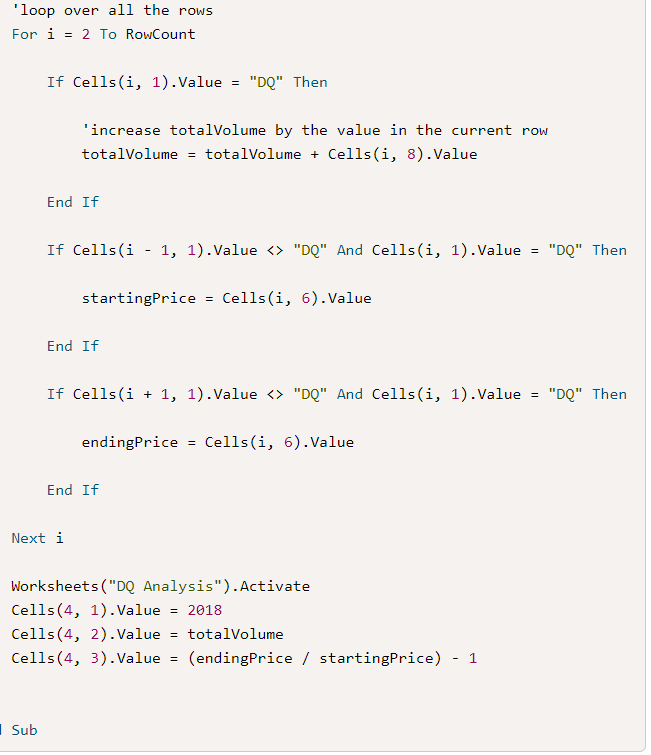


With the starting and ending prices stored, our for loop is finished. We can now add a line to our output to show the yearly return for DQ:



The finished DQAnalysis macro should look like this:



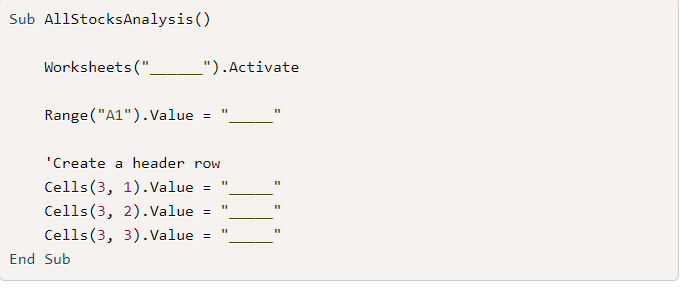


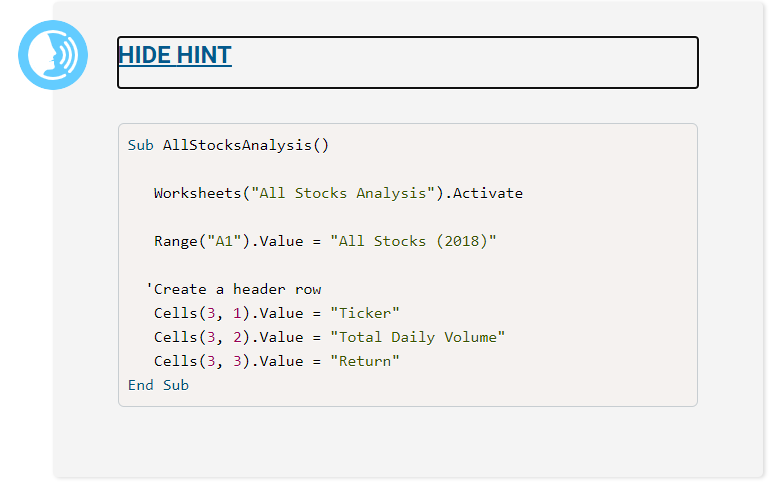
First, create a new worksheet called "All Stocks Analysis." This is where we're going to put the output for the analysis of multiple stocks. Next, create a new subroutine called "AllStocksAnalysis." The first thing we're going to do in AllStocksAnalysis() is format the output worksheet. Follow these steps:

1. Make the title in cell A1 "All Stocks (2018)."
2. Add three columns with the following headers:
   * Ticker
   * Total Daily Volume
   * Return

Before looking at the code, try writing the code on your own. Here are a few hints:

* We can reuse the code from DQAnalysis and change the text to work on AllStocksAnalysis.
* We need to complete three tasks to make the header in AllStocksAnalysis:
  + Activate the worksheet.
  + Set the text in the title cell (A1).
  + Set the text of the header row.
* Fill in the blanks of the following code:





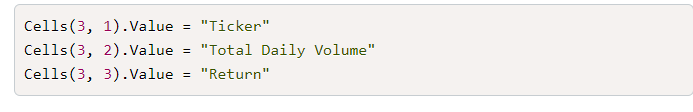
Let's go over what this subroutine is doing:

Worksheets("All Stocks Analysis").Activate

This code sets the output worksheet to be the active worksheet so that we don't accidentally overwrite cells in the wrong worksheet.

Range("A1").Value = "All Stocks (2018)"

This line of code sets the value of the cell in A1 to the string "All Stocks (2018)."



Finally, this code sets the column headings in the third row, and the first, second, and third columns (A3, B3, and C3).

**NOTE**

If your code looks slightly different than ours but still works, that's perfectly fine. Imagine if multiple authors were given the same story to tell; they would likely all tell the story in a different way. Writing code is no different. There's almost always more than one way to accomplish a given task, and every programmer will write their code a different way.

One way to accomplish our task—running an analysis of all stocks—is to copy the code from the Daqo analysis and paste it over and over, changing the ticker and the line to output each time. However, computers are better at repeating themselves than humans are, so we'll let VBA handle that part.

