# 2.0.1: Make Your Way With VBA

Welcome to Module 2! In this module, we'll explore Visual Basic for Applications, or VBA, a programming language for Microsoft Office applications. Within the context of VBA, you'll learn core coding concepts that will help you perform more complex analyses. Watch the following video to get started.

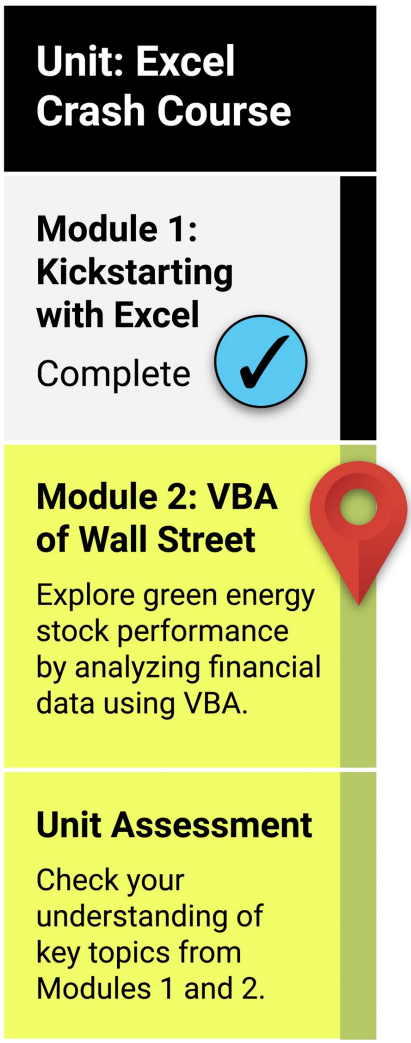
# 2.0.2: Module 2 Roadmap

## Looking Ahead

In this module we'll use Visual Basic for Applications, or VBA, to add even more analytical power to Excel. VBA is a programming language that interacts directly with Excel worksheets and cells, allowing us to write scripts to automate simple tasks. It also allows us to write complicated scripts to perform complex analyses, which we wouldn't be able to do with Excel alone.

Within the context of VBA, you'll learn the fundamental building blocks of programming languages. Learning to code is essentially learning how to deconstruct a problem and translate the solution into simple instructions. Previously, Excel helped us start thinking about data differently; now, VBA will help us start thinking about how to programmatically analyze that data.

Before beginning this module, you should have a strong command of Excel workbooks, Excel cell references, and built-in Excel functions.




## What You Will Learn

By the end of this module, you will be able to:

- Create a VBA macro that can trigger pop-ups and inputs, read and change cell values, and format cells.
- Use `for` loops and conditionals to direct logic flow.
- Use nested `for` loops.
- Apply coding skills such as syntax recollection, pattern recognition, problem decomposition, and debugging.

---

# Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction to VBA (15 minutes)
- Become an Excel Developer with VBA (1 hour)
- Analyze Stock Data with VBA (2 hours)
- Analyze Multiple Stocks (2 hours)
- Make the Worksheet Readable (2 hours)
- Make the Worksheet Interactive (1 hour 15 minutes)
- Application (5 hours)

# 2.0.3: VBA of Wall Street

VBA is often used in the financial industry, so you'll be working with stock data in this module. Watch the following video to learn about the background of the project you'll be working on.

# 2.1.1: Install Developer Tools

Steve is somewhat savvy in Excel, but he’s asked you to help him with his analysis. This is a great opportunity to use VBA . But first, you’ll need to install the tools necessary to access it in Excel.

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.

## macOS

# Windows

### Add, Commit, Push

Put your new GitHub skills to the test by creating a new repository for this project. Name the repository "stock-analysis." We'll use this repository to back up our work. As you work through this module, remember to commit your work often! Reminders to "add, commit, push" are provided periodically at natural stopping points.

# 2.1.2: Download the Dataset and Set Up the VBA Editor

In order to help Steve with the analysis, we need to make sure that VBA is up and running correctly on our machine. We also need to download the file containing the stock data.

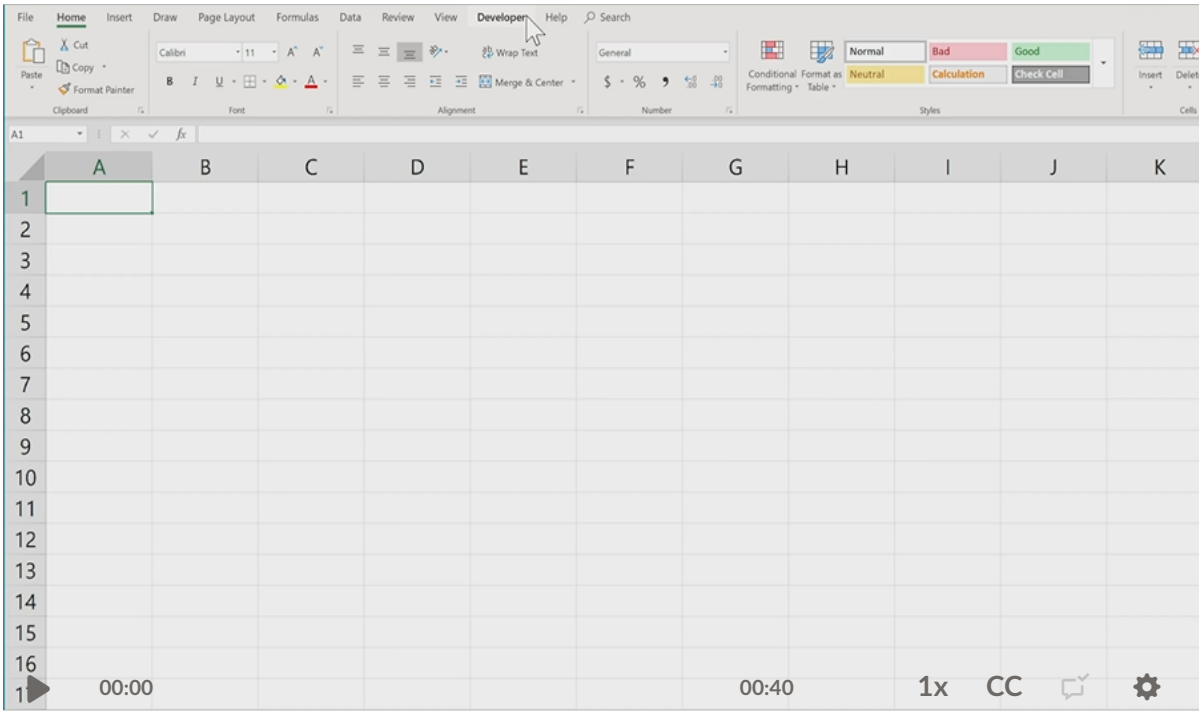
Before getting started, click the link below to download the dataset you'll be working within this module: green_stocks.xlsx.

