GOOGLE SHEETS PROGRAMMING WITH GOOGLE APPS SCRIPT

Your Guide To Building Spreadsheet Applications In The Cloud

MICHAEL MAGUIRE

Google Sheets Programming With Google Apps Script (2015 Revision In Progress)

Your Guide To Building Spreadsheet Applications In The Cloud

Michael Maguire

This book is for sale at
http://leanpub.com/googlespreadsheetprogramming

This version was published on 2015-09-29

Tweet This Book!

Please help Michael Maguire by spreading the word about this book on Twitter!

The suggested hashtag for this book is #googlespreadsheetprogramming.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#googlespreadsheetprogramming

Contents

Chapter 1: Introduction **1**
1.1 Google Sheets 1
1.2 Google Apps Script (GAS) 2
1.3 JavaScript or Google Apps Script? 3
1.4 Summary Of Topics Covered 3
1.5 Software Requirements For This Book 5
1.6 Intended Readership 5
1.7 Book Code Available On GitHub 6
1.8 My Blog On Google Spreadsheet Programming . . . 7
1.8 Guideline On Using This Book 7
1.9 2015 Update Notes 8

Chapter 2: Getting Started **9**
2.1 Introduction 9
2.2 Google Apps Script Examples 9
2.2 Executing Code – One Function At A Time 11
2.3 Summary 14

Chapter 3: User-Defined Functions **15**
3.1 Introduction 15
3.2 Built-in Versus User-Defined Functions 16
3.3 Why Write User-Defined Functions 17
3.4 What User-Defined Functions Cannot Do 18
3.5 Introducing JavaScript Functions 21
3.6 User-Defined Functions Versus JavaScript Functions 25
3.7 Using JSDoc To Document Functions 26

3.8 Checking Input And Throwing Errors 26
3.9 Encapsulating A Complex Calculation 29
3.10 Numeric Calculations 31
3.11 Date Functions 33
3.12 Text Functions 39
3.13 Using JavaScript Built-In Object Methods 42
3.14 Using A Function Callback 43
3.15 Extracting Useful Information About The Spreadsheet 45
3.16 Using Google Services 49
3.18 Summary 50

Appendix A: Excel VBA And Google Apps Script Comparison 52
Introduction 52
Spreadsheets and Sheets 53
Ranges 64

Chapter 1: Introduction

1.1 Google Sheets

Google Sheets is one of the core components of Google cloud applications. If you have a Gmail account, you can create and share your spreadsheets with others, even with those who do not have a Gmail account. Google Sheets offers a comprehensive set of standard spreadsheet features and functions similar to those found in other spreadsheet applications such as Microsoft Excel. In addition, it also supports some novel features such as the very versatile *QUERY* function and regular expression functions such *REGEXMATCH*.

What really distinguished Google Sheets from desktop spreadsheet applications like Excel is its cloud nature. The spreadsheet application runs in a browser and the spreadsheet files themselves are stored remotely. The spreadsheet files can be shared with others in read-only or read-edit modes making them ideal collaborative tools. Spreadsheets form just one part, albeit an important one, of the Google suite of products. Others are Google Documents, Gmail, calendars, forms, and so on and all of these products are inter-operable at least to some degree resulting in a very productive environment perfectly suited to collaborative work.

When I began using Google Sheets back in 2010 it was quite limited in terms of data volume, speed and functionality. It has undergone significant development since then and got a major upgrade in March 2014[1]. If your experience of Google Sheets was negatively influenced by experience with earlier versions, I encourage you to try it again, I think you will notice a big improvement. The

[1] https://support.google.com/docs/answer/3544847?hl=en

old 400,000 cell limit per spreadsheet is gone and is now at least 2,000,000. It will comfortably deal with tens of thousands of rows which is, I believe, quite acceptable for any spreadsheet. Other spreadsheet applications such as Excel can handle a million plus rows but when data volumes grow to this size, it is advisable to switch to a database or a dedicated statistical application to handle such data sizes.

1.2 Google Apps Script (GAS)

The Google Sheets application also hosts a programming language called Google Apps Script (GAS) that is executed, not in the browser but remotely on the Google cloud. Google define Google Apps Script as follows:

"Google Apps Script is a JavaScript cloud scripting language that provides easy ways to automate tasks across Google products and third party services."

If Google Sheets is so feature-rich, you might wonder why it needs to host a programming language. Here are few reasons why GAS is needed:

- Write user-defined functions for Google Sheets
- Write simple "macro" type applications
- Develop spreadsheet-based applications
- Integrate other Google products and services
- Develop Graphical User Interfaces (GUIs) that can be run as web applications
- Interact with cloud-based relational databases via Google *JDBC* Services.

GAS plays a similar role in Google Sheets to that played by Visual Basic for Applications (VBA) in Excel. Both are hosted by their

respective applications and both are used to extend functionality and integrate with other applications and services.

1.3 JavaScript or Google Apps Script?

The emphasis here is on using GAS to enhance and control Google Sheets. Other Google services are discussed in the context of how they can be used with Google Sheets. Since GAS is JavaScript (Google describe it as a sub-set of JavaScript 1.8), there will inevitably be discussion of JavaScript as a programming language. There is, therefore, some discussion of JavaScript topics as they relate to the code examples given.

Regarding terminology, when discussing a general JavaScript feature, the code may be referred to as "JavaScript" but when dealing with a Google App specific example, it may be referred to as "Google Apps Script" or "GAS". The meaning of whichever term is used should be clear from the context. For example, the *Spreadsheet* object is central to Google Sheets programming. It is, however, provided by the hosting environment and is not part of JavaScript itself. This duality of the programming language and the objects provided by the hosting environment is similar to JavaScript running on the web client and the Document Object Model (DOM) entities that it manipulates.

1.4 Summary Of Topics Covered

This book aims to provide the reader with a solid knowledge of the GAS language both as it applies to Google Sheets and how it is used to allow Google Sheets to inter-operate with other Google products and services as well as with relational databases.

Chapter 2 introduces the GAS language and sets the scene for the chapters that follow. One of the most important applications

of the hosted spreadsheet language, be it VBA or GAS, is to allow users to write user-defined functions (also known as custom functions). These are covered in depth in chapter 3 and this chapter merits careful reading for any readers not familiar with JavaScript. Functions are central to JavaScript and, by extension, to GAS.

The *Spreadsheet*, *Sheet*, and *Range* objects are crucial for manipulating Google Sheets with GAS and these objects are covered in depth in chapters 4 and 5. All subsequent chapters assume that the reader is comfortable with these objects, their methods and their uses.

Having covered the basics of user-defined functions and the fundamental spreadsheet objects, chapter 6 explains how GAS can be used to work with a back-end relational database (MySQL). Spreadsheets are great tools but they have their limitations and, as applications increase both in complexity and in data volume, there comes a point where a more robust data storage solution is needed that that offered by spreadsheets.

In order to build spreadsheet *applications* some sort of Graphical User Interface (GUI) is usually required. Chapters 7 and 8 cover menus, alerts, prompts and user forms created using Google *Html Service*. Creating forms with *Html Service* offers the opportunity to develop web skills such as HTML, CSS and client-side JavaScript. For those not experienced in frontend development, this material offers a gentle introduction to a very important skill.

In addition to being an excellent collaborative tool, Google Sheets is part of a larger set of applications with which it can interact. GAS written in Google Sheets can be used to manipulate other Google products such as Google Drive, and Google Calendar and this aspect of GAS programming is covered in chapters 9 and 10.

For those coming to GAS from Excel VBA, appendix A will be of especial interest as it gives example code and explanations of how to perform common spreadsheet programming tasks in both languages. Appendix B gives an example of a quite complex spread-

sheet application written in GAS that brings together much of the material covered earlier. It describes how to build an application that takes spreadsheet data as input and uses it to generate SQL for table creation and row insertion. Appendix C discusses additional GAS and JavaScript resources that will be of interest to readers.

1.5 Software Requirements For This Book

Not many! Modern cloud-based applications such as Google Sheets greatly reduce the technical barriers for new entrants. In the old days, you might have needed a specific operating system running some proprietary and often expensive software to get started. Not anymore! Anyone with a modern browser, an internet connection and a Gmail account running on Windows, Mac OS X or any version of Linux will be able to follow along with this book and run the code examples given. The code examples should run in any modern browser but I mainly use Chrome and Firefox and never use Internet Explorer (IE) although I expect that all the code will run fine in any post IE7 version.

1.6 Intended Readership

This book is written for those who wish to learn how to programmatically manipulate Google Sheets using GAS. I began learning GAS for two reasons. Firstly, I was using Google Sheets as a collaborative tool in my work and, secondly, I had become aware of the increasing importance of JavaScript generally and was keen to learn it. Being able to use the JavaScript-based language that is GAS in spreadsheets appealed to me because I was already quite experienced in programming Excel using VBA so I felt that learning GAS to manipulate Google Sheets would offer a familiar

environment. I reasoned that there are many experienced Excel VBA programmers around who might feel similarly so that is why I wrote this book. There are of course now many people who use Google products who are familiar with JavaScript but who may not know much about spreadsheets. This book might also be of interest to this group of users. This book assumes programming knowledge in some programming language though not necessarily JavaScript.

1.7 Book Code Available On GitHub

The emphasis on this book is on practical code examples. Some of the code examples are short and only practical in the sense that they exemplify a GAS feature. GAS is a "moving target" in that new features are added and deprecated frequently thereby making it difficult to keep all code up-to-date. At the time of writing, all the examples worked as expected but please email me[2] if something is broken or if you get a warning of something having been deprecated. To allow readers to follow along, I have tried to document the code extensively using JSDoc[3] and in-line code comments. To run the examples, you can copy and paste from the book to the GAS Script Editor. This will work but I recommend getting the code from GitHub. All the code examples in this book are available on a Github repository created specifically for this updated version of the book. The user name is **Rotifer** and the repository name is **GoogleSpreadsheetProgramming_2015**. The full URL is here.[4] You can use the Git command line tool to check out the repository to your local machine or simply copy the examples directly from the GitHub repository.

[2] mick@javascript-spreadsheet-programming.com

[3] usejsdoc.org/

[4] https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015

1.8 My Blog On Google Spreadsheet Programming

I maintain a blog[5] that pre-dates this book. The blog began in late 2010 so some of the early examples use deprecated or unsupported features. That said, the blog inspired this book and is actively maintained so it can be viewed as a complementary resource to this book. It is worth checking for new entries from time to time because I use it to explore and discuss new spreadsheet and GAS features and I cover some more advanced JavaScript material there.

1.8 Guideline On Using This Book

I learn best from examples so this book is heavily example-driven. I first like to see something working and then examine it and learn from it. In order to derive maximum benefit from this book, it is important to execute the code from your own spreadsheets. Read the book descriptions and code documentation carefully and make sure that you understand both the objective of the code as well as how it works. Chapter 3 goes in to some depth on core JavaScript concepts such as functions, arrays and objects but if you are relatively inexperienced in JavaScript, then some background reading will help enormously. Remember, GAS is JavaScript so the better you understand JavaScript, the better you will be at writing and understanding GAS. Do not be afraid to experiment with code, change or re-write my examples at will! If you find better ways of doing things, please let me know!

[5] http://www.javascript-spreadsheet-programming.com

1.9 2015 Update Notes

The first version of this book was released in November 2013 and the feedback I have received on it has been largely positive. However, I realise that some of the material is now out-of-date. For example, *UiApp* and *DocsList* have both been deprecated since that version of the book was released. I am also aware that the book could be improved by better explanations, better examples and generally better writing. I hope this version delivers a better product to its readers.