**NOTE**

In general, when it comes to programming, follow the advice of the acronym **DRY: Don't Repeat Yourself**. If you're reusing a piece of code over and over again, there's probably a better way to do it. After all, if you have to go back and edit your code, it's better to do it in just one place and then be done with it. When there are multiple places where you need to make the same edit, it's easy to miss one—which could result in a bug that's difficult to troubleshoot.

**Arrays in Programming**

Instead of repeating our Daqo analysis code over and over and changing the bits that are stock-specific, we need to create a list of tickers and have VBA handle reusing the code. Luckily, we can do this with a for loop and an **array**.

**IMPORTANT**

**Arrays** are another fundamental building block in programming languages. In some languages, arrays are known as **lists** because they work like a shopping list. Arrays hold an arbitrary number of variables of the same type. Instead of giving each variable a different name, we give the array a name and then access the individual variables by their index.

An **index** of a variable is its position in the array. Almost all programming languages start their index at zero, which means that a variable at index 1 will be the second element in the list. To conceptualize this, think about how floors of buildings in the U.S. are numbered compared to buildings in Great Britain. In the U.S., the floor level of a building is usually referred to as the "first floor," and the floor above that the "second floor," and so on. In Great Britain, however, the floor level is called the "ground floor," and the floor above it is considered the "first floor." In VBA, we could say indexes follow the British model.

## Arrays in VBA

In VBA, arrays are initialized with the Dim keyword, but with a couple of key changes:

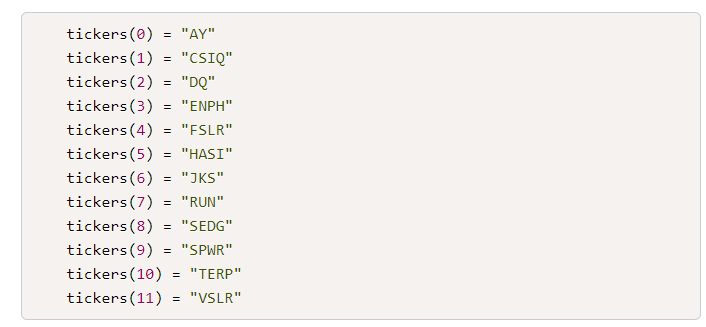
* Insert a number in parentheses after the array name that represents the number of elements in the array.
* Specify the type of variable for each element in the array.

For example, the code to create an array to hold 12 tickers would be:

Dim tickers(12) As String

To access the elements in the array, we'll put the index in parentheses after the array name tickers. Don't forget that we're starting at zero. This means tickers(0) will be the first element in the list and tickers(11) will be the 12th and final element of the array. When we reference a single element in the array, we treat it like any other variable: we assign it a value, check if it equals another variable, and so on.

Let's assign each of the tickers to an element in the array. That code should look like this:

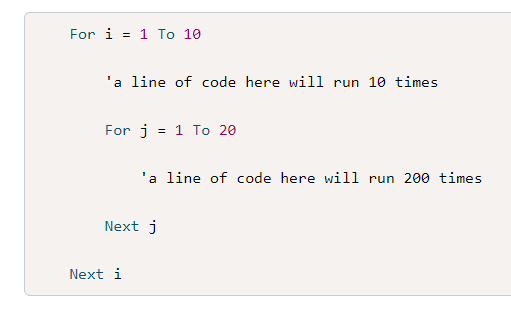


## Nested Loops

Now we're going to loop through the array. For each element in the array, we'll run the same analysis we did for DQ. This means we'll be running a for loop inside of another for loop! Loops inside loops are called **nested** **loops**.

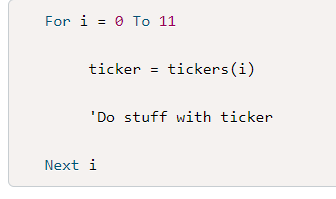
In theory, we can nest as many for loops as we want—but that means it can quickly become difficult to keep track of which loop you're working in. This is another scenario where clean code formatting comes into play. Remember whitespace? This is a great time for it. We'll add one more indentation in the code each time we begin a new inner loop. This way, when we're writing our code, we'll know which loop the code belongs to.

Additionally, we need to make a new iterator variable for our inner loop. A simple nested for loop with proper indenting will look something like this:



Note that we must end the inner loop before we can end the outer loop; otherwise, VBA will get confused, give up, and throw a compile error.

To loop through all of the stocks, let's use the index i to access each element in the array. The code will look like this:



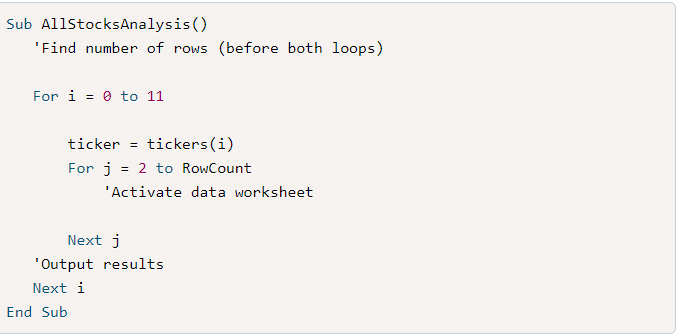
To write this macro, we can reuse a lot of the code we wrote for DQ. However, we'll need to determine which lines of code should go inside the loop, and which lines of code should go outside the loop. For example, we don't need to get the number of rows in the 2018 sheet for every ticker; it will be the same, so we'll put that outside both loops.

**REWIND**

When you reuse code, you are essentially recognizing abstract patterns: you are using code already written to solve one problem and applying it to a different problem.