**Download the dataset.**
**(https://courses.bootcampspot.com/courses/138/files/16079/download?wrap=1)**
**(https://courses.bootcampspot.com/courses/138/files/16079/download?wrap=1)**

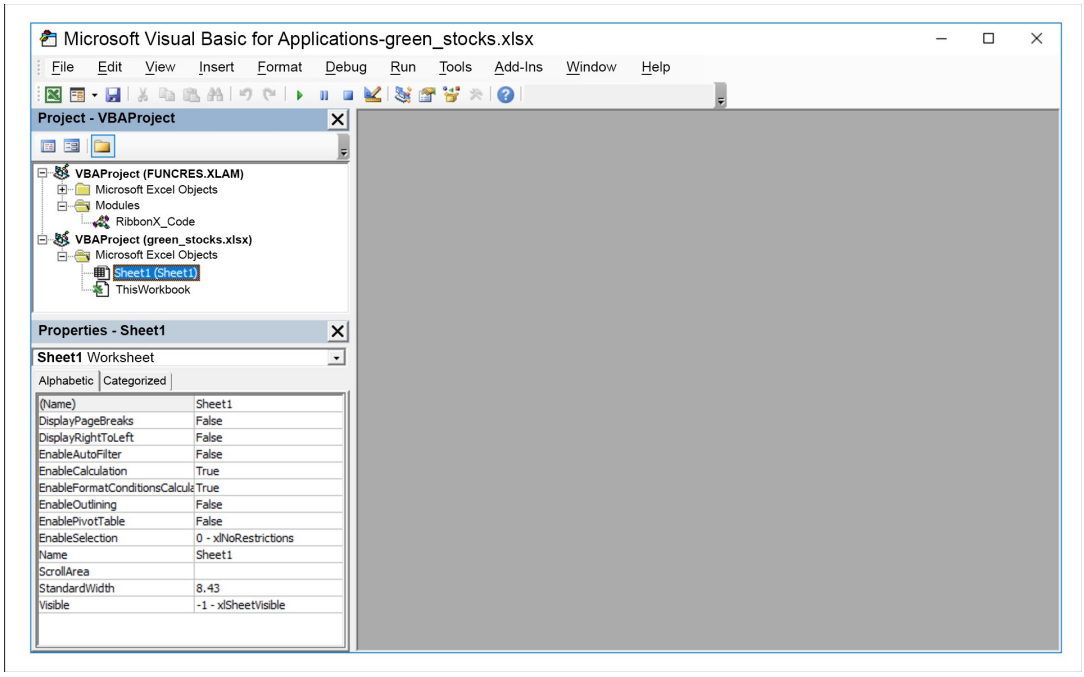
When the file is downloaded, we can get started with VBA. The best way to test if a new language is working correctly is to write a simple program and run it. So let's write our first program! The first step is to open the VBA editor. The following videos walk you through the process of opening and setting up the VBA editor. Watch the video for your operating system.

## macOS

# Windows




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.

# 2.1.3: Enable Macros

Steve realizes he will need to enable macros if he wants to run the analysis on new data in the future. Let's help him enable macros on his computer now so that he can run the analysis whenever he wants.

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.

## macOS

## Windows

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.

# 2.1.4: Create a Simple Test Macro

We need to make sure we installed VBA correctly and that we'll be able to write VBA code for Steve. The best way to make sure that a new programming language is installed correctly is to write a simple program (or macro), run it, and make sure that everything ran correctly. So let's write a simple macro and make sure that it runs correctly.

## 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 ($10^2$), 4 tens ($10^1$), and 7 ones ($10^0$). 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 ($2^5$), 1 four ($2^2$), and 1 two ($2^1$), 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 $2^{15}$, 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.xlsm` window. Watch the following demo to see this process in action. Choose the video for your operating system.

## macOS

## Windows

You've written your first VBA program. Nice work!

> ### Add, Commit, Push
>
> Be sure to save your changes to `green_stocks.xlsm`. After you've saved the file, upload it to the "stocks-analysis" repository you created on GitHub.

# 2.2.1: Create a Worksheet for Your Analysis

Now that we know that VBA is working correctly, let’s start analyzing some stock data. Steve wants to find the total daily volume and yearly return for each stock. Daily volume is the total number of shares traded throughout the day; it measures how actively a stock is traded. The yearly return is the percentage difference in price from the beginning of the year to the end of the year. Steve’s parents are starting to pester him about Daqo’s stock, so we’ll start with DQ.

## Work with Worksheets

We've enabled and tested macros, so now we can start analyzing actual stock data. First, we'll need a worksheet to hold this data. The following video walks you through the process of setting up a worksheet for your analysis. Watch the video for that corresponds to your operating system.

### macOS

## Windows

# 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:

```
Sub DQAnalysis()
    Worksheets("DQ Analysis").Activate

    Range("A1").Value = "DAQO (Ticker: DQ)"

End Sub
```

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.

```
Sub DQAnalysis()
    Worksheets("DQ Analysis").Activate

    Range("A1").Value = "DAQO (Ticker: DQ)"

    Cells(3, 1).Value = "Year"
    Cells(3, 2).Value = "Total Daily Volume"
    Cells(3, 3).Value = "Return"
End Sub
```

Let's get some practice using the `Range()` and `Cells()` methods.

### Skill Drill

Using only `Range()` method, rewrite `DQAnalysis` so that it creates the same output. Then rewrite it again, using only the `Cells()` method.

### Add, Commit, Push

Save your changes to `green_stocks.xlsm` and upload it to the "stocks-analysis" repository in GitHub. GitHub will update the file to the new version while keeping a history of all the changes we've made.

# 2.2.2: Write Readable Code

Readable code is code that is clear and well-documented. Writing readable code helps not just the person reading it, but also the programmer who is writing it in the first place—especially if he or she needs to take a break from the project for any period of time. You may already know from firsthand experience that stepping away from a project, even for a short time, can cause you to forget everything you worked on! Readable code helps us get up to speed more quickly. This is something to keep in mind as we work on our project for Steve. He might ask us to change or revisit the code in the future, so let's focus on writing clean, readable code. How do we do that? Let's find out.

