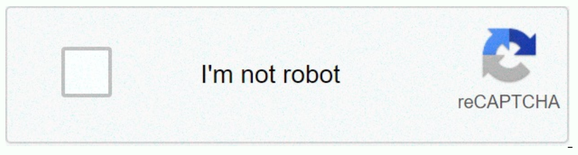

Continue

Google sheets api run script

[{ type: thumbs-down, id: missingTheInformationINeed, label:Missing the information I need },{ type: thumbs-down, id: teComplicatedTooManySteps, label:Too complicated/too many steps },{ type: thumb-down, id: outOfDate, label:Out of date },{ typeof: thumb-down, id: samplesCodeIssue, label::Samples/Code issue },{ type: thumb-down, id: samplesCodeIssue, label:Samples/Code issue },{ type: thumbs-down, id: samplesCodeIssue, label::Samples/Code issue },{ type: thumb-down, id: otherDown, label: [{ type: thumbs-up, id: easyToUnderstand, label:Easy-to-understand },{ type: thumbs-up, id: resolvedMyProblem, label:Resolved my problem },{ type: thumbs-up, id: anderUp, label:Other }] Complete the steps described in the rest of this page to create a simple Google Apps script that requests to the Google Sheets Prerequisites To perform this quick start, you need the following prerequisites: A Google Account Access to Google Drive Step 1: Create the script Create a new script by exporting to Google script.google.com/create. Replace the contents of the script editor with the following code: Click Save . Click Untitled project at the top left, type Quickstart, and click Rename. Click File > Save, name your project Quickstart, and click OK. Enable the Google Sheets API advanced service in your script. Step 3: Run the sample in the Apps Script editor, click Run. The first time you run the sample, it will ask you to authorize access: Click Review permissions. Select an account. Click Allow. The script's execution log appears at the bottom of the window. In the Apps Script editor, click Start > logNamesAndMajors. The first time you run the sample, it will ask you to authorize access: Click the Continue button. Click the Accept button. To view the script's output, click View > Logs. Large! Check out the further reading section below to learn more. Warning There was a problem Bummer, let us know what went wrong. Check out our troubleshooting section below for some common mistakes and solutions. If you found an error in the code, report the issue on GitHub or submit a pull request. This section describes some common issues you may encounter while trying to perform this quick start and sets possible solutions. ReferenceError: [API NAME] is not defined This error occurs when the API is not flucgrated into the Apps script editor. Revisit Step 2.b ensure that the corresponding interchange is set. This application has not been verified. The OAuth permission screen presented to the user can show the alert This app is not verified if it requests scopes that provide access to sensitive user data. These applications must eventually go through the verification process to remove that alert and other restrictions. During the development phase, you can continue past this alert by clicking > Go to {Project Name} (unsafe). [{ type: thumbs-down, id: missingTheInformationINeed, the information I need },{ type: thumbs-down, id: teComplicatedTooManySteps, label:Too label:Too label:Te / too many steps },{ type: thumbs-down, id: outOfDate, label:Out or date },{ type: thumb-down, id: samplesCodeIssue, label:Samples/Code issue },{ type: thumb-down, id: otherDown, label:Other }] [{ type: thumbs-up, id: easyTostand, easyTostand, label:Other }] [{ type: thumbs-up, id: easyTostandder, easyTostandder, label:Other }] [{ type: thumbs-up, id: easyTostandder, label:Other }] [{ type: thumbs-up, id: easyTostandder, label:Other }] [{ type: thumbs-up, id: otherDown, label:Other }] [{ type: thumbs-up, id: otherDown, label:Other }] [{ typeof-up, id: easyTostandder, label:Other }] [{ type: thumbs-up, id: otherDown, label:Easy to understand },{ type: thumb Up, id: resolvedMyProblem, label:Resolved my problem },{ type: thumbs-up, id: anderUp , label:Other }] Except as otherwise noted, the contents of this page are licensed under the Creative Commons Attribution 4.0 License, and code samples are licensed under the Apache 2.0 License. For details, see the Google Developer Site Policies. Java is a registered trademark of Oracle and/or its affiliates. Last updated 2020-12-07 UTC. Welcome to the third part of Fundamental Programs Script with Sheets playlist! By completing this codelab, you can learn how to use data manipulations, custom menus, and public API data retrieval in Apps script to improve your Sheets' experience. And you can continue to work with the SpreadsheetApp, Spreadsheet, Sheet, and Series classes that introduced the previous codelabs in this playlist. What you will learn How to import data from a personal or shared spreadsheet into Drive. How to create a custom menu with the onOpen() function. How to expropriate and manipulate string data values in Google Sheet cells. How to pull and manipulate JSON object data from a public API source. Before you start This is the third codelab in the Fundamentals of Apps Script playlist. Before you start this codelab, you must complete the previous code sheets: Macros and custom functions spreadsheets, pages, and ranges you need to have an understanding of the basic Programs Script topics explored in the previous code