When we're done with the analysis for a ticker, we'll need to output the results, which means we'll be activating the output worksheet. When we start on a new ticker, we'll need to reactivate the data worksheet, so that code goes inside the innermost loop.

Using comments to show where we're going to put our code is a good idea. So far it would look something like the following. (Hold off on making any changes to your code just yet!)



**NOTE**

We can't initialize a variable more than once because VBA will assume that we're trying to create a new variable and accidentally gave it the name of an existing variable. Therefore, we don't want to put our Dim statements inside loops.

Before we start putting code into our new loop structure, we should formulate a plan. We'll use this plan to keep our code blocks organized, using comments as our structure.

We can reuse a lot of the code we've already written in the DQAnalysis subroutine, but we'll need to rearrange it to fit our new ticker loop. Remember, we want to perform the same kind of analysis we did for DQ, but for every stock in our ticker list. We also don't want to waste time rewriting code that we've already written.

Let's write our plan.

## Map Out a Plan

Since this code might get a little complicated, we should start by writing a basic outline of the program flow. Then we'll use comments to organize it all before we write the actual code.

Our new macro should do the following:

1. Format the output sheet on the "All Stocks Analysis" worksheet.
2. Initialize an array of all tickers.
3. Prepare for the analysis of tickers.
   * Initialize variables for the starting price and ending price.
   * Activate the data worksheet.
   * Find the number of rows to loop over.
4. Loop through the tickers.
5. Loop through rows in the data.
   * Find the total volume for the current ticker.
   * Find the starting price for the current ticker.
   * Find the ending price for the current ticker.
6. Output the data for the current ticker.

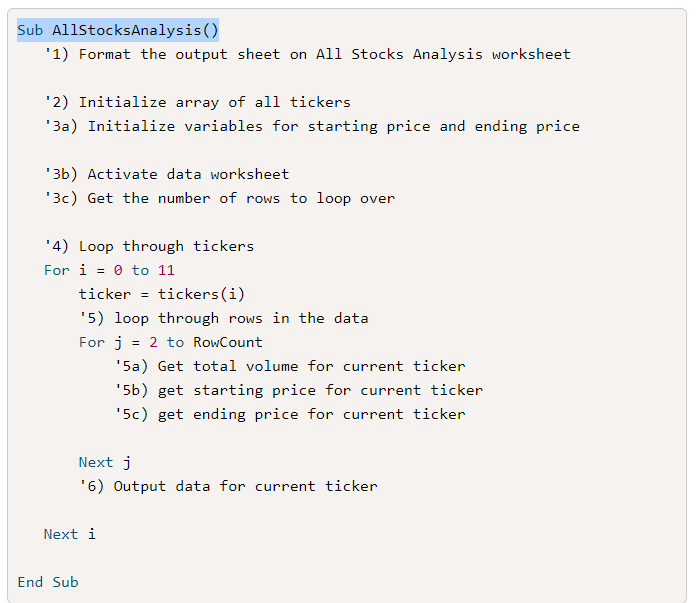
## Write the Macro

Let's put the plan into action. The following video provides an overview of this process from start to finish so you know what your code should look like.

Now you are going to put fingers to keys. Every coder goes through the experience of planning out what they're going to code, writing every line, and then running their program—only to find out it doesn't even run. Or it runs, but gives the wrong answer. This is where debugging comes into play.

**Debugging** is the process of going through your code, figuring out why it's not doing what you expected, and fixing the error. Debugging is something you've probably done already, but maybe you didn't know it had a name. If this happens to you, just remember that debugging is a big part of being a coder.

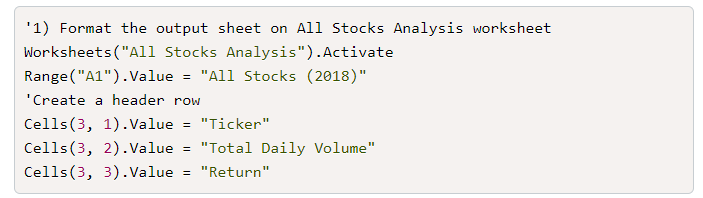
With both loops written in code, our game plan looks like this:



### Step 1: Format the Output Sheet on the "All Stocks Analysis" Worksheet

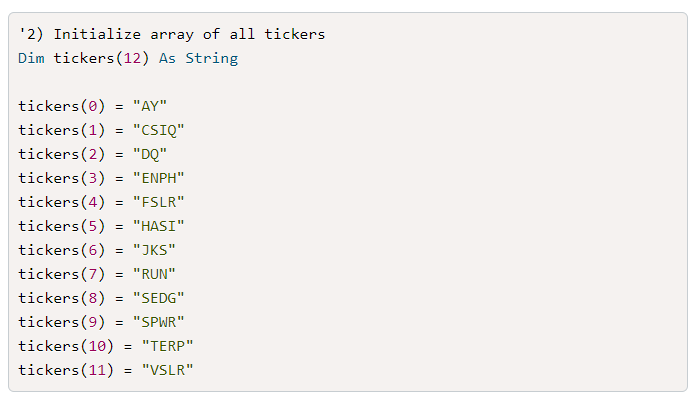
Copy the code from DQAnalysis and make the following changes:

* Activate "All Stocks Analysis" instead of "DQ Analysis."
* Change the A1 value to "All Stocks (2018)."
* Change the first column header to "Ticker."