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:

```
Sub DQAnalysis()
    Worksheets("DQ Analysis").Activate

    Range("A1").Value = "DAQO (Ticker: DQ)"
```

```
    Cells(3, 1).Value = "Year"
    Cells(3, 2).Value = "Total Daily Volume"
    Cells(3, 3).Value = "Return"
End Sub
```

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 `Cells` lines are doing.

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

```
Sub DQAnalysis()
    Worksheets("DQ Analysis").Activate

    Range("A1").Value = "DAQO (Ticker: DQ)"

    'Create a header row
    Cells(3, 1).Value = "Year"
    Cells(3, 2).Value = "Total Daily Volume"
    Cells(3, 3).Value = "Return"
End Sub
```

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.

# 2.2.3: Find Total Daily Volume for DQ in 2018

Steve's parents want to know how actively DQ was traded in 2018. They believe that if a stock is traded often, then the price will accurately reflect the value of the stock. If we sum up all of the daily volume for DQ, we'll have the yearly volume and a rough idea of how often it gets traded.

At this point, we have an output worksheet to hold our analysis and know how to write readable code. Now we can start writing more complex code to perform our analyses. In this section, we'll dive into loops, conditionals, code patterns, and how to find coding help on the internet. Let's get started!

## 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 it 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:

```
For i = 1 To 3
    MsgBox (i)
```

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:

```
For i = 1 To 3
    MsgBox (i)

Next i
```

Make a new macro and 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)`.

```
For i = 1 To 8
    MsgBox (Cells(1, i))

Next i
```

Make sure the 2018 sheet is active and then run the `for` loop. 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:

```
totalVolume = 0
For i = 2 To 3013
    'increase totalVolume

Next i
```

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:

```
rowStart = 2
rowEnd = 3013
```

```
totalVolume = 0

For i = rowStart To rowEnd
    'increase totalVolume

Next i
```

> **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)`:

```
rowStart = 2
rowEnd = 3013
totalVolume = 0

For i = rowStart To rowEnd
    'increase totalVolume
    totalVolume = totalVolume + Cells(i, 8).Value

Next i
```

To check that `totalVolume` increased, display the value with `MsgBox()`.

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.

# 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`.

```
If condition Then
    'Run code if condition is true
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.

```
If 2 < 3 Then
    'This code will always run
End If
```

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

```
If 3 < 2 Then
    'This code will never run
End If
```

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:

```
rowStart = 2
rowEnd = 3013
totalVolume = 0

For i = rowStart To rowEnd
    'increase totalVolume if ticker is "DQ"
    If Cells(i, 1).Value = "DQ" Then
        totalVolume = totalVolume + Cells(i, 8).Value
    End If

Next i
```

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:

```
    Worksheets("DQ Analysis").Activate
    Cells(4, 1).Value = 2018
    Cells(4, 2).Value = totalVolume
```

Now is a good time to save your work. Don't forget to save your work often! Then run your code to make sure everything is working.

> **Finding**
> Check the total daily volume in the "DQ Analysis" worksheet. You should see that DQ traded 107,873,900 shares in 2018.

> **Add, Commit, Push**
> If everything is working correctly, save your changes to `green_stocks.xlsm` and upload it again to the "stocks-analysis" repository in GitHub. Saving incremental changes to files on GitHub is called **pushing** to the repository.

# 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:

```
totalValue = 0

For i = Start To finish
```

```
    If Condition Then
        totalValue = totalValue + currentValue
    End If

Next i
```

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** **(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 (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.
- `.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.
- `.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.

```
'rowEnd code taken from https://stackoverflow.com/questions/18088729/row-cou
rowEnd = Cells(Rows.Count, "A").End(xlUp).Row
```

# 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
```

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.

# 2.2.4: Get DQ's Yearly Return for 2018

Steve wants to know how DQ performed in 2018. One way to measure this is to calculate the yearly return for DQ. The **yearly return** is the percentage increase or decrease in price from the beginning of the year to the end of the year. In other words, if you invested in DQ at the beginning of the year and never sold, the yearly return is how much your investment grew or shrunk by the end of the year.

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.

```
If Cells(i, 1).Value = "DQ" And Cells(i - 1, 1).Value <> "DQ" Then

    'set starting price

End If
```

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:

```
If Cells(i, 1).Value = "DQ" And Cells(i - 1, 1).Value <> "DQ" Then

    startingPrice = Cells(i, 6).Value

End If
```

To find the ending price, follow the same code pattern.

> **Rewind**
>
> That's right: we're reusing a design pattern! We'll apply the pattern used to find the starting price to our process for finding the ending price by following these steps:
>
> 1. Initialize a variable to store ending price as a double data type.
> 2. Check if the current row is the last row of DQ's data:
>    - Check that the current row's ticker is DQ.
>    - Check that the next row's ticker is not DQ.
> 3. If so, set the ending price to the current row's closing price.

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

```
    Dim startingPrice As Double
    Dim endingPrice As Double
```

```
If Cells(i, 1).Value = "DQ" And Cells(i + 1, 1).Value <> "DQ" Then

    endingPrice = Cells(i, 6).Value

End If
```

With the starting and ending prices stored, we can now add a line to our output to show the yearly return for DQ:

```
    Worksheets("DQ Analysis").Activate
    Cells(4, 1).Value = 2018
    Cells(4, 2).Value = totalVolume
    Cells(4, 3).Value = endingPrice / startingPrice - 1
```

The finished `DQAnalysis` macro should look like this:

```
Sub DQAnalysis()
    Worksheets("DQ Analysis").Activate

    Range("A1").Value = "DAQO (Ticker: DQ)"

    'Create a header row
    Cells(3, 1).Value = "Year"
    Cells(3, 2).Value = "Total Daily Volume"
    Cells(3, 3).Value = "Return"
    Worksheets("2018").Activate

    'set initial volume to zero
    totalVolume = 0

    Dim startingPrice As Double
    Dim endingPrice As Double

    'find the number of rows to loop over
    RowCount = Cells(Rows.Count, "A").End(xlUp).Row

    'loop over all the rows
    For i = 2 To RowCount

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

            'increase totalVolume by the value in the current row
            totalVolume = totalVolume + Cells(i, 8).Value

        End If

        If Cells(i - 1, 1).Value <> "DQ" And Cells(i, 1).Value = "DQ" Then
```

```
            startingPrice = Cells(i, 6).Value

        End If

        If Cells(i + 1, 1).Value <> "DQ" And Cells(i, 1).Value = "DQ" Then

            endingPrice = Cells(i, 6).Value

        End If

    Next i

    Worksheets("DQ Analysis").Activate
    Cells(4, 1).Value = 2018
    Cells(4, 2).Value = totalVolume
    Cells(4, 3).Value = endingPrice / startingPrice - 1


End Sub
```

Whew! That's a long piece of code. Let's walk through it again and remind ourselves what it's doing.

