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.