### Step 2: Initialize an Array of All Tickers

There's no shortcut for this step. We have to type every assignment of a ticker to the array:

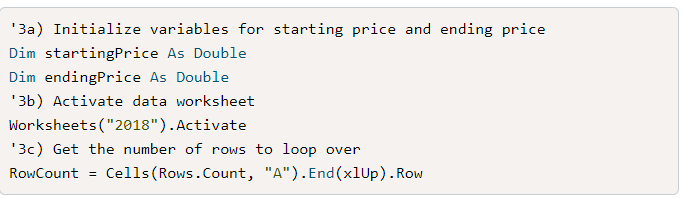


### Step 3: Prepare for the Analysis of Tickers

In this step, we'll do the following:

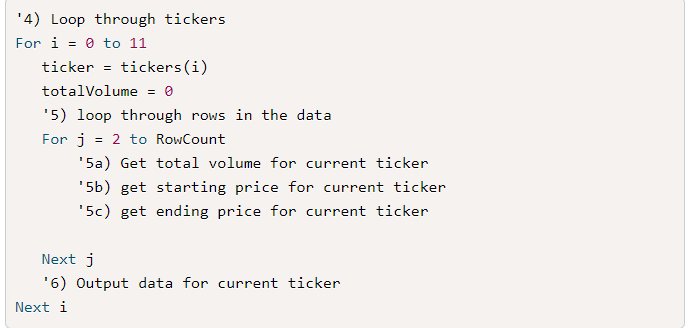
* Initialize variables for the starting price and ending price.
* Activate the data worksheet.
* Get the number of rows to loop over.

We can use the some of the code from DQAnalysis as-is.



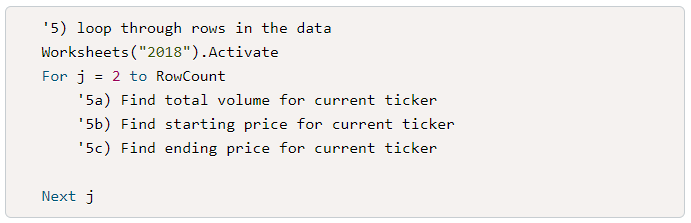
### Step 4: Loop Through the Tickers

Before we get to the inner loop, we need to consider any values that need to be initialized before the inner loop starts. Every time we finish analysis on one ticker, we need to reset the total volume to zero. This means the line totalVolume = 0 is inside the ticker loop, but outside of the row loop. Add it now.



### Step 5: Loop Through Rows in the Data

Now we can focus on the inner loop. Before starting the loop, make sure that the right worksheet is active by using the Worksheets().Activate method.

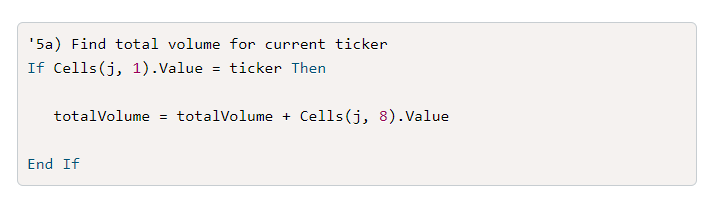


Step 5 consists of three parts:

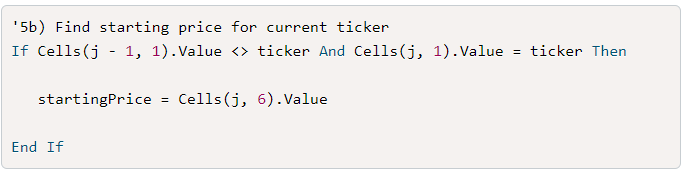
* Find the total volume for the current ticker.
* Find the starting price for the current ticker.
* Find the ending price for the current ticker.

For these steps, we can copy the code from DQAnalysis, but be careful! Now j is the variable iterating over the tickers, so we'll have to change every i reference to j after we copy and paste.

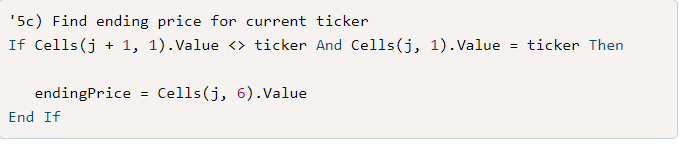
##### Find the total volume for the current ticker:



##### Find the starting price for the current ticker:

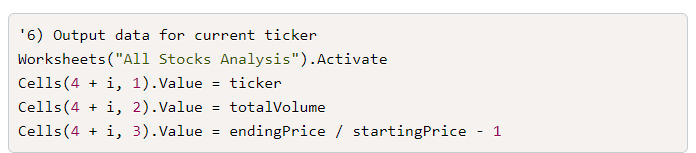


##### Find the ending price for the current ticker:

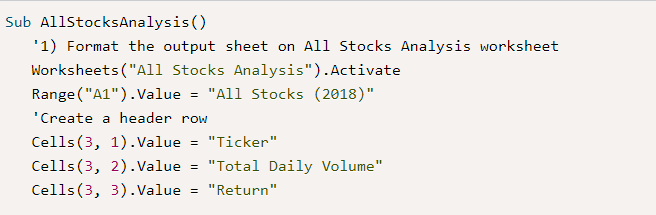


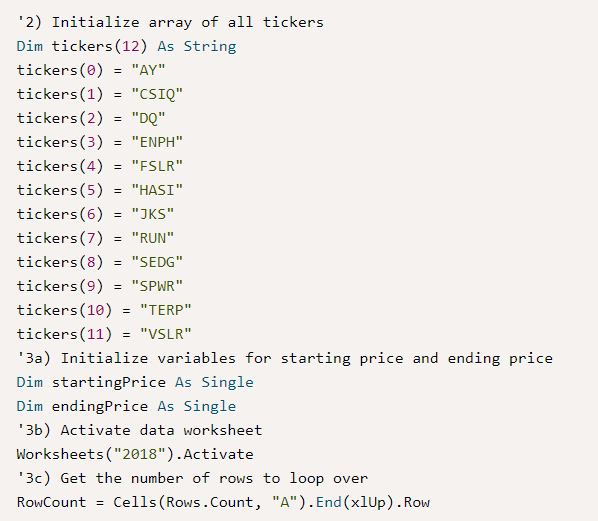
### Step 6: Output the Data for the Current Ticker

