OVER 150+ Exercises

Getting started with Google Apps Script SHEETS

Basic Read and Write

Objective: Read data from one range and write it to another range in the same sheet.

Code:

```
function readAndWrite() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const data = sheet.getRange("A1:B2").getValues();
 sheet.getRange("C1:D2").setValues(data);
}
```

Explanation: This script reads data from the range A1:B2 and writes it to C1:D2. It introduces basic read-write operations in Google Sheets using Apps Script.

Append Row

Objective: Append a new row of data to the end of a sheet.

Code:

```
function appendRow() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.appendRow(["New Data", new Date()]);
}
```

Explanation: This function appends a row with static text and the current date, demonstrating how to add data dynamically to the end of a sheet.

Format Cells

Objective: Change the background color of a range of cells.

Code:

```
function formatCells() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.getRange("A1:A5").setBackground("yellow");
}
```

Explanation: This script changes the background color of cells A1 through A5 to yellow, showcasing cell formatting capabilities.

Create a New Sheet

Objective: Create a new sheet in the active spreadsheet with a specified name.

Code:

```
function createNewSheet() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const sheetName = "New Sheet";
 if (!ss.getSheetByName(sheetName)) {
 ss.insertSheet(sheetName);
 }
}
```

Explanation: Checks if a sheet with the given name exists before creating it, to avoid duplication errors.

Delete Specific Rows

Objective: Delete rows that contain a specific value in a certain column.

Code:

```
function deleteRowsWithValue() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const data = sheet.getDataRange().getValues();
 data.reverse().forEach((row, index) => {
 if (row[0] === "Delete Me") { // Assuming the value is in the first column
 sheet.deleteRow(data.length - index);
```

```
 }
 });
}
```

Explanation: Iterates through the sheet in reverse to avoid index shifting when deleting rows, targeting rows where the first column contains "Delete Me".

Sort Data

Objective: Sort the sheet based on values in a specific column.

Code:

```
function sortSheet() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.sort(2); // Sorts by the second column.
}
```

Explanation: This function sorts all the data in the active sheet based on the values in the second column.

Filter and Copy Data

Objective: Copy rows to another sheet based on a filter criterion.

Code:

```
function filterAndCopy() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const sourceSheet = ss.getSheetByName("Source");
 const targetSheet = ss.getSheetByName("Target") ||
ss.insertSheet("Target");
 const data = sourceSheet.getDataRange().getValues();
 const filteredData = data.filter(row => row[1] > 100); // Assuming we're
filtering based on the second column
 targetSheet.getRange(1, 1, filteredData.length,
filteredData[0].length).setValues(filteredData);
```

```
}
```

Explanation: Filters rows from the "Source" sheet where the second column's value is greater than 100 and copies them to the "Target" sheet.

Custom Function

Objective: Create a custom function to calculate the sum of two numbers in Google Sheets.

Code:

```
/**
 * Calculates the sum of two numbers.
 *
 * @param {number} number1 The first number.
 * @param {number} number2 The second number.
 * @return The sum of the two numbers.
 * @customfunction
 */
function SUM_TWO_NUMBERS(number1, number2) {
 return number1 + number2;
}
```

Explanation: Demonstrates how to create a custom function that can be used directly in Google Sheets as a formula.

Batch Update

Objective: Perform a batch update to set values for multiple ranges at once.

Code:

```
function batchUpdate() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const rangeList = sheet.getRangeList(['A1:B2', 'C3:D4', 'E5:F6']);
 rangeList.getRanges().forEach((range, index) => {
```

```
 range.setValues([[`Start ${index}`, `End ${index}`], [`Start ${index}`, `End ${index}`]]);
 });
}
```

Explanation: Sets values for multiple ranges in a single operation, showcasing efficient data manipulation over multiple areas.

Use Google Sheets API

Objective: Use the Google Sheets API to retrieve the titles of all sheets in the spreadsheet.

Code:

```
function getSheetTitles() {
 const sheets = SpreadsheetApp.getActiveSpreadsheet().getSheets();
 const titles = sheets.map(sheet => sheet.getName());
 Logger.log(titles);
}
```

Explanation: Utilizes the Google Sheets API to work with sheet objects, extracting their names. This function is useful for managing sheets based on their titles.

Convert Sheet Data to JSON

Objective: Write a script to convert the data in a sheet to JSON format and log the JSON string.

Explanation: This exercise demonstrates how to work with data in a structured format, which is useful for integration with web applications and APIs.

Code:

```
function sheetDataToJson() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
```

```
 const data = sheet.getDataRange().getValues();
 const headers = data.shift(); // First row as headers
 const jsonData = data.map(row => {
 let obj = {};
 row.forEach((value, index) => {
 obj[headers[index]] = value;
 });
 return obj;
 });
 Logger.log(JSON.stringify(jsonData));
}
```

Insert Checkbox

Objective: Insert checkboxes in a specific range within a sheet.

Explanation: Learn how to programmatically add UI elements like checkboxes, enhancing interactivity within sheets.

Code:

```
function insertCheckboxes() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.getRange("A1:A10").insertCheckboxes();
}
```

Create a Dropdown List

Objective: Create a dropdown list in a range of cells using data validation.

Explanation: This task explores data validation and how to enforce data integrity in user inputs with dropdown lists.

Code:

```
function createDropdownList() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const range = sheet.getRange("B1:B10");
```

```
 const rule =
SpreadsheetApp.newDataValidation().requireValueInList(["Option 1",
"Option 2", "Option 3"], true).build();
 range.setDataValidation(rule);
}
```

Highlight Duplicate Values

Objective: Programmatically highlight duplicate values in a column.

Explanation: Enhances data analysis capabilities by visually identifying duplicates directly within the spreadsheet.

Code:

```
function highlightDuplicates() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const data = sheet.getRange("C1:C" + sheet.getLastRow()).getValues();
 let valueCounts = {};
 data.forEach((row, index) => {
 const cellValue = row[0];
 if (valueCounts[cellValue]) {
 valueCounts[cellValue].push(index + 1);
 } else {
 valueCounts[cellValue] = [index + 1];
 }
 });
 Object.keys(valueCounts).forEach(value => {
 if (valueCounts[value].length > 1) {
 valueCounts[value].forEach(rowNum => {
 sheet.getRange("C" + rowNum).setBackground("red");
 });
 }
 });
}
```

Auto-Resize Columns

Objective: Automatically resize columns in a sheet based on their content.

Explanation: Focuses on improving the presentation of data in Sheets by ensuring content is fully visible.

Code:

```
function autoResizeColumns() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.autoResizeColumns(1, sheet.getLastColumn());
}
```

Merge Cells

Objective: Merge a range of cells in a sheet both horizontally and vertically.

Explanation: Teaches manipulation of cell properties to customize the layout of data in a spreadsheet.

Code:

```
function mergeCells() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.getRange("D1:E2").merge();
}
```

Unmerge Cells

Objective: Unmerge any merged cells within a specified range.

Explanation: Complements the previous exercise by demonstrating how to reverse cell merging, restoring individual cell boundaries.

Code:

```
function unmergeCells() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.getRange("D1:E2").breakApart();
}
```

Create a Chart

Objective: Programmatically create a chart from data in a sheet.

Explanation: Expands visualization capabilities by embedding charts within Sheets, offering insights at a glance.

Code:

```
function createChart() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const chartBuilder = sheet.newChart();
 chartBuilder.addRange(sheet.getRange("A1:B10"))
 .setChartType(Charts.ChartType.LINE)
 .setOption("title", "Sample Chart");
 sheet.insertChart(chartBuilder.build());
}
```

Set Sheet Tab Color

Objective: Change the color of the current sheet's tab.

Explanation: Introduces customization of the sheet's appearance for better organization and visual management.

Code:

```
function setSheetTabColor() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.setTabColor("yellow");
}
```

Protect a Range

Objective: Protect a range in the sheet, preventing it from being edited by anyone except the script runner.

Explanation: Enhances data integrity by limiting modifications to sensitive areas of the spreadsheet.

Code:

```
function protectRange() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const range = sheet.getRange("A1:A10");
 const protection = range.protect().setDescription("Sample Protection");
 protection.removeEditors(protection.getEditors());
 if (protection.canDomainEdit()) {
 protection.setDomainEdit(false);
 }
}
```

Automatically Email Sheet Data

Objective: Write a script to email the contents of a Google Sheet as an HTML table.

Explanation: Demonstrates how to send emails programmatically from Google Sheets data, integrating Google Sheets with Gmail.

Code:

```
function emailSheetData() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const data = sheet.getDataRange().getValues();
 let htmlMessage = "<table border='1'><tr>";
 data[0].forEach(header => htmlMessage += `<th>${header}</th>`);
 htmlMessage += "</tr>";
 data.slice(1).forEach(row => {
 htmlMessage += "<tr>";
 row.forEach(cell => htmlMessage += `<td>${cell}</td>`);
 htmlMessage += "</tr>";
 });
 htmlMessage += "</table>";
 MailApp.sendEmail({
 to: "recipient@example.com",
 subject: "Sheet Data",
```

```
 htmlBody: htmlMessage,
 });
}
```

Import CSV Data into a Sheet

Objective: Fetch CSV data from a URL and import it into a Google Sheet.

Explanation: Explores how to work with external data sources and parse CSV data for use in Sheets.

Code:

```
function importCsvFromUrl() {
 const url = "http://example.com/data.csv"; // Replace with your actual URL
 const csvData = UrlFetchApp.fetch(url).getContentText();
 const csvRows = Utilities.parseCsv(csvData);
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.getRange(1, 1, csvRows.length,
csvRows[0].length).setValues(csvRows);
}
```

Sync Sheet Data with Calendar

Objective: Create Google Calendar events based on data from a Google Sheet.

Explanation: Integrates Google Sheets with Google Calendar, automating the event creation process based on a dataset.

Code:

```
function createCalendarEvents() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const data = sheet.getDataRange().getValues(); // Assuming data has
columns: Event Name, Date, Description
 const calendar = CalendarApp.getDefaultCalendar();
 data.forEach((row, index) => {
```

```
 if (index > 0) { // Skip header row
 const [eventName, eventDate, description] = row;
 calendar.createEvent(eventName, eventDate, eventDate, {description});
 }
 });
}
```

Conditional Row Deletion

Objective: Delete rows in a Google Sheet where the date in a specific column is older than one week.

Explanation: Focuses on date manipulation and conditional logic to manage and clean up data in Sheets.

Code:

```
function deleteOldRows() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const data = sheet.getDataRange().getValues();
 const oneWeekAgo = new Date(new Date().setDate(new Date().getDate()
- 7));
 data.reverse().forEach((row, index) => {
 const rowDate = new Date(row[0]); // Assuming the date is in the first
column
 if (rowDate < oneWeekAgo) {
 sheet.deleteRow(data.length - index);
 }
 });
}
```

Updating Cell Comments

Objective: Programmatically add comments to cells based on their values.

Explanation: Teaches how to use comments for additional context or notes, enhancing the informational value of cell content.

Code:

```
function updateCellComments() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const range = sheet.getDataRange();
 const values = range.getValues();
 values.forEach((row, rowIndex) => {
 row.forEach((cell, colIndex) => {
 if (cell > 100) { // Example condition
 const cellRange = sheet.getRange(rowIndex + 1, colIndex + 1);
 cellRange.setNote(`Value exceeds 100: ${cell}`);
 }
 });
 });
}
```

Generate PDF from Sheet

Objective: Convert a Google Sheet to a PDF file and save it to Google Drive.

Explanation: Showcases how to create PDFs from Sheets data, useful for reporting or archival purposes.

Code:

```
function generatePdfFromSheet() {
 const spreadsheet = SpreadsheetApp.getActiveSpreadsheet();
 const sheet = spreadsheet.getActiveSheet();
 const url =
`https://docs.google.com/spreadsheets/d/${spreadsheet.getId()}/export?exp
ortFormat=pdf&gid=${sheet.getSheetId()}`;
 const options = {
 headers: {
 'Authorization': 'Bearer ' + ScriptApp.getOAuthToken()
 }
 };
 const response = UrlFetchApp.fetch(url, options);
```

```
 const blob = response.getBlob();
 DriveApp.createFile(blob).setName(`${sheet.getName()}.pdf`);
}
```

Link Data Between Sheets

Objective: Write a script to copy data from one sheet to another within the same spreadsheet based on a condition.

Explanation: Introduces techniques for linking and synchronizing data across multiple sheets.

Code:

```
function linkDataBetweenSheets() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const sourceSheet = ss.getSheetByName("Source");
 const targetSheet = ss.getSheetByName("Target");
 const data = sourceSheet.getDataRange().getValues();
 const filteredData = data.filter(row => row[2] === "Yes"); // Assuming the condition is in the third column
 targetSheet.getRange(1, 1, filteredData.length, filteredData[0].length).setValues(filteredData);
}
```

Create a Data Entry Form

Objective: Use Google Forms to create a data entry form that populates a Google Sheet.

Explanation: Demonstrates the integration between Google Forms and Sheets for data collection and management.

Code:

```
function createDataEntryForm() {
 const form = FormApp.create('Data Entry Form');
 form.addTextItem().setTitle('Name');
```

```
 form.addDateItem().setTitle('Date');
 form.addMultipleChoiceItem().setTitle('Category')
 .setChoiceValues(['Category 1', 'Category 2', 'Category 3']);
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 form.setDestination(FormApp.DestinationType.SPREADSHEET,
ss.getId());
}
```

Automate Sheet Archiving

Objective: Archive old data from the active sheet to a new sheet within the same spreadsheet based on a date criterion.

Explanation: Teaches how to manage and archive data dynamically, keeping the active sheet focused on current information.

Code:

```
function archiveOldData() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const sourceSheet = ss.getActiveSheet();
 const archiveSheetName = "Archive " + new Date().toDateString();
 let archiveSheet = ss.getSheetByName(archiveSheetName) ||
ss.insertSheet(archiveSheetName);
 const data = sourceSheet.getDataRange().getValues();
 const currentDate = new Date();
 const oldData = data.filter(row => {
 const rowDataDate = new Date(row[0]); // Assuming date is in the first
column
 return rowDataDate.setHours(0,0,0,0) < currentDate.setHours(0,0,0,0);
 });
 if (oldData.length > 0) {
 archiveSheet.getRange(archiveSheet.getLastRow() + 1, 1, oldData.length,
oldData[0].length).setValues(oldData);
 // Optionally, delete old data from source sheet
 }
}
```

Sheet Data Validation Script

Objective: Implement a script to validate data in a Google Sheet, highlighting any errors.

Explanation: Focuses on ensuring data quality through programmatic validation, critical for maintaining accurate and reliable datasets.