sheets of this playlist. Basic familiarity with the Apps Script editor Basic familiarity with Google Sheets Ability to read Sheets A1 Notation Basic familiarity with JavaScript and its String class The exercises in this codelab require spreadsheet to work in. Follow these steps to create a new spreadsheet to use in these exercises: Create a new spreadsheet in your Google Drive. You can do this from the Drive interface by selecting New Pages > By default, the new spreadsheet is placed in your root drive. Click the spreadsheet title and change it from Untitled Spreadsheet to Data Manipulation and Custom Menus. Your page should look like this: Select Tools > Script Editor to open the script editor. Click on the project title and change it from Project to Data Manipulation and Custom Menus. Click Rename to save the title change. With a blank spreadsheet and project you are ready to start the lab! Move to the next section to start learning about custom menus. Note: If you are using gmail.com account, you may not be able to use a This program verified dialog when you first use your script. Google uses it to alert users who might be using code from unknown or untrusted authors. If you see this dialogue, it's it to continue since you are the script writer. In these cases, continue to authorize the script by doing the following: In this information is not verified dialog, click Advanced. Click Go to Data Manipulation and Custom Menus (unsafe). In the next screen, click Allow. Throughout this codelab, you can see multiple permission prompts. Follow the on-screen directions to continue to authorize the code. You can read more about this process in the Apps Script authorization directory. Apps script gives you the ability to define custom menus that can appear in Google Sheets. You can also use custom menus in Google Docs, Google Slides, and Google Forms. When you define a custom menu item, you create a text label and connect it to an Apps script feature in your script project. When you add the menu to the UI, it appears in Google Sheets: When a user clicks a custom menu item, run the Apps script feature you associated with it. This is a handy way to cause Apps to run Script functions without opening the script editor. It also allows other users of the spreadsheet to run your code without knowing anything about how it or Apps Script works-on them, it's just another menu item. Custom menu items are defined in the onOpen() simple trigger function, which you will learn about in the next section. Triggers in Apps Script provides a way to run specific Apps Script code in response to specific conditions or events. When you create a trigger, you define what event causes the trigger to fire and provide an Apps Script function that runs when that event happens. onOpen() is an example of a simple trigger. Simple triggers are easy to set up — all you have to do is write an Apps Script function named opOpen()and Apps Script will run it every time the associated spreadsheet is opened or rebooted: /** * A special function that runs when the spreadsheet is first* opened or restarted. onOpen() is used to add custom menu * items to the spreadsheet. */ function onOpen() { /* ... */ } Key concept: Simple triggers are easy to set up and use, but have a number of restrictions. Because of these restrictions, onOpen() is used only to create menu items. Triggers are a complex topic and as such are beyond the scope of this codelab. For now, you can only think of onOpen() as an event handler that adds menu items when loading a spreadsheet. Implementation Let's Create a Custom Menu! Replace the code in your script project with the following: /** A special function that runs when the spreadsheet is first * opened or restarted. onOpen() is used to add custom menu * items to the spreadsheet. */ function onOpen() { var ui = ui.createMenu('Booklist') .addItem('Load booklist', 'loadBookList') .addToUi(); } Save your script project. Code Review Let's review this code to understand how it works. In onOpen(), the first line uses the witness() method to acquire a Onion object that represents the user interface of the active spreadsheet this script represents is bound bound The following three lines create a new menu (Book list), add a menu item (Load Book List) to that menu, and then add the menu to the spreadsheet's interface. This is done with the createMenu(caption), addItem(caption, functionName), and addToUi() methods, respectively. The addItem (caption, functionName) creates a connection between the menu item label and an Apps scripting feature that runs when that menu item is selected. In this case, selecting the Load Book List menu item causes Sheets to try to run the loadBookList() function (which does not yet exist). Results Let's Run this feature now to see that it works! Do the following: In Google Sheets, restart your spreadsheet. Note: this usually closes the tab with your script editor. Reopen your script editor by selecting Tools editor > After your spreadsheet is rebooted, you'll see a new menu called Book List on your menu bar: By clicking Booklist, you can see the resulting drop-down list: And that's how you