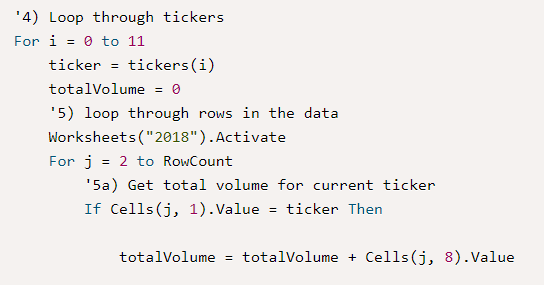
Finally, we need to slightly alter the code so that the output for each ticker prints on a new row. This is a case where using Cells() is much easier than using Range(). Instead of printing on the 4th row only, we print on the 4th row plus i.

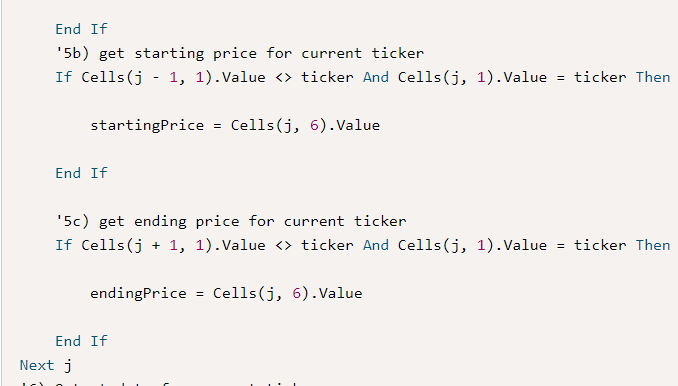


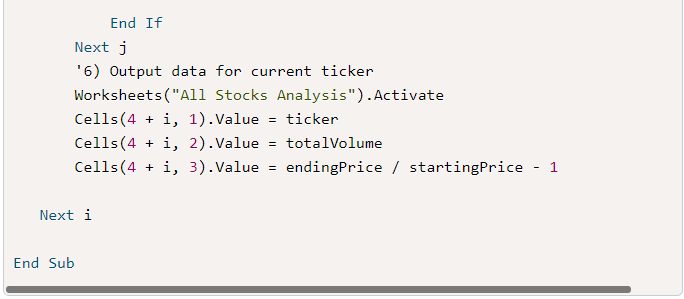
The macro should now look like the following. Compare your code to this and make sure you haven't missed anything.

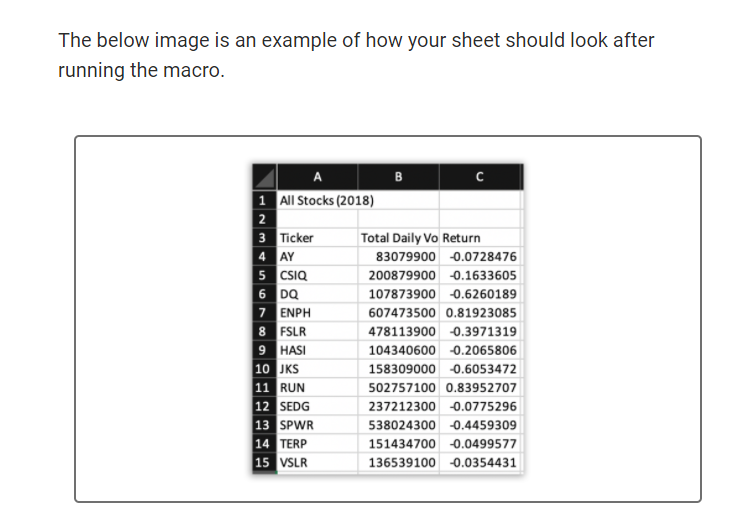






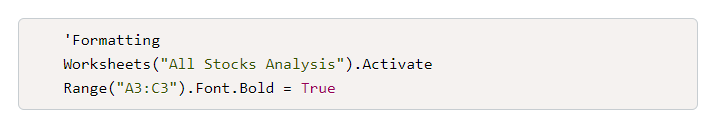






## Visual Style Formatting

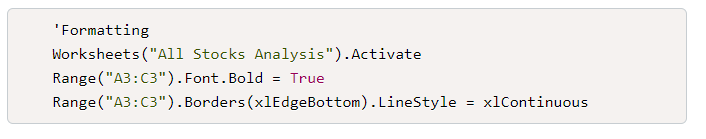
Select the header range and make the text bold by setting the Bold property of the Font property to True.



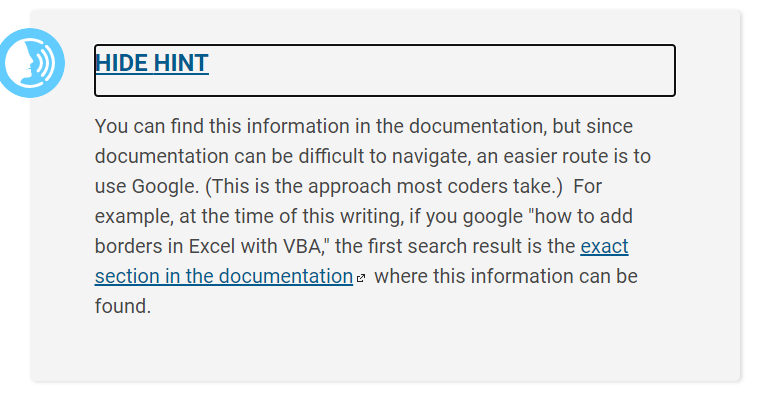
Select the same range and add a border to the bottom edge with the following code:

.Borders(xlEdgeBottom).LineStyle = xlContinuous

Here's how it looks within the context of the surrounding code:



This is a small example of how to format cells (go ahead and run your macro to test it!). There are many sources of information about Excel formatting. Think about where you could find them. If you need some help, here is a hint.