Code:

```
function validateSheetData() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const range = sheet.getDataRange();
 const data = range.getValues();
 data.forEach((row, rowIndex) => {
 row.forEach((cell, colIndex) => {
 if (typeof cell !== 'number' || cell < 0) { // Example validation: ensure positive numbers
 const cellRange = sheet.getRange(rowIndex + 1, colIndex + 1);
 cellRange.setBackground('red').setNote('Invalid data: Expected a positive number.');
 }
 });
 });
}
```

Update Sheet Based on Email Content

Objective: Write a script to update a Google Sheet based on specific information received in Gmail messages.

Explanation: Demonstrates how to integrate Gmail with Google Sheets, parsing email content to update sheet data automatically.

Code:

```
function updateSheetFromEmail() {
```

```
 const label = GmailApp.getUserLabelByName("Updates");
 const threads = label.getThreads();
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 threads.forEach(thread => {
 const message = thread.getMessages()[0]; // Get the first message
 const body = message.getPlainBody();
 // Example: Parse email body for specific information
 const matches = body.match(/Name: (.+)\nAmount: (\d+)/);
 if (matches) {
 const [, name, amount] = matches;
 // Append to sheet
 sheet.appendRow([new Date(), name, amount]);
 }
 thread.removeLabel(label); // Optional: remove label after processing
 });
}
```

Auto-generate Google Forms from Sheet Data

Objective: Create Google Forms automatically based on the options listed in a Google Sheet.

Explanation: Focuses on using Sheets data to dynamically generate forms, useful for surveys or quizzes that frequently update.

Code:

```
function generateFormFromSheet() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Form
Options");
 const data = sheet.getDataRange().getValues(); // Assuming the first row is
headers
 const form = FormApp.create('New Form from Sheet');
 data[0].forEach((question, index) => {
 if (index > 0) { // Skip the first column assuming it's not a question
```

```
 const choices = data.slice(1).map(row => row[index]).filter(choice =>
choice);

form.addMultipleChoiceQuestion().setTitle(question).setChoiceValues(choi
ces);
 }
 });
}
```

Sync Sheet with External Database

Objective: Synchronize a Google Sheet with data from an external database using JDBC.

Explanation: Explores connecting Google Sheets to an external SQL database, allowing for data synchronization.

Code:

```
// Note: This script requires setting up JDBC connection including adding
appropriate JDBC URL in script properties
function syncWithDatabase() {
 const conn = Jdbc.getConnection('JDBC_URL', 'USERNAME',
'PASSWORD');
 const stmt = conn.createStatement();
 const resultSet = stmt.executeQuery('SELECT * FROM myTable');
 const metaData = resultSet.getMetaData();
 const numCols = metaData.getColumnCount();
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.clear();
 // Write column names
 const columnNames = [];
 for (let i = 1; i <= numCols; i++) {
 columnNames.push(metaData.getColumnName(i));
 }
 sheet.appendRow(columnNames);
 // Write data
 while (resultSet.next()) {
```

```
 const row = [];
 for (let col = 0; col < numCols; col++) {
 row.push(resultSet.getString(col + 1));
 }
 sheet.appendRow(row);
 }
 resultSet.close();
 stmt.close();
 conn.close();
}
```

Batch Process Images in Drive Folder

Objective: Resize all images in a specified Google Drive folder and save them to another folder.

Explanation: Demonstrates interacting with Google Drive to process and manage files, particularly images.

Code:

```
// Note: Requires enabling Advanced Drive Service
function resizeImages() {
 const sourceFolder = DriveApp.getFolderById('SOURCE_FOLDER_ID');
 const targetFolder = DriveApp.getFolderById('TARGET_FOLDER_ID');
 const images = sourceFolder.getFilesByType(MimeType.IMAGE_JPEG);
 while (images.hasNext()) {
 const image = images.next();
 const imageBlob = image.getBlob();
 const resizedImage = ImagesService.newImage(imageBlob).resize(100,
100).getBlob();

targetFolder.createFile(resizedImage.setName(`resized-${image.getName(
)}`));
 }
}
```

Monitor Sheet Changes and Log

Objective: Log changes to a specific range in a Google Sheet to a separate "Log" sheet, including the timestamp of the change.

Explanation: Focuses on creating an audit trail for changes in Sheets, useful for tracking edits or updates over time.

Code:

```
function logSheetChanges(e) {
 const logSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Log");
 const range = e.range;
 const oldValue = e.oldValue;
 const newValue = range.getValue();
 const timestamp = new Date();
 logSheet.appendRow([timestamp, range.getA1Notation(), oldValue,
newValue]);
}
```

Generate Sheet Summaries

Objective: Create a summary of data from multiple sheets within a spreadsheet, aggregating key metrics in a "Summary" sheet.

Explanation: Teaches how to compile and summarize data from various sources within a single spreadsheet.

Code:

```
function generateSummary() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const summarySheet = ss.getSheetByName("Summary") ||
ss.insertSheet("Summary");
 const sheets = ss.getSheets();
 summarySheet.clear();
 summarySheet.appendRow(['Sheet Name', 'Total Rows', 'Total Columns']);
 sheets.forEach(sheet => {
 if (sheet.getName() !== "Summary") {
```

```
 const dataRange = sheet.getDataRange();
 const numRows = dataRange.getNumRows();
 const numCols = dataRange.getNumColumns();
 summarySheet.appendRow([sheet.getName(), numRows, numCols]);
 }
 });
}
```

Auto-archive Completed Tasks

Objective: Move rows from a "Tasks" sheet to an "Archive" sheet based on a "Status" column indicating completion.

Explanation: Automates task management within Sheets, streamlining the process of archiving completed tasks.

Code:

```
function archiveCompletedTasks() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const tasksSheet = ss.getSheetByName("Tasks");
 const archiveSheet = ss.getSheetByName("Archive") ||
ss.insertSheet("Archive");
 const tasks = tasksSheet.getDataRange().getValues();
 tasks.forEach((row, index) => {
 if (row[2] === "Completed" && index !== 0) { // Assuming status is in the
third column and skipping header
 archiveSheet.appendRow(row);
 tasksSheet.deleteRow(index + 1); // Adjusting for zero-based index
 }
 });
}
```

Dynamic Sheet Name Creation

Objective: Create new sheets with names based on the current month and year automatically.

Explanation: Enhances automation capabilities, allowing for dynamic creation of time-based organizational structures within spreadsheets.

Code:

```
function createMonthlySheet() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const date = new Date();
 const sheetName = Utilities.formatDate(date,
Session.getScriptTimeZone(), "MMMM yyyy");
 if (!ss.getSheetByName(sheetName)) {
 ss.insertSheet(sheetName);
 }
}
```

Custom Email Alerts for Low Inventory

Objective: Send a custom email alert when the inventory level of any item in a "Stock" sheet falls below a minimum threshold.

Explanation: Utilizes Google Sheets for inventory management, integrating custom alerts for better stock control.

Code:

```
function sendInventoryAlerts() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Stock");
 const data = sheet.getDataRange().getValues();
 let message = "";
 data.forEach((row, index) => {
 if (index > 0) { // Skip header row
 const [item, quantity] = row;
 if (quantity < 10) { // Assuming the threshold is 10
 message += `Low stock alert for ${item}: only ${quantity} left.\n`;
 }
 }
 });
```

```
 if (message) {
 MailApp.sendEmail("manager@example.com", "Inventory Alert",
message);
 }
}
```

Automate Expense Tracking

Objective: Automatically categorize expenses entered into a "Expenses" sheet and calculate totals by category in a "Summary" sheet.

Explanation: Showcases how to automate financial tracking and reporting within Google Sheets.

Code:

```
function categorizeAndSummarizeExpenses() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const expensesSheet = ss.getSheetByName("Expenses");
 const summarySheet = ss.getSheetByName("Summary") ||
ss.insertSheet("Summary");
 const expenses = expensesSheet.getDataRange().getValues();
 const categories = {};
 expenses.forEach((row, index) => {
 if (index > 0) { // Skip header
 const [date, category, amount] = row;
 if (categories[category]) {
 categories[category] += amount;
 } else {
 categories[category] = amount;
 }
 }
 });
 summarySheet.clear();
 summarySheet.appendRow(["Category", "Total"]);
 for (const [category, total] of Object.entries(categories)) {
 summarySheet.appendRow([category, total]);
 }
```

```
}
```

Track Sheet Edits in Real-time

Objective: Implement a script to track and log every edit made in a Google Sheet, including the cell reference, old value, new value, and timestamp.

Explanation: Enhances data governance by providing a detailed audit trail of all changes made to a spreadsheet.

Code:

```
function onEdit(e) {
 const logSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Edit Log") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Edit Log");
 const {range, oldValue, value} = e;
 const timestamp = new Date();
 const cell = range.getA1Notation();
 logSheet.appendRow([timestamp, cell, oldValue, value]);
}
```

Automated Sheet Cleanup

Objective: Create a script that runs daily to clean up a specific Google Sheet, removing rows where the date in a specific column is older than 30 days.

Explanation: Keeps data in Sheets relevant and manageable by automatically removing outdated information.

Code:

```
function dailyCleanup() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data");
 const dataRange = sheet.getDataRange();
```

```
 const data = dataRange.getValues();
 const cutoffDate = new Date(new Date().setDate(new Date().getDate() -
30));
 data.reverse().forEach((row, index) => {
 const rowDate = new Date(row[0]); // Assuming date is in the first column
 if (rowDate < cutoffDate) {
 sheet.deleteRow(data.length - index);
 }
 });
}
```

Summarize Data by Category

Objective: Write a script to summarize data in a Google Sheet, calculating the total amount spent per category and outputting the summary to a new sheet.

Explanation: Demonstrates data aggregation techniques, essential for financial and analytical reporting.

Code:

```
function summarizeByCategory() {
 const sourceSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Expenses");
 const summarySheet =
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Summary");
 const data = sourceSheet.getDataRange().getValues();
 const summary = {};
 data.forEach((row, index) => {
 if (index > 0) { // Skip header row
 const [category, amount] = row;
 summary[category] = summary[category] ? summary[category] + amount :
amount;
 }
 });
 summarySheet.appendRow(["Category", "Total"]);
 for (const [category, total] of Object.entries(summary)) {
```

```
 summarySheet.appendRow([category, total]);
 }
}
```

Auto-Generate Document from Sheet Data

Objective: Automatically generate a Google Doc report from data in a Google Sheet, including a summary and detailed table of contents.

Explanation: Explores integrating Google Sheets with Docs to create dynamic, data-driven documents.

Code:

```
function generateDocReport() {
 const dataSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Report Data");
 const data = dataSheet.getDataRange().getValues();
 const doc = DocumentApp.create("Report " + new
Date().toISOString().slice(0, 10));
 const body = doc.getBody();
 body.appendParagraph("Report
Summary").setHeading(DocumentApp.ParagraphHeading.HEADING1);
 // Example summary generation
 body.appendParagraph("This report generated on " + new
Date().toString());
 body.appendParagraph("Detailed
Data").setHeading(DocumentApp.ParagraphHeading.HEADING1);
 const table = [];
 data.forEach((row, index) => {
 if (index === 0) {
 table.push(row.map(header => header.toUpperCase()));
 } else {
 table.push(row);
 }
 });
```

```
 body.appendTable(table);
 doc.saveAndClose();
}
```

Dynamic Range Name Creation

Objective: Write a script to dynamically create named ranges in a Google Sheet based on content in specific cells.

Explanation: Improves data management and accessibility by utilizing named ranges, facilitating easier reference to specific data segments.

Code:

```
function createDynamicNamedRanges() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const data = sheet.getDataRange().getValues();
 data.forEach((row, index) => {
 if (row[0] !== "") { // Assuming the name for the range is in the first column
 const rangeName = row[0];
 const range = sheet.getRange(index + 1, 1, 1, sheet.getLastColumn());
 SpreadsheetApp.getActiveSpreadsheet().setNamedRange(rangeName,
range);
 }
 });
}
```

Link Data Across Spreadsheets

Objective: Develop a script to link data from a master spreadsheet to multiple child spreadsheets based on specific criteria.

Explanation: Showcases how to manage data across multiple spreadsheets, ensuring consistency and reducing manual data entry.

Code:

```
function linkDataToChildren() {
```

```
 const masterSheet =
SpreadsheetApp.openById("MASTER_SPREADSHEET_ID").getSheetByN
ame("Data");
 const childSpreadsheetIds = ["CHILD_SPREADSHEET_ID_1",
"CHILD_SPREADSHEET_ID_2"]; // Example IDs
 const masterData = masterSheet.getDataRange().getValues();
 childSpreadsheetIds.forEach(spreadsheetId => {
 const childSheet =
SpreadsheetApp.openById(spreadsheetId).getSheetByName("Data");
 childSheet.clear(); // Optional: clear existing data
 childSheet.getRange(1, 1, masterData.length,
masterData[0].length).setValues(masterData);
 });
}
```

Analyze Text Sentiment

Objective: Use Google's Natural Language API to analyze the sentiment of text entries in a Google Sheet and write the results to another column.

Explanation: Integrates Google Cloud Natural Language API for sentiment analysis, adding a layer of text analytics to sheet data.

Code:

```
// Note: This requires enabling Google Cloud Natural Language API and
setting up Google Cloud Project
function analyzeTextSentiment() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Feedback");
 const texts = sheet.getRange("A2:A" + sheet.getLastRow()).getValues();
 const apiKey = "YOUR_API_KEY"; // Replace with your API key
 const apiEndpoint =
"https://language.googleapis.com/v1/documents:analyzeSentiment?key=" +
apiKey;
 texts.forEach((text, index) => {
 if (text[0] !== "") {
 const apiPayload = {
```

```
 document: {
 type: "PLAIN_TEXT",
 content: text[0]
 }
 };
 const options = {
 method: "post",
 contentType: "application/json",
 payload: JSON.stringify(apiPayload)
 };
 const response = UrlFetchApp.fetch(apiEndpoint, options);
 const sentimentScore =
JSON.parse(response.getContentText()).documentSentiment.score;
 sheet.getRange("B" + (index + 2)).setValue(sentimentScore); // Assuming
sentiment scores are written in column B
 }
 });
}
```

Automate Meeting Minutes Generation

Objective: Automate the creation of meeting minutes in Google Docs from scheduled events in Google Calendar, populated with attendee names and topics from a Google Sheet.

Explanation: Showcases integration between Google Calendar, Sheets, and Docs for automated meeting management.