can create your own custom menu in Sheets! The following section defines the loadBookList() function and introduces one way you can communicate with data in Apps Script: read other spreadsheets. Now that you've created a custom menu, you can create functions that can be performed to just click the menu item itself. At the moment, the custom menu book list has one menu item: Load Book List. But the feature mentioned when you use the Load Book List menu item, loadBookList(), doesn't exist in your script, so select Booklist > Load Booklist throws an error: You can fix this error by implementing the loadBook list() function. Implementation You want the new menu item to fill the spreadsheet with data to collaborate, so you'll implement LoadBookList() to read data from another spreadsheet and copy it into this one: Add the following code to your script under onOpen(): /** * Create a template book list based on the *provide 'codelab-book-list' sheet. */ function loadBookList(){ // Get the active sheet. var skin = SpreadsheetApp.getActiveSheet(); // Get another spreadsheet of Drive via the // spreadsheet's ID. var bookSS = SpreadsheetApp.openById(1c0GvbVUDeBmhTpq_A3vJh2xsebtLuwGwpBYqcOBqGvo); // Get the tab, data series, and values of the // spreadsheet stored in bookSS. var bookSheet = bookSS.getSheetByName(codelab-book-list); var bookRange = bookSheet.getDataRange(); var bookListValues = bookRange.getValues(); // Add those values to the active sheet in the current // spreadsheet. It overwrites any values that already // there exist. sheet.getRange(1, 1, bookRange.getHeight(), bookRange.getWidth()) .setValues(bookListValues); Rename the landing page and change the data // columns for easier reading. sheet.setName(Booklist); sheet.autoResizeColumns(1, 3); } Save your script project. Code Review So How Does This Feature Work? The loadBookList() function uses methods from the Spreadsheet, Sheet, and Series classes that codesheets the previous codesheets With those concepts in mind, you can break down the loadBookList() code in the following four sections: 1: Identify the landing page The first line uses SpreadsheetApp.getActiveSheet() to get and save a reference to the current page object in the variable sheet. This is where the data will be copied. 2: Identify the source data The following few lines establish four variables that refer to the source data you retrieve: bookSS stores a reference to the spreadsheet from which the code reads data. The code finds the spreadsheet through its spreadsheet ID. In this example, we provided the ID of a source spreadsheet to read from, and open the spreadsheet using the SpreadsheetApp.openById(id) method. booksHeet stores a reference to a page within books THAT contains the data you want. The code identifies the page to read last name by its name, codelab-book-list. bookRange stores a reference to a variety of data in booksHeet. The method Sheet.getDataRange() returns the range that contains all the non-empty cells in the sheet. This is a handy way to make sure you get a series that covers all the data in a sheet without including many empty rows and columns. bookListValues is a 2D array that contains all the values taken from the cells in bookRange. The Range.getValues() method generates this array by reading data from the source page. 3: Copy the data from source to destination The following code section copies the bookListValues data into page, and then renames the page as well: 4: Format the landing page The Sheet.setName(name) is used to change the landing page name to Booklist. The last line in the feature uses Sheet.autoResizeColumns(startColumn, numColumns) to change the first three columns in the landing page, so you can read the new data more easily. Results You can see this feature in action! In Google Sheets, select Book List > Download Book List to run the feature to fill your spreadsheet: You now have a page with a list of book titles, authors, and 13-digit ISBN numbers! In the next section, you learn how to change and update the data in this book list using string manipulations and custom menus. You can now see book information on your page. Each row refers to a specific book, the list of its title, author, and ISBN number in separate columns. However, you can also see issues with this raw data: For some rows, the title and author are placed together in the title column, linked by a comma or the string by . Some sequions are missing information for their book's titles and authors. In the following sections, you will correct these issues by clearing the data. For the first edition, you will create functions that read the title column and split the text when a comma or by delimiter is found, place the corresponding author and title substrings in the correct columns. For the second problem, you'll write code that automatically looks up missing book information through an external and fill that information into your page. Fill. want to create three new menu items to control the data cleaning operations that you will implement. Implementation Let's update OpOpen() to include the extra menu items you need. Do the following: In your script