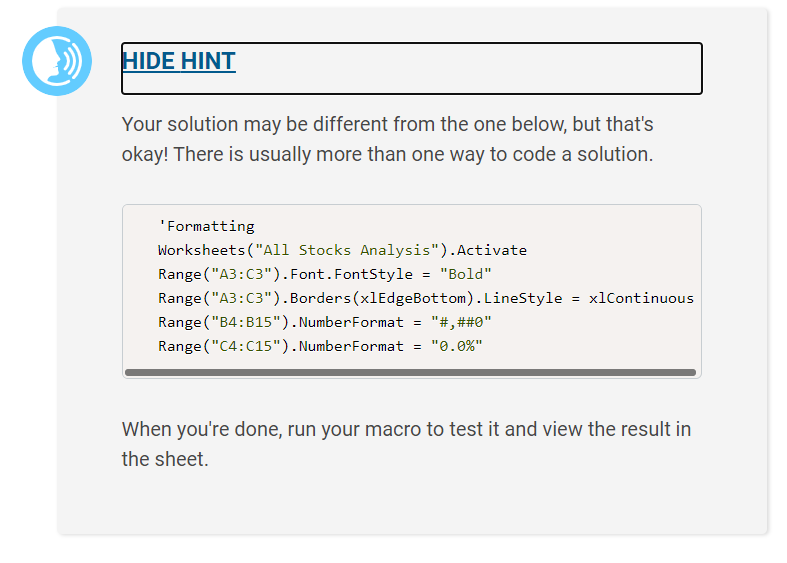
**Numeric Formatting**

Range objects have a NumberFormat property that we can change by setting it equal to a special string of characters. The NumberFormat string for separating digits with commas is "#,##0", which we'll use for the volume. To make a single-digit percentage for the return, we'll use the format "0.0%".

**NOTE**

In VBA number formats, the number sign (#) is used if you want to display only a significant digit (i.e., any digit that isn't a trailing zero). If you want to display a trailing zero, use the numerical symbol zero (0) instead. Adding a percent sign will move the decimal point two digits to the right.

Now let's write some code that will format the numeric cells in your sheet. Try it out on your own first, then check your solution against the one in the hint below.



Here we've used one **digit of precision** in the return amount. This implies that the percentage change is accurate up to a tenth of a percent. Excel will round the value when displaying it, but it won't affect the actual value stored in the cell.

**NOTE**

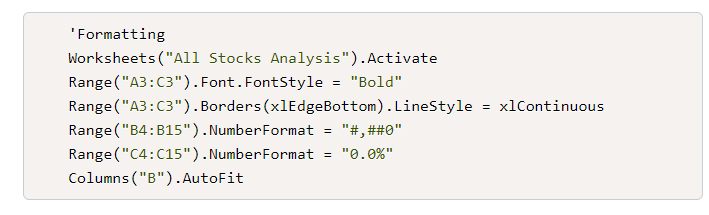
Are 1.0 and 1.00 the same? Yes and no. They both store the same exact value: 1. However, these numbers imply different things.

"1.0" implies accuracy up to the tenths place; the true value could be anywhere between 0.95 and 1.05. "1.00" implies that the value is accurate up to the hundredths place, so its implied true value lives in a much narrower range: 0.995 to 1.005.

In scientific settings, the number of **digits of precision** is the total number of significant digits, but in financial and engineering applications, the digits of precision refer to how many digits are written after the decimal point.

## Automatically Fit Column Widths

We can change the width of a column to auto-fit the data using the AutoFit property, which is convenient. However, we'll need to select the column, not just the range. Luckily, that's pretty convenient too: it's just the function Columns. Select column B with Columns("B"), then run the macro and check its result.



Now that we've made static formatting updates, let's use this tool with the programming tools we already know to create conditional formatting.

## Color Formatting

The color of a cell belongs to the Interior property of the cell. We'll need to access the Color property of the Interior to change the color of the cell. There are a few ways to refer to colors in VBA, but the simplest and easiest to read are the VBA standard colors. For most basic color names, you can put vb before the color name, and VBA will refer to that color. For example, the standard VBA color green is vbGreen. However, the color of an empty cell isn't set to white, even though it's usually displayed as white. To clear the color, it should be set to xlNone.