Code:

```
function generateMeetingMinutes() {
 const events = CalendarApp.getEventsForDay(new Date());
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Meeting
Topics");
 const topics = sheet.getDataRange().getValues();
 events.forEach(event => {
```

```
 const doc = DocumentApp.create("Minutes: " + event.getTitle() + " - " +
Utilities.formatDate(new Date(), Session.getScriptTimeZone(),
"yyyy-MM-dd"));
 const body = doc.getBody();
 body.appendParagraph("Meeting Title: " + event.getTitle());
 body.appendParagraph("Attendees: " + event.getGuestList().map(guest =>
guest.getEmail()).join(", "));
 body.appendParagraph("Topics:");
 topics.forEach((topic, index) => {
 if (index > 0) { // Skip header
 body.appendListItem(topic[0]);
 }
 });
 doc.saveAndClose();
 });
}
```

Custom Dashboard in Google Sites

Objective: Embed a Google Sheet as a live, interactive chart in a Google Site, creating a custom dashboard that updates in real-time.

Explanation: Utilizes Google Sites and Sheets to create dynamic, data-driven dashboards for reporting and visualization.

Code:

```
// Note: This script assumes you have edit access to a Google Site and a
chart in a Google Sheet ready to be embedded.
function embedSheetChartInSite() {
 const site = SitesApp.getSite("example.com", "site-path");
 const sheet = SpreadsheetApp.getActiveSpreadsheet();
 const sheetId = sheet.getId();
 const chart = sheet.getSheets()[0].getCharts()[0]; // Get the first chart in
the first sheet
 const imageUrl =
chart.getAs('image/png').getBlob().getAs('image/png').getDataAsString();
 const image = '<img src="' + imageUrl + '" />';
```

```
 site.createAnnouncement("Dashboard Update", image,
SitesApp.PageType.WEB_PAGE, []);
}
```

Dynamic Project Timeline

Objective: Create a dynamic project timeline in Google Sheets, using conditional formatting and formulas to update based on project start dates and durations.

Explanation: Enhances project management within Google Sheets, allowing for visual representation and automatic updates of project timelines.

Code:

```
function setupProjectTimeline() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Projects");
 const projects = sheet.getDataRange().getValues();
 // Assuming projects data starts from the second row and has the
structure: [Project Name, Start Date, Duration (days)]
 projects.forEach((project, index) => {
 if (index > 0) { // Skip header
 const [name, startDate, duration] = project;
 const startColumn = 2; // Assuming timeline starts from column B
 const startRow = index + 1;
 const startOffset = (new Date(startDate) - new Date(projects[1][1])) / (1000
* 60 * 60 * 24); // Days from the first project start date
 const endOffset = startOffset + duration;
 // Apply conditional formatting to visualize the timeline
 const range = sheet.getRange(startRow, startColumn + startOffset, 1,
duration);
 range.setBackground("green");
 // Extend this example to dynamically adjust based on project data and
apply to your specific needs
 }
 });
```

```
}
```

Import JSON Data into Sheets

Objective: Write a script to import JSON data from a public API into a Google Sheet.

Explanation: Demonstrates how to fetch data from external APIs and parse JSON data, an essential skill for integrating third-party services with Google Sheets.

Code:

```
function importJsonData() {
 const url = 'https://api.example.com/data';
 const response = UrlFetchApp.fetch(url);
 const jsonData = JSON.parse(response.getContentText());
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.clear(); // Clear existing data
 // Assuming jsonData is an array of objects
 const headers = Object.keys(jsonData[0]);
 sheet.appendRow(headers); // Append headers
 jsonData.forEach(item => {
 const row = headers.map(header => item[header]);
 sheet.appendRow(row);
 });
}
```

Automated Email Reports from Sheets

Objective: Create a script that sends daily email reports summarizing data from a specific Google Sheet.

Explanation: Integrates Google Sheets with Gmail to automate the process of generating and sending reports, showcasing the potential for automated communication based on spreadsheet data.

Code:

```
function sendDailyEmailReport() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName('Sales Data');
 const dataRange = sheet.getDataRange();
 const data = dataRange.getValues();
 let report = 'Daily Sales Report:\n\n';
 // Generate report content from data
 data.forEach((row, index) => {
 if (index === 0) return; // Skip header
 const [date, sales] = row;
 report += `Date: ${date}, Sales: ${sales}\n`;
 });
 MailApp.sendEmail({
 to: 'manager@example.com',
 subject: 'Daily Sales Report',
 body: report,
 });
}
```

Sync Google Calendar with Sheets

Objective: Synchronize Google Calendar events with a Google Sheet, listing upcoming events for the next 7 days.

Explanation: Showcases how to integrate Google Calendar with Sheets, enabling the creation of custom event management and tracking solutions.

Code:

```
function syncCalendarWithSheet() {
 const calendar = CalendarApp.getDefaultCalendar();
 const today = new Date();
 const oneWeekLater = new Date(today.getTime() + 7 * 24 * 60 * 60 *
1000);
 const events = calendar.getEvents(today, oneWeekLater);
```

```
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName('Upcoming
Events');
 sheet.clear(); // Clear existing events
 sheet.appendRow(['Event Title', 'Start Date', 'End Date']);
 events.forEach(event => {
 const title = event.getTitle();
 const startDate = event.getStartTime();
 const endDate = event.getEndTime();
 sheet.appendRow([title, startDate, endDate]);
 });
}
```

Custom Sheet Functions with Cache

Objective: Develop a custom Google Sheets function that uses the Cache service to improve performance for repeated calculations.

Explanation: Introduces the concept of caching in Google Apps Script to optimize custom function performance, particularly beneficial for resource-intensive operations or external data fetches.

Code:

```
function MY_CUSTOM_FUNCTION(input) {
 const cache = CacheService.getScriptCache();
 const cachedResult = cache.get(input);
 if (cachedResult) {
 return JSON.parse(cachedResult);
 } else {
 // Perform some calculations or fetch external data
 const result = performExpensiveCalculation(input);
 cache.put(input, JSON.stringify(result), 1500); // Cache for 25 minutes
 return result;
 }
 function performExpensiveCalculation(input) {
 // Placeholder for actual calculation logic
 return input * 2; // Example calculation
```

```
 }
}
```

Generate Google Docs from Sheet Rows

Objective: Automatically generate Google Docs for each row in a Google Sheet, using the row data to populate a document template.

Explanation: Explores the automation of document creation based on spreadsheet data, illustrating how to streamline reporting, invoicing, or personalized document generation.

Code:

```
function generateDocsFromSheetRows() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName('Data');
 const rows = sheet.getDataRange().getValues();
 rows.forEach((row, index) => {
 if (index === 0) return; // Skip header row
 const doc = DocumentApp.create(`Document for ${row[0]}`); // Assuming
first column is a name or identifier
 const body = doc.getBody();
 body.appendParagraph(`Hello, ${row[0]}! Your data is: ${row.join(', ')}.`);
 // Further customize document as needed
 });
}
```

Advanced Data Validation

Objective: Implement advanced data validation in a Google Sheet that checks for duplicate entries in a specific column.

Explanation: Demonstrates the use of Apps Script to enforce data integrity beyond the built-in data validation features, ensuring unique values in a dataset.

Code:

```
function checkForDuplicates() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName('Registrations')
;
 const column = sheet.getRange('A:A').getValues(); // Check column A for
duplicates
 const unique = new Set();
 column.forEach((cell, index) => {
 if (unique.has(cell[0])) {
 sheet.getRange(index + 1, 1).setBackground('red');
 } else if (cell[0] !== '') {
 unique.add(cell[0]);
 }
 });
}
```

Merge Sheet Data into Master Sheet

Objective: Create a script to merge data from multiple sheets into a master sheet within the same Google Spreadsheet.

Explanation: Useful for consolidating data from various sources, this exercise highlights how to programmatically combine datasets in Google Sheets.

Code:

```
function mergeDataIntoMasterSheet() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const masterSheet = ss.getSheetByName('Master') ||
ss.insertSheet('Master');
 masterSheet.clear(); // Optional: Clear existing data in Master sheet
 ss.getSheets().forEach(sheet => {
 if (sheet.getName() !== 'Master') {
 const data = sheet.getDataRange().getValues();
 masterSheet.getRange(masterSheet.getLastRow() + 1, 1, data.length,
data[0].length).setValues(data);
 }
```

```
 });
}
```

Dynamic Form Responses Sheet Selection

Objective: Dynamically select the destination sheet for Google Forms responses based on the form submission date.

Explanation: Tailors data organization within a spreadsheet based on submission timing, showcasing dynamic interaction between Google Forms and Sheets.

Code:

```
function onFormSubmit(e) {
 const responseSheetName = Utilities.formatDate(new Date(),
Session.getScriptTimeZone(), "MMMM yyyy");
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const sheet = ss.getSheetByName(responseSheetName) ||
ss.insertSheet(responseSheetName);
 const formResponse = e.values;
 sheet.appendRow(formResponse);
}
```

Automated Data Backfill

Objective: Write a script to automatically backfill missing data in a Google Sheet based on predefined rules or placeholders.

Explanation: Addresses data completeness challenges by automatically populating missing information, enhancing data analysis readiness.

Code:

```
function backfillMissingData() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName('Inventory');
```

```
 const range = sheet.getDataRange();
 const data = range.getValues();
 data.forEach((row, rowIndex) => {
 row.forEach((cell, colIndex) => {
 if (cell === '' || cell === 'UNKNOWN') { // Identify missing data
 const backfillValue = getBackfillValueForColumn(colIndex); // Custom
function to determine backfill value
 sheet.getRange(rowIndex + 1, colIndex + 1).setValue(backfillValue);
 }
 });
 });
 function getBackfillValueForColumn(colIndex) {
 // Placeholder for logic to determine backfill value based on column
 // Example: return colIndex === 2 ? 'N/A' : 0;
 return 'Backfilled'; // Default backfill value
 }
}
```

Sheet to Calendar Event Sync with Reminders

Objective: Synchronize Google Sheet entries with Google Calendar, creating calendar events with reminders based on the sheet's data.

Explanation: Enhances personal or team productivity by automating the creation of calendar events, including setting reminders from a schedule or task list maintained in Google Sheets.

Code:

```
function syncSheetToCalendarWithReminders() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName('Events');
 const events = sheet.getDataRange().getValues();
 const calendar = CalendarApp.getDefaultCalendar();
 events.forEach((event, index) => {
 if (index > 0) { // Skip header row
```

```
 const [title, startDate, endDate, reminderMinutes] = event;
 const newEvent = calendar.createEvent(title, new Date(startDate), new
Date(endDate));
 newEvent.addPopupReminder(reminderMinutes); // Set a popup reminder
 }
 });
}
```

Create Conditional Dropdown Lists

Objective: Programmatically create conditional dropdown lists in Google Sheets, where the options in one dropdown determine the choices in another.

Explanation: Demonstrates advanced data validation techniques, allowing for dynamic and interactive spreadsheets.

Code:

```
function createConditionalDropdowns() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const mainCategory = ['Fruit', 'Vegetable'];
 const dependentOptions = {
 'Fruit': ['Apple', 'Banana', 'Cherry'],
 'Vegetable': ['Carrot', 'Lettuce', 'Onion']
 };
 // Main category dropdown
 const mainDropdown = sheet.getRange("A1");
 mainDropdown.setDataValidation(SpreadsheetApp.newDataValidation()
 .requireValueInList(mainCategory, true).build());
 // Dependent dropdown setup
 const cell = sheet.getRange("B1");
 sheet.getRange("A1").onEdit = function(e) {
 const selectedMainCategory = e.value;
 const options = dependentOptions[selectedMainCategory];
 if (options) {
 cell.setDataValidation(SpreadsheetApp.newDataValidation()
```

```
 .requireValueInList(options, true).build());
 }
 };
}
// Note: This script outline assumes an onEdit trigger setup for dynamic functionality.
```

Auto-format New Rows

Objective: Automatically apply specific formatting to new rows added to a Google Sheet.

Explanation: Utilizes the onEdit or onChange trigger to maintain consistent styling, ensuring data uniformity across a sheet.

Code:

```
function autoFormatNewRows(e) {
 const sheet = e.source.getActiveSheet();
 const addedRange = e.range;
 if (sheet.getName() === "Data" && addedRange.getRow() > 1) { // Assuming "Data" is the target sheet
 addedRange.setBackground('#ffff99'); // Example: Set background color
 addedRange.setFontWeight('bold'); // Set text bold
 }
}
// Note: Set up an installable trigger for onChange to capture row additions.
```

Summarize Data Across Sheets

Objective: Aggregate and summarize data from multiple sheets within a spreadsheet into a summary sheet.

Explanation: Provides a holistic view of data scattered across multiple sheets, centralizing information for analysis or reporting.

Code:

```
function summarizeAcrossSheets() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const summarySheet = ss.getSheetByName("Summary") ||
ss.insertSheet("Summary");
 summarySheet.clear(); // Clear existing data
 ss.getSheets().forEach(sheet => {
 if (sheet.getName() !== "Summary") {
 const total = sheet.getRange("B2:B" + sheet.getLastRow()) // Assuming
data to sum is in column B
 .getValues()
 .reduce((sum, [value]) => sum + value, 0);
 summarySheet.appendRow([sheet.getName(), total]);
 }
 });
}
```

Monitor Sheet for External Changes

Objective: Create a script to monitor a sheet for changes made outside of the Google Sheets UI, such as those from API calls or form submissions, and log these changes.

Explanation: Ensures data integrity by tracking external modifications, important for auditing and compliance purposes.

Code:

```
function monitorForExternalChanges() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const logSheet = ss.getSheetByName("Change Log") ||
ss.insertSheet("Change Log");
 const triggerSheet = ss.getSheetByName("Data");
 // Placeholder for logic to detect changes
 // Example: Compare cached data snapshot with current data
 // On detecting changes, log them
 logSheet.appendRow(["Detected external change", new Date()]);
}
```

```
// Note: Implement detailed comparison logic based on specific use case requirements.
```

Dynamic Gantt Chart Creation

Objective: Generate a Gantt chart in Google Sheets based on project task data, including start dates and durations.

Explanation: Visualizes project timelines, providing a graphical representation of tasks and their schedules, aiding in project management.

Code:

```
function createGanttChart() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Projects");
 const chartBuilder = sheet.newChart();
 chartBuilder
 .setChartType(Charts.ChartType.BAR)
 .addRange(sheet.getRange("A2:C" + sheet.getLastRow())) // Assuming A:
Task, B: Start Date, C: Duration
 .setPosition(5, 5, 0, 0)
 .setOption('title', 'Project Gantt Chart')
 .setOption('height', 600)
 .setOption('width', 800);
 sheet.insertChart(chartBuilder.build());
}
```

Link Sheets with Dynamic Hyperlinks

Objective: Use scripts to create dynamic hyperlinks in a master sheet, linking to specific ranges in other sheets based on criteria.

Explanation: Enhances navigation within complex spreadsheets, streamlining access to related data across multiple sheets.