project, update your onOpen() code to match the following: /** A special function that runs when the spreadsheet is first * opened or restarted. onOpen() is usually used to add custom * menu items to the spreadsheet. */ function onOpen() { var ui = SpreadsheetApp.getUi(); ui.createMenu('Booklist') .addItem('Load booklist', 'loadBookList') .addSeparator() .addItem('Separate title/author at first comma') .splitAtFirstComma') .addItem('Separate title/author finally by, 'splitAtLastBy') .addSeparator() .addItem('Fill in empty titles and writer cells', 'fillinTheBlanks') .addToUi(); } Save your script project. In the script editor, choose onOpen in the dropdown feature and click Run. It will run opOpen() to rebuild the spreadsheet menu, so you don't have to restart the spreadsheet. In this new code, the Menu.addSeparator() method creates a horizontal diveler in the drop-down list so that you can visually keep groups of related menu items organized. The new menu items are then added below, with the labels Separate title/Author at the first comma, Separate title/author finally at, and Fill in blank titles and writer cells. note: The menu items you add will appear in Sheets in the same order you add them in the onOpen() code. Results In your spreadsheet, you can click the Book List menu to see the new menu items: If you click on these items now, this will cause an error since you haven't implemented their corresponding features yet, so let's do it next. The dataset you entered in your spreadsheet has some cells where the author and title are incorrectly combined into one cell using a comma: Split text strings into separate columns is a common spreadsheet task. Google Sheets provides a SPLIT() feature that divides strings into columns. However, often datasets have issues that can't be easily resolved with Sheets features. In these cases, you can write Apps Script code to perform the complex operations necessary to clean and organize your data. You begin to clear your data by first implementing a function called splitAtFirstComma() that divides the author and title into their respective cells when found commas. The splitAtFirstComma() function must take the following steps: Get the range that represents the currently selected cells. See if cells in that range have a comma. Where commas are found, divide the string into two (and only two) substrings at the location of the first comma. To make things simpler, you might assume that any comma indicates a [writers], [title] string pattern. You can also assume that if several commas appear in the cell, it is appropriate to click on the first comma in the divide. Sets the substrings as the new content of the respective title and author cells. Implementation To implement these steps, you will the same Spreadsheet service methods you used previously, but you also need to use some simple JavaScript to manipulate the string data. Take the following steps: In the Apps Script editor, add the following function at the end of your script project: /** * Reshape title and author columns by dividing the title column* at the first comma, if present. */ function splitAtFirstComma(){ // Get the active (currently highlighted) range. var activeRange = SpreadsheetApp.getActiveRange(); var titleAuthorRange = activeRange.offset(0, 0, activeRange.getHeight(), activeRange.getWidth() + 1); // Get the current values of the selected title column cells. // This is a 2D array. var titleAuthorValues = titleAuthorRange.getValues(); // Update values where commas are found. Accepting the presence // of a comma indicates a writers, title pattern. for (var row = 0; row < titleAuthorValues.length; row ++){ var indexOfFirstComma = titleAuthorValues[row][0].indexOf(,); if (indexOfFirstComma >= 0){ // Has a comma found, so divide and update the values in // the values array. var titlesAndAuthors = titleAuthorValues[row][0] = titlesAndAuthors.slice(indexOfFirstComma + 2); // Update the author value in the array. titleAuthorValues[row][1] = titlesAndAuthors.slice(0, indexOfFirstComma); } Put the updated values back in the spreadsheet. titleAuthorRange.setValues(titleAuthorValues); } Save your script project. Code Review Let's review the new code. The code consists of three main sections: 1: Retrieves the highlighted title values The first three lines establish three variables that refer to the current data in the page: activeRange represents the range that the user currently highlighted when the splitAtFirstComma() function was called. To keep this exercise simple, we can assume that the user does this only when cells are highlighted in column A. titleAuthorRange represents a new range that covers the same cells as activeRange, but also includes one more column on the right. titleAuthorRange was created using the Range.offset(rowOffset, columnOffset, numRows, numColumns) method. The code needs this extended range because it needs a place to place any discovered authors it finds in the title column. titleAuthorValues is a 2D array of data extracted from titleAuthorRange using Range.getValues(). 