As I stated earlier, Google Sheets and GAS are "moving targets" that are subject to constant change. I intend to release a new version of this book yearly from now on so that the material remains current and the overall quality of the book improves. I will continue to blog about new features and topics that I have not covered so far and, if these look like a good fit and attract interest from readers, I will incorporate such subjects into later version of the book.

Something I find annoying and expensive is when I buy a technical book only to learn a few months later that a new edition has been released. I have one shell programming book from 1990 that I still use but shell programming is one of the few stable technologies that I use (SQL is another, yes, features are added but the core is quite stable). Most technical books that I have bought in the past become obsolete, at least in part, in a year or two. My idea with this book is to make sure those who buy it once get all the updates for free and I plan to keep updating it indefinitely. Leanpub make this possible, so a big thanks to them and thanks also to all of you who bought the first version of this book!

Time to write some GAS!

Chapter 2: Getting Started

2.1 Introduction

The best way to learn JavaScript/Google Apps Script is to write some code. Getting started is very straightforward: All that is needed is a Gmail account and a browser with an Internet connection. To run some example code, first go to Google Drive and create a spreadsheet. To view the script editor, select *Tools->Script editor...* from the spreadsheet menu bar. The first time you do this in a new spreadsheet file, you will be presented with a pop-up window entitled "Google Apps Script", just ignore and close it for now. Give the project a name, any name you like, by hovering over and replacing the text "untitled project" on the top left. Delete the code stub entitled "myFunction" so that the script editor is now blank. Paste in the example code in the following sections and save (save icon or menu action *File->Save*).

2.2 Google Apps Script Examples

Here are four example functions. When pasted into the script editor, the code formatting applied by the editor becomes evident and makes the code easier to read. The code for this chapter can be viewed and downloaded from GitHub here.[6]

[6] https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch02.gs

```
function sayHelloAlert() {
  // Declare a string literal variable.
  var greeting = 'Hello world!',
      ui = SpreadsheetApp.getUi();
  // Display a message dialog with the greeting
  //(visible from the containing spreadsheet).
  // Older versions of Sheets used Browser.msgBox()
  ui.alert(greeting);
}

function helloDocument() {
  var greeting = 'Hello world!';
  // Create DocumentApp instance.
  var doc =
    DocumentApp.create('test_DocumentApp');
  // Write the greeting to a Google document.
  doc.setText(greeting);
  // Close the newly created document
  doc.saveAndClose();
}

function helloLogger() {
  var greeting = 'Hello world!';
  //Write the greeting to a logging window.
  // This is visible from the script editor
  //   window menu "View->Logs...".
  Logger.log(greeting);
}


function helloSpreadsheet() {
  var greeting = 'Hello world!',
      sheet = SpreadsheetApp.getActiveSheet();
  // Post the greeting variable value to cell A1
  // of the active sheet in the containing
```

```
36    //  spreadsheet.
37    sheet.getRange('A1').setValue(greeting);
38    // Using the LanguageApp write the
39    //  greeting to cell:
40    // A2 in Spanish,
41    //  cell A3 in German,
42    //  and cell A4 in French.
43    sheet.getRange('A2')
44          .setValue(LanguageApp.translate(
45                    greeting, 'en', 'es'));
46    sheet.getRange('A3')
47          .setValue(LanguageApp.translate(
48                    greeting, 'en', 'de'));
49    sheet.getRange('A4')
50           .setValue(LanguageApp.translate(
51                     greeting, 'en', 'fr'));
52  }
```

2.2 Executing Code – One Function At A Time

In order to execute code, there must be at least one valid function in the script editor. After pasting the code above, there are four functions that will each be executed in turn.

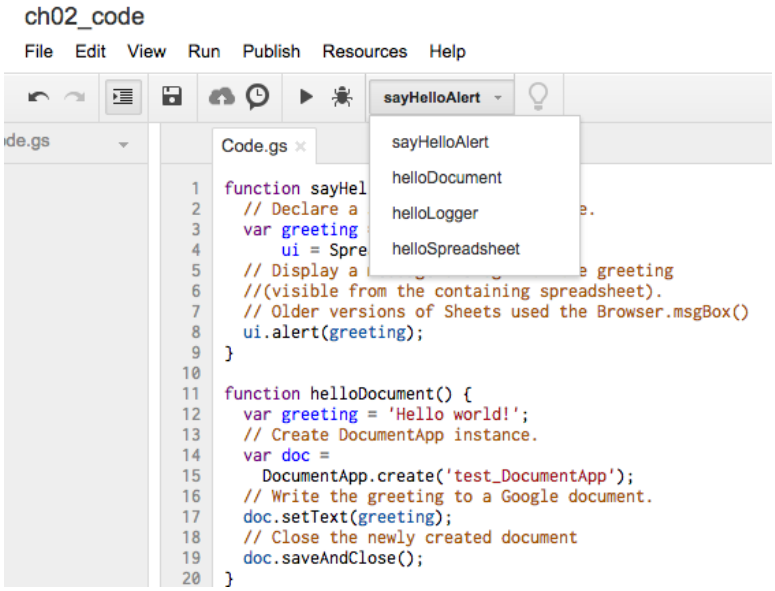


Figure 2-1: Google Apps Script Editor displaying the code and the "Select function" drop-down list.

Select function *sayHelloAlert()* from the *"Select function"* drop-down list on the script editor toolbar and then press the execute icon (to the left of the function list drop-down). You will need to authorize the script when you first try to execute it. Subsequent executions do not require authorisation. Once authorised, switch to the spreadsheet and you will see a small window with the greeting "Hello world". These browser popup displays are modal meaning that they block all subsequent code execution until they are closed. For this reason, their use should be limited. The *Logger* is generally a better tool for writing and displaying output.

New Sheets Feature

Browser.msgBox instead of *alert()* is used in older versions and still works.

Now select the second function named *helloDocument()* and execute it. This is a much more interesting example than the previous one because it shows how GAS written in one application can be used to manipulate other applications. The first time you try to execute it, you will get a message saying, "Authorization required". Once you authorise it and then execute, it will create a new Google *Document* and write the message to it. This example, though trivial and useless, does demonstrate how GAS code written in one application can manipulate other applications. This is a very powerful feature and will be a recurring theme of this book.

The *helloLogger()* function, when executed, writes the message to a logging area that is viewable from the script editor menu *"View->Logs..."*. It is equivalent to the "console.log" in Firebug and Node.js. It will be used frequently in later code examples for output and for error reporting.

The final function *helloSpreadsheet()* demonstrates two important aspects of Google Apps Script:

Firstly, spreadsheets can be manipulated via the *Spreadsheet* object (the Google documentation refers to *Spreadsheet* as a "class"). It provides a method that returns an object representing the active sheet (*getActiveSheet()*) and that this returned object has a method that returns a range (*getRange()*), in this instance a single cell with the address *"A2"*. The returned range object method, *setValue()*, is then called with a string argument that is written to cell A1 of the active sheet. These types of chained method calls look daunting at first. The method call chain described above could be re-written as:

```
var greeting = 'Hello world!',
    activeSpreadsheet =
      SpreadsheetApp.getActiveSpreadsheet(),
    activeSheet =
      activeSpreadsheet.getActiveSheet(),
    rng = activeSheet.getRange('A1'),
    greeting = 'Hello world!';
rng.setValue(greeting);
```

The code above uses a number of intermediate variables and may be easier to understand initially but after experience with GAS, the chained method call will start to feel easier and more natural. The objects referenced in this example will be discussed in detail in chapters 4 and 5.

Secondly, the example code shows how easy it is to call another service from a Google Apps Script function. Here the *LanguageApp* was used to translate a simple text message into Spanish, German, and French. This ability to seamlessly access other Google services is extremely powerful.

2.3 Summary

This chapter has shown how to access the GAS Script Editor and execute functions from it. The examples demonstrated how GAS can display simple alerts, write messages to the *Logger*, manipulate ranges in spreadsheets and use other Google applications such as *Document* and Google services such as *LanguageApp*. These examples barely scrape the surface of what can be achieved using GAS. The next chapter uses GAS to write user-defined functions that can be called in the same manner as built-in spreadsheet functions.

Chapter 3: User-Defined Functions

3.1 Introduction

User-defined functions allow spreadsheet users and developers to extend spreadsheet functionality. No spreadsheet application can cater for all requirements for all users so a mechanism is provided that allows users to write their own customised functions in the hosted language, be that VBA in Excel or GAS in Google Sheets.

Definition

A user-defined function is one that can be called using the *equals* (=) sign in a spreadsheet cell and writes its return value or values to the spreadsheet.

User-defined functions are also known as *custom functions* and this is the term used by the Google.

Google Custom Function Documentation

Worth Reading![7].

The source code for this chapter can be found here on GitHub.[8] I have adopted the convention of writing user-defined functions in

[7] https://developers.google.com/apps-script/guides/sheets/functions

[8] https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch03.gs

all upper case, other fiunctions that are not intended to be called as user-defined functions are written in camel case. This chapter is quite long and covers a lot of material. It is also the chapter where I put most effort into explaining basic JavaScript concepts such as functions, arrays, objects and so on. Therefore, if you are unfamiliar with GAS/JavaScript, then pay close attention to the code examples and explanatory text. If my explanations are inadequate, then you can take advantage of the extensive on-line JavaScript resources available for all levels of user in addition to the many excellent JavaScript textbooks (see Appendix B for information on resources that I have found useful).

3.2 Built-in Versus User-Defined Functions

Modern spreadsheet applications, including Google Sheets, supply a large number of built-in functions and they owe much of their utility and widespread usage in diverse domains of finance, science, engineering and so on to these functions. There is also a high degree of standardisation between spreadsheet applications regarding function names, arguments, and usage so that they generally work uniformly in Google Sheets, Microsoft Excel, OpenOffice Calc and Gnumeric. Built-in Google Sheets functions are spread over multiple categories, see Google Spreadsheet function list [9]. Most of the standard spreadsheet text, date, statistical, logic and lookup functions are present and their usage is, in all cases that I have encountered, identical to equivalents in other spreadsheet applications.

Google Sheets also implements a number of novel functions. Some of these are very convenient and it is worth being familiar with them before you embark on writing your own functions so that you do not end up implementing something that is already present. One

[9] https://support.google.com/drive/bin/static.py?hl=en&topic=25273&page=table.cs

of my favourites is the *QUERY* function and a good description of its syntax and use can be found here[10]. This function comes from the Google Visualization API and it uses an SQL-like syntax to extract sub-sets of values from input spreadsheet data cells. If you are already familiar with SQL, then you will feel right at home with this function. Another group of functions worth exploring are those that use **regular expressions**. The functions concerned are *REGEXMATCH*, *REGEXEXTRACT* and *REGEXREPLACE* and I have described them in a blog entry[11]. Regular expressions are very useful and GAS implements the full standard JavaScript regular expression specification.

New Functionality

Google Sheets continues to add new functions

3.3 Why Write User-Defined Functions

The two main reasons for writing user-defined functions are for clarity and to extend functionality. User-defined functions can add to clarity by wrapping complex computations in a named and documented function. Spreadsheet power-users can often ingeniously combine the built–in functions to effectively create new ones. The disadvantage of this approach is that the resulting formulas can be very difficult to read, understand and debug. When such functionality is captured in a function, it can be given a name, documented and, tested. Secondly, user-defined functions allow developers to customise spreadsheet applications by adding functionality for their particular domain.