Code:

```
function linkSheetsWithHyperlinks() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const masterSheet = ss.getSheetByName("Master");
 const sheets = ss.getSheets();
 sheets.forEach(sheet => {
 if (sheet.getName() !== "Master") {
 const sheetName = sheet.getName();
 const hyperlinkFormula = `=HYPERLINK("#gid=${sheet.getSheetId()}","Go
to ${sheetName}")`;
 masterSheet.appendRow([sheetName, hyperlinkFormula]);
 }
 });
}
```

Automate Email Digest of Sheet Updates

Objective: Send a weekly email digest summarizing changes made in a Google Sheet, including added rows and modified data.

Explanation: Keeps stakeholders informed about spreadsheet updates, facilitating communication and collaboration.

Code:

```
function sendWeeklyUpdateDigest() {
 const changesLogSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Changes
Log");
 const changes = changesLogSheet.getRange("A2:B" +
changesLogSheet.getLastRow()).getValues(); // Assuming A: Description,
B: Timestamp
 let emailBody = "Weekly Update Digest:\n\n";
 changes.forEach(([description, timestamp]) => {
 emailBody += `${timestamp}: ${description}\n`;
 });
 MailApp.sendEmail({
 to: "team@example.com",
 subject: "Weekly Spreadsheet Update Digest",
```

```
 body: emailBody,
 });
}
```

Implement Row-Level Access Control

Objective: Create a script that hides or shows rows in a Google Sheet based on the current user's email address, implementing basic access control.

Explanation: Enables personalized views of spreadsheet data, ensuring users only access information relevant or permissible to them.

Code:

```
function implementRowLevelAccessControl() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const userEmail = Session.getActiveUser().getEmail();
 const dataRange = sheet.getDataRange();
 const data = dataRange.getValues();
 data.forEach((row, index) => {
 const accessEmail = row[0]; // Assuming the user's email is in the first column
 if (accessEmail !== userEmail) {
 sheet.hideRows(index + 1);
 } else {
 sheet.showRows(index + 1);
 }
 });
}
```

Auto-Update Charts Based on Filters

Objective: Dynamically update charts in a Google Sheet based on data filtered by the user, reflecting real-time changes in visualizations.

Explanation: Enhances data analysis by automatically adjusting visual representations as underlying data changes, improving insights.

Code:

```
function autoUpdateChartsOnFilter() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data");
 const charts = sheet.getCharts();
 charts.forEach(chart => {
 const chartRange = chart.getRanges()[0]; // Assuming single range source
for simplicity
 const filteredRange =
chartRange.getSheet().getRange(chartRange.getA1Notation());
 const newChart = chart.modify().setRange(filteredRange).build();
 sheet.updateChart(newChart);
 });
}
```

Collaborative Task Assignment

Objective: Develop a system within Google Sheets for task assignment, allowing users to claim tasks by entering their names, which then locks the task row for editing by others.

Explanation: Facilitates collaborative work management directly within Google Sheets, streamlining task distribution and ownership.

Code:

```
function collaborativeTaskAssignment(e) {
 const sheet = e.source.getActiveSheet();
 if (sheet.getName() === "Tasks" && e.range.getColumn() === 3) { //
Assuming column C is for claiming tasks
 const claimedBy = e.value;
 if (claimedBy) {
 const row = e.range.getRow();
 const protection = sheet.getRange(row, 1, 1,
sheet.getLastColumn()).protect();
 protection.removeEditors(protection.getEditors());
 protection.addEditor(claimedBy);
```

```
 }
 }
}
// Note: Set up an onEdit trigger for this function to automatically execute
on user edits.
```

Automated Data Cleanup

Objective: Develop a script that automatically cleans up a dataset in a Google Sheet, removing any rows that contain specific keywords in a designated column.

Explanation: Demonstrates how to programmatically ensure data quality by filtering out unwanted or irrelevant data entries based on predefined criteria.

Code:

```
function automatedDataCleanup() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data
Cleanup");
 const range = sheet.getDataRange();
 const data = range.getValues();
 const keywords = ["Irrelevant", "Remove", "Unwanted"]; // Keywords to
trigger row removal
 for (let i = data.length - 1; i >= 0; i--) {
 if (keywords.includes(data[i][2])) { // Assuming the keywords are in the third
column
 sheet.deleteRow(i + 1);
 }
 }
}
```

Track Sheet Access

Objective: Implement a system to log every instance of a Google Sheet being opened, recording the timestamp and user email to a separate "Access Log" sheet.

Explanation: Enhances security and auditing by providing a historical record of sheet access, useful for monitoring and compliance.

Code:

```
function onOpen(e) {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const logSheet = ss.getSheetByName("Access Log") ||
ss.insertSheet("Access Log");
 const userEmail = Session.getActiveUser().getEmail();
 const timestamp = new Date();
 logSheet.appendRow([timestamp, userEmail]);
}
```

Generate Invoice PDFs

Objective: Automatically generate and email PDF invoices from data entered in a Google Sheet, including customer information and purchase details.

Explanation: Streamlines the invoicing process by leveraging Google Sheets data to create personalized, ready-to-send invoices.

Code:

```
function generateAndEmailInvoices() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Invoices");
 const invoices = sheet.getDataRange().getValues();
 invoices.forEach((invoice, index) => {
 if (index === 0) return; // Skip header row
 const [customerEmail, customerName, amountDue] = invoice;
 const docTemplate = DocumentApp.openById("TEMPLATE_DOC_ID");
 const docCopy = docTemplate.copy("Invoice for " + customerName);
```

```
 const body = docCopy.getBody();
 body.replaceText("{{CustomerName}}", customerName);
 body.replaceText("{{AmountDue}}", amountDue.toString());
 const pdfBlob = docCopy.getAs(MimeType.PDF);
 MailApp.sendEmail(customerEmail, "Your Invoice", "Please find attached
your invoice.", {
 attachments: [pdfBlob],
 name: "Automated Emailer Script"
 });
 // Optionally, delete the temporary document
 DriveApp.getFileById(docCopy.getId()).setTrashed(true);
 });
}
```

Synchronize Sheet to Database

Objective: Create a script to synchronize data from a Google Sheet to an external SQL database, ensuring the database reflects the current sheet data.

Explanation: Facilitates data consistency between Google Sheets and external databases, critical for applications relying on up-to-date information across platforms.

Code:

```
// Note: This script requires setting up Google Cloud SQL and JDBC
connection.
function syncSheetToDatabase() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sync Data");
 const data = sheet.getDataRange().getValues();
 const conn = Jdbc.getConnection('JDBC_URL', 'USERNAME',
'PASSWORD');
 data.forEach((row, index) => {
 if (index > 0) { // Skip header row
 const [id, name, value] = row;
```

```
 const stmt = conn.prepareStatement('REPLACE INTO tableName (id,
name, value) VALUES (?, ?, ?)');
 stmt.setString(1, id);
 stmt.setString(2, name);
 stmt.setString(3, value);
 stmt.execute();
 }
 });
}
```

Dynamic Project Dashboard

Objective: Utilize Google Sheets to create a dynamic project management dashboard, automatically updating with project progress and milestones.

Explanation: Demonstrates the use of formulas, scripts, and charts in Sheets to provide real-time visibility into project status, enhancing project tracking and stakeholder communication.

Code:

```
function updateProjectDashboard() {
 const projectsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Projects");
 const dashboardSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Dashboard");
 const projectData = projectsSheet.getDataRange().getValues();
 dashboardSheet.clear(); // Prepare for fresh update
 dashboardSheet.appendRow(["Project", "Status", "Completion %",
"Milestones Completed/Total"]);
 projectData.forEach((project, index) => {
 if (index === 0) return; // Skip header
 const [projectName, status, completion, milestonesCompleted,
totalMilestones] = project;
 dashboardSheet.appendRow([projectName, status, completion,
`${milestonesCompleted}/${totalMilestones}`]);
 });
```

```
 // Additional steps could include setting up conditional formatting rules or generating charts based on the updated data.
}
```

Optimize Large Sheet Performance

Objective: Implement strategies in a Google Sheet script to optimize performance when dealing with large datasets, minimizing execution time and resource consumption.

Explanation: Focuses on efficient data processing techniques, such as batch operations and selective data loading, to enhance script performance.

Code:

```
function optimizePerformanceForLargeSheets() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Large Data");
 const range = sheet.getRange("A2:B" + sheet.getLastRow()); // Assuming data resides in columns A and B
 const data = range.getValues();
 // Example optimization: Process data in chunks
 const chunkSize = 100; // Process 100 rows at a time
 for (let i = 0; i < data.length; i += chunkSize) {
 const chunk = data.slice(i, i + chunkSize);
 // Process chunk
 // Example processing: Log chunk or perform calculations
 }
 // Note: This is a generic template. Specific optimizations depend on the task at hand.
}
```

Custom Analytics from Sheet Data

Objective: Calculate and display custom analytics in a Google Sheet, such as moving averages or growth rates, based on historical data entries.

Explanation: Enhances data analysis capabilities within Sheets, allowing for sophisticated statistical calculations and trend analysis.

Code:

```
function calculateCustomAnalytics() {
 const dataSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sales Data");
 const analyticsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Analytics") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Analytics");
 const salesData = dataSheet.getRange("B2:B" +
dataSheet.getLastRow()).getValues(); // Assuming sales figures are in
column B
 analyticsSheet.clear(); // Clear existing analytics
 analyticsSheet.appendRow(["Month", "Moving Average"]);
 const movingAveragePeriod = 3; // Example: 3-month moving average
 for (let i = movingAveragePeriod; i <= salesData.length; i++) {
 const slice = salesData.slice(i - movingAveragePeriod, i);
 const sum = slice.reduce((acc, val) => acc + val[0], 0);
 const movingAverage = sum / movingAveragePeriod;
 analyticsSheet.appendRow([`Month ${i}`, movingAverage]);
 }
}
```

Implement Version Control for Sheet Edits

Objective: Create a script to implement basic version control for a Google Sheet, logging each edit with a timestamp, the editor's email, and a snapshot of the edited range.

Explanation: Provides a rudimentary form of version control, crucial for tracking changes and facilitating collaboration in shared documents.

Code:

```
function logSheetEdits(e) {
```

```
 const editLogSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Edit Log") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Edit Log");
 const {range, value, oldValue, user} = e;
 const timestamp = new Date();
 const userEmail = user.getEmail();
 const cellReference = range.getA1Notation();
 editLogSheet.appendRow([timestamp, userEmail, cellReference, oldValue,
value]);
}
```

Sheet-based Task Scheduler

Objective: Use Google Sheets as a task scheduler, where tasks and their scheduled times are listed, and a script triggers task execution at the specified times.

Explanation: Showcases how to use Google Sheets for simple task scheduling, leveraging Google Apps Script triggers for task execution.

Code:

```
// Note: Due to Apps Script limitations, exact execution at specified times
may require setting up time-driven triggers manually or using a workaround
to check periodically.
function executeScheduledTasks() {
 const scheduleSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Schedule");
 const tasks = scheduleSheet.getDataRange().getValues();
 const currentTime = new Date();
 tasks.forEach((task, index) => {
 if (index === 0) return; // Skip header
 const [taskName, scheduledTime] = task;
 const taskTime = new Date(scheduledTime);
 if (taskTime <= currentTime) {
 // Placeholder for task execution logic
 // Example: if (taskName === "Send Email") sendEmail();
 }
```

```
 });
}
```

Real-time Data Visualization

Objective: Dynamically create and update data visualizations in a Google Sheet based on real-time data changes, such as live sales figures or sensor data.

Explanation: Enhances the interactivity and responsiveness of data visualizations in Sheets, enabling real-time monitoring and analysis.

Code:

```
function updateRealTimeDataVisualization() {
 const dataSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Real-time
Data");
 const data = dataSheet.getDataRange().getValues();
 const chartSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Live Chart") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Live Chart");
 // Assume existing chart is to be updated
 let chart = chartSheet.getCharts()[0]; // Get the first chart for update
 if (chart) {
 chart = chart.modify()
 .setOption('title', 'Real-time Data Visualization')
 .addRange(dataSheet.getRange(1, 1, data.length, data[0].length))
 .build();
 chartSheet.updateChart(chart);
 } else {
 // Create new chart if not existing
 const newChart = chartSheet.newChart()
 .setChartType(Charts.ChartType.LINE)
 .setOption('title', 'Real-time Data Visualization')
 .addRange(dataSheet.getRange(1, 1, data.length, data[0].length))
 .setPosition(5, 5, 0, 0)
 .build();
```

```
 chartSheet.insertChart(newChart);
 }
}
```

Batch Update Sheet Formatting

Objective: Write a script to apply conditional formatting rules across multiple ranges in a Google Sheet based on specific data criteria.

Explanation: Demonstrates how to programmatically enhance data readability and visual analysis through conditional formatting, applied in bulk for efficiency.

Code:

```
function batchUpdateFormatting() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const range = sheet.getDataRange();
 const rules = range.getConditionalFormatRules();
 const newRule = SpreadsheetApp.newConditionalFormatRule()
 .whenNumberGreaterThan(100)
 .setBackground("#FFEB3B")
 .setRanges([sheet.getRange("A1:A10"), sheet.getRange("B1:B10")])
 .build();
 rules.push(newRule);
 range.setConditionalFormatRules(rules);
}
```

Automate Document Assembly from Sheets

Objective: Create a script to assemble a Google Doc report from various Google Sheets, aggregating data into a comprehensive document.

Explanation: Explores the integration between Google Sheets and Docs to automate the creation of complex documents, such as reports or proposals, based on sheet data.

Code:

```
function automateDocumentAssembly() {
 const dataSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Report Data");
 const rows = dataSheet.getDataRange().getValues();
 const doc = DocumentApp.create("Assembled Report");
 const body = doc.getBody();
 rows.forEach((row, index) => {
 body.appendParagraph(row.join(", "));
 if (index === 0) body.appendHorizontalRule(); // Example separator after
header
 });
 doc.saveAndClose();
}
```

Create a Custom Data Entry Form

Objective: Develop a custom data entry form using Google Apps Script's HTML Service, submitting form data back to a Google Sheet.

Explanation: Showcases how to create and deploy custom web apps or forms for data collection, providing a tailored user interface for data entry.

Code:

```
function showCustomForm() {
 const html = HtmlService.createHtmlOutputFromFile('Form.html')
 .setWidth(400)
 .setHeight(300);
 SpreadsheetApp.getUi().showModalDialog(html, 'Enter Data');
}
// Assume 'Form.html' exists in the script project, containing form elements
and a submission handler that calls the below function
function submitFormData(formData) {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.appendRow([formData.name, formData.email, formData.comment]);
// Matching form input names
```

```
}
```

Sync Sheet Data with External API

Objective: Write a script to synchronize Google Sheet data with an external REST API, both sending and receiving data to keep the sheet updated.