2: Examine each title and split on any comma delimiter found The following section examines the values in titleAuthorValues to search for commas. A JavaScript for loop is used to examine all the values in the first column titleAuthorValues. When a comma substring is found (,) with the JavaScript String indexOf() method, the code does the following: The cell string value is copied to the titlesAndAuthors variable. The comma location is determined with of the JavaScript String indexOf() method. The JavaScript String cut() method is used twice to obtain the substring before the comma delimiter and the to the demarcation. The substrings are copied back into the titleAuthorValues 2D array, overwriting the existing values in that position. Since we assume that a [writers], [title] pattern, the order of the two substrings is reversed to place the title in the first column and the writers in the second. Note: When the code doesn't find a comma, it leaves the data in that row unchanged. 3: Copy the new values back into the page Once all the title cell values are examined, the updated titleAuthorValues 2D array is copied back into the spreadsheet using the Range.setValues (values) method. Key point: The code could have updated each spreadsheet individually as it found each comma, which Range.setValue(value) calls each time a cell had to be changed. However, this will result in much slower code, as each Range.setValue (value) call requires server communication. When pages update data, it's almost always better to get the values into a 2D array first, update the array, and then push the data back to the tab with a single Range.setValues (values) call. This way, you reduce the amount of server communication your code needs to do, and speed up your code. Results You can now see the effects of the splitAtFirstComma() function in action! Try running it by selecting the separate title/writer at the first comma menu item after choosing... one cell: ... or multiple cells: Note: You can select any column of text in your spreadsheet and use this function to search for commas and split the text according to the rules you set. Just be aware that, if the code finds a comma delimiter, it will overwrite the cell to the immediate right of the original. You've now built an Apps Script feature that processes Google Sheets data! Next you will implement the second splitter feature. If you take another look at the original data, you can see a different problem. Just like some of the data formats titles and authors in a single cell as [writers], [title], other cells format writer and title as [title] by [writers]: Implementing You can solve this problem using the same technique used in the last section, creating a new feature called splitAtLastBy(). This feature has a very similar job to splitAtFirstComma()—the only real difference is that it's looking for a slightly different pattern of text. Implement this function by doing the following: In the Apps Script editor, add the following function at the end of your script project: /** * Reshape title and author columns by partitioning title column * on the last instance of the string by . */ function splitAtLastBy(){ // Get the active (currently highlighted) range. var activeRange = SpreadsheetApp.getActiveRange(); var titleAuthorRange = activeRange.offset(0, 0, activeRange.getHeight(), + 1); Gets the current values of the selected title column cells. This is a 2D array. var titleAuthorValues = titleAuthorRange.getValues(); // Update values where found by substrings. Found. the presence of a door indicates a title by writers // pattern. for (var row = 0; row < titleAuthorValues.length; row ++){ var indexOfLastBy = titleAuthorValues[row][0].lastIndexOf(by); if (indexOfLastBy >= 0){ // Has a door , so divide and update the values in // our array. var titlesAndAuthors = titleAuthorValues[row][0]; Updates the title value. titleAuthorValues[row][0] = titlesAndAuthors.slice(0, indexOfLastBy); Update the author value. titleAuthorValues[row][1] = titlesAndAuthors.slice(indexOfLastBy + 4); } } // Put the values back in the spreadsheet. titleAuthorRange.setValues(titleAuthorValues); } Save your script project. Code review There are some key differences between this code and splitAtFirstComma(): The substring door is used as a string delimiter, instead of , . Here, the JavaScript String.lastIndexOf(substring) is used instead of String.indexOf(substring). This means that if there are multiples by substrings in the initial string, all but the last by assumed to be part of the title. After splitting the string, the first substring is set as the title and the second as the author string (this is the opposite order of the one used in splitAtFirstComma()). Results You can now see the effects of the splitAtLastBy() function in action! Try running it by selecting the Separate title/writer finally by menu item after selecting... one cell: ... or multiple cells: You have completed this section of the codelab. You can now use Apps script to read and mod change the string data in a page and use custom menus to run different Apps script commands! In the next section, you learn how to further enhance this dataset by filling in the empty cells with data drawn from a public API. So far, you've