---

Save your code and run it to make sure everything works correctly.

> **Finding**
> Daqo dropped over 63% in 2018—yikes! Steve will definitely want to offer some better stocks to his parents.

> **Add, Commit, Push**
> Once everything is working correctly, save `green_stocks.xlsm` and push the changes to the "stocks-analysis" repository in GitHub.

# 2.3.1: Create a New Worksheet and Subroutine

Since Daqo might not be the best option for Steve's parents to invest in, let's analyze multiple stocks to find some better choices for them. A lot of the work we've already done to analyze DQ can be repurposed to analyze any stock. With a little more code, we can analyze a whole list of stocks.

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:

```
Sub AllStocksAnalysis()

    Worksheets("______").Activate

    Range("A1").Value = "_____"

    'Create a header row
    Cells(3, 1).Value = "_____"
    Cells(3, 2).Value = "_____"
    Cells(3, 3).Value = "_____"
End Sub
```

Your code should look something like this:

```
Sub AllStocksAnalysis()

    Worksheets("All Stocks Analysis").Activate

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

    'Create a header row
    Cells(3, 1).Value = "Ticker"
    Cells(3, 2).Value = "Total Daily Volume"
    Cells(3, 3).Value = "Return"
End Sub
```

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)."

```
Cells(3, 1).Value = "Ticker"
Cells(3, 2).Value = "Total Daily Volume"
Cells(3, 3).Value = "Return"
```

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.

# 2.3.2: Loop Over All Tickers

To run analyses on all of the stocks, we need to create a program flow that loops through all of the tickers.

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.

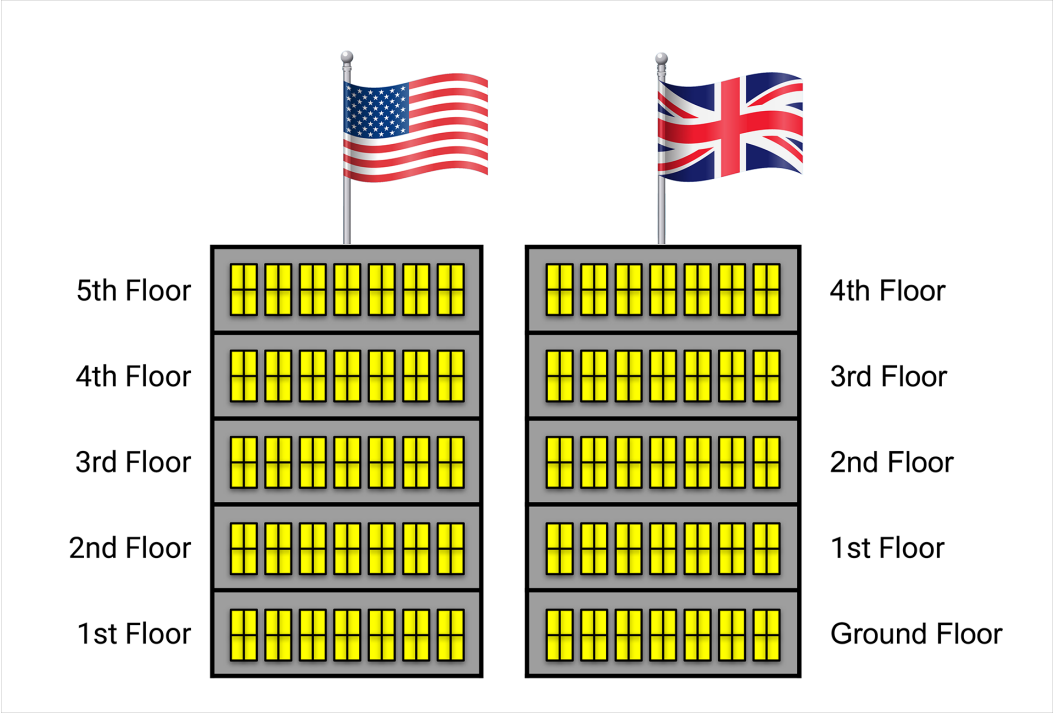
## Arrays in Programming

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:

```
tickers(0) = "AY"
tickers(1) = "CSIQ"
tickers(2) = "DQ"
tickers(3) = "ENPH"
tickers(4) = "FSLR"
tickers(5) = "HASI"
tickers(6) = "JKS"
tickers(7) = "RUN"
tickers(8) = "SEDG"
tickers(9) = "SPWR"
tickers(10) = "TERP"
tickers(11) = "VSLR"
```

# 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:

```
For i = 1 To 10

    'a line of code here will run 10 times

    For j = 1 To 20

        'a line of code here will run 200 times

    Next j

Next i
```

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:

```
For i = 0 To 11

    ticker = tickers(i)

    'Do stuff with ticker
```

```
Next i
```

Let's get some practice using nested `for` loops.

### Skill Drill

1. Create a nested `for` loop that puts a 1 in each cell from A1 to J10.
2. Modify the nested `for` loop to put the sum of the row number and column number into each cell.
3. Modify the nested `for` loop again to fill in any rectangular block of cells for *m* columns and *n* rows.

### Add, Commit, Push

Save `green_stocks.xlsm` and push the changes to the “stocks-analysis” repository in GitHub.

# 2.3.3: Reuse Code

Steve may want to look at a different set of stocks in the future. With this in mind, we should create a flexible macro for running multiple stocks. By carefully reusing the code we've already written for DQ, we can write a macro with this flexibility.