Explanation: Enhances external data integration capabilities, allowing for real-time data exchange between a Google Sheet and third-party services.

Code:

```
function syncWithExternalApi() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("External
Data");
 const data = sheet.getDataRange().getValues();
 // Send data to API
 const response = UrlFetchApp.fetch("https://api.example.com/data", {
 method: "post",
 contentType: "application/json",
 payload: JSON.stringify({data: data}),
 });
 // Receive and update sheet with new data
 const newData = JSON.parse(response.getContentText());
 sheet.clear();
 newData.forEach(row => sheet.appendRow(row));
}
```

Advanced Sheet Data Analysis

Objective: Implement a script to perform advanced data analysis on a dataset within Google Sheets, such as regression analysis or forecasting.

Explanation: Pushes the boundary of data analysis within Google Sheets using Apps Script, applying statistical methods or predictive modeling directly on sheet data.

Code:

```
function performAdvancedAnalysis() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data
Analysis");
 const values = sheet.getDataRange().getValues();
 // Placeholder for analysis logic, e.g., regression analysis
 const analysisResults = values.map(row => {
 // Perform calculation, e.g., simple linear regression
 return [row[0], row[1] * 2]; // Example transformation
 });
 // Output results to a new sheet
 const resultsSheet =
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Analysis Results");
 analysisResults.forEach(result => resultsSheet.appendRow(result));
}
```

Real-time Collaboration Dashboard

Objective: Create a real-time collaboration dashboard in Google Sheets, displaying user edits, comments, and active viewers.

Explanation: Aims to enhance team collaboration within Google Sheets by providing a real-time overview of user interactions and contributions.

Code:

```
function setupCollaborationDashboard() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const logSheet = ss.getSheetByName("Collaboration Log") ||
ss.insertSheet("Collaboration Log");
 // Setup initial log structure
 logSheet.appendRow(["Timestamp", "User", "Action", "Range", "Value"]);
 // Additional functionality would require setting up triggers for onEdit,
onComment, etc.
}
```

Custom Email Campaign from Sheets

Objective: Utilize Google Sheets to manage and send a custom email campaign, allowing for personalized emails based on sheet data.

Explanation: Demonstrates leveraging Google Sheets as a CRM or marketing tool, sending out customized emails to a list of recipients maintained in a sheet.

Code:

```
function sendCustomEmailCampaign() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Email
Campaign");
 const recipients = sheet.getDataRange().getValues();
 recipients.forEach((recipient, index) => {
 if (index > 0) { // Skip header
 const [email, name, customMessage] = recipient;
 const subject = `Special Offer for ${name}`;
 const body = `Hello ${name},\n\n${customMessage}`;
 MailApp.sendEmail(email, subject, body);
 }
 });
}
```

Automated Sheet Archiving

Objective: Develop a script for automated archiving of old data in a Google Sheet, moving rows older than a certain date to an archive sheet.

Explanation: Aids in data management and organization within Google Sheets, ensuring active sheets remain focused and uncluttered.

Code:

```
function automateSheetArchiving() {
```

```
 const sourceSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Main Data");
 const archiveSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Archive") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Archive");
 const cutoffDate = new Date();
 cutoffDate.setMonth(cutoffDate.getMonth() - 6); // Archive entries older
than 6 months
 const rows = sourceSheet.getDataRange().getValues();
 const rowsToArchive = [];
 rows.forEach((row, index) => {
 const rowDate = new Date(row[0]); // Assuming date is in the first column
 if (rowDate < cutoffDate) {
 rowsToArchive.push(row);
 sourceSheet.deleteRow(index + 1); // Adjust for array starting at 0,
whereas Sheets start at 1
 }
 });
 rowsToArchive.forEach(row => archiveSheet.appendRow(row));
}
```

Dynamic Range Summation

Objective: Create a script to dynamically sum ranges in a Google Sheet based on criteria in adjacent cells, updating a summary cell with the total.

Explanation: Enhances data manipulation capabilities in Sheets, allowing for flexible and dynamic calculations based on changing data or criteria.

Code:

```
function dynamicRangeSummation() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Financials");
 const data = sheet.getDataRange().getValues();
 let total = 0;
 data.forEach(row => {
```

```
 // Assuming criteria is in the second column, and values to sum are in the
third
 if (row[1] === "Eligible") {
 total += row[2];
 }
 });
 sheet.getRange("F1").setValue(total); // Assuming F1 is the summary cell
}
```

Sheet Data Encryption & Decryption

Objective: Implement a script to encrypt sensitive data before storing it in a Google Sheet and decrypt it when accessed.

Explanation: Introduces basic data security practices within Google Sheets, protecting sensitive information through encryption.

Code:

```
const SECRET_KEY = "your-secret-key";
function encryptData() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sensitive
Data");
 const data = sheet.getDataRange().getValues();
 data.forEach((row, rowIndex) => {
 row.forEach((cell, colIndex) => {
 const encryptedText = Utilities.encrypt(Utilities.newBlob(cell.toString()),
SECRET_KEY);
 sheet.getRange(rowIndex + 1, colIndex + 1).setValue(encryptedText);
 });
 });
}
function decryptData() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sensitive
Data");
 const data = sheet.getDataRange().getValues();
```

```
 data.forEach((row, rowIndex) => {
 row.forEach((cell, colIndex) => {
 try {
 const decryptedText = Utilities.newBlob(Utilities.decrypt(cell,
SECRET_KEY)).getDataAsString();
 sheet.getRange(rowIndex + 1, colIndex + 1).setValue(decryptedText);
 } catch(e) {
 // Handle decryption error
 }
 });
 });
}
```

Custom Sheet Data Importer

Objective: Build a script that imports data from multiple external sources into a Google Sheet, transforming and consolidating the data according to specified rules.

Explanation: Showcases the ability to aggregate and process data from various origins, streamlining data collection and preparation for analysis or reporting.

Code:

```
function customDataImporter() {
 const sources = [
 {url: 'https://api.example.com/data1', transform: (data) => data.map(item
=> [item.date, item.value])},
 {url: 'https://api.example2.com/data', transform: (data) =>
data.results.map(item => [item.timestamp, item.amount])}
 ];
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Imported
Data");
 sources.forEach(source => {
 const response = UrlFetchApp.fetch(source.url);
```

```
 const jsonData = JSON.parse(response.getContentText());
 const transformedData = source.transform(jsonData);
 transformedData.forEach(row => sheet.appendRow(row));
 });
}
```

Sheet Change Notification System

Objective: Develop a system that sends notifications (e.g., email, SMS) when specific changes are made to a Google Sheet, such as updates to critical cells.

Explanation: Enhances monitoring of key data points within a spreadsheet, ensuring stakeholders are promptly informed about significant updates.

Code:

```
function setupChangeNotifications() {
 // This example uses email for notifications; integration with SMS APIs
would require additional setup
 const watchCells = ['A1', 'B2']; // Cells to monitor for changes
 const notificationEmail = 'notify@example.com';
 ScriptApp.newTrigger('notifyOnCellChange')
 .forSpreadsheet(SpreadsheetApp.getActiveSpreadsheet())
 .onEdit()
 .create();
}
function notifyOnCellChange(e) {
 const editedRange = e.range.getA1Notation();
 if (watchCells.includes(editedRange)) {
 MailApp.sendEmail(notificationEmail, 'Sheet Update Notification', `Cell
${editedRange} has been updated to: ${e.value}`);
 }
}
```

Advanced Data Cleanup Tool

Objective: Create a sophisticated data cleanup tool in Google Sheets that identifies and corrects common data inconsistencies (e.g., formatting errors, duplicates).

Explanation: Demonstrates complex data validation and correction strategies, crucial for maintaining high-quality data in business or research contexts.

Code:

```
function advancedDataCleanup() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data
Cleanup");
 const data = sheet.getDataRange().getValues();
 const cleanedData = data.map(row => {
 // Example cleanup operations
 const cleanedRow = row.map(cell => {
 if (typeof cell === 'string') {
 // Trim whitespace and correct common formatting issues
 return cell.trim().replace(/\s+/g, ' ');
 }
 return cell;
 });
 return cleanedRow;
 });
 // Remove duplicates
 const uniqueData = cleanedData.filter((row, index, self) =>
 index === self.findIndex((t) => (t.join(',') === row.join(',')))
 );
 sheet.clearContents();
 uniqueData.forEach(row => sheet.appendRow(row));
}
```

Dynamic Content Generator

Objective: Implement a script to generate dynamic content in a Google Sheet based on user inputs, creating customized reports or documents.

Explanation: Explores the creation of interactive tools within Sheets, where user inputs can trigger the generation of tailored content, enhancing user engagement and productivity.

Code:

```
function dynamicContentGenerator() {
 const ui = SpreadsheetApp.getUi();
 const response = ui.prompt('Enter Report Date (YYYY-MM-DD)',
ui.ButtonSet.OK_CANCEL);
 if (response.getSelectedButton() === ui.Button.OK) {
 const date = response.getResponseText();
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Reports");
 const reportData = generateReportDataForDate(date); // Assume this
function generates report data
 sheet.clear();
 sheet.appendRow(["Report for Date", date]);
 reportData.forEach(item => sheet.appendRow([item.metric, item.value]));
 }
}
function generateReportDataForDate(date) {
 // Placeholder for report data generation logic
 return [
 {metric: "Total Sales", value: "$10,000"},
 {metric: "New Customers", value: 25}
 // More data...
 ];
}
```

Multi-Sheet Data Aggregator

Objective: Build a script that aggregates data from multiple sheets into a summary sheet, performing calculations to summarize the data (e.g., totals, averages).

Explanation: Demonstrates handling data across multiple sheets, useful for consolidating information in large spreadsheets with data spread across numerous tabs.

Code:

```
function multiSheetDataAggregator() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const summarySheet = ss.getSheetByName("Summary") ||
ss.insertSheet("Summary");
 summarySheet.clear();
 const sheets = ss.getSheets();
 let aggregatedData = {};
 sheets.forEach(sheet => {
 if (sheet.getName() !== "Summary") {
 const data = sheet.getDataRange().getValues();
 data.forEach((row, index) => {
 if (index > 0) { // Skip headers
 const category = row[0];
 const amount = row[1];
 if (aggregatedData[category]) {
 aggregatedData[category] += amount;
 } else {
 aggregatedData[category] = amount;
 }
 }
 });
 }
 });
 summarySheet.appendRow(["Category", "Total"]);
 Object.keys(aggregatedData).forEach(category => {
 summarySheet.appendRow([category, aggregatedData[category]]);
 });
}
```

Automated Data Validation Reports

Objective: Design a script to automatically validate data within a sheet against specific rules and generate a report detailing any discrepancies found.

Explanation: Facilitates data integrity checks by automating the validation process, crucial for datasets requiring adherence to strict standards or formats.

Code:

```
function automatedDataValidationReports() {
 const dataSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data");
 const reportSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Validation
Report") || SpreadsheetApp.getActiveSpreadsheet().insertSheet("Validation
Report");
 const data = dataSheet.getDataRange().getValues();
 const validationRules = [
 {column: 1, rule: value => value !== "", message: "Name cannot be
empty"},
 {column: 2, rule: value => !isNaN(value) && value > 0, message: "Age
must be a positive number"}
 // Add more rules as needed
 ];
 reportSheet.clear();
 reportSheet.appendRow(["Row", "Issue"]);
 data.forEach((row, rowIndex) => {
 validationRules.forEach(({column, rule, message}) => {
 if (!rule(row[column - 1])) {
 reportSheet.appendRow([rowIndex + 1, `Column ${column}:
${message}`]);
 }
 });
 });
}
```

Dynamic Chart Updates Based on Filters

Objective: Create a script to dynamically update chart data in Google Sheets based on user-applied filters, reflecting the filtered data in real-time.

Explanation: Enhances data visualization by linking charts to dynamically filtered data ranges, providing users with immediate visual feedback on their data exploration.

Code:

```
function dynamicChartUpdates() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const dataSheet = ss.getSheetByName("Sales Data");
 const chartSheet = ss.getSheetByName("Sales Chart");
 const filteredRange = dataSheet.getRange("A1:B" +
dataSheet.getLastRow()); // Assume data is filtered here
 const charts = chartSheet.getCharts();
 if (charts.length > 0) {
 const chart = charts[0];
 chartSheet.updateChart(chart.modify().setRange(filteredRange).build());
 } else {
 // Create a new chart if not present
 const chartBuilder = chartSheet.newChart()
 .setChartType(Charts.ChartType.COLUMN)
 .addRange(filteredRange)
 .setPosition(5, 5, 0, 0);
 chartSheet.insertChart(chartBuilder.build());
 }
}
```

Sheet-to-Sheet Data Sync with Transformation

Objective: Implement a script to synchronize data between two sheets, applying a transformation function to the data before it's copied over.

Explanation: Showcases data synchronization between sheets while allowing for data processing or transformation, useful in workflows requiring data manipulation before use.

Code:

```
function sheetToSheetSyncWithTransformation() {
 const sourceSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Source");
 const targetSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Target");
 const data = sourceSheet.getDataRange().getValues();
 const transformedData = data.map(row => {
 // Apply transformation to each row
 return row.map(cell => typeof cell === 'number' ? cell * 2 : cell); // Example
transformation
 });
 targetSheet.clear();
 transformedData.forEach(row => targetSheet.appendRow(row));
}
```

Implement Custom Sheet Functions

Objective: Develop custom Google Sheets functions using Google Apps Script to perform unique calculations or text manipulations not available in built-in functions.

Explanation: Expands the functionality of Google Sheets with bespoke formulas, tailored to specific analytical or processing needs.

Code:

```
/**
 * Custom function to calculate the Fibonacci number.
 *
 * @param {number} n The position in the Fibonacci sequence.
 * @return The Fibonacci number.
 * @customfunction
```

```
 */
function FIBONACCI(n) {
 function fib(n) {
 return n < 2 ? n : fib(n - 1) + fib(n - 2);
 }
 return fib(n);
}
```

Real-time Collaboration Visualizer

Objective: Design a script to visualize real-time collaboration in a Google Sheet, highlighting cells currently being edited by users.

Explanation: Aims to enhance the collaborative experience by providing visual cues of active participation, fostering awareness among concurrent users.

Code:

```
// Note: Google Apps Script currently does not support triggers or methods for real-time cell edit detection for collaboration visualization.
// This exercise is conceptual and aims to inspire thinking about possible extensions or external integrations.
function realTimeCollaborationVisualizer() {
 // Conceptual implementation
 Logger.log("This is a conceptual exercise. Implementing real-time collaboration visualization would require external integration or API support.");
}
```

Conditional Row Hiding Based on User Input

Objective: Develop a script that dynamically hides rows in a Google Sheet based on user input, such as hiding all rows where a specific column's value does not match the user-provided criterion.