refined your dataset to resolve title and author formatting issues, but the dataset is still missing information, highlighted in the cells below: The missing data cannot be accessed by string operations on the data you have here. Instead, you should get the missing data from another source. In this case, you can do this in Apps script by requesting information from an external API that can provide your script additional book information. APIs are application programming interfaces. This is a pretty common term, but basically means that someone has provided a service that can call your apps and scripts to do specific things like request information or take certain actions. In this section, you call a publicly available API to request additional book information, which you then place in the vacant cells in your page. This section teaches you how to: Request book data from an external API source. Extract title and author information from the returned data and write that information to your spreadsheet. Before you delve into code that's directly with your work, you can learn about the details of working with external APAs in Apps Script through a helper feature specifically for for book information from the public Open Library API. Note: Open Library is an Internet archive that strives to create a web page for each book publication. If you're interested, you can read more about Open Library on their official website and their API documentation. This helper function, fetchBookData_ (ISBN) takes a 13-digit ISBN number from a book as a parameter and returns data about that book by linking and retrieling information from the Open Library API and then analyzing the returned JSON object. Implementation Implement this helper function by doing the following: In the Apps Script editor, add the following code at the end of your script: /* * Helper function that retrieves book data from the Open Library * public API. * * @param N {number} ISBN - The ISBN number of the book to look up. * @return {object} The book's data, in JSON format. */ function fetchBookData_ (ISBN){ // Connect to the public API. var url = + ISBN + &jscmd=details&format=json; var response = UrlFetchApp.fetch(url, {'muteHttpExceptions': true}); Make request to API and get response before this point. var json = response.getContentText(); var bookData = JSON.parse(json); Only return the data we are interested in. } Save your script project. Note: In Apps Script, functions that make names end in _ are considered private. These functions cannot be called by other scripts if present in a library, or by clients during server client communication. However, these are advanced topics, so for now just know it's a best practice to end a function name in _ if it's a helper feature that only the current script will ever have to use. Code Overview This code is divided into two main sections: 1: Make an API request In the first two lines fetchBookData_ (ISBN) connect to the public Open Library API, using the API's URL endpoint and Apps Script's UrlFetch service. The url variable is just a URL string, such as a web address. This points to a location on the Open Library Servers. It also includes three parameters (bibkeys, jscmd, and format) that tell the Open Library servers what information you are requesting and how to structure the response. In this case, you provide the book's ISBN number to identify it, and ask for detailed information to be returned in JSON format. Note: Each public API defines its own formats for making requests and returning results. To effectively communicate your code with an API, you must conform to the formats that define it. For the Open Library, the request and response formats are described in the Open LIbrary Books API documentation. The response format here is JSON (JavaScript Object Notation), which is just a particular way of organizing data; JSON is similar to HTML or XML, but tends to be easier for humans to read. Once you have the URL string built send the code a request to that location and receive a response. This is done with the params) method. This method sends an information request to the external URL that you provide and stores the resulting response in the response variable. In addition to the URL, the code sets the optional parameter muteHttpExceptions as true. This setting only means that your code will not stop if the requests result in an API error; instead, the error response is returned. The request returns an HTTPResponse object that is stored in the response variable. HTTP responses include a response code, HTTP headers, and the main response content. The information of interest here is the main JSON content, so the code should extract it and then analyze the JSON to track down and return the desired information. In the last three lines of the code, the HTTPResponse.getContentText() method returns the main contents of the response as a string. This string is in JSON format, but the exact content and format is defined by the Open Library API. The JSON.parse(jsonString) method converts the JSON string to a JavaScript object so that different parts of the data can be easily