Therefore, to set the color of the cell at row 4, column 3 to green, we write:

Cells(4, 3).Interior.Color = vbGreen

To set the color of that cell to red, we write:

Cells(4, 3).Interior.Color = vbRed

And to clear the cell, we write:

Cells(4, 3).Interior.Color = xlNone

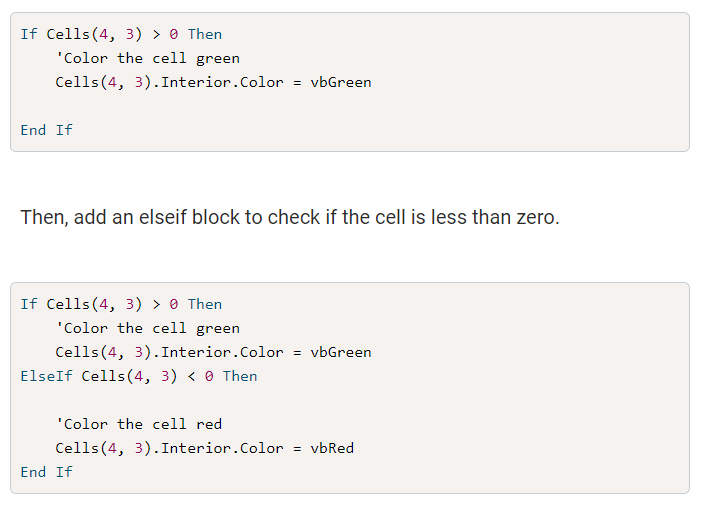
## Conditionals Revisited

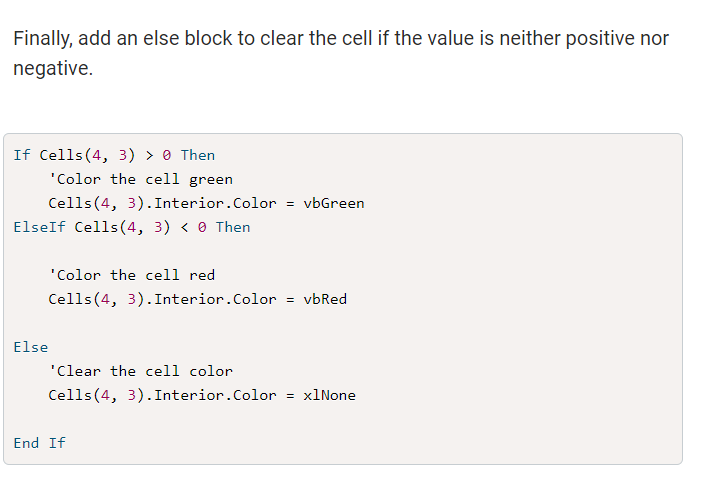
We could use separate if-then statements for each condition, but we want to cover all cases, including ones we may not have thought of. For example, what if the formula returns an error? How would this be formatted?

If-then statements have a couple more tricks up their sleeve, namely, the else statement and the elseif statement.

Just as the if-then statement creates code that runs if a condition is true, the **else statement** runs a separate block of code if a condition is false. The **elseif statement** introduces another condition: it runs if the first condition is false and the second condition is true.

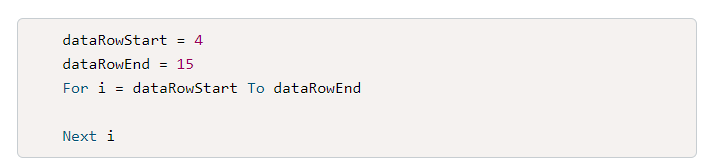
To see an example, let's apply conditional formatting to the cell in row 4, column 3. First, to color the cell green, create an if-then block that checks if the value is greater than zero.





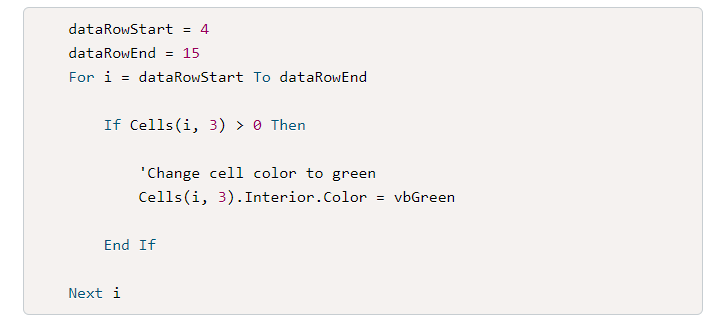
## Set Up the Loop

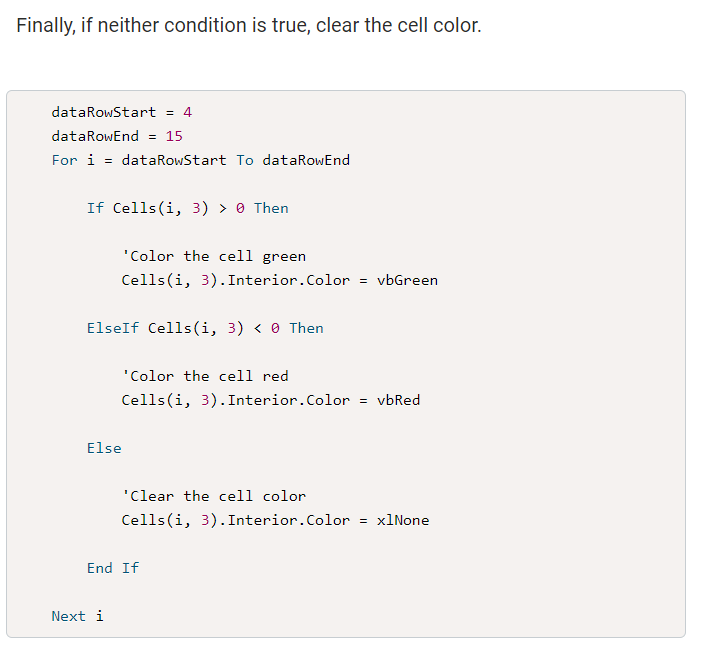
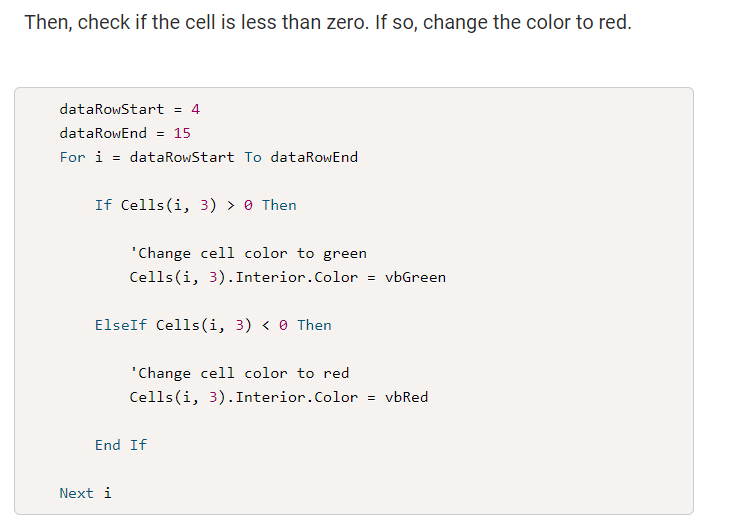
Now that we know how to conditionally format one cell, we can conditionally format all the cells we need with a loop. The data starts on row 4 and goes to row 15, so we'll create two new variables to hold those values—remember, we don't want unnamed magic numbers in our code—and have the for loop iterate through each row.