Explanation: Enhances user interaction with large datasets by allowing dynamic filtering of information directly within the Google Sheets interface.

Code:

```
function hideRowsBasedOnInput() {
 const ui = SpreadsheetApp.getUi();
 const response = ui.prompt('Enter the value to filter by:',
ui.ButtonSet.OK_CANCEL);
 if (response.getSelectedButton() === ui.Button.OK) {
 const filterValue = response.getResponseText();
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const rows = sheet.getDataRange().getValues();
 rows.forEach((row, index) => {
 // Assuming the column to check is the first one (index 0)
 if (row[0] !== filterValue) {
 sheet.hideRows(index + 1);
 } else {
 sheet.showRows(index + 1);
 }
 });
 }
}
```

Auto-Generate Slide Presentations from Sheets Data

Objective: Create a script that auto-generates Google Slides presentations based on the data and charts found in a Google Sheet, for periodic reporting purposes.

Explanation: Streamlines the creation of presentation materials, automating the transfer of key data points and visualizations into a slide deck format.

Code:

```
function generateSlidesFromSheetData() {
```

```
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const dataRange = sheet.getDataRange();
 const charts = sheet.getCharts();
 const slides = SlidesApp.create("Sheet Data Presentation");
 slides.appendSlide().insertTextBox("Data Summary");
 dataRange.getValues().forEach((row, index) => {
 const slide =
slides.appendSlide(SlidesApp.PredefinedLayout.TITLE_AND_BODY);
 slide.getShapes()[0].getText().setText(`Row ${index + 1}`);
 slide.getShapes()[1].getText().setText(row.join(", "));
 });
 charts.forEach((chart, index) => {
 const slide =
slides.appendSlide(SlidesApp.PredefinedLayout.TITLE_ONLY);
 slide.getShapes()[0].getText().setText(`Chart ${index + 1}`);
 slide.insertSheetsChart(chart);
 });
}
```

Custom Error Checking Tool

Objective: Implement a custom error checking tool in Google Sheets that scans for common data entry mistakes in specified columns and highlights them for review.

Explanation: Aids in maintaining data quality by providing a tailored mechanism for identifying and correcting errors beyond the built-in data validation features.

Code:

```
function customErrorChecking() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const range = sheet.getDataRange();
 const data = range.getValues();
 // Example: Check for numerical values in a column expected to contain
only text
```

```
 const textColumnIndex = 1; // Assuming the second column should only
contain text
 data.forEach((row, rowIndex) => {
 const cellValue = row[textColumnIndex];
 if (!isNaN(cellValue)) {
 // Highlight cell with error
 sheet.getRange(rowIndex + 1, textColumnIndex +
1).setBackground("yellow");
 }
 });
}
```

Automated Data Segmentation

Objective: Write a script to automatically segment a dataset into multiple sheets based on a key column's value, creating or updating sheets for each unique value found.

Explanation: Facilitates data organization and analysis by dynamically separating a larger dataset into categorized subsets, each within its own sheet.

Code:

```
function automatedDataSegmentation() {
 const sourceSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Master Data");
 const data = sourceSheet.getDataRange().getValues();
 const header = data.shift(); // Remove the header row
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const segmentationColumnIndex = 0; // Assuming the first column
determines segmentation
 const segmentMap = {};
 data.forEach(row => {
 const segmentKey = row[segmentationColumnIndex];
 if (!segmentMap[segmentKey]) {
 segmentMap[segmentKey] = [header]; // Initialize with header
 }
```

```
 segmentMap[segmentKey].push(row);
 });
 Object.keys(segmentMap).forEach(segment => {
 let segmentSheet = ss.getSheetByName(segment);
 if (!segmentSheet) {
 segmentSheet = ss.insertSheet(segment);
 } else {
 segmentSheet.clear(); // Clear existing data
 }
 segmentMap[segment].forEach(row => segmentSheet.appendRow(row));
 });
}
```

Linking Sheets Data with Maps

Objective: Create a script that takes geographic data from a Google Sheet and visualizes it on a Google Map, embedding the map within a Google Sites page or sending it via email.

Explanation: Leverages geographic data for visual analysis and sharing, integrating Google Sheets with Google Maps and other Google services for enhanced data presentation.

Code:

```
// Note: This exercise outlines a conceptual approach due to the complexity of integrating multiple services.
function visualizeDataOnMap() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Locations");
 const locations = sheet.getDataRange().getValues();
 let mapUrl = "https://www.google.com/maps/d/u/0/edit?mid=YOUR_MAP_ID&ll=YOUR_DEFAULT_LAT_LNG&z=6";
 locations.forEach((location, index) => {
 if (index > 0) { // Skip header
 const [name, latitude, longitude] = location;
```

```
 // Append each location as a marker on the map URL (simplified for
illustration)
 mapUrl +=
`&markers=color:red%7Clabel:${index}%7C${latitude},${longitude}`;
 }
 });
 // Example: Embed map URL in a Google Sites page or send via email
 Logger.log(mapUrl);
}
```

Dynamic Team Roster from Directory

Objective: Automatically generate a team roster in a Google Sheet based on user profiles in a Google Workspace Directory, updating it as users are added or removed.

Explanation: Utilizes Google Workspace Directory information to maintain an up-to-date team roster, reducing manual management of team member details.

Code:

```
// Note: Requires Google Workspace Admin privileges to access Directory
API
function updateTeamRosterFromDirectory() {
 const directoryUsers = AdminDirectory.Users.list({domain:
'example.com'}).users;
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Team
Roster");
 sheet.clear();
 sheet.appendRow(["Name", "Email", "Title"]);
 directoryUsers.forEach(user => {
 sheet.appendRow([user.name.fullName, user.primaryEmail,
user.organizations ? user.organizations[0].title : ""]);
 });
}
```

Auto-Update Calendar Events from Sheet

Objective: Synchronize Google Calendar events with a schedule maintained in a Google Sheet, automatically creating, updating, or removing events as the sheet changes.

Explanation: Streamlines event management by directly linking a Google Sheet schedule with Google Calendar, ensuring calendar events reflect the most current plans.

Code:

```
function syncCalendarFromSheet() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Event
Schedule");
 const events = sheet.getDataRange().getValues();
 const calendar = CalendarApp.getDefaultCalendar();
 events.forEach((event, index) => {
 if (index === 0) return; // Skip header
 const [title, description, startDate, endDate] = event;
 const existingEvents = calendar.getEvents(new Date(startDate), new
Date(endDate), {search: title});
 if (existingEvents.length === 0) {
 calendar.createEvent(title, new Date(startDate), new Date(endDate),
{description});
 } else {
 // Optionally update existing events or handle duplicates
 }
 });
}
```

Sheets Data Anonymization

Objective: Implement a script that anonymizes sensitive information in a Google Sheet, replacing identifiable data with pseudonyms or generic identifiers.

Explanation: Essential for protecting privacy and complying with data protection regulations, this script modifies personal data to prevent identification of individuals.

Code:

```
function anonymizeSheetData() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sensitive
Data");
 const dataRange = sheet.getDataRange();
 const data = dataRange.getValues();
 const anonymizedData = data.map((row, index) => {
 if (index > 0) { // Skip header
 // Example: Anonymize email address and name (columns B and C)
 row[1] = `User${index}@example.com`; // Anonymized email
 row[2] = `User ${index}`; // Anonymized name
 }
 return row;
 });
 dataRange.setValues(anonymizedData);
}
```

Enhance Sheet with AI-generated Content

Objective: Utilize an external AI text generation API to enrich a Google Sheet with AI-generated content based on keywords or topics listed in the sheet.

Explanation: Explores the integration of AI services with Google Sheets, enabling automatic generation of creative content or summaries for listed topics.

Code:

```
function generateAiContent() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Topics");
 const topics = sheet.getDataRange().getValues();
 const apiKey = "YOUR_AI_SERVICE_API_KEY";
 const apiUrl = "https://api.example-ai-service.com/generate";
 topics.forEach((topic, index) => {
 if (index > 0) { // Skip header
 const response = UrlFetchApp.fetch(apiUrl, {
 method: "post",
 contentType: "application/json",
 payload: JSON.stringify({prompt: topic[0], length: 100}), // Example
payload
 headers: {Authorization: `Bearer ${apiKey}`}
 });
 const content = JSON.parse(response.getContentText()).generatedText;
 sheet.getRange(index + 1, 2).setValue(content); // Assuming the AI
content goes in the second column
 }
 });
}
```

Custom Feedback Analysis Dashboard

Objective: Create a script that compiles and analyzes customer feedback stored in a Google Sheet, generating a dashboard with insights such as sentiment scores and common themes.

Explanation: Demonstrates the use of Google Sheets as a platform for aggregating and interpreting feedback, employing text analysis techniques to extract actionable insights.

Code:

```
function createFeedbackAnalysisDashboard() {
```

```
 const feedbackSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Customer
Feedback");
 const dashboardSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Dashboard") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Dashboard");
 const feedbackData = feedbackSheet.getDataRange().getValues();
 // Placeholder for sentiment analysis and theme extraction logic
 // Example: Aggregate feedback by theme and calculate average
sentiment score for each
 dashboardSheet.clear();
 dashboardSheet.appendRow(["Theme", "Average Sentiment Score",
"Feedback Count"]);
 // Example dashboard data
 const themes = {
 "Product Quality": {score: 4.2, count: 25},
 "Customer Service": {score: 3.8, count: 40},
 // Add more themes and scores
 };
 Object.keys(themes).forEach(theme => {
 const {score, count} = themes[theme];
 dashboardSheet.appendRow([theme, score, count]);
 });
}
```

Automated Sheet Translation

Objective: Develop a script that translates text in specified columns of a Google Sheet into a different language using Google's Translation service, storing the translated text in adjacent columns.

Explanation: Showcases the integration of Google Cloud Translation API with Sheets for automating the localization of content within spreadsheets.

Code:

```
function translateSheetContent() {
```

```
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Localization");
 const range = sheet.getDataRange();
 const values = range.getValues();
 const targetLanguage = 'fr'; // Target language code (French in this
example)
 values.forEach((row, index) => {
 const textToTranslate = row[0]; // Assuming text to translate is in the first
column
 const translation = LanguageApp.translate(textToTranslate, '',
targetLanguage);
 sheet.getRange(index + 1, 2).setValue(translation); // Storing translation in
the second column
 });
}
```

Generate Conditional Email Reports

Objective: Create a script that sends email reports based on conditional logic applied to data within a Google Sheet, such as sending weekly sales performance summaries if targets are not met.

Explanation: Utilizes Google Sheets data to drive conditional reporting via email, enhancing communication and responsiveness based on data-driven triggers.

Code:

```
function sendConditionalEmailReports() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sales
Summary");
 const data = sheet.getDataRange().getValues();
 const salesTarget = 10000; // Example sales target
 data.forEach((row, index) => {
 if (index === 0) return; // Skip header row
 const [week, sales] = row;
 if (sales < salesTarget) {
```

```
 MailApp.sendEmail({
 to: "salesmanager@example.com",
 subject: `Weekly Sales Report - Week ${week}`,
 body: `Sales target not met: $${sales} out of $${salesTarget}.`
 });
 }
 });
}
```

Synchronize Contacts with Sheets

Objective: Automate the synchronization of Google Contacts with a Google Sheet, updating the sheet whenever a new contact is added or existing contacts are modified.

Explanation: Enhances contact management by keeping a Google Sheet in sync with Google Contacts, providing a backup and an easy-to-access view of contact information.

Code:

```
function syncContactsToSheet() {
 const contacts = ContactsApp.getContacts();
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Contacts
Sync");
 sheet.clear();
 sheet.appendRow(["Name", "Email", "Phone Number"]);
 contacts.forEach(contact => {
 const name = contact.getFullName();
 const emails = contact.getEmails();
 const phones = contact.getPhones();
 const email = emails.length > 0 ? emails[0].getAddress() : "";
 const phone = phones.length > 0 ? phones[0].getPhoneNumber() : "";
 sheet.appendRow([name, email, phone]);
 });
}
```

Auto-generate Forms from Sheet Data

Objective: Build a script that automatically generates Google Forms based on the structure and content specified in a Google Sheet, creating a dynamic way to create surveys or quizzes.

Explanation: Demonstrates dynamic creation of Google Forms for surveys, quizzes, or feedback collection, based on configurations or content outlined in a Google Sheet.

Code:

```
function generateFormsFromSheet() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Form
Configurations");
 const data = sheet.getDataRange().getValues();
 const form = FormApp.create('New Dynamic Form');
 data.forEach((row, index) => {
 if (index === 0) return; // Skip header row
 const [question, option1, option2, option3] = row;
 const item = form.addMultipleChoiceQuestion().setTitle(question);
 item.setChoiceValues([option1, option2, option3]);
 });
}
```

Dynamic Resource Allocation

Objective: Implement a script for dynamic resource allocation in a Google Sheet, adjusting assignments based on availability and workload captured within the sheet.

Explanation: Facilitates efficient resource management by automating the assignment process based on current workload and resource availability, optimizing distribution of tasks.

Code:

```
function dynamicResourceAllocation() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Resource
Allocation");
 const resources = sheet.getRange("A2:C10").getValues(); // A: Resource
Name, B: Availability, C: Current Workload
 resources.forEach((resource, index) => {
 let [name, availability, workload] = resource;
 if (availability > workload) {
 // Logic to assign new tasks based on availability
 const newWorkload = workload + 1; // Example increment
 sheet.getRange(index + 2, 3).setValue(newWorkload); // Update workload
in the sheet
 }
 });
}
```

Custom Inventory Management

Objective: Create a script to manage inventory levels in a Google Sheet, automatically updating stock quantities based on sales data entries and alerting when restocking is needed.

Explanation: Streamlines inventory tracking and management directly within Google Sheets, providing real-time insights into stock levels and restock requirements.

Code:

```
function manageInventory() {
 const inventorySheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Inventory");
 const salesSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sales");
 const salesData = salesSheet.getDataRange().getValues();
 salesData.forEach((sale, index) => {
 if (index === 0) return; // Skip header row
 const [productID, quantitySold] = sale;
```

```
 const inventoryRange = inventorySheet.getDataRange();
 const inventoryData = inventoryRange.getValues();
 const productRow = inventoryData.findIndex(row => row[0] ===
productID);
 if (productRow !== -1) {
 const currentStock = inventoryData[productRow][2];
 const updatedStock = currentStock - quantitySold;
 inventorySheet.getRange(productRow + 1, 3).setValue(updatedStock);
 // Alert for restocking
 if (updatedStock < 5) { // Example threshold
 MailApp.sendEmail("inventorymanager@example.com", "Restock Alert",
`Product ID ${productID} needs restocking.`);
 }
 }
 });
}
```

Automated Meeting Minutes

Objective: Develop a script that compiles meeting minutes into a Google Sheet from notes taken in Google Docs during meetings, categorized by date and topic.