[10] https://anandexcels.wordpress.com/2013/11/01/query-function-in-google-sheets/

[11] http://www.javascript-spreadsheet-programming.com/2013/09/regular-expressions.html

3.4 What User-Defined Functions Cannot Do

An important point about user-defined functions is that they cannot be used to alter any properties, such as formats, of the spreadsheet or any of its cells. They cannot be used to send e-mails and cannot be used to insert new worksheets. Functions can be used to do these things but they cannot be called as user-defined functions. This is a common area of misunderstanding where users attempt to call functions from the spreadsheet using the *equals* operator (=). This results in an error because the function is trying to set some spreadsheet property.

User-defined functions should be designed to work just like built-in functions in that you pass in zero or more values as arguments and they return a value or values in the form of an array. Their purpose is their return values, not their side effects. Excel VBA makes a distinction between subroutines and functions. Subroutines do not return a result and cannot be called as user-defined functions. In GAS we only have functions that either return values or do not (void functions).

To illustrate how a user-defined function cannot alter spreadsheet properties, consider the following function that takes a range address argument (a string, not a *Range* object) and sets the font for the range to bold:

Code Example 3.1

```
/**
* Simple function that cannot be called from
* the spreadsheet as a user-defined function
* because it sets a spreadsheet property.
*
* @param {String} rangeAddress
* @return {undefined}
*/
function setRangeFontBold (rangeAddress) {
  var sheet =
    SpreadsheetApp.getActiveSheet();
  sheet.getRange(rangeAddress)
    .setFontWeight('bold');
}
```

The code uses objects that have not yet been discussed but the idea is simple; Take a range address as input and set the font to bold for that range. However, when the function is called from a spreadsheet, it does not work.

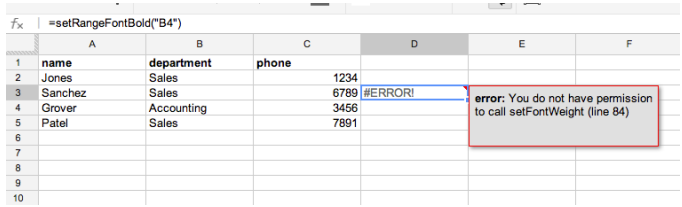


Figure 3-1: Error displayed when calling a function with side effects from a spreadsheet.

The error shown above refers to permissions but the problem is that the called function is attempting to modify sheet properties.

To prove that the function *setRangeFontBold()* is valid, here is a function that prompts for a range address using an *prompt* dialog. It calls the *setRangeFontBold()* function passing the given range address as an argument.

To See The Prompt

Call the function in the Script Editor Then switch to the spreadsheet

Code Example 3.2

```
/**
* A function that demonstrates that function
*   "setRangeFontBold() is valid although it
*   cannot be called as a user-defined function.
*
* @return {undefined}
*/
function call_setCellFontBold () {
  var ui = SpreadsheetApp.getUi(),
      response = ui.prompt(
                    'Set Range Font Bold',
                    'Provide a range address',
                    ui.ButtonSet.OK_CANCEL),
      rangeAddress = response.getResponseText();
  setRangeFontBold(rangeAddress);
}
```

When this function is called, the following prompt is displayed in the spreadsheet view:

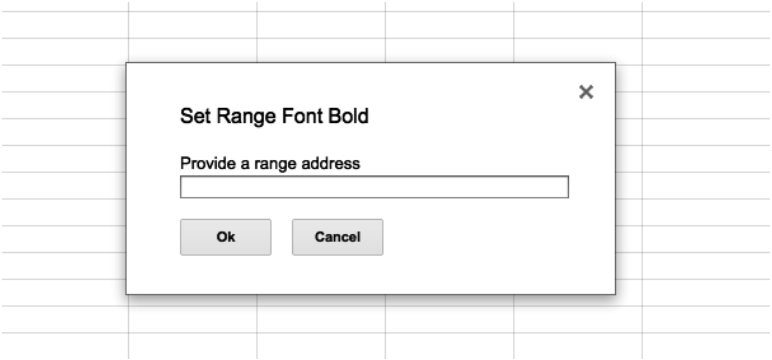


Figure 3-2: *Prompt dialog display for a range address.

The *prompt* mechanism for requesting user input will be discussed in a later chapter. The important point here is that a function may be valid but, if it has side effects such as altering range properties, then it cannot be called as a user-defined function.

New Sheets Feature

Browser.inputBox instead of *prompt()* is used in older versions and still works.

3.5 Introducing JavaScript Functions

User-defined functions can be called exactly like built-in ones using the *equals* (=) operator in a spreadsheet cell. They can take zero or more arguments of different types and these arguments can be either cell references or literal values, just as with built-in functions. The user-defined functions themselves are written in GAS and, as I have repeated multiple times already, GAS is JavaScript. To understand and write user-defined functions in Google Sheets requires an understanding of JavaScript functions. The better you understand JavaScript functions, the better you will be at crafting your own user-defined GAS functions for Google Sheets!

JavaScript functions are immensely powerful and flexible. They play a central role in many JavaScript idioms and patterns and mastery of them is a prerequisite to becoming an advanced JavaScript programmer. Only the basics of JavaScript functions are required for the purposes of this chapter. There are different ways of defining JavaScript functions but in this chapter most of the functions are defined as **function declarations** in form:

```
function functionName(comma-separated
                      parameters) {
  statements
}
```

Other types of function definition will be described as they are introduced later in the book.

JavaScript Functions

Read this Mozilla resource[12]

The Mozilla resource above discusses various means of defining JavaScript functions. For user-defined functions the decision on the syntax to use to define our functions is made for us because only function declarations work in this setting, **named function expressions** and **object methods** will not work.

For the purposes of this chapter, here are the principal additional points of note regarding JavaScript functions:

- When the function is called, the arguments passed in are assigned to the parameters in the function definition.
- These arguments can then be used within the function body.

[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

- Parameters are always passed by value in JavaScript but when reference types, such as arrays, are used, the behaviour can appear pass-by-reference.
- If the number of arguments passed in is less than the number of parameters in the function definition, the unassigned parameters are given the value *undefined*.
- Variables declared within the function using the *var* statement are local. That means that they are scoped to that function and are not visible outside the function.
- Functions in JavaScript are objects and they define a number of pre-defined properties.
- Two function properties are important for this chapter. These are the *arguments* object and the *length* property.
- The *arguments* object is an array-like list that stores all the arguments passed to the function when it is called.
- The *arguments* object is not an array but, like the *Array* type, it does have a *length* property that stores the number of arguments that were actually given when the function was called and its elements can be accessed using array-like indexing
- The *length* property of the function stores the number of arguments the function expects based on the number of parameters in the function definition.
- Since JavaScript functions can accept any number of arguments of any type regardless of the parameter list, the *arguments.length* value can be compared to the function *length* property to check if the argument count is as expected.
- Functions can have an explicit *return* statement. If no *return* statement is specified, the function will return the value *undefined*.
- User-defined functions without a *return* statement are pointless.
- A *return* statement on its own can be used to exit the function

and will return *undefined*, again, not much use in user-defined functions.

- User-defined functions should always return some value other than *undefined*.
- The returned value can be a primitive such as a string, Boolean, or a number. It can also be a reference type such as a JavaScript object or an array.

To see some of these points in action, paste the following code example 3.3 into the Script Script editor and choose and run the second function named *call_testFunc*. The point of this example is to show how the number of function arguments and the argument data types can be determined in JavaScript.

Code Example 3.3

```
/**
 * Function to demonstrate how to check
 * the number and types of passed arguments.
 *
 *
 * @return {undefined}
 */
function testFunc(arg1, arg2) {
  var i;
  Logger.log('Number of arguments given: ' +
              arguments.length);
  Logger.log('Number of arguments expected: ' +
              testFunc.length);
  for (i = 0; i< arguments.length; i += 1) {
    Logger.log('The type of argument number ' +
                (i + 1) + ' is ' +
                       typeof arguments[i]);
  }
}
```

The function output demonstrates how JavaScript functions can check the number of arguments passed in and the type of each argument by:

1. Checking the argument count given when the function is called (*arguments.length*) against the argument count expected based on the parameter list in the function definition (the function *length* property).
2. Using the *typeof* operator to determine the type of each given argument. The *typeof* operator can also be used to identify missing argument values where the missing value will be of type *undefined*. This check can be used to assign defaults using the statement *if (typeof arg === 'undefined') { arg = default; }*.

3.6 User-Defined Functions Versus JavaScript Functions

When user-defined functions are called from within spreadsheet cells, their string arguments **must be provided in double quotes**. In JavaScript, either single or double quotes can be used to enclose a string literal but only double quotes are acceptable for string literals in user-defined functions. Single quote usage can lead to puzzling errors so beware!

JavaScript function names and variable names are case-sensitive, *function afunc () { … }* and *function aFunc () { … }* are two different functions (unlike VBA). However, when called from a spreadsheet, the function names are **case-insensitive**. The implication of this is to be very careful when naming functions to be called from a spreadsheet. Camel casing is the JavaScript standard and is used throughout here except when defining testing functions that may be given the prefix "testing"/"*run*" or when writing user-defined functions where the function names are all in uppercase.

3.7 Using JSDoc To Document Functions

JSDoc is a markup language, inspired by its Java equivalent called Javadoc, for adding comments and structured annotation to JavaScript code. A sub-set of JSDoc can be used to document user-defined functions. The JSDoc markup can then be extracted using various tools to auto-generate documentation. It is defined by a special type of multi-line commenting where the opening comment tag is defined with a special opening tag of a forward slash followed by a double asterisk:

```
1  /**
2  * Line describing the function
3  * @param {data type} parameter name
4  * @return {data type}
5  * @customfunction
6  */
```

The documentation example above contains a description line and three types of annotation tags denoted by a leading "@". Each function paramater should be assigned an *@param* tag denoting the expected data type and the parameter name. Unsurprisingly, the return data type is given the *@return* tag. The final tag is called *@customfunction* and it is used to signify that the function is is intended to be called as a user-defined function. When JSDoc is used with this tag, the user-defined functions appear in the autocomplete on entering the function name preceded by an equals sign in a spreadsheet cell.

3.8 Checking Input And Throwing Errors

JavaScript functions are flexible and can be called with any number of arguments of any type. Stricter languages offer some protection

against bad argument values. Most languages throw errors if the passed argument count does not match the parameter count while static languages such as Java check the given data types of function and method arguments. JavaScript takes a very relaxed approach but inappropriate argument types can cause errors or, more seriously, bugs where the return value may not be what you expect. For example:

Code Example 3.4

```
// Function that is expected
// to add numbers but will also
// "add" strings.
function adder(a, b) {
  return a + b;
}
// Test "adder()" with numeric arguments
// and with one numeric and one string
// argument.
function run_adder() {
  Logger.log(adder(1, 2));
  Logger.log(adder('cat', 1));
}
```

The problem here is twofold. Firstly, the *adder()* function takes arguments of any type. Secondly, the + operator can do string concatenation as well as numeric addition. When one of the operands is a string, JavaScript *silently* coerces the other argument to type *string*. Although this example is contrived, it does illustrate a potential source of bugs. One way to guard against this is to explicitly check each argument type and then throw an error if any of the arguments is of the wrong type. The *typeof* operator can be used to do this. Here is a version of the *adder()* function that does exactly this:

Code Example 3.5

```
// Function that is expected
// to add numbers but will also
// "add" strings.
function adder(a, b) {
  return a + b;
}

// Function that checks that
// both arguments are of type number.
// Throws an error if this is not true.
function adder(a, b) {
  if (!(typeof a === 'number' &&
        typeof b === 'number')) {
    throw TypeError(
          'TypeError: ' +
         'Both arguments must be numeric!');
  }
  return a + b;
}

// Test "adder()" with numeric arguments
// Thrown error is caught, see logger.
function run_adder() {
  Logger.log(adder(1, 2));
  try {
    Logger.log(adder('cat', 1));
  } catch (error) {
    Logger.log(error.message);
  }
}
```

Now the function *adder()* throws an error if either of the arguments is non-numeric. The test function *run_adder()* uses a *try .. catch* construct to deal with the error. Detecting and dealing with incorrect arguments in functions may or may not be a priority depending

on circumstances. The *typeof* operator is adequate for primitive data types such as numbers and strings but limited for objects. For example, it reports arrays as type *object*, true but not very useful. When passing objects to functions, the *instanceof* operator is better suited for type checking.