extracted. Finally, the feature returns the share of the data that matches the ISBN number for the book of interest. Note: If you feel like you need to further requalify yourself from JSON, see the JavaScriptdocumentation for working with JSON or the JavaScript JSON class reference. Results Now that you've implemented fetchBookData_ (ISBN), other features in your code can quickly view book information using any book using its ISBN number. You will use this feature to help fill in the spaces in your spreadsheet. You can now implement a new fillInTheBlanks() function that will do the following: Identify the missing title and author data within the active data series. Retrieve a specific book's missing data by linking the Open Library API via fetchBookData_ (ISBN) helper method. Update the missing title and/or writer's values in their respective cells. Implementation Implement this new feature by doing the following: In the Apps Script editor, insert the following code at the end of your script project: /** * Fill in missing title and author data using Open Library API * calls. */ function fillInTheBlanks(){ // Constants that identify the index of the title, author, // and ISBN columns (in the 2D bookValues array below). var TITLE_COLUMN = 0; var AUTHOR_COLUMN = 1; var ISBN_COLUMN = 2; // Get the current book information in the active page. The data // is placed in a 2D array. var dataRange = SpreadsheetApp.getActiveSpreadsheet() .getDataRange(); var bookValues = dataRange.getValues(); Examine each row of the data (excluding the header row). If an ISBN is present and a title or author is missing, // use the fetchBookData_(isbn) method to retrieve the // missing data from the Open Library API. Fill the // missing titles or authors when they are found. for (var row = 1; row < bookValues.length; row ++){ var isbn = bookValues[row][ISBN_COLUMN]; var title = bookValues[row][TITLE_COLUMN]; var writer = = if (isbn != & (title === || writer ===)) { // Dial the API only if you have an ISBN number and // either the title or author is missing. var bookData = fetchBookData_ (isbn); Sometimes the API does not return the information needed. In those cases, don't try to update the row further. if (!bookData || !bookData.details) { continue; } // The API may not have a title, so just fill it in // if the API returns one and the title is empty in the // skin. if (title === &bookData.details.title){ bookValues[row][TITLE_COLUMN] = bookData.details.title; } The API may not have a writer name, so fill it in only // if the API returns one and the author is empty in the // skin. if (author === & bookData.details.authors && bookData.details.authors[0].name){ bookValues[row][AUTHOR_COLUMN] = bookData.authors.authors[0].name } } } Put the updated book data values back into the spreadsheet. dataRange.setValues(bookValues); } Save your script project. Code Overview This code is divided into three sections: The first three lines of the function define some constants that help make the code more human-readable. In the following two lines, the bookValues variable is used to maintain a local copy of the page's book information. The code will read information from bookValues, use the API to fill out missing information, and then write these values back to the spreadsheet. Note: This arrangement differs from what you used for splitAtFirstComma() and splitAtLastBy(). In those functions, the code examined only the currently marked titles and their sequents. Here you examine the entire page using Spreadsheet.getDataRange(). The code then runs across each row in bookValues to search for missing titles or authors. To reduce the number of API calls the code needs to make (and therefore waste time), the code only calls the API if the following is true: The row has a value in the ISBN column (that is, the book's ISBN is known). Either the title or writer cell in that row is empty. If the conditions are all where the code calls the API using the fetchBookData_ (isbn) helper function that you implemented, and stores the result in the bookData variable. This variable should now write the missing information you want back into the skin. The only task that remains now is to add the bookData information to our spreadsheet. However, there is a caveat. Unfortunately, public APIs like the Open Library Book API sometimes don't have the information you request, or can sometimes have another problem that prevents it from providing that information. If you blindly assume that each API request will succeed, your code won't be robust enough to handle unexpected errors. To make sure your code is robust in the face of API errors, the code needs to make sure that the API response is valid before trying to use it. Once the code has bookData, it does a simple confirming that bookData and bookData.details exist before attempting to read from them. Ashes missing, that means the API didn't have the data you wanted. In that case, the ongoing command tells the code to skip that row — you can't fill in the missing cells, but at least your script won't collapse. The following part of the code has similar checks that verify the API-returned title and author