Explanation: Automates the organization of meeting notes, centralizing the information in a Google Sheet for easy access and review.

Code:

```
function compileMeetingMinutes() {
 const minutesFolderId = "YOUR_FOLDER_ID_HERE";
 const folder = DriveApp.getFolderById(minutesFolderId);
 const files = folder.getFiles();
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Meeting
Minutes");
 while (files.hasNext()) {
 const file = files.next();
 const doc = DocumentApp.openById(file.getId());
```

```
 const body = doc.getBody().getText();
 const [date, topic] = doc.getName().split(" - "); // Assuming naming
convention "Date - Topic"
 sheet.appendRow([date, topic, body]);
 }
}
```

Real-time Stock Market Dashboard

Objective: Create a script to fetch real-time stock market data using a finance API, displaying the information in a Google Sheet as a dashboard with automatic updates.

Explanation: Leverages external financial data sources to build a real-time stock market dashboard within Google Sheets, providing instant access to market movements.

Code:

```
function updateStockMarketDashboard() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Stock
Dashboard");
 const stocks = ["AAPL", "GOOGL", "MSFT"]; // Example stock symbols
 const apiKey = "YOUR_FINANCE_API_KEY";
 const apiUrl = "https://api.examplefinance.com/stock/";
 stocks.forEach(symbol => {
 const response =
UrlFetchApp.fetch(`${apiUrl}${symbol}?apiKey=${apiKey}`);
 const data = JSON.parse(response.getContentText());
 const {price, change} = data;
 const row = sheet.createTextFinder(symbol).findNext().getRow();
 sheet.getRange(row, 2).setValue(price);
 sheet.getRange(row, 3).setValue(change);
 });
}
```

Sheet-based Project Time Tracking

Objective: Implement a time tracking system in a Google Sheet for project management, enabling start and stop timestamps for tasks and calculating total time spent.

Explanation: Provides a simple yet effective way to track time spent on various projects or tasks directly within Google Sheets, facilitating project time management.

Code:

```
function startTimeTracking(taskId) {
 const startTime = new Date();
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Time
Tracking");
 // Check if task is already being tracked
 const existingRow = sheet.createTextFinder(taskId).findNext();
 if (existingRow) {
 sheet.getRange(existingRow.getRow(), 3).setValue(startTime); //
Assuming start time is in column C
 } else {
 sheet.appendRow([taskId, "", startTime]);
 }
}
function stopTimeTracking(taskId) {
 const stopTime = new Date();
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Time
Tracking");
 const row = sheet.createTextFinder(taskId).findNext().getRow();
 const startTime = new Date(sheet.getRange(row, 3).getValue()); //
Assuming start time is in column C
 const duration = (stopTime - startTime) / (1000 * 60 * 60); // Duration in
hours
 sheet.getRange(row, 2).setValue(stopTime); // Assuming stop time is in
column B
 sheet.getRange(row, 4).setValue(duration); // Assuming total duration is in
column D
```

}

Personal Finance Dashboard

Objective: Build a personal finance dashboard in Google Sheets using Apps Script, aggregating data from various accounts and categories to provide insights into spending and savings.

Explanation: Utilizes Google Sheets as a powerful tool for personal finance management, automatically updating financial data to help track and analyze personal spending and saving habits.

Code:

```
function updatePersonalFinanceDashboard() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Finance
Dashboard");
 const accounts = ["Checking", "Savings", "Credit Card"]; // Example
accounts
 const categories = ["Rent", "Groceries", "Utilities"]; // Example expense
categories
 // Placeholder for fetching financial data from external sources or other
sheets
 const financialData = {
 "Checking": 5000,
 "Savings": 15000,
 "Credit Card": -2000,
 "Expenses": {
 "Rent": 1200,
 "Groceries": 300,
 "Utilities": 150
 }
 };
 accounts.forEach((account, index) => {
 const balance = financialData[account];
 sheet.getRange(2, index + 2).setValue(balance); // Assuming account
balances start on row 2
```

```
 });
 categories.forEach((category, index) => {
 const expense = financialData.Expenses[category];
 sheet.getRange(index + 4, 2).setValue(expense); // Assuming expenses
start on row 4, column 2
 });
 // Additional logic for calculating totals, averages, or other insights
}
```

Automated Document Assembly from Sheet Data

Objective: Write a script to automatically assemble documents in Google Docs based on structured data from a Google Sheet, including text and images, for reports or contracts.

Explanation: Demonstrates integrating Google Sheets with Google Docs to dynamically create comprehensive documents, automating routine reporting or contract generation tasks.

Code:

```
function assembleDocuments() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Report Data");
 const rows = sheet.getDataRange().getValues();
 rows.forEach((row, index) => {
 if (index === 0) return; // Skip header row
 const [title, description, imageUrl] = row;
 const doc = DocumentApp.create(title);
 const body = doc.getBody();

body.appendParagraph(title).setHeading(DocumentApp.ParagraphHeading
.HEADING1);
 body.appendParagraph(description);
 if (imageUrl) body.appendImage(UrlFetchApp.fetch(imageUrl).getBlob());
```

```
 Logger.log(`Document created: ${doc.getUrl()}`);
 });
}
```

Real-time Data Dashboard in Sheets

Objective: Create a real-time data dashboard in Google Sheets that automatically updates with live data from an external API, such as stock market prices or weather conditions.

Explanation: Explores fetching and displaying live data within Google Sheets, enabling real-time monitoring and analysis directly from a spreadsheet.

Code:

```
function updateRealTimeDashboard() {
 const apiUrl = "https://api.example.com/live-data";
 const response = UrlFetchApp.fetch(apiUrl);
 const jsonData = JSON.parse(response.getContentText());
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Live
Dashboard");
 sheet.getRange("A2").setValue(new Date()); // Timestamp
 sheet.getRange("B2").setValue(jsonData.stockPrice);
 sheet.getRange("C2").setValue(jsonData.weatherCondition);
 // Add more data fields as needed
}
```

Sheet-based Workflow Automation

Objective: Automate a multi-step workflow in Google Sheets, where the completion of one task triggers the next step, including notifications and task assignments.

Explanation: Leverages Google Sheets as a central platform for managing and automating workflows, coordinating tasks, and communications within teams or projects.

Code:

```
function automateWorkflow() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Workflow
Steps");
 const steps = sheet.getDataRange().getValues();
 steps.forEach((step, index) => {
 if (index === 0 || step[2] !== "Completed") return; // Skip header and
incomplete steps
 const nextStep = steps[index + 1];
 if (nextStep && nextStep[2] === "Pending") {
 // Notify assignee of the next step
 MailApp.sendEmail(nextStep[1], "Workflow Task Assignment", `You are
assigned to: ${nextStep[0]}`);
 sheet.getRange(index + 2, 3).setValue("In Progress"); // Update status of
the next step
 return;
 }
 });
}
```

Automated Client Reporting System

Objective: Develop a system within Google Sheets that generates and emails customized client reports at scheduled intervals, pulling data from various sheets.

Explanation: Automates client communication by generating personalized reports based on up-to-date data, enhancing client engagement and service delivery.

Code:

```
function sendClientReports() {
 const clientSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Clients");
 const clients = clientSheet.getDataRange().getValues();
```

```
 clients.forEach((client, index) => {
 if (index === 0) return; // Skip header row
 const [clientEmail, reportType] = client;
 const reportData = generateReportData(reportType); // Assume a function
that generates report data based on type
 const reportDoc = DocumentApp.create(`Report for ${clientEmail}`);
 const body = reportDoc.getBody();
 body.appendParagraph(`Report Type: ${reportType}`);
 reportData.forEach(data => body.appendParagraph(data));
 const pdfBlob = reportDoc.getAs(MimeType.PDF);
 MailApp.sendEmail(clientEmail, "Your Custom Report", "Please find
attached your requested report.", {attachments: [pdfBlob]});
 DriveApp.getFileById(reportDoc.getId()).setTrashed(true); // Clean up by
deleting the temporary document
 });
}
```

Dynamic Event Scheduler in Sheets

Objective: Implement a dynamic event scheduler in Google Sheets that allows users to input events, automatically avoiding scheduling conflicts and optimizing event times.

Explanation: Provides a solution for managing events or appointments within Google Sheets, including logic to prevent overlaps and suggest optimal scheduling.

Code:

```
function dynamicEventScheduler() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Event
Schedule");
 const events = sheet.getDataRange().getValues();
 const calendar = CalendarApp.getDefaultCalendar();
 events.forEach((event, index) => {
 if (index === 0) return; // Skip header row
 const [eventName, eventDate] = event;
```

```
 const dayEvents = calendar.getEventsForDay(new Date(eventDate));
 let eventAdded = false;
 for (let hour = 9; hour <= 17 && !eventAdded; hour++) { // Business hours:
9 AM to 5 PM
 const startTime = new Date(eventDate);
 startTime.setHours(hour);
 const endTime = new Date(startTime);
 endTime.setHours(startTime.getHours() + 1); // 1-hour long events
 if (!dayEvents.some(e => e.getStartTime() <= startTime &&
e.getEndTime() > startTime)) {
 calendar.createEvent(eventName, startTime, endTime);
 eventAdded = true;
 sheet.getRange(index + 1, 3).setValue(`Scheduled at ${hour}:00`); //
Indicate scheduled time
 }
 }
 if (!eventAdded) {
 sheet.getRange(index + 1, 3).setValue("Could not schedule");
 }
 });
}
```

Inventory Level Alerts

Objective: Set up a script in Google Sheets to monitor inventory levels and automatically send alerts via email when stock for any item falls below a predefined threshold.

Explanation: Ensures timely replenishment of inventory by proactively monitoring stock levels and alerting responsible parties to prevent stockouts.

Code:

```
function inventoryLevelAlerts() {
 const inventorySheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Inventory");
 const items = inventorySheet.getDataRange().getValues();
```

```
 items.forEach((item, index) => {
 if (index === 0) return; // Skip header row
 const [itemName, stockLevel] = item;
 const threshold = 10; // Example threshold for stock level
 if (stockLevel < threshold) {
 MailApp.sendEmail("inventory@example.com", "Inventory Alert", `Stock
for ${itemName} is below threshold at ${stockLevel} units.`);
 }
 });
}
```

Collaborative Document Editing Tracker

Objective: Design a system within Google Sheets for tracking edits made to a collection of Google Docs, logging each edit with details like document name, editor, and timestamp.

Explanation: Facilitates document management and review in collaborative environments by tracking modifications across multiple documents within a centralized Google Sheet.

Code:

```
function trackDocumentEdits() {
 // Note: Google Docs doesn't directly support triggering scripts on edits.
 // This example outlines a conceptual approach for periodic checks or
manual triggers.
 const docsFolder = DriveApp.getFolderById("YOUR_FOLDER_ID");
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Edit Log");
 const files = docsFolder.getFiles();
 while (files.hasNext()) {
 const file = files.next();
 const doc = DocumentApp.openById(file.getId());
 const docName = doc.getName();
 const revisionHistory = Docs.Documents.get(file.getId()).revisionId; //
Requires Docs API enabled
 revisionHistory.forEach(revision => {
```

```
 const editor = revision.lastModifyingUser.emailAddress;
 const timestamp = revision.modifiedTime;
 sheet.appendRow([docName, editor, timestamp]);
 });
 }
}
```

Automated Survey Analysis

Objective: Automate the analysis of survey data collected in a Google Sheet, calculating summary statistics and generating insights, such as top preferences or trends.

Explanation: Streamlines the processing of survey results, providing quick and actionable insights from raw response data.

Code:

```
function analyzeSurveyData() {
 const surveySheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Survey
Responses");
 const responses = surveySheet.getDataRange().getValues();
 const analysisSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Survey
Analysis") || SpreadsheetApp.getActiveSpreadsheet().insertSheet("Survey
Analysis");
 // Example analysis: Count responses per option
 const responseCounts = {};
 responses.forEach((response, index) => {
 if (index === 0) return; // Skip header
 response.forEach(option => {
 responseCounts[option] = (responseCounts[option] || 0) + 1;
 });
 });
 analysisSheet.clear();
 Object.keys(responseCounts).forEach(option => {
 analysisSheet.appendRow([option, responseCounts[option]]);
```

```
 });
}
```

Dynamic Gantt Chart Generator

Objective: Create a script to generate dynamic Gantt charts in Google Sheets based on project timelines, automatically adjusting as project dates or milestones change.

Explanation: Enhances project management capabilities within Google Sheets by visualizing project timelines and dependencies through automatically updated Gantt charts.

Code:

```
function generateDynamicGanttChart() {
 const projectsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Projects");
 const chartSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Gantt Chart")
|| SpreadsheetApp.getActiveSpreadsheet().insertSheet("Gantt Chart");
 const projects = projectsSheet.getDataRange().getValues();
 // Assuming projects data includes: Project Name, Start Date, End Date
 // This example outlines setting up data for a chart; specific Gantt chart
setup requires third-party tools or complex chart customization
 projects.forEach((project, index) => {
 if (index === 0) {
 chartSheet.appendRow(["Project Name", "Start Date", "Days"]);
 } else {
 const [name, startDate, endDate] = project;
 const start = new Date(startDate);
 const end = new Date(endDate);
 const duration = (end - start) / (1000 * 60 * 60 * 24); // Duration in days
 chartSheet.appendRow([name, start, duration]);
 }
 });
 // Additional steps to customize the chart visualization based on sheet data
}
```

Sheet-Based Email Campaign Manager

Objective: Develop a script for managing email campaigns directly from a Google Sheet, including scheduling, recipient list management, and performance tracking.

Explanation: Utilizes Google Sheets as a comprehensive platform for email campaign management, streamlining the process from planning to analysis.

Code:

```
function manageEmailCampaigns() {
 const campaignSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Email
Campaigns");
 const campaigns = campaignSheet.getDataRange().getValues();
 campaigns.forEach((campaign, index) => {
 if (index === 0) return; // Skip header row
 const [campaignName, recipientEmail, emailSubject, emailBody,
scheduledDate] = campaign;
 const currentDate = new Date();
 const scheduled = new Date(scheduledDate);
 if (currentDate >= scheduled) {
 MailApp.sendEmail(recipientEmail, emailSubject, emailBody);
 campaignSheet.getRange(index + 1, 6).setValue("Sent"); // Mark as sent in
column F
 }
 });
 // Further enhancements could include tracking opens, clicks, and other
metrics if integrated with an email service that supports these features.
}
```

Consolidate Multiple Sheets into One Master Sheet

Objective: Develop a script to automatically consolidate data from multiple sheets within a spreadsheet into a single master sheet, including handling duplicates and maintaining data integrity.