3.9 Encapsulating A Complex Calculation

The **relative standard deviation**[13] (RSD) is frequently used in statistics to express and compare variability of data sets. It is not provided as a built-in spreadsheet function. The formula to calculate it is simply the sample standard deviation divided by the sample mean multiplied by 100. Given the following values in cells A1 to A10: 19.81 18.29 21.47 22.54 20.17 20.1 17.61 20.91 21.62 19.17 The RSD rounded to two decimal places can be calculated using this spreadsheet formula:

```
1  =ROUND(100*(STDEV(A1:A10)/AVERAGE(A1:A10)),2)
```

The functionality expressed in this spreadsheet formula can be encapsulated in a user-defined function as follows:

Code Example 3.6

[13] http://en.wikipedia.org/wiki/Relative_standard_deviation

```
/**
 * Given the man and standard deviation
 * return the relative standard deviation.
 *
 * @param {number} stdev
 * @param {number} mean
 * @return {number}
 * @customfunction
*/
function RSD (stdev, mean) {
  return 100 * (stdev/mean);
}
```

This function can be called as follows:

```
=RSD(STDEV(A1:A10),AVERAGE(A1:A10))
```

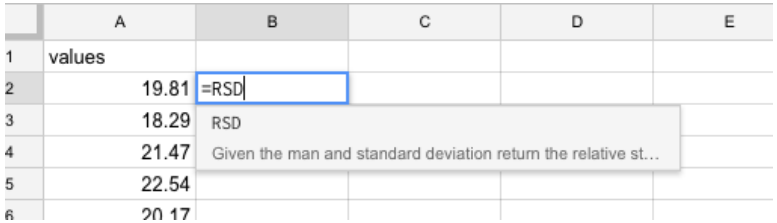


Figure 3-3: *Calling the RSD function in Google Sheets with provided autocomplete.

The above function can be pasted into the Script Editor and called as described in the documentation. So what was gained by writing a user-defined function when a combination of spreadsheet built-in functions can be used and the user-defined version still uses the spreadsheet STDEV and AVERAGE built-ins? Firstly, the calculation now has a meaningful name. Secondly, argument checking can be added as described in the previous section. Lastly, the logic is captured in one place and is documented. These are all good reasons

to consider replacing complex spreadsheet formulas with user-defined functions. In addition, our JSDoc description now appears in the autocomplete.

3.10 Numeric Calculations

The following example takes a single numeric argument as degrees Centigrade and uses that argument to return the temperature in Fahrenheit. To use, paste the function into the Google Apps Script editor and then call it by name in a cell in the containing spreadsheet. For example, if cell A1 contains the value 0, then the formula *=CELSIUSTOFAHRENHEIT(A1)* in cell B1 will return the result 32 as expected.

Code Example 3.7

```
/**
 * Given a temperature in Celsius, return Fahrenheit va\
lue.
 *
 * @param {number} celsius
 * @return {number}
 * @customfunction
 */
function CELSIUSTOFAHRENHEIT(celsius) {
  if (typeof celsius !== 'number') {
    throw TypeError('Celsius value must be a number');
  }
  return ((celsius * 9) / 5) + 32;
}
```

Function *CELSIUSTOFAHRENHEIT* takes a numeric value representing degrees Celsius and returns the Fahrenheit equivalent. Before performing the calculation, it first checks that the given value is

numeric, if not, it throws a type error. Here is a screenshot showing some input values and the outputs after calling this function:

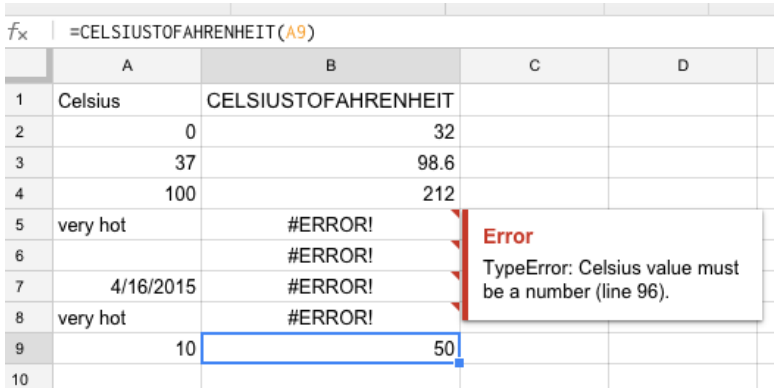


Figure 3-4: Input and output for user-defined function *CELSIUSTOFAHRENHEIT* and .

See how the thrown error is reported as a comment in cells A5 to A8. Without the *typeof* check, the date value in cell A7 would be interpreted as a number of seconds and would return an unfeasibly large value. The outcome of evaluating the cell A6 is even more insidious because the **blank is interpreted as zero** and the function evaluates to 32. Indeed, 0 Celsius is equal to 32 Fahrenheit but this is highly misleading. Remove or comment out the *typeof* check and see the effect! The text value in cell A5 will throw an error regardless of the *typeof* check, which is to be expected. However, the default interpretation of dates as numbers and blank cells as zero is a serious trap for the unwary. The same problem occurs in Excel VBA and is more difficult to guard against. This example of a user-defined function is very simple but it highlights the need for rigorous checking of inputs.

Verifying that arguments are within a valid range may also be required to avoid invalid return values. As an example, the formula to calculate the area of a circle is simple: $\Pi * radius^2$. The following function does this but it also checks that the given argument is both

numeric and not less than zero.

Code Example 3.8

```
/**
 * Given the radius, return the area of the circle.
 * @param {number} radius
 * @return {number}
 * @customfunction
 */
function AREAOFCIRCLE (radius)  {
  if (typeof radius !== 'number' || radius < 0){
    throw Error('Radius must be a positive number');
  }
  return Math.PI * (radius * radius);
}
```

In example 3.8, the function expects a single numeric argument for the circle radius. However, since a negative radius makes no sense in this context, there is an additional check to ensure that the radius given is not negative. If either of these conditions fails, an error is thrown (|| is the *or* operator). In this example, both checks were performed togeter using the *or* operator. If greater granularity is required, the checks could be performed in two separate checks that would yield two different errors. This approach helps when functions are part of larger applications.

3.11 Date Functions

Google Sheets provides a comprehensive set of date functions, see here[14] for the full list. However, if the provided functions do not cover your requirements, you can write your own in GAS and

[14] https://support.google.com/docs/table/25273?hl=en&rd=2

call them from your spreadsheet as user-defined functions. Some cautionary words: Dates and times can be quite complicated in any application or programming language where you have to allow for time zones, daylight saving time changes and leap years so rigorous testing of inputs and outputs is highly advidable.

The examples given here also cover some important basic JavaScript concepts such as date manipulation, objects and methods and JavaScript arrays.

The first example (3.9) takes a date as its sole argument and returns the day name (in English) for that date.

Code Example 3.9

```
/**
 * Given a date, return the name of the
 * day for that date.
 *
 * @param {Date} date
 * @return {String}
 * @customfunction
 */
function DAYNAME(date) {
  var dayNumberNameMap = {
    0: 'Sunday',
    1: 'Monday',
    2: 'Tuesday',
    3: 'Wednesday',
    4: 'Thursday',
    5: 'Friday',
    6: 'Saturday'},
      dayName,
      dayNumber;
  if(! date.getDay ) {
    throw TypeError(
```

```
22          'TypeError: Argument is not of type "Date"!');
23      }
24      dayNumber = date.getDay();
25      dayName = dayNumberNameMap[dayNumber];
26      return dayName;
27  }
```

The most interesting points in the code example 3.9 are the use of the object named *dayNumberNameMap*, how the argument type is checked and how the *Date* object method *getDay()* is used. The variable *dayNumberNameMap* is a JavaScript object. Since objects are cornerstones of JavaScript programming, here is a short detour that describes them. At their simplest, JavaScript objects provide a mapping of keys to values. The values are called "properties" and they can be other objects, scalar values, arrays or functions. When object properties are functions, they are known as **methods**. JavaScript objects are analogous to hashes or dictionaries in other languages. The very popular data interchange format called JavaScript Object Notation (JSON) is based on them. Excel VBA offers a *Dictionary Object* from the Microsoft Scripting Library[15] that provides similar functionality but is much less flexible. The object in the example above maps index numbers to the names of the days of the week. Returning to the code example above, the *date* argument is expected to be of type *Date* and the mechanism used to test for it, that is the line *if(! date.getDay) {*, merits some explanation. If the argument is a *Date* object, then it will implement a *getDate()* method. The test here is to see if the property exists, notice how the *getDate()* method is not called because there are no parentheses. If the argument does not have a *getDate* property, then an error is raised. This type of checking for methods on objects is common in client-side (browser) JavaScript where implementations from different vendors may or may not provide certain methods. If the argument passes the type check, then we are satisified that it is a

[15] https://support.microsoft.com/en-us/kb/187234/

Date object so we can call the *getDay()* method on it. The *getDay()* method returns the week day as an integer starting with 0 (zero) for Sunday. Having the day number, we can use it as key for our object *dayNumberNameMap* to return the day name. To use the function *DAYNAME* as a user-defined function, copy the code into the Script Editor and call it from the spreadsheet like this where the return value of the built-in *TODAY()* function is given as an argument:

```
=DAYNAME(TODAY())
```

All the user-defined functions given so far return a scalar value but they can also return arrays where each array element is entered into its own cell. An example will help to illustrate this. In this example, we write a function that returns all dates that fall on a given day between two dates.

Code Example 3.10

```
/**
* Add a given number of days to the given date
* and return the new date.
*
* @param {Date} date
* @param {number} days
* @return {Date}
*/
function addDays(date, days) {
  // Taken from Stackoverflow:
  // questions/563406/add-days-to-datetime
    var result = new Date(date);
    result.setDate(result.getDate() + days);
    return result;
}

/**
```

```
18  * Given a start date, an end date and a day name,
19  * return an array of all dates that fall (inclusive)
20  * between those two dates for the given day name.
21  *
22  * @param {Date} startDate
23  * @param {Date} endDate
24  * @param {String} dayName
25  * @return {Date[]}
26  * @customfunction
27  */
28 function DATESOFDAY(startDate, endDate, dayName) {
29   var dayNameDates = [],
30       testDate = startDate,
31       testDayName;
32   while(testDate <= endDate) {
33     testDayName = DAYNAME(testDate);
34     if(testDayName.toLowerCase() ===
35       dayName.toLowerCase()) {
36       dayNameDates.push(testDate);
37     }
38     testDate = addDays(testDate, 1);
39   }
40   return dayNameDates;
41 }
```

The code in example 3.10 defines a user-defined function called *DATESOFDAY()* that calls two other GAS functions, *addDays()* and *DAYNAME()* to perform some of the computation. Function *DAYNAME()* was described in code example 3.9 above will be needed when you paste this code into the Script Editor. The function *addDays()* was copied from this location on Stackoverflow.[16]. I stated earlier that date manipulation can be complex and the discussion thread on this Stackoverflow entry certainly proves that.