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!)

```
Sub AllStocksAnalysis()
   'Find number of rows (before both loops)

   For i = 0 to 11

       ticker = tickers(i)
       For j = 2 to RowCount
           'Activate data worksheet

       Next j
   'Output results
   Next i
End Sub
```

> **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:

```
Sub AllStocksAnalysis()
   '1) Format the output sheet on All Stocks Analysis worksheet

   '2) Initialize array of all tickers
   '3a) Initialize variables for starting price and ending price

   '3b) Activate data worksheet
   '3c) Get the number of rows to loop over

   '4) Loop through tickers
   For i = 0 to 11
       ticker = tickers(i)
       '5) loop through rows in the data
       For j = 2 to RowCount
           '5a) Get total volume for current ticker
           '5b) get starting price for current ticker
           '5c) get ending price for current ticker

       Next j
       '6) Output data for current ticker

   Next i

End Sub
```

# 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."

```
'1) Format the output sheet on All Stocks Analysis worksheet
Worksheets("All Stocks Analysis").Activate
Range("A1").Value = "All Stocks (2018)"
'Create a header row
Cells(3, 1).Value = "Year"
Cells(3, 2).Value = "Total Daily Volume"
Cells(3, 3).Value = "Return"
```

# 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:

```
'2) Initialize array of all tickers
Dim tickers(12) As String

tickers(0) = "AY"
tickers(1) = "CSIQ"
tickers(2) = "DQ"
tickers(3) = "ENPH"
tickers(4) = "FSLR"
tickers(5) = "HASI"
tickers(6) = "JKS"
tickers(7) = "RUN"
tickers(8) = "SEDG"
tickers(9) = "SPWR"
tickers(10) = "TERP"
tickers(11) = "VSLR"
```

# 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 code from `DQAnalysis` as-is.

```
'3a) Initialize variables for starting price and ending price
Dim startingPrice As Single
Dim endingPrice As Single
'3b) Activate data worksheet
Worksheets("2018").Activate
'3c) Get the number of rows to loop over
RowCount = Cells(Rows.Count, "A").End(xlUp).Row
```

# 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.

```
'4) Loop through tickers
For i = 0 to 11
   ticker = tickers(i)
   totalVolume = 0
   '5) loop through rows in the data
   For j = 2 to RowCount
       '5a) Get total volume for current ticker
       '5b) get starting price for current ticker
       '5c) get ending price for current ticker

   Next j
   '6) Output data for current ticker
Next i
```

# 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.

```
    '5) loop through rows in the data
    Worksheets("2018").Activate
    For j = 2 to RowCount
        '5a) Find total volume for current ticker
        '5b) Find starting price for current ticker
        '5c) Find ending price for current ticker

    Next j
```

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:

```
'5a) Find total volume for current ticker
If Cells(j, 1).Value = ticker Then

   totalVolume = totalVolume + Cells(j, 8).Value

End If
```

## Find the starting price for the current ticker:

```
'5b) Find starting price for current ticker
If Cells(j - 1, 1).Value <> ticker And Cells(j, 1).Value = ticker Then

   startingPrice = Cells(j, 6).Value
```

```
End If
```

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

```
'5c) Find ending price for current ticker
If Cells(j + 1, 1).Value <> ticker And Cells(j, 1).Value = ticker Then

   endingPrice = Cells(j, 6).Value
End If
```

## 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`.

```
'6) Output data for current ticker
Worksheets("All Stocks Analysis").Activate
Cells(4 + i, 1).Value = ticker
Cells(4 + i, 2).Value = totalVolume
Cells(4 + i, 3).Value = endingPrice / startingPrice - 1
```

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

```
Sub AllStocksAnalysis()
   '1) Format the output sheet on All Stocks Analysis worksheet
   Worksheets("All Stocks Analysis").Activate
   Range("A1").Value = "All Stocks (2018)"
   'Create a header row
   Cells(3, 1).Value = "Year"
   Cells(3, 2).Value = "Total Daily Volume"
   Cells(3, 3).Value = "Return"

   '2) Initialize array of all tickers
   Dim tickers(12) As String
   tickers(0) = "AY"
   tickers(1) = "CSIQ"
```

```
    tickers(2) = "DQ"
    tickers(3) = "ENPH"
    tickers(4) = "FSLR"
    tickers(5) = "HASI"
    tickers(6) = "JKS"
    tickers(7) = "RUN"
    tickers(8) = "SEDG"
    tickers(9) = "SPWR"
    tickers(10) = "TERP"
    tickers(11) = "VSLR"
    '3a) Initialize variables for starting price and ending price
    Dim startingPrice As Single
    Dim endingPrice As Single
    '3b) Activate data worksheet
    Worksheets("2018").Activate
    '3c) Get the number of rows to loop over
    RowCount = Cells(Rows.Count, "A").End(xlUp).Row

    '4) Loop through tickers
    For i = 0 to 11
        ticker = tickers(i)
        totalVolume = 0
        '5) loop through rows in the data
        Worksheets("2018").Activate
        For j = 2 to RowCount
            '5a) Get total volume for current ticker
            If Cells(j, 1).Value = ticker Then

                totalVolume = totalVolume + Cells(j, 8).Value

            End If
            '5b) get starting price for current ticker
            If Cells(j - 1, 1).Value <> ticker And Cells(j, 1).Value = ticker

                startingPrice = Cells(j, 6).Value

            End If

            '5c) get ending price for current ticker
            If Cells(j + 1, 1).Value <> ticker And Cells(j, 1).Value = ticker

                endingPrice = Cells(j, 6).Value
```

```
            End If
        Next j
        '6) Output data for current ticker
        Worksheets("All Stocks Analysis").Activate
        Cells(4 + i, 1).Value = ticker
        Cells(4 + i, 2).Value = totalVolume
        Cells(4 + i, 3).Value = endingPrice / startingPrice - 1

    Next i

End Sub
```

### Add, Commit, Push

Don't forget to save your changes and push `green_stocks.xlsm` to the "stocks-analysis" repository in GitHub.

# 2.4.1: Static Formatting

Now that we've run the analysis, let's make it easier for Steve to read by adding some formatting to our table. This is the same type of formatting we did in the last module—changing font styles, adding borders, setting number formats, and so on—but we can automate formatting with VBA.

The first step to begin formatting your table is to make sure the right worksheet is active in VBA. This might seem redundant because the worksheet should be active after the loop finishes, but we should always write code expecting that it will be changed in the future. Maybe we'll come back in six months and add another loop that activates a different worksheet. If we haven't specifically stated that the "All Stocks Analysis" worksheet should be active, we'll start formatting the wrong sheet. It's a good practice to explicitly activate the sheet we need before we start formatting.