Inside the loop, we'll use the same code from the previous section but replace the row number with the iterator.

First, check if the cell is greater than zero. If so, change the color to green.





Let's get some added practice with for loops, conditionals, and formatting.

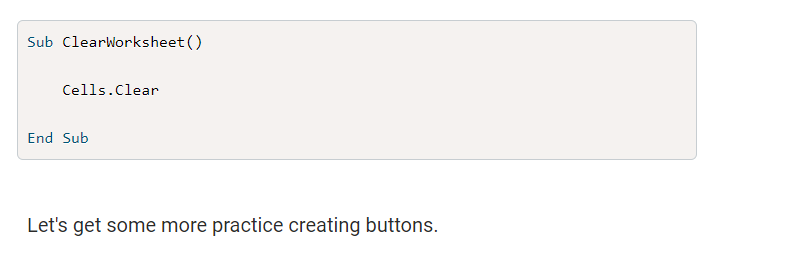
## Create a Button

The following videos walk you through the process of creating a button to run your analysis. Choose the video appropriate for your operating system.

## Clear the Worksheet

To test if the button is working correctly, first clear the worksheet. This is something that can be done by hand, but since we're now VBA pros, let's create a new macro and button to clear the worksheet for us.

Return to the VBA code and add a simple macro:



**Store the Value from InputBox**

We can store the value entered by assigning InputBox to a variable. Add the following line of code at the very beginning of a new macro (named yearValueAnalysis):

yearValue = InputBox("What year would you like to run the analysis on?")

Note that even though we're entering a number, InputBox will return a value with a string data type. Luckily, the lines of code we'll change already treat the year as a string.

**Replace Hard-Coded Values**

In this macro, there are three lines of code that depend on the year; let's change all of those to use the yearValue variable instead. The first line is:

Range("A1").Value = "All Stocks (2018)"

To change this from assigning a hard-coded string literal to using the dynamic value stored in yearValue, we need to build the string in pieces. The tool to do this is **concatenation**.

Concatenation refers to the process of joining two or more strings together. In VBA, we can concatenate strings using the "+" operator. The new line will read:

Range("A1").Value = "All Stocks (" + yearValue + ")"

There are two other lines where we need to replace the hard-coded year value. These lines activate the worksheet with the stock data: first to get the row count, and then inside the for loop to make sure we're in the right worksheet. Those lines should now both read:

Worksheets(yearValue).Activate

**NOTE**

Make sure to leave the initial .Activate in the macro as-is: Worksheets("All Stocks Analysis").Activate.

**Run the Macro**

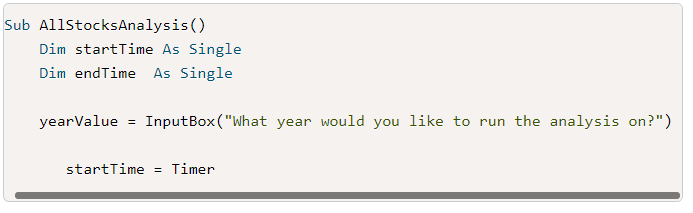
You're done. Go ahead and run the macro for both years to make sure that it's working correctly.

To get the amount of time it will take to run Steve’s AllStocksAnalysis script, we need to capture the start time and end time of the executed code. Lucky for us, VBA has a Timer function!

## Get the Start and End Time

To get the start and end time we will need to initialize two variables, startTime and endTime, and then set each variable equal to the Timer function.

In the AllStocksAnalysis script, add the startTime and endTime variables as Single data types underneath the Sub AllStocksAnalysis() subroutine. Next, underneath the yearValuevariable set the startTime variable equal to the Timer function, which will allow us to start the clock!



In the code above, we are setting the startTime variable equal to the Timer function after the yearValue variable because we want to start the clock after we have entered the year in the InputBox() command.

Next, scroll to the end of your AllStocksAnalysis script. After the last Next iand before the End Sub command, set the endTime variable equal to the Timer function. Finally, create a message box statement that will tell us how long the code took to run by subtracting the startTime from the endTime for the analysis.



endTime = Timer

MsgBox "This code ran in " & (endTime - startTime) & " seconds for the year " & (yearValue)

End Sub

**Displaying the Elapsed Time**

When you run the macro for both years, a message box similar to the images below will pop up and show the elapsed time for 2017 and 2018.

**NOTE**

The first time you run a macro, the elapsed time may be longer than subsequent runs because computer resources need to be allocated to run the macro. Once allocated, these resources are ready for subsequent runs.