[16] http://stackoverflow.com/questions/563406/add-days-to-datetime

Before discussing the actual code in example 3.10 in detail, first note how a user-defined function can call other GAS functions to perform some of its tasks as can be seen in this example where two other functions are called and their return values are used by *DATESOFDAY()*. By off-loading much of its work, *DATESOFDAY()* remains small on manageable.

Make Functions Small

De-composing a programming task into multiple small functions makes each function easier to test, de-bug and re-use

DATESOFDAY() takes two dates and a day name as argument. I have not added any checking of types or numbers of argument, these could be easily added using the techniques discussed earlier. The function declares two local variables, *dayNameDates* and *testDate*. It initialises *testDate* to *startDate* and then enters the *while* loop. Each round of the *while* loop gets the name of the day for the *testDate* using the *DAYNAME* function described in code example 3.9. The returned day name is then compared to the given argument day name (*dayName*) using a **case-insensitive** comparison by calling the string method *toLowerCase()* on both string operands. If they are equal, the date is added to the array using the *Array push()* method. This technique of building arrays in loops will be seen throughout the book. JavaScript arrays are extremely flexible and implement a range of very useful methods. Many of these methods will be encountered later in this book. After testing the day of the date, the test date is incremented by one day using the *addDays()* function. The condition *testDate <= endDate* is tested in the next round of the *while* loop and the loop finishes when the condition evaluates to *false* and the *dayNameDates* array is returned.

When the function *DATESOFDAY()* is called as a user-defined

function, it writes the entire array to a single column, see the screenshot below:

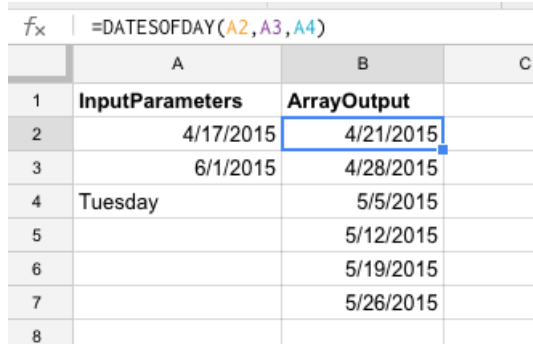


Figure 3-5: Calling user-defined function that returns an array of values.

3.12 Text Functions

Google Spreadsheets text manipulation functions can also be complemented and enhanced by writing user-defined functions. JavaScript, like most programming languages, refers to the text type as "string". Strings in JavaScript are primitive types but they can be wrapped to behave like objects. The wrapping of a primitive so that it becomes an object is known as **auto-boxing**. Auto-boxing can also be applied to *Boolean* and *Number* primitive types and though I have never seen any use for it, with *Boolean*. The *Number* type has some useful methods like *toFixed()*. Since auto-boxing is transparent to the user/programmer, all that really matters is that strings can be treated as though they have methods that can be used to extract information from them and to generate new strings.

To see all the String properties supported by GAS/JavaScript, paste the following code into the script editor and execute it. An alphabetical list of string properties is sent to the logger.

Code Example 3.11

```
/**
 * Print all string properties to the
 *  Script Editor Logger
 * @return {undefined}
 */
function printStringMethods() {
  var strMethods =
    Object.getOwnPropertyNames(
      String.prototype);
  Logger.log('String has ' +
              strMethods.length +
             ' properties.');
  Logger.log(strMethods.sort().join('\n'));
}
```

The code above uses JavaScript's very powerful introspection capabilities to extract an array of all the property names defined for object *String* on what is known as its **prototype**. The prototype is a property that refers to an object where the *String* methods are defined in the prototype. Prototypes are quite an advanced topic but they are very important and provide the basis for inheritance in JavaScript. All JavaScript objects (that includes arrays and functions) have a prototype property. The logger output shows that there are **38** *String* properties. Most are methods but the important *length* property, for example, is not. Code example 3.11 also uses two important *Array* methods: *sort()* and *join()*. The *sort()* method does an alphabetical sort in this example and *join()* then makes a string of the array using the given argument (new line '\n' in this example) to separate the concatenated array elements. Notice how the method calls are "chained" one after the other. This is a common approach in JavaScript. As an exercise, you could try writing an equivalent function to print out all the array properties.

Since the *String* object has so many methods, we can use these methods in user-defined functions. Here is a function that returns

the argument string in reverse character order.

Code Example 3.12

```
/**
 * Given a string, return a new string
 * with the characters in reverse order.
 *
 * @param {String} str
 * @return {String}
 * @customfunction
 */
function REVERSESTRING(str) {
  var strReversed = '',
      lastCharIndex = str.length - 1,
      i;
  for (i = lastCharIndex; i >= 0; i -= 1) {
    strReversed += str[i];
  }
  return strReversed;
}
```

Reversing strings might not strike you as the most useful functionality unless of course you work with DNA or are interested in palindromes. To see a palindrome example, call the function with the word *detartrated*, yes, that word really exists. The example function uses a JavaScript *for* loop to extract the string characters in reverse order and appends them to a new string that is built within the loop. The newly built string is returned upon completion of the loop. If you declare the variable *strReversed* but do not assign it, something unexpected happens. As the variable is unassigned, it is *undefined*. If you concatenate a string to *undefined* you get another of those troublesome implicit JavaScript conversions that I warned about earlier where *undefined* becomes the string "undefined! Beware!

Spreadsheets generally offer a large set of built-in text functions and Google Sheets augments these with regular expression functions. I have covered regular expressions in two blog entries (see part 1here[17] and part 2 here[18].) I will include a chapter on regular expression later in this book and will give some regular expression-based user-defined function examples there.

3.13 Using JavaScript Built-In Object Methods

JavaScript provides a small number of built-in objects that provide additional functionality that can be used in user-defined functions. For example, the JavaScript *Math* object is available in GAS and provides some useful methods. Simulation of a die throw can be easily implemented using its methods:

Code Example 3.13

```
/**
 * Simulate a throw of a die
 * by returning a number between
 * and 6.
 *
 * @return {number}
 * @customfunction
 */
function THROWDIE() {
  return 1 + Math.floor(Math.random() * 6);
}
```

[17] http://www.javascript-spreadsheet-programming.com/2013/09/regular-expressions.html

[18] http://www.javascript-spreadsheet-programming.com/2014/09/regular-expressions-part-2.html

Try out the function by calling *=THROWDIE()* from any spreadsheet cell.

This function uses two methods defined in the JavaScript built-in *Math* object to simulate dice throwing. The *Math round()* method could also be used here but, as is very well described here[19], it leads to biased results. This becomes apparent when the function is called hundreds of times. The important point to note is the need for testing and reading of documentation to avoid inadvertent errors and biases.

3.14 Using A Function Callback

The object nature of JavaScript functions allows them to be both passed as arguments to other functions and to be returned by other functions. The following example provides an example of the power and flexibility of JavaScript. It also shows how easy it is to use a range of cells as an argument. The function concatenates an input range of cells using a delimiter argument. There is an in-built Google Spreadsheet function called JOIN that performs this task but the user-defined version below has one additional feature: It provides an option to enclose each individual element in single quotes. Here is the code:

Code Example 3.14

[19] http://www.the-art-of-web.com/javascript/random/#.UPU8tKHC-nY

```
/** Concatenate cell values from
* an input range.
* Single quotes around concatenated
* elements are optional.
*
* @param {String[]} inputFromRng
* @param {String} concatStr
* @param {Boolean} addSingleQuotes
* @return {String}
* @customfunction
*/
function CONCATRANGE(inputFromRng, concatStr,
                     addSingleQuotes) {
  var cellValues;
  if (addSingleQuotes) {
    cellValues =
      inputFromRng.map(
        function (element) {
          return "'" + element + "'";
        });
    return cellValues.join(concatStr);
 }
   return inputFromRng.join(concatStr);
}
```

An example of a call to this function in the spreadsheet is *=CONCATRANGE(A1:A5, ",", true)*. This returns output like 'A','B','C','D','E'. There are two interesting aspects to this code:

1. The range input is converted to a JavaScript array. In order to work correctly, all the input must be from the same column. If the input range is two-dimensional or from multiple cells in the same row, it will generate a JavaScript array-of-arrays. That point is ignored here, but try giving it a two-dimensional range input or something like A1:C2" and observe the output!

2. The more interesting point is the use of the *Array map()* method. This method iterates the array and applies its function argument to each element of the array and returns a new array. The function used here is an **anonymous function**. It takes one argument, the array element, and returns it enclosed in single quotes. The anonymous function operates as a **callback**. More examples of callbacks will be given in later chapters. This example just hints at the power of this approach.

A version of this function was written in Excel VBA as a quick way of generating input for the *IN* clause in SQL. Lists of string values (VARCHARs in database terminology) would be provided in spreadsheets so an Excel user-defined function was written to allow the lists to be concatenated in comma-separated strings with the constituent values enclosed in single quotes. There are better ways of running such queries that will be discussed in the JDBC chapter but this is an approach that can be useful.

It is worth noting that built-in Google Spreadsheets can be combined to get the same output:

```
1  =CONCATENATE("'", JOIN("','", A1:A5), "'")
```

However, no introduction to JavaScript functions would be complete without at least a brief mention of anonymous functions and callbacks.

3.15 Extracting Useful Information About The Spreadsheet

All the user-defined functions discussed up to this point were written in host-independent JavaScript. They should all work in modern browsers or in Node.js scripts.

Running Examples In Node.js

Replace *Logger.log()" with *console.log()"*

The restrictions on the actions of user-defined functions were also discussed. Despite these restrictions, useful information about a spreadsheet can be returned to spreadsheet cells by means of user-defined functions. The examples given below are all functions that take no arguments and return some useful information about the active spreadsheet. It is worth noting how they all use *get* methods, any attemp to call *set* methods would not work.

Code Example 3.15

```
/**
 * Return the ID of the active
 *  spreadsheet.
 *
 * @return {String}
 * @customfunction
 */
function GETSPREADSHEETID() {
  return SpreadsheetApp
    .getActiveSpreadsheet().getId();
}

/**
 Return the URL of the active
 *  spreadsheet.
 *
 * @return {String}
 * @customfunction
 */
function GETSPREADSHEETURL() {
  return SpreadsheetApp
```

```
22      .getActiveSpreadsheet().getUrl();
23  }
24
25  /**
26   * Return the owner of the active
27   *  spreadsheet.
28   *
29   * @return {String}
30   * @customfunction
31   */
32  function GETSPREADSHEETOWNER() {
33    return SpreadsheetApp
34      .getActiveSpreadsheet().getOwner();
35  }
36
37  /**
38   *Return the viewers of the active
39   *  spreadsheet.
40   *
41   * @return {String}
42   * @customfunction
43   */
44
45  function GETSPREADSHEETVIEWERS() {
46    var ss =
47      SpreadsheetApp.getActiveSpreadsheet();
48    return ss.getViewers().join(', ');
49  }
50
51  /**
52   * Return the locale of the active
53   *  spreadsheet.
54   *
55   * @return {String}
56   * @customfunction
```

```
 */
function GETSPREADSHEETLOCALE() {
  var ss = SpreadsheetApp.getActiveSpreadsheet();
  return ss.getSpreadsheetLocale();
}
```

These examples use GAS host objects and methods. The objects in the code examples are discussed extensively in the following two chapters, so read ahead if you wish to understand them right now. The purpose here is to just show how user-defined functions can be utilised to extract information about a spreadsheet using host object methods.