Use the following code:

```
    'Formatting
    Worksheets("All Stocks Analysis").Activate
```

## Visual Style Formatting

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

```
    'Formatting
    Worksheets("All Stocks Analysis").Activate
    Range("A3:C3").Font.Bold = 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:

```
    'Formatting
    Worksheets("All Stocks Analysis").Activate
    Range("A3:C3").Font.Bold = True
    Range("A3:C3").Borders(xlEdgeBottom).LineStyle = xlContinuous
```

This is a small example of how to format cells.

> **Hint**
> Wondering how to find all the different ways to format cells in Excel? You can find this information in the documentation, but since documentation can be difficult to navigate, an easier route is to use Google. (This is the approach most coders take.) For example, at the time of this writing, if you google "how to add borders in Excel with VBA," the first search result is the **exact section in the documentation (https://docs.microsoft.com/en-us/office/vba/api/excel.range.borders)** where this information can be found.

Let's try to put a few more styles into place.

> **Skill Drill**
> Using the properties listed for the `Font` object in the **official VBA documentation (https://docs.microsoft.com/en-us/office/vba/api/excel.font%28object%29)**, change three of the font properties of the header.

# 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.

The code should look like this:

```
    'Formatting
    Worksheets("All Stocks Analysis").Activate
    Range("A3:C3").Font.FontStyle = "Bold"
    Range("A3:C3").Borders(xlEdgeBottom).LineStyle = xlContinuous
    Range("B4:B15").NumberFormat = "#,##0"
    Range("C4:C15").NumberFormat = "0.0%"
```

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.

> **Skill Drill**
>
> 1. Add one more digit of precision to the return percentage.
> 2. Format the price column to use a dollar sign and two significant digits 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")`.

```
    'Formatting
    Worksheets("All Stocks Analysis").Activate
    Range("A3:C3").Font.FontStyle = "Bold"
    Range("A3:C3").Borders(xlEdgeBottom).LineStyle = xlContinuous
    Range("B4:B15").NumberFormat = "#,##0"
    Range("C4:C15").NumberFormat = "0.0%"
    Columns("B").AutoFit
```

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

# 2.4.2: Conditional Formatting

The table is a lot easier to read now, but it's still difficult to determine at a glance which stocks performed well and which ones did not. Let's format our data so that Steve can determine stock performance at a glance.

If we make positive returns green and negative returns red, it'll be a lot easier to determine which stocks did well and which ones didn't. Let's add some formatting based on the values of the returns. We'll loop through each of the returns, and if the return is positive, we'll make the background color green; if the return is negative, we'll make the background red; otherwise, we'll clear the background color.

> **Rewind**
> Earlier, we used Excel's built-in conditional formatting to accomplish this kind of highlighting. However, since we're now building the output sheet programmatically, we're going to handle the conditional formatting ourselves with code.

## 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.

```
If Cells(4, 3) > 0 Then
    'Color the cell green
    Cells(4, 3).Interior.Color = vbGreen

End If
```

Then, add an elseif block to check if the cell is less than zero.

```
If Cells(4, 3) > 0 Then
    'Color the cell green
    Cells(4, 3).Interior.Color = vbGreen
ElseIf Cells(4, 3) < 0 Then

    'Color the cell red
    Cells(4, 3).Interior.Color = vbRed
End If
```

Finally, add an else block to clear the cell if the value is neither positive nor negative.

```
If Cells(4, 3) > 0 Then
    'Color the cell green
    Cells(4, 3).Interior.Color = vbGreen
ElseIf Cells(4, 3) < 0 Then

    'Color the cell red
    Cells(4, 3).Interior.Color = vbRed

Else
    'Clear the cell color
    Cells(4, 3).Interior.Color = xlNone

End If
```

---

# 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.

```
    dataRowStart = 4
    dataRowEnd = 15
    For i = dataRowStart To dataRowEnd

    Next i
```

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.

```
dataRowStart = 4
dataRowEnd = 15
For i = dataRowStart To dataRowEnd

    If Cells(i, 3) > 0 Then

        'Change cell color to green
        Cells(i, 3).Interior.Color = vbGreen

    End If

Next i
```

Then, check if the cell is less than zero. If so, change the color to red.

```
dataRowStart = 4
dataRowEnd = 15
For i = dataRowStart To dataRowEnd

    If Cells(i, 3) > 0 Then

        'Change cell color to green
        Cells(i, 3).Interior.Color = vbGreen

    ElseIf Cells(i, 3) < 0 Then

        'Change cell color to red
        Cells(i, 3).Interior.Color = vbRed

    End If

Next i
```

Finally, if neither condition is true, clear the cell color.

```
    dataRowStart = 4
    dataRowEnd = 15
    For i = dataRowStart To dataRowEnd

        If Cells(4, 3) > 0 Then

            'Color the cell green
            Cells(4, 3).Interior.Color = vbGreen

        ElseIf Cells(4, 3) < 0 Then

            'Color the cell red
            Cells(4, 3).Interior.Color = vbRed

        Else

            'Clear the cell color
            Cells(4, 3).Interior.Color = xlNone

        End If

    Next i
```

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

> ### Skill Drill
>
> Using nested `for` loops and conditionals, format an 8 x 8 square of cells with a checkerboard pattern.
>
> **Hint:** The `mod` function checks if a number is even; `n mod 2` evaluates `True` if `n` is even and `False` if `n` is odd.

It's always a good idea to save your work.

> ### Add, Commit, Push
>
> Save your changes and push to the “stocks-analysis” repository in GitHub.

# 2.5.1: Make a Run Button

Steve needs a way to run these analyses. He could install the Developer tab, but a button would be easier and more user-friendly. Let's make a button for Steve.

Now that we've written and tested a significant amount of code, running a macro might seem like a simple task. But this might not be the case for Steve. He wants to focus on financial analysis, not installing Developer tools, determining the correct macro to use, and then figuring out how to run the macro. To make life easier for the end-users of our code like Steve, we can create buttons in the worksheet.

## 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.

### macOS

## Windows

# 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:

```
Sub ClearWorksheet()

    Cells.Clear

End Sub
```

Let's get some more practice creating buttons.

> **Skill Drill**

1. Create a button to run the `ClearWorksheet()` macro. (**Hint:** Try selecting the button we've already created, copying it, and then pasting it.)
2. Create a button on the DQ worksheet to run the `DQAnalysis()` macro.
3. Answer this question: What happens if you create a button to run `ClearWorksheet()` on the DQ worksheet?
4. Create a button to run `ClearWorksheet()` on the DQ worksheet. Does its behavior match what you thought would happen?

# 2.5.2: Run the Analysis for Any Year

Steve will probably want to run this analysis for each year, so let's update our code to run for any year, not just 2018.

Our macro needs interactivity; specifically, we want Steve to be able to input the year he wants to run the analysis for. To get his input, we'll use the `InputBox()` command. `InputBox` works like `MsgBox` but contains a text box to get input from the user.

## 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 the macro:

```
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
```

---

# Run the Macro

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

> ### Add, Commit, Push
> You know the drill. Push your changes to the "stocks-analysis" repository in GitHub.

# Module 2 Challenge

Submit Assignment

**Due** Jan 26 by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

In this challenge, you will edit your code to perform the same analysis of stocks in 2018, but you will switch the nesting order of your `for` loops in order to run the analysis faster. Reworking code is known as **refactoring** and is a key part of the coding process.

You can think of refactoring as rewriting a draft of an essay. The first time we write code, our focus is on getting it to work correctly. After we're done with the first "draft" of code, we can then look to see how we could accomplish the same task with code that runs more quickly or is easier to understand. Therefore, refactoring is the process of accomplishing the same task with our code, but in a more efficient way.

# Background

Steve is grateful for your help and loves the workbook you prepared for him. What he doesn't love, however, are the returns on the stocks in 2018. Steve wants to do a little more research for his parents and is gathering data for the entire stock market over the last few years. While your code works great for a dozen stocks, it might not work so well for thousands of stocks. (Remember, you had to loop through all of the stock data for each ticker you analyzed.)

To help Steve, **refactor** your code. Remember, refactoring is like writing another draft of your code. You aren't trying to add new functionality—you just want to make the code more efficient, whether by taking fewer steps, using less memory, or making the code easier to read.

Refactoring is common on the job, as first attempts at code won't always be the best way to accomplish a task. Sometimes, refactoring someone else's code will be your entry point to working with the existing code at a job.

You're going to refactor your code for Steve to loop through the data only once and collect all of the information it needs in a single pass.

# Objectives

The goals of this challenge are for you to:

- Analyze code and explain how it accomplishes a task.
- Evaluate logic flow of `for` loops and conditionals.
- Extend your pattern recognition skills to refactor existing code.

# Instructions

Refactor the code below, which you created in this module.

As you refactor your code, complete the following:

- Create arrays for all the volumes, starting prices, and ending prices.

  **Hint:** You'll need to create a `for` loop to set all volumes to zero.

- Set a `tickerIndex` equal to zero before the loop. If the next row's ticker doesn't match, increase the `tickerIndex`. Here, we're taking advantage of the fact that the tickers are all in alphabetical order.
- Use `tickerIndex` to access the correct index across the four different arrays you'll be using (the `tickers` array and the three output arrays). `tickers(tickerIndex)` will give you the current ticker you're working on.
- Once you're done collecting the data, loop through your arrays to output all of the information you've collected. The formatting code should still work without needing to be changed.

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

    Worksheets("All Stocks Analysis").Activate

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

    'Create a header row
    Cells(3, 1).Value = "Ticker"
    Cells(3, 2).Value = "Total Daily Volume"
```

```
    Cells(3, 3).Value = "Return"
    Dim tickers(12) As String

    tickers(0) = "AY"
    tickers(1) = "CSIQ"
    tickers(2) = "DQ"
    tickers(3) = "ENPH"
    tickers(4) = "FSLR"
    tickers(5) = "HASI"
    tickers(6) = "JKS"
    tickers(7) = "RUN"
    tickers(8) = "SEDG"
    tickers(9) = "SPWR"
    tickers(10) = "TERP"
    tickers(11) = "VSLR"


    Worksheets(yearValue).Activate

    'get the number of rows to loop over
    RowCount = Cells(Rows.Count, "A").End(xlUp).Row

    Dim startingPrice As Single
    Dim endingPrice As Single

    For i = 0 To 11

        ticker = tickers(i)
        totalVolume = 0

        Worksheets(yearValue).Activate

        'loop over all the rows
        For j = 2 To RowCount

            If Cells(j, 1).Value = ticker Then

                'increase totalVolume by the value in the current row
                totalVolume = totalVolume + Cells(j, 8).Value
```

```
            End If

            If Cells(j - 1, 1).Value <> ticker And Cells(j, 1).Value = ticker
Then

                startingPrice = Cells(j, 6).Value

            End If

            If Cells(j + 1, 1).Value <> ticker And Cells(j, 1).Value = ticker
Then

                endingPrice = Cells(j, 6).Value

            End If

        Next j

        Worksheets("All Stocks Analysis").Activate
        Cells(4 + i, 1).Value = ticker
        Cells(4 + i, 2).Value = totalVolume
        Cells(4 + i, 3).Value = endingPrice / startingPrice - 1

    Next i

    'Formatting
    Worksheets("All Stocks Analysis").Activate
    Range("A3:C3").Font.FontStyle = "Bold"
    Range("A3:C3").Borders(xlEdgeBottom).LineStyle = xlContinuous
    Range("B4:B15").NumberFormat = "#,##0"
    Range("C4:C15").NumberFormat = "0.0%"
    Columns("B").AutoFit
    dataRowStart = 4
    dataRowEnd = 15
    For i = dataRowStart To dataRowEnd

        If Cells(i, 3) > 0 Then

            Cells(i, 3).Interior.Color = vbGreen
```

```
        Else

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

        End If

    Next i

End Sub
```

Not only have you created VBA code that automates financial analysis, but you've also reworked the code to be more efficient, saving Steve time and frustration. Your code is also flexible enough that Steve can add data for other years and rerun the analysis for any given year. Job well done!

# Submission

Save your worksheet and upload it to a new GitHub repository. Make sure the "Initialize this repository with a README" option is checked.

Next, update your README.md file as follows:

- Add a header for the challenge, i.e., `# Challenge`
- Under this section, add a short description of the new analysis

Save your README file. Make sure the most recent copy is in your GitHub repository. Then, copy the link to your Challenge from your repo and paste it into the submission here.