information. The code only updates the bookValues array if the original title or author cell was empty and the API returned a value that you can place there. Exit the loop after examining all rows in the page. The last step is to write back the now-updated bookValues array to the spreadsheet using Range.setValues(values). Results Now you can finish cleaning your book data! Do the following: If you haven't yet, mark the A2:A15 series in your page, and then select Booklist > Separate title/writer with the first comma to clear the comma problems. If you haven't yet, highlight the A2:A15 series in your page and then select Booklist > Separate title/writer finally by clearing the door problems. Select Book Lists > Fill out titles and author cells to fill out your code all the remaining cells: Congratulations on completing this codelab! You've learned how to create custom menus to enable different parts of your Apps Script code. You've also seen how to import data into Google Sheets using Apps Script services and public APOs. This is a very common operation in spreadsheet processing, and Apps Script enables you to import data from a wide range of sources. Finally, you've seen how you can use Apps Script services and JavaScript to read, process, and write spreadsheet data! No you learned How to import data from a Google spreadsheet. How to create a custom menu in onOpen (). How to expropriate and manipulate string data values. How to call public APIs using the UrlFetch service. How to expropriate JSON object data retrieved from a public API source. Key concepts: onOpen() is a simple trigger used to create menu items. Simple triggers have a variety of limitations that you need to adhere to. When reading or writing spreadsheet data, it's a best practice to read or write an entire series at once with Range.getValues() or Range.setValues(values). Do not read or set each cell individually, as it will be much slower. Apps Script functions that end names with _ are considered private. These features cannot be called by other scripts if included as part of a library or by clients through client server communication. If you know a function is only going to be used by the script its in, it's usually a good idea to end its name with _. When using data returned by external APOs, make sure that the data returned, is before trying to use it. This will make your code less likely to have errors if the API returns an unexpected result. Key terms: Active (status): Indicates that the specified spreadsheet, skin, range, or cell is the one currently viewed or highlighted by the user of the spreadsheet. Active Cell: The Single Highlighted Highlighted Highlighted within the active page that has the pointer focus Active range: The group of one or more cells currently highlighted within the active page. API: Application programming interface; this is a service that can contact apps or scripts to retrieve information or take specific actions. Authorization: The process of the user granting permissions to allow Apps script to access private data in the relevant Google services. Container-bound script: Any script that is bound to and created from a Google Workspace document, such as a Google Sheet or Google Doc. Helper feature: A feature, usually private, that is called by another to complete a small sub-task. Helper functions are usually used when the sub-task needs to be done many times in different locations. JSON: File format for defining data objects and their properties from text. onOpen(): If defined in a script, a simple trigger that burns when a file opens or restarts. Private functions: A function that cannot be called by client server communication as part of a library or by client server communications. Apps Script private functions have names that end with _. Range: A series represents a group of one or more adjacent cells within a sheet. The Range class gives you the ability to read and work ranges within a sheet. Script editor: The code editor for Apps Script. Simple triggers: A subtype of triggers in Apps Script that has trigger function names and certain constraints reserved; for example, onOpen(). Page: A single page of a spreadsheet. The Sheet class allows you to access and modify sheets. SpreadsheetApp: This class serves as the parent class for the Spreadsheet Service and provides a starting point for code that reads or manipulates Google Sheets data. Spreadsheet: A Google Sheets file that resides within Google Drive. The Spreadsheet class allows you to access and modify spreadsheets. Triggers: An event that was keyed to an Apps scripting feature. When the event (such as a spreadsheet being opened) occurs, the trigger fires and automatically performs the associated Apps Script function. UrlFetch: The Apps Script service that lets scripts connect to URL endpoints to make requests and receive responses. For more information about how Apps Script communicates with Google Sheets, see the Spreadsheet Service Reference Documentation. What's next The following codelab in this playlist is more in depth about how to format data within a spreadsheet. Locate the following codelab at Data formatting! Formatting!