All the functions given above can extract the required information and return it in a single statement. The functions *GETSPREADSHEETVIEWERS()* and *GETSPREADSHEETLOCALE* use a variable to reference the active spreadsheet and the value returned using a method call to the variable. The reason for doing this here was simply to make the code more readable by avoiding a wrapped line in the book text.

To see these functions in action, just paste the code above into any Google Spreadsheet Script Editor and then invoke each of them from the spreadsheet using the usual syntax, example: *=GETSPREADSHEETID()*. All of the examples except *GETSPREADSHEETVIEWERS()* use a *Spreadsheet* method that returns a primitive JavaScript value, namely a string. The *getUsers()* method however returns an array and the function uses the *Array* type *join()* method to convert the array to a string. The *join()* call could be omitted and the function would still work. However, each user would be written to a separate cell. To experiment, remove the *join()* and then call the function.

3.16 Using Google Services

Some Google Services can be used in user-defined functions. The following example is taken from the Google Language service.

User-defined functions can be used as a sort of dictionary to translate words and phrases. The following function can all be called from the spreadsheet to perform translations from English to French.

Code Example 3.16

```
/**
 * Return French version
 * of English input.
 *
 * @param {String} input
 * @return {String}
 */
function ENGLISHTOFRENCH(input) {
  return LanguageApp
    .translate(input, 'en', 'fr');
}
```

Example invocation:

```
=ENGLISHTOFRENCH("Try closing the file!")
```

The output: “Essayez de fermer le fichier!”

If you wish to translate into, for example, Spanish or German, replace “fr” with “es” and “de”, respectively. These functions appear to work well for single words and straight-forward, non-idiomatic, phrases. However, only a fluent speaker could comment on the translation quality. A good check is to re-translate back into the

original and see if it makes sense. I have tried some idiomatic input such as “JavaScript rocks!” and “JavaScript sucks!” - translated into standard English as “JavaScript is really great” and “JavaScript is very bad!”- and the translations, unsurprisingly, missed the point.

3.18 Summary

- Google Spreadsheets provides a large number of built-in functions.
- User-defined functions can be written to complement these functions and to provide new functionality.
- Before writing a user-defined function, ensure that Google Spreadsheets does not already provide an equivalent.
- User-defined functions cannot be used to set cell properties or display dialogs.
- User-defined functions are written in the Google Apps Scripting version of JavaScript.
- JavaScript functions are very powerful and very flexible.
- User-defined functions should be rigorously tested.
- There should be code to check and verify the number and type of arguments that are passed when the user-defined function is called. The JavaScript *typeof* and *instanceof* operators can be used for these purposes.
- A variety of invalid data should be passed as arguments to ensure that the user-defined function performs as expected by throwing an error.
- Use the *throw* statement with an error object (example *TypeError*) with a message for the error. Calling code can then use *try catch* statements to deal with the error.
- Ensure that the user-defined function deals appropriately with blank cells.
- Ensure that user-defined functions expecting numeric arguments do not misinterpret date input.

- Watch out for implicit JavaScript or spreadsheet type conversions.
- When using JavaScript built-in functions or methods provided by the *Math* library for example, read the documentation and test function output against expected results to avoid errors and biases.
- The object nature of JavaScript functions allows them to be used as callbacks.
- Useful information about a Google Spreadsheet document can be retrieved with user-defined functions that call methods of the *SpreadsheetApp* class.
- User-defined functions can be used to query Google services such as *LanguageApp* in order to return useful information to a spreadsheet.

Appendix A: Excel VBA And Google Apps Script Comparison

Introduction

Microsoft Excel™ remains the dominant spreadsheet application so many of those coming to Google Apps Script programming will be very familiar with it. It hosts a programming language called Visual Basic for Applications™ (VBA) that can be used to extend functionality, build user interfaces, integrate with other Microsoft Office™ applications, and as a front end to relational databases. This appendix aims to provide a quick reference for Excel VBA programmers by giving some VBA examples in parallel with equivalent Google Apps Script (basically JavaScript) code for some common spreadsheet programming tasks. VBA and Google Apps Script are very different languages and the APIs they each use for spreadsheet programming have been developed independently so their respective methods and properties also differ. Despite these differences, the objectives of both languages and the tasks they are applied to are similar to the extent that an experienced VBA programmer should be able to pick up Google Apps Script quite quickly. This appendix assumes knowledge of VBA and aims to facilitate the reader's transition to the Google Apps Script environment.

The examples below are given in pairs: First the VBA code and then the **functionally equivalent** Google Apps Script version with some explanatory comments. The VBA is generally not commented because the examples assume VBA expertise. Comments are added,

however, for trickier and longer examples or when the code examples in the two languages are very different due to inherent VBA/JavaScript differences or divergent API approaches. The code examples should perform as described but in many instances alternative approaches can be used and the examples may not be optimal.

The VBA examples given generally write their output to the **Immediate Window** while the Google Apps Script equivalents write to the **Logger**. Some examples write to or format spreadsheet cells.

> **Google Apps Script Or JavaScript**
>
> "JavaScript" is used here to refer to general JavaScript concepts such as arrays.
>
> Google Apps Script refers to the specific Google implementation as it applies to spreadsheet programming and Google App Script APIs. The context should make the usage clear.

Spreadsheets and Sheets

Spreadsheets and sheets are handled similarly. One difference of note is that Google Apps Script does not make a distinction between *Sheets* and *Worksheets* as VBA does.

Active Spreadsheet

Multiple spreadsheet files can be open at a given time but only one is *active*.

VBA

```
Public Sub SpreadsheetInstance(()
    Dim ss As Workbook
    Set ss = Application.ActiveWorkbook
    Debug.Print ss.Name
End Sub
```

Google Apps Script

```
function spreadsheetInstance() {
  var ss = SpreadsheetApp.getActiveSpreadsheet();
  Logger.log(ss.getName());
}
```

VBA uses the *Name* property while Google Apps Script uses the method *getName()* to return the value.

Sheet/Worksheet

Spreadsheets contain sheets. In VBA these are stored as *collections* and in Google Apps Script as JavaScript arrays of *Sheet* objects. The pairs of examples given here call some *Sheet* methods and print the output.

VBA

```
Public Sub FirstSheetInfo()
    Dim sh1 As Worksheet
    Set sh1 = ActiveWorkbook.Worksheets(1)
    Dim usedRng As Range
    Set usedRng = sh1.UsedRange
    Debug.Print sh1.Name
    Debug.Print usedRng.Address
End Sub
```

Google Apps Script

```
function firstSheetInfo() {
  var ss = SpreadsheetApp.getActiveSpreadsheet(),
      sheets = ss.getSheets(),
      // getSheets() returns an array
      // JavaScript arrays are always zero-based
      sh1 = sheets[0];
  Logger.log(sh1.getName());
  // getDataRange is analagous to UsedRange
  //in VBA
  // getA1Notation() is functional equivalent to
  //  Address in VBA
  Logger.log(sh1.getDataRange().getA1Notation());
}
```

Sheet Collections

The previous examples extracted a single *Sheet* object and called some of its methods. The example pairs here loop over the all the sheets of the active spreadsheet and print the sheet names.

VBA

```
Public Sub PrintSheetNames()
    Dim sheets As Worksheets
    Dim sheet As Worksheet
    For Each sheet In ActiveWorkbook.Sheets
        Debug.Print sheet.Name
    Next sheet
End Sub
```

Google Apps Script

```
// Print the names of all sheets in the active
//  spreadsheet.
function printSheetNames() {
  var ss = SpreadsheetApp.getActiveSpreadsheet(),
      sheets = ss.getSheets(),
      i;
  for (i = 0; i < sheets.length; i += 1) {
    Logger.log(sheets[i].getName());
  }
}
```

Adding And Removing Sheets

Spreadsheet applications may need to add new sheets to an existing spreadsheet file and then, after some processing, they may need to then remove one or more sheets. Both tasks are easily achieved in both VBA and and Google Apps Script.

VBA

```
' Add a new sheet to a workbook.
' Call the Add method of the
'  Worksheets collection
' Assign a name to the returned
'  Worksheet instance
'   Name property.
Sub AddNewSheet()
    Dim newSheet As Worksheet
    Set newSheet = ActiveWorkbook.Worksheets.Add
    newSheet.Name = "AddedSheet"
    MsgBox "New Sheet Added!"
End Sub

' Delete a named sheet from the
'  active spreadsheet.
' The sheet to delete is identified
'   in the Worksheets collection
' by name. The returned instance
'  is deleted by calling its
' Delete method.
' MS Excel will prompt to confirm.
Sub RemoveSheet()
    Dim sheetToRemove As Worksheet
    Set sheetToRemove = _
     ActiveWorkbook.Worksheets("AddedSheet")
    sheetToRemove.Delete
    MsgBox "Sheet Deleted!"
End Sub
```

Google Apps Script

```
// Add a new sheet to the active spreadsheet.
// Get an instance of the active spreadsheet.
// Call its insertSheet method.
// Call the setName method of the
//  returned instance.
function addNewSheet() {
  var ss =
     SpreadsheetApp.getActiveSpreadsheet(),
      newSheet;
  newSheet = ss.insertSheet();
  newSheet.setName("AddedSheet");
  Browser.msgBox("New Sheet Added!");
}

// Remove a named sheet from the
//  active spreadsheet.
// Get an instance of the active
//  spreadsheet.
// Get an instance of the sheet to remove.
// Activate the sheet to remove
// Call the spreadsheet instance method
//   deleteActiveSheet.
function removeSheet() {
  var ss =
   SpreadsheetApp.getActiveSpreadsheet(),
   sheetToRemove =
       ss.getSheetByName("AddedSheet");
  sheetToRemove.activate();
  ss.deleteActiveSheet();
  Browser.msgBox("SheetDeleted!");
}
```

The code comments in both languages should adequately describe the actions and objects required to add and remove sheets from both spreadsheet applications. The Google Apps Script mechanism

appears a little more complicated than its VBA equivalent. In order to remove a sheet, it first has to be activated so that the *Spreadsheet* instance method *deleteActiveSheet()* can be called. Otherwise, both languages work quite similarly.

Hiding And Unhiding Sheets

Hiding sheets can help to keep a spreadsheet uncluttered and easy to use while also helping to prevent inadvertent changes to important data. Lists of values that the application uses may not be important to the users so they can be hidden from their view while still remaining available to the application code. The VBA and Google Apps Script code required to do the hiding, unhiding and listing of hidden sheets is very similar.

Hiding a sheet identified by name.