**Note:** You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing "I choose to skip this assignment" in the text box.

| **Module 2 Challenge Rubric** |
| --- |

| Criteria | Ratings | | | | | Pts |
|---|---|---|---|---|---|---|
| Create Arrays | **25.0 pts**<br>**Mastery**<br>Arrays are correctly created for tickers, volume, starting prices, and ending prices. | **20.0 pts**<br>**Approaching Mastery**<br>Arrays are correctly created for tickers as well as two of three output arrays (volume, starting prices, and ending prices) with one or two minor errors. | **15.0 pts**<br>**Progressing**<br>Arrays are created for tickers and two of the three output arrays (volume, starting prices, and ending prices) with two or three errors. | **10.0 pts**<br>**Emerging**<br>Arrays are created for tickers and one of the three output arrays (volume, starting prices, and ending prices), with more than three errors. | **0.0 pts**<br>**Incomplete**<br>No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty | 25.0 pts |
| Set & Match tickerIndex | **25.0 pts**<br>**Mastery**<br>tickerIndex is set to equal to zero before all four loops and tickers are correctly matched across rows. | **20.0 pts**<br>**Approaching Mastery**<br>tickerIndex is set to equal to zero before all four loops and tickers are correctly matched across rows, with one or two minor errors. | **15.0 pts**<br>**Progressing**<br>tickerIndex is set to equal to zero before three loops and tickers are correctly matched across rows, with two or three errors. | **10.0 pts**<br>**Emerging**<br>tickerIndex is set to equal to zero before two loops and tickers are correctly matched across rows, with more than three errors. | **0.0 pts**<br>**Incomplete**<br>No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty | 25.0 pts |
| Apply tickerIndex to Arrays | **25.0 pts**<br>**Mastery**<br>Use tickerIndex to access the correct index across the tickers array and the three output arrays. | **20.0 pts**<br>**Approaching Mastery**<br>Use tickerIndex to access the correct index across the tickers array and the three output arrays, with one or two minor errors. | **15.0 pts**<br>**Progressing**<br>Use tickerIndex to access the correct index across the tickers array and two output arrays, with two or three errors. | **10.0 pts**<br>**Emerging**<br>Use tickerIndex to access the correct index across the tickers array and one output array, with more than three errors. | **0.0 pts**<br>**Incomplete**<br>No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty | 25.0 pts |
| Loop Arrays | **25.0 pts**<br>**Mastery**<br>The script loops through stock data and reads/ stores all of the following values from each row: tickers, volume, | **20.0 pts**<br>**Approaching Mastery**<br>The script loops through stock data and reads/ stores three of the four following values from each row: tickers, volume, starting prices, ending prices, with | **15.0 pts**<br>**Progressing**<br>The script loops through stock data and reads/ stores two of the four following values from each row: tickers, volume, starting prices, ending prices, with two or three errors. | **10.0 pts**<br>**Emerging**<br>The script loops through stock data and reads/ stores one of the following values from each row: tickers, volume, starting prices, ending prices, | **0.0 pts**<br>**Incomplete**<br>No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty | 25.0 pts |

| Criteria | Ratings | | | Pts |
|---|---|---|---|---|
| | starting prices, ending prices. | one or two minor errors. | with more than three errors. | Total Points: 100.0 |

# Module 2 Career Connection

## Learn the Difference Between Employer-Ready and Employer-Competitive

According to a recent Jobvite Recruiter Nation Report, 74% of recruiters believe hiring will become more competitive in the next 12 months. Based on our conversations with employers, we know there's a difference between being *ready* to enter the market and being *competitive* in the market.

Take a few moments to review the employer-ready vs. employer-competitive framework for your industry.

**Career Services Next Step:** **Employer-Ready vs. Employer-Competitive Framework (https://courses.bootcampspot.com/courses/138/pages/employer-ready-vs-employer-competitive-framework)**

# Employer-Ready vs. Employer-Competitive Framework

Based on employer feedback, we split our curriculum into two parts: **employer-ready** and **employer-competitive**. It is a natural progression for students to move to being employer-ready and then to become employer-competitive.

- **Employer-ready** means that you have the minimum requirements to enter the job search. Your professional materials––including your resume and brand statement––are strong and complete.
- **Employer-competitive** means that you are taking a proactive approach to the job-search process by preparing for interviews, attending career workshops, networking, and continually fine-tuning your resume and brand statement.

If you complete the Career Services milestones (checkpoint assignments) by the end of the course, you will be considered **employer-ready**.

If you continue to use Career Services by attending workshops and working with your career director to further understand the industry, you will be considered **employer-competitive** and have the knowledge you need to make your desired career move in the data industry in the near future.

## What to Do to Be Employer-Ready

Employer-readiness criteria falls into the following categories.

---

## Professional Materials

Create a resume that is clear, concise, and compelling.

# GitHub

- Add at least 200 commits and refactor previously submitted code.
- Maintain 3–6 repositories that include a descriptive title, thorough README, and clean code.

# What to Do to Be Employer-Competitive

Being employer-competitive is a more involved process that will continue after your completion of this boot camp. Employer-competitive criteria is divided into the following categories.

# Professional Materials

- Create a professional resume that is clear, concise, compelling, and tailored to the role you are applying for.
- Write targeted cover letters that explain why you want the particular role you are applying for and why you want to work at that specific company.
- Update your LinkedIn profile to include a compelling bio or personal statement, professional photo, link to your GitHub, and Projects section.

# GitHub

- Add at least 200 commits and refactor previously submitted code.
- Maintain 3–6 repositories that include a descriptive title, thorough README, and clean code.
- Contribute regularly to open-source projects.

# Industry Visibility

- Attend all employer-facing events organized by this boot camp’s Career Services team.

- Attend one or two local Meetup or Eventbrite events per week. Make two or three solid connections per event.
- Create business cards that include your desired title, links to your GitHub and LinkedIn, and QR code for resume.
- Reach out to your network to let them know what types of roles you're looking for.
- Use LinkedIn to connect weekly with approximately 5 employees or decision-makers.
- Conduct at least three informational interviews weekly with industry leaders.
- Follow companies you are interested in applying to on social media.

---

# Interview Preparation

- Research common interview questions and practice answering them.
- Schedule a one-on-one mock interview with your Career Director.

---

# Job Applications

- Apply to at least 10 jobs per week.
- Set up email notifications for your common keyword searches from job boards.
- Share application status with your Career Director so they can leverage possible employer relationships or alumni connections.
- Follow up on all applications within 1 week.

---

# Professional Development

- Continue learning after you complete this boot camp by researching technology that is in demand in your market.
- Build a solo project.

# Unit Assessment: Excel & VBA

You are about to complete your first Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 1 and 2.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Some of the questions on this assessment require specific resources. Download the following resources before you get started.

***Top5000Songs_Unsolved.xlsx***

***conditional_formatting.xlsx***

***ProductionPivot_Unsolved.xlsx***

***house_data.csv***

***ePhone_prices.xlsx***

***star_counter.xlsm***

# Unit Assessment: Excel & VBA

Please click **Start** when you are ready to begin the activity.

Start