VBA

```
1  Public Sub SheetHide()
2      Dim sh As Worksheet
3      Set sh = Worksheets.Item("ToHide")
4      sh.Visible = False
5  End Sub
```

Google Apps Script

```
// Hide a sheet specified by its name.
function sheetHide() {
  var ss =
    SpreadsheetApp.getActiveSpreadsheet(),
      sh = ss.getSheetByName('ToHide');
  sh.hideSheet()
}
```

Listing hidden sheets

VBA

```
Public Sub ListHiddenSheetNames()
    Dim sheet As Worksheet
    For Each sheet In Worksheets
        If sheet.Visible = False Then
            Debug.Print sheet.Name
        End If
    Next sheet
End Sub
```

Google Apps Script

```
// Write a list of hidden sheet names to log.
function listHiddenSheetNames() {
  var ss =
    SpreadsheetApp.getActiveSpreadsheet(),
      sheets = ss.getSheets();
  sheets.forEach(
    function (sheet) {
      if (sheet.isSheetHidden()) {
        Logger.log(sheet.getName());
      }
    });
}
```

Unhiding hidden sheets

VBA

```
Public Sub SheetsUnhide()
    Dim sheet As Worksheet
    For Each sheet In Worksheets
        If sheet.Visible = False Then
            sheet.Visible = True
        End If
    Next sheet
End Sub
```

Google Apps Script

```
// Unhide all hidden sheets.
function sheetsUnhide() {
  var ss =
    SpreadsheetApp.getActiveSpreadsheet(),
      sheets = ss.getSheets();
  sheets.forEach(
    function (sheet) {
      if (sheet.isSheetHidden()) {
        sheet.showSheet();
      }
    });
}
```

The main difference in the approach taken by each language in these examples is how they iterate over the *Worksheets* collection in VBA and the array of *Sheet* objects in Google Apps Script. Newer versions of JavaScript, including Google Apps Script, have added some very powerful methods to arrays. Included in these are methods that take a callback function as an argument that is invoked for each element in the array. The *forEach()* method above

is an example. It operates in a similar manner to the VBA *For Each* loop but unlike it, *forEach()* needs a function as an argument. In the examples above anonymous functions were used. This type of approach where functions take other functions as arguments is very powerful but may be unfamiliar to VBA programmers.

Protecting Sheets

Hiding sheets provides a type of "security through obscurity" but does not prevent deliberate tampering. Both VBA and Google Apps Script allow you to protect individual worksheets within a spreadsheet but they take very approaches to this.

VBA

```
' Password-protect rotect a sheet identified
' by name
Public Sub SheetProtect()
    Dim sh As Worksheet
    Dim pwd As String: pwd = "secret"
    Set sh = Worksheets.Item("ToProtect")
    sh.Protect pwd
End Sub
```

Google Apps Script

```
// Identify a sheet by name to protect
// When this code runs, the lock icon
// will appear on the sheet name.
// Share the spreadsheet with another user
// as an editor. That user can edit all
// sheets except the protected one. The user
// can still edit the protected sheet.
function sheetProtect() {
  var ss =
    SpreadsheetApp.getActiveSpreadsheet(),
      sh = ss.getSheetByName('ToProtect'),
      permissions = sh.getSheetProtection();
  ss.addEditor(<gmail address goes here>);
  permissions.setProtected(true);
  sh.setSheetProtection(permissions);
}
```

In VBA, a password is set and the protected is using the *Worksheet Protect* method passing it the password string as an argument. Once protected even the spreadsheet file owner needs to know the password to do anything to the sheet. Google Apps Script takes a different approach. By default, only the file creator can see or edit the spreadsheet. The owner can then add editors or viewers to the spreadsheet. A viewer can see all the sheets but not edit them while the editor can, as the name suggests, edit the sheet contents. However, a single sheet can be protected so that it can be viewed by a user granted editor privilege but is not editable by them. The code example given above shows how this can be done. Unlike in VBA, however, the owner of the spreadsheet will always have full edit permissions on all sheets. In other words, the owner cannot remove permissions from themselves.

Ranges

Spreadsheet programming is largely about manipulating ranges so this is a long section.

Selection

Requiring a user to select an input range is a common feature of spreadsheet applications. In order to process the selected cells, the application needs to determine:

- The sheet containing the selection
- The location of the selection within the sheet as given by its address
- The dimensions of the selection, that is the number of rows and columns in the selection

This information is extracted and printed in the following examples

VBA

```
1  Public Sub PrintSelectionDetails()
2      Debug.Print "Selected Range Details: "
3      Debug.Print "-- Sheet: " & _
4                  Selection.Worksheet.Name
5      Debug.Print "-- Address: " & _
6                  Selection.Address
7      Debug.Print "-- Row Count: " & _
8                  Selection.Rows.Count
9      Debug.Print "'-- Column Count: " & _
10                  Selection.Columns.Count
11 End Sub
```

Google Apps Script

```
// Prints details about selected range in
//  active spreadsheet
// To run, paste code into script editor,
//   select some cells on any sheet,
//    execute code and
//   check log to see details
// Prints details about selected range
//  in active spreadsheet
// To run, paste code into script editor,
//   select some cells on any sheet,
//  execute code and
//   check log to see details
function printSelectionDetails() {
  var ss =
      SpreadsheetApp.getActiveSpreadsheet(),
      selectedRng = ss.getActiveRange();
  Logger.log('Selected Range Details:');
  Logger.log('-- Sheet: '
             + selectedRng
                .getSheet()
                .getSheetName());
  Logger.log('-- Address: '
             + selectedRng.getA1Notation());
  Logger.log('-- Row Count: '
             + ((selectedRng.getLastRow() + 1)
             - selectedRng.getRow()));
  Logger.log('-- Column Count: '
             + ((selectedRng.getLastColumn() + 1)
             - selectedRng.getColumn()));
}
```

VBA provides the handy *Selection* object which is of type *Range* and its methods can be used to extract the required information. The Google Apps Script *Spreadsheet* object provides the *getActiveSelection()* method to return the Google Spreadsheets equivalent to

the VBA *Selection*. Its *getRow()* and *getColumn()* methods return the row number of the first row and first column, respectively, for the *Range* object on which they are invoked. The purpose of the *getLastRow()* and *getLastColumn()* *Range* methods is clear from their names. By using a combination of these methods the VBA *Selection.Rows.Count* and *Selection.Columns.Count* properties can be mimicked as was done above.

Used Range

To retrieve the very useful equivalent of the VBA *UsedRange* object in Google Apps Script, use the *Sheet getDataRange()* method. In both languages it is easy to transfer the cell contents of a range into an array. JavaScript arrays are a lot more flexible than those in VBA and they are always zero-based. JavaScript's dynamic typing also makes matters more straightforward. VBA is a typed language but its *Variant* type negates the all the type-checking. However, it has to be used to receive the *Range value* property. Another fundamental language difference is that JavaScript does not distinguish functions and subroutines. Instead functions are always used and if there is no explicit *return* statement, *undefined* is the return value.

VBA

```
1  Public Function GetUsedRangeAsArray(sheetName  _
2                              As String) As Variant
3      Dim sh As Worksheet
4      Set sh = _
5        ActiveWorkbook.Worksheets(sheetName)
6      GetUsedRangeAsArray = sh.UsedRange.value
7  End Function
8  Sub test_GetUsedRangeAsArray()
9      Dim sheetName As String
10     Dim rngValues
```

```
    Dim firstRow As Variant
    sheetName = "Sheet1"
    rngValues = GetUsedRangeAsArray(sheetName)
    Debug.Print rngValues(1, 1)
    Debug.Print UBound(rngValues)
    Debug.Print UBound(rngValues, 2)
End Sub
```

Google Apps Script

```
function getUsedRangeAsArray(sheetName) {
  var ss =
    SpreadsheetApp.getActiveSpreadsheet(),
      sh = ss.getSheetByName(sheetName);
  // The getValues() method of the
  //   Range object returns an array of arrays
  return sh.getDataRange().getValues();
}
// JavaScript does not distinguish between
//  subroutines and functions.
// When the return statement is omitted,
//  functions return undefined.
function test_getUsedRangeAsArray() {
  var ss = SpreadsheetApp.getActiveSpreadsheet(),
      sheetName = 'Sheet1',
      rngValues = getUsedRangeAsArray(sheetName);
  // Print the number of rows in the range
  // The toString() call to suppress the
  // decimal point so
  //  that, for example, 10.0, is reported as 10
  Logger.log((rngValues.length).toString());
  // Print the number of columns
  // The column count will be the same
  // for all rows so only need the first row
  Logger.log((rngValues[0].length).toString());
```

```
  // Print the value in the first cell
  Logger.log(rngValues[0][0]);
}
```

Add Colours To Range In First Sheet

Cells and their contents can be programmatically formatted just as easily in Google Spreadsheets as in Excel.

VBA

```
Sub AddColorsToRange()
    Dim sh1 As Worksheet
    Dim addr As String: addr = "A4:B10"
    Set sh1 = ActiveWorkbook.Worksheets(1)
    sh1.Range(addr).Interior.ColorIndex = 3
    sh1.Range(addr).Font.ColorIndex = 10
End Sub
```

Google Apps Script

```
// Select a block of cells in the first sheet.
// Use Range methods to set both the font and
//   background colors.
function addColorsToRange() {
  var ss =
      SpreadsheetApp.getActiveSpreadsheet(),
      sheets = ss.getSheets(),
      sh1 = sheets[0],
      addr = 'A4:B10',
      rng;
  // getRange is overloaded. This method can
  //  also accept row and column integers
  rng = sh1.getRange(addr);
```

```
  rng.setFontColor('green');
  rng.setBackgroundColor('red');
}
```

Range Offsets

The *offset Range* property in VBA is implemented in Google Apps Script as the *Range offset()* method. In its basic form, the Google Apps Script version can be used to exactly mimic its VBA namesake as the following code demonstrates.

VBA

```
Public Sub OffsetDemo()
    Dim sh As Worksheet
    Dim cell As Range
    Set sh = _
      ActiveWorkbook.Worksheets(1)
    Set cell = sh.Range("B2")
    cell.value = "Middle"
    cell.Offset(-1, -1).value = "Top Left"
    cell.Offset(0, -1).value = "Left"
    cell.Offset(1, -1).value = "Bottom Left"
    cell.Offset(-1, 0).value = "Top"
    cell.Offset(1, 0).value = "Bottom"
    cell.Offset(-1, 1).value = "Top Right"
    cell.Offset(0, 1).value = "Right"
    cell.Offset(1, 1).value = "Bottom Right"
End Sub
```

Google Apps Script

```
// The Spreadsheet method getSheets() returns
//  an array.
// The code "ss.getSheets()[0]"
//  returns the first sheet and is equivalent to
// "ActiveWorkbook.Worksheets(1)" in VBA.
// Note that the VBA version is 1-based!
function offsetDemo() {
  var ss =
   SpreadsheetApp.getActiveSpreadsheet(),
      sh = ss.getSheets()[0],
      cell = sh.getRange('B2');
  cell.setValue('Middle');
  cell.offset(-1,-1).setValue('Top Left');
  cell.offset(0, -1).setValue('Left');
  cell.offset(1, -1).setValue('Bottom Left');
  cell.offset(-1, 0).setValue('Top');
  cell.offset(1, 0).setValue('Bottom');
  cell.offset(-1, 1).setValue('Top Right');
  cell.offset(0, 1).setValue('Right');
  cell.offset(1, 1).setValue('Bottom Right');
}
```

Pasting and executing these code snippets in either spreadsheet application writes the location of cell B2's neighbours relative to its location. The Google Apps Script *offset()* method is, however, **overloaded**. This concept was discussed in chapter 5 in relation to the *Sheet getRange()* method but it merits re-visiting here to show how the functionality of its overloaded versions can be implemented in VBA.

VBA

```
' Mimicking Google Apps Script
'  offset() method overloads.
Public Sub OffsetOverloadDemo()
    Dim sh As Worksheet
    Dim cell As Range
    Dim offsetRng2 As Range
    Dim offsetRng3 As Range
    Set sh = ActiveWorkbook.Worksheets(1)
    Set cell = sh.Range("A1")
    'Offset returns a Range so Offset
    ' can be called again
    ' on the returned Range from
    '  first Offset call.
    Set offsetRng2 = Range(cell.Offset(1, 4), _
                     cell.Offset(1, 4).Offset(1, 0))
    Set offsetRng3 = Range(cell.Offset(10, 4), _
                     cell.Offset(10, 4).Offset(3, 4))
    Debug.Print offsetRng2.Address
    Debug.Print offsetRng3.Address
End Sub
```

Google Apps Script

```
// Demonstrating overloaded versions of offset()
// Output:
// Address of offset() overload 2
//  (rowOffset, columnOffset, numRows) is: E2:E3
//  Address of offset() overload 3 (rowOffset,
//    columnOffset, numRows, numColumns)
//     is: E11:I14
function offsetOverloadDemo() {
  var ss =
      SpreadsheetApp.getActiveSpreadsheet(),
      sh = ss.getSheets()[0],
      cell = sh.getRange('A1'),
```

```
13        offsetRng2 = cell.offset(1, 4, 2),
14        offsetRng3 = cell.offset(10, 4, 4, 5);
15    Logger.log('Address of offset() overload 2 ' +
16           '(rowOffset, columnOffset, numRows) is: '
17               + offsetRng2.getA1Notation());
18    Logger.log('Address of offset() overload 3 ' +
19           '(rowOffset, columnOffset, numRows, ' +
20           'numColumns) is: '
21           + offsetRng3.getA1Notation());
22  }
```

While the VBA version defines the same ranges as the Google Apps Script version, it is not exactly clear. The key point to realise is that the VBA *Range Offset* property returns another *Range* so there is no reason why *Offset* cannot be invoked again on this returned *Range*. However, code like this VBA example should be avoided where possible and, if it cannot be avoided, it had better be well commented and documented! It was given here purely for demonstration purposes.

Named Ranges

The advantages of using named ranges were outlined in chapter 5. Google Apps Script provides *Spreadsheet* methods for setting named ranges and for retrieving the *Range* objects that the names refer to, see chapter 5 for a full discussion. However, there does not appear to be a way to implement the following VBA functionality.

VBA

```
Public Sub PrintRangeNames()
  Dim namedRng As Name
  For Each namedRng In ActiveWorkbook.Names
    Debug.Print "The name of the range is: " & _
         namedRng.Name & _
        " It refers to this address: " & _
            namedRng.RefersTo
    Next namedRng
End Sub
```

This VBA code prints details for all named ranges in the active Excel file. This functionality can be very useful but at the time of writing, I was unable to duplicate it in Google Apps Script.

Cell Comments

Cell comments are a good way to document spreadsheets and add useful metadata that can describe the meaning of cell contents. They are also amenable to programmatic manipulation.

The Google Apps Script equivalent to Excel comments are **notes**. These are *Range* attributes that can be set and retrieved with with the *Range* getters and setters *setNote()* and *getNote()*, respectively.

Cell Comments

Comments in Google Spreadsheets set from the spreadsheet are **not** the same as notes set programmatically. There does not appear to be a way to programmatically manipulate comments set from the spreadsheet by users.

Setting Cell Comments

VBA

```
Public Sub SetCellComment(sheetName As String, _
                          cellAddress As String, _
                          cellComment As String)
    Dim sh As Worksheet
    Dim cell As Range
    Set sh = ActiveWorkbook.Worksheets(sheetName)
    Set cell = sh.Range(cellAddress)
    cell.AddComment cellComment
End Sub
Public Sub test_SetCellComment()
    Dim sheetName As String
    sheetName = "Sheet1"
    Dim cellAddress As String
    cellAddress = "C10"
    Dim cellComment As String
    cellComment = "Comment added: " & Now()
    Call SetCellComment(sheetName, _
                          cellAddress, _
                          cellComment)
End Sub
```

Google Apps Script

```
function setCellComment(sheetName, cellAddress,
                          cellComment) {
  var ss =
      SpreadsheetApp.getActiveSpreadsheet(),
      sh = ss.getSheetByName(sheetName),
      cell = sh.getRange(cellAddress);
  cell.setNote(cellComment);
}
function test_setCellComment() {
  var sheetName = 'Sheet1',
      cellAddress = 'C10',
      cellComment = 'Comment added ' + Date();
```

```
  setCellComment(sheetName, cellAddress, cellComment);
}
```

Removing Cell Comments

VBA

```
' Need to check if the cell has a comment.
' If it does not, then exit the sub but if
' it does, then remove it.
Public Sub RemoveCellComment(sheetName _
                             As String, _
                     cellAddress As String)
    Dim sh As Worksheet
    Dim cell As Range
    Set sh = ActiveWorkbook.Worksheets(sheetName)
    Set cell = sh.Range(cellAddress)
    If cell.Comment Is Nothing Then
        Exit Sub
    Else
        cell.Comment.Delete
    End If
End Sub
Public Sub test_RemoveCellComment()
    Dim sheetName As String
    sheetName = "Sheet1"
    Dim cellAddress As String
    cellAddress = "C10"
    Call RemoveCellComment(sheetName, _
                           cellAddress)
End Sub
```

Google Apps Script

```
// To remove a comment, just pass an empty string
//  to the setNote() method.
function removeCellComment(sheetName, cellAddress) {
  var ss =
      SpreadsheetApp.getActiveSpreadsheet(),
      sh = ss.getSheetByName(sheetName),
      cell = sh.getRange(cellAddress);
  cell.setNote('');
}
function test_removeCellComment() {
  var sheetName = 'Sheet1',
      cellAddress = 'C10';
  removeCellComment(sheetName, cellAddress);
}
```

Selectively Copy Rows From One Sheet To A New Sheet

Copying rows from one sheet to another based on some pre-determined criterion is a common spreadsheet task. Given the input in the figure below, the code examples given do the following:

- Insert a new sheet named "Target" into which rows will be copied
- Copy the header row to the new sheet
- Check each of the data rows and if the second column value is less than or equal to 10000 then copy the row to the new sheet.

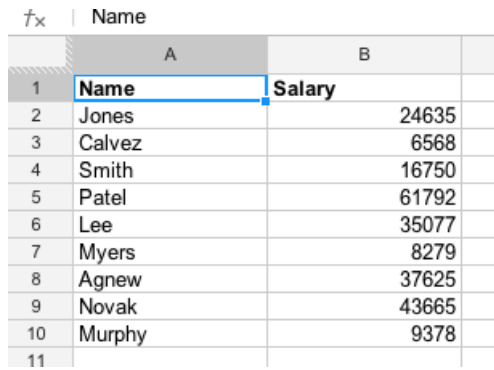


Figure Appendix A.1: Data input sheet

VBA

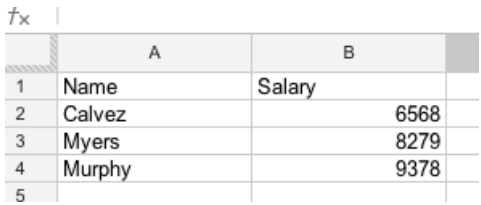
```
' This VBA code is commented because the
'  VBA approach differs
'   considerably from the Google Apps Script one.
' Note: the Offset() method of the Range
'  object uses 0-based indexes.
Public Sub copyRowsToNewSheet()
    Dim sourceSheet As Worksheet
    Dim newSheet As Worksheet
    Dim newSheetName As String
    newSheetName = "Target"
    Dim sourceRng As Range
    Dim sourceRows As Variant
    Dim i As Long
    Set sourceSheet = _
        Application.Worksheets("Source")
    Set newSheet = ActiveWorkbook.Worksheets.Add
    newSheet.Name = newSheetName
    ' Use a named range as marker
    '  for row copying (VBA hack!)
    newSheet.Range("A1").Name = "nextRow"
```

```
    Set sourceRng = sourceSheet.UsedRange
    ' Copy the header row
    sourceRng.Rows(1).Copy Range("nextRow")
    ' Moved the named range marker down one row
    Range("nextRow").Offset(1, 0).Name = _
       "nextRow"
    'Skip header row by setting i,
    ' the row counter, = 2
    ' i starts at 2 to skip header row
    For i = 2 To sourceRng.Rows.Count
        If sourceRng.Cells(i, 2).value _
            <= 10000 Then
            ' Define the row range to copy
            ' using the first and
            '   last cell in the row.
            Range(sourceRng.Cells(i, 1), _
                  sourceRng.Cells(i, _
            sourceRng.Columns.Count)).Copy _
              Range("nextRow")
            Range("nextRow").Offset(1, 0).Name _
             = "nextRow"
        End If
    Next i
End Sub
```

Google Apps Script

```
// Longer example
// Copy rows from one sheet named "Source" to
//  a newly inserted
//   one based on a criterion check of second
//   column.
// Copy the header row to the new sheet.
// If Salary <= 10,000 then copy the entire row
function copyRowsToNewSheet() {
  var ss =
      SpreadsheetApp.getActiveSpreadsheet(),
      sourceSheet = ss.getSheetByName('Source'),
      newSheetName = 'Target',
      newSheet = ss.insertSheet(newSheetName),
      sourceRng = sourceSheet.getDataRange(),
      sourceRows = sourceRng.getValues(),
      i;
  newSheet.appendRow(sourceRows[0]);
  for (i = 1; i < sourceRows.length; i += 1) {
    if (sourceRows[i][1] <= 10000) {
      newSheet.appendRow(sourceRows[i]);
    }
  }
}
```

The output from these code examples is shown below.



	A	B
1	Name	Salary
2	Calvez	6568
3	Myers	8279
4	Murphy	9378
5		

Figure Appendix A.1: Data output sheet

The Google Apps Script *Sheet appendRow()* is very convenient and significantly simplifies the code when compared to the VBA

example. Taking an array of values, it just adds a row to the sheet that contains these values. In VBA the range name "next" is used as a marker for the destination to where the selected row is copied. After each copying operation, it has to be moved down by one row to be ready for the next row to copy.

Print Addresses And Formulas For Range

The code examples below demonstrate how to loop over a range of cells one cell at a time.

VBA

```
Public Sub test_PrintSheetFormulas()
    Dim sheetName As String
    sheetName = "Formulas"
    Call PrintSheetFormulas(sheetName)
End Sub
Public Sub PrintSheetFormulas(sheetName _
                              As String)
    Dim sourceSheet As Worksheet
    Dim usedRng As Range
    Dim i As Long
    Dim j As Long
    Dim cellAddr As String
    Dim cellFormula As String
    Set sourceSheet = _
       ActiveWorkbook.Worksheets(sheetName)
    Set usedRng = sourceSheet.UsedRange
    For i = 1 To usedRng.Rows.Count
        For j = 1 To usedRng.Columns.Count
            cellAddr = _
              usedRng.Cells(i, j).Address
            cellFormula = _
```

```
                usedRng.Cells(i, j).Formula
            If Left(cellFormula, 1) = "=" Then
                Debug.Print cellAddr & _
                  ": " & cellFormula
            End If
        Next j
    Next i
End Sub
```

Google Apps Script

```
function test_printSheetFormulas() {
  var sheetName = 'Formulas';
  printSheetFormulas(sheetName);
}
function printSheetFormulas(sheetName) {
  var ss =
      SpreadsheetApp.getActiveSpreadsheet(),
      sourceSheet = ss.getSheetByName(sheetName),
      usedRng = sourceSheet.getDataRange(),
      i,
      j,
      cell,
      cellAddr,
      cellFormula;
  for (i = 1; i <= usedRng.getLastRow();
            i += 1) {
    for (j = 1; j <= usedRng.getLastColumn();
                j += 1) {
      cell = usedRng.getCell(i, j);
      cellAddr = cell.getA1Notation();
      cellFormula = cell.getFormula();
      if (cellFormula) {
        Logger.log(cellAddr +
          ': ' + cellFormula);
```

```
25            }
26        }
27    }
28 }
```

The Google Apps Script *Range getCell()* method is analogous to the VBA *Range Cells* property. Both expect two integer arguments for the row and column indexes and both are one-based.