Explanation: Demonstrates how to programmatically merge data from various sources within Google Sheets, useful for creating comprehensive reports or dashboards from fragmented data sets.

Code:

```
function consolidateSheetsIntoMaster() {
 const ss = SpreadsheetApp.getActiveSpreadsheet();
 const allSheets = ss.getSheets();
 const masterSheet = ss.getSheetByName("Master Sheet") ||
ss.insertSheet("Master Sheet");
 masterSheet.clear();
 allSheets.forEach(sheet => {
 if (sheet.getName() !== "Master Sheet") {
 const data = sheet.getDataRange().getValues();
 data.forEach(row => {
 if (!row.every(cell => cell === "")) { // Skip empty rows
 masterSheet.appendRow(row);
 }
 });
 }
 });
}
```

Auto-Generate Calendar Events from Sheet Entries

Objective: Create a script that reads entries from a Google Sheet and automatically creates corresponding events in Google Calendar, including handling event updates and cancellations.

Explanation: Automates the scheduling process by syncing calendar events with data entered in a Google Sheet, streamlining event management and scheduling tasks.

Code:

```
function generateCalendarEventsFromSheet() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Events");
 const events = sheet.getDataRange().getValues();
 const calendar = CalendarApp.getDefaultCalendar();
 events.forEach((event, index) => {
 if (index > 0) { // Skip header row
 const [title, description, startDate, endDate, eventId] = event;
 if (eventId) {
 try {
 const existingEvent = calendar.getEventById(eventId);
 existingEvent.setTitle(title);
 existingEvent.setDescription(description);
 existingEvent.setTime(new Date(startDate), new Date(endDate));
 } catch (e) {
 // Event ID might be invalid or deleted; handle as needed
 }
 } else {
 const newEvent = calendar.createEvent(title, new Date(startDate), new
Date(endDate), {description});
 sheet.getRange(index + 1, 6).setValue(newEvent.getId()); // Store new
event ID in column F
 }
 }
 });
}
```

Dynamic Survey Result Analysis

Objective: Implement a script to dynamically analyze survey results stored in a Google Sheet, generating insights such as average scores, top choices, and trends over time.

Explanation: Enables advanced analysis of survey data, facilitating the extraction of meaningful insights and trends directly within Google Sheets.

Code:

```
function analyzeSurveyResults() {
 const surveySheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Survey
Results");
 const analysisSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Analysis") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Analysis");
 const responses = surveySheet.getDataRange().getValues();
 analysisSheet.clear();
 const summary = {}; // Example structure: {question1: {choice1: count,
choice2: count}, question2: {...}}
 responses.forEach((response, index) => {
 if (index === 0) return; // Skip header row
 response.forEach((answer, i) => {
 const question = responses[0][i]; // Get question from header
 summary[question] = summary[question] || {};
 summary[question][answer] = (summary[question][answer] || 0) + 1;
 });
 });
 Object.keys(summary).forEach((question, index) => {
 analysisSheet.appendRow([`Question: ${question}`]);
 Object.keys(summary[question]).forEach(choice => {
 analysisSheet.appendRow([choice, summary[question][choice]]);
 });
 analysisSheet.appendRow([""]); // Add a blank row for spacing
 });
}
```

Automated Invoice Generation and Tracking

Objective: Build a script for automated generation of invoices based on Google Sheets data, including tracking payments and sending reminders for overdue invoices.

Explanation: Streamlines the invoicing process, leveraging Google Sheets for invoice creation, management, and follow-up, ensuring timely payments and financial tracking.

Code:

```
function generateAndTrackInvoices() {
 const invoiceSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Invoices");
 const invoices = invoiceSheet.getDataRange().getValues();
 invoices.forEach((invoice, index) => {
 if (index === 0 || invoice[4] !== "") return; // Skip header and already
processed invoices
 const [customerName, amountDue, dueDate, invoiceId] = invoice;
 // Assume a function to generate PDF invoice (not shown)
 generatePdfInvoice(invoiceId, customerName, amountDue, dueDate);
 const today = new Date();
 if (new Date(dueDate) < today) {
 // Assume a function to send email reminder (not shown)
 sendInvoiceReminder(customerName, invoiceId);
 }
 });
}
```

Sheet-Based Project Cost Tracker

Objective: Develop a script to track project costs within a Google Sheet, categorizing expenses, calculating totals, and comparing against budgets.

Explanation: Facilitates efficient management of project finances, using Google Sheets to monitor expenses, track budget adherence, and highlight cost overruns.

Code:

```
function trackProjectCosts() {
 const expenseSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Expenses");
 const budgetSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Budgets");
 const expenses = expenseSheet.getDataRange().getValues();
 const budgets = budgetSheet.getDataRange().getValues();
 const budgetMap = {}; // Key: project, Value: budget amount
 budgets.forEach((budget, index) => {
 if (index > 0) { // Skip header row
 const [project, amount] = budget;
 budgetMap[project] = amount;
 }
 });
 const expenseSummary = {}; // Key: project, Value: total expense
 expenses.forEach((expense, index) => {
 if (index > 0) { // Skip header row
 const [project, amount] = expense;
 expenseSummary[project] = (expenseSummary[project] || 0) + amount;
 }
 });
 // Compare expenses against budgets and log overruns
 Object.keys(expenseSummary).forEach(project => {
 if (expenseSummary[project] > budgetMap[project]) {
 Logger.log(`Budget overrun in project ${project}: Budget -
${budgetMap[project]}, Expenses - ${expenseSummary[project]}`);
 }
 });
}
```

Client Portfolio Management in Sheets

Objective: Create a system within Google Sheets to manage a portfolio of clients, including tracking interactions, project statuses, and financial transactions.

Explanation: Utilizes Google Sheets as a CRM tool, enabling efficient management of client information, project tracking, and financial oversight.

Code:

```
function manageClientPortfolio() {
 const clientSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Clients");
 const interactionSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Interactions");
 const clients = clientSheet.getDataRange().getValues();
 clients.forEach((client, index) => {
 if (index === 0) return; // Skip header row
 const [clientId, clientName] = client;
 // Logic to track interactions (calls, emails, meetings)
 const interactions = getInteractionsForClient(clientId); // Assume this
function fetches interaction data
 interactions.forEach(interaction => {
 interactionSheet.appendRow([clientName, interaction.type,
interaction.date, interaction.notes]);
 });
 // Additional logic for project status and financial transactions
 });
}
```

Dynamic Resource Scheduling System

Objective: Implement a dynamic resource scheduling system in Google Sheets, allocating resources to projects based on availability, project needs, and priority levels.

Explanation: Enhances resource management efficiency by using Google Sheets to dynamically allocate and optimize resource assignments across various projects.

Code:

```
function dynamicResourceScheduling() {
 const resourceSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Resources");
```

```
 const projectSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Projects");
 const allocationSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Allocations");
 const resources = resourceSheet.getDataRange().getValues();
 const projects = projectSheet.getDataRange().getValues();
 projects.forEach((project, index) => {
 if (index === 0) return; // Skip header row
 const [projectId, projectName, priority, resourceNeeds] = project;
 resources.forEach(resource => {
 const [resourceId, resourceName, availability] = resource;
 if (availability === "Available") {
 // Example allocation logic based on priority and needs
 allocationSheet.appendRow([projectId, projectName, resourceId,
resourceName]);
 // Update resource availability
 updateResourceAvailability(resourceId, "Allocated");
 }
 });
 });
}
```

Financial Scenario Planning Tool

Objective: Develop a financial scenario planning tool in Google Sheets that models various financial outcomes based on adjustable assumptions and inputs.

Explanation: Enables financial planning and analysis within Google Sheets, providing a flexible tool for forecasting and scenario analysis based on user-defined parameters.

Code:

```
function financialScenarioPlanning() {
 const assumptionsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Assumptions")
;
```

```
 const scenariosSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Scenarios");
 const assumptions = assumptionsSheet.getDataRange().getValues();
 const baseScenario = { revenueGrowthRate: 0.1, costGrowthRate: 0.05,
initialRevenue: 100000, initialCost: 50000 };
 scenariosSheet.clear();
 assumptions.forEach((assumption, index) => {
 if (index === 0) return; // Skip header row
 const [scenarioName, revenueGrowthRate, costGrowthRate] =
assumption;
 const revenue = baseScenario.initialRevenue * (1 + revenueGrowthRate);
 const cost = baseScenario.initialCost * (1 + costGrowthRate);
 const profit = revenue - cost;
 scenariosSheet.appendRow([scenarioName, revenue, cost, profit]);
 });
}
```

Custom Task Prioritization Matrix

Objective: Create a script to generate a custom task prioritization matrix within Google Sheets, classifying tasks based on urgency and importance criteria.

Explanation: Facilitates effective task management by categorizing and prioritizing tasks within a Google Sheet, aiding in decision-making and productivity enhancement.

Code:

```
function generateTaskPrioritizationMatrix() {
 const tasksSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Tasks");
 const matrixSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Prioritization
Matrix") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Prioritization
Matrix");
 const tasks = tasksSheet.getDataRange().getValues();
```

```
 matrixSheet.clear();
 matrixSheet.appendRow(["Task", "Urgency", "Importance", "Category"]);
 tasks.forEach((task, index) => {
 if (index === 0) return; // Skip header row
 const [taskName, urgency, importance] = task;
 let category = "";
 if (urgency >= 8 && importance >= 8) category = "Do First";
 else if (urgency < 8 && importance >= 8) category = "Schedule";
 else if (urgency >= 8 && importance < 8) category = "Delegate";
 else category = "Don't Do";
 matrixSheet.appendRow([taskName, urgency, importance, category]);
 });
}
```

Automated Asset Tracking System

Objective: Implement an automated asset tracking system in Google Sheets, monitoring the status, location, and usage of various assets, with alerts for maintenance or replacement.

Explanation: Streamlines asset management by utilizing Google Sheets to track and manage assets efficiently, ensuring timely maintenance and reducing operational risks.

Code:

```
function automatedAssetTracking() {
 const assetSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Assets");
 const assets = assetSheet.getDataRange().getValues();
 assets.forEach((asset, index) => {
 if (index === 0) return; // Skip header row
 const [assetId, assetName, status, location, lastMaintenanceDate] = asset;
 const today = new Date();
 const maintenanceDue = new Date(lastMaintenanceDate);
 maintenanceDue.setFullYear(maintenanceDue.getFullYear() + 1); //
Example: annual maintenance
 if (status === "In Use" && today >= maintenanceDue) {
```

```
 // Send maintenance alert
 MailApp.sendEmail("maintenance@example.com", "Maintenance Alert",
`Asset ${assetName} (ID: ${assetId}) located at ${location} is due for
maintenance.`);
 }
 });
}
```

Time-Driven Data Refresh

Objective: Create a script that automatically refreshes external data in a Google Sheet at scheduled intervals, ensuring the data remains current without manual updates.

Explanation: Utilizes Apps Script's time-driven triggers to periodically fetch and update data from external sources, keeping the spreadsheet's information up-to-date for analysis or reporting.

Code:

```
function setupAutomaticDataRefresh() {
 ScriptApp.newTrigger('refreshExternalData')
 .timeBased()
 .everyHours(1) // Adjust interval as needed
 .create();
}
function refreshExternalData() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("External
Data");
 const apiUrl = "https://api.example.com/data";
 const response = UrlFetchApp.fetch(apiUrl);
 const jsonData = JSON.parse(response.getContentText());
 // Assuming jsonData is an array of objects
 sheet.clear();
 jsonData.forEach((item, index) => {
 if (index === 0) {
 // Add headers based on object keys
```

```
 sheet.appendRow(Object.keys(item));
 }
 // Add data rows
 sheet.appendRow(Object.values(item));
 });
}
```

Automated Email Digest Based on Sheet Data

Objective: Develop a script that compiles a weekly email digest from a Google Sheet's data, summarizing key metrics or updates and sending it to a list of subscribers.

Explanation: Demonstrates how to leverage sheet data to automate the creation and distribution of informative email digests, enhancing communication and engagement.

Code:

```
function sendWeeklyEmailDigest() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Weekly
Updates");
 const data = sheet.getDataRange().getValues();
 let emailBody = "This week's key updates:\n\n";
 data.forEach((row, index) => {
 if (index > 0) { // Skip header row
 const [updateTitle, updateDetail] = row;
 emailBody += `* ${updateTitle}: ${updateDetail}\n`;
 }
 });
 MailApp.sendEmail({
 to: "subscribers@example.com",
 subject: "Weekly Digest",
 body: emailBody,
 });
```

```
}
```

Sync Google Forms Responses to Sheet

Objective: Create a script to automatically transfer new Google Forms responses to a designated Google Sheet, including processing or formatting responses as required.

Explanation: Streamlines the management of form responses by automating the transfer and initial processing of data collected via Google Forms.

Code:

```
function onFormSubmit(e) {
 const formResponsesSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Form
Responses");
 const responses = e.values; // e.values contains form response data
 // Process or format responses if necessary
 const processedResponses = responses.map(response => {
 // Example processing step
 return response.toUpperCase();
 });
 formResponsesSheet.appendRow(processedResponses);
}
```

Generate Custom Reports from Sheets

Objective: Implement a script to generate custom reports in Google Docs or Sheets based on user-selected criteria and data filters within a Google Sheet.

Explanation: Enables dynamic report generation based on specified data segments or analysis criteria, facilitating tailored reporting and insights.

Code:

```
function generateCustomReports() {
 const dataSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sales Data");
 const reportCriteria =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Report
Criteria");
 const criteria = reportCriteria.getRange("A2").getValue(); // Assuming
criteria is specified in A2
 const data = dataSheet.getDataRange().getValues().filter(row => {
 // Filter data based on criteria
 return row.includes(criteria);
 });
 const reportDoc = DocumentApp.create(`Custom Report - ${criteria}`);
 const body = reportDoc.getBody();
 data.forEach((row, index) => {
 if (index === 0) {
 // Add headers
 body.appendParagraph(row.join("\t")).setBold(true);
 } else {
 // Add data rows
 body.appendParagraph(row.join("\t"));
 }
 });
 Logger.log(`Report generated: ${reportDoc.getUrl()}`);
}
```

Implement Version Control for Sheet Edits

Objective: Develop a script that tracks changes made to a Google Sheet, logging each edit with details like the cell range, previous value, new value, and timestamp, effectively creating a version history within a separate sheet.

Explanation: Enhances collaboration and accountability by maintaining a detailed log of changes, supporting version control within collaborative spreadsheet environments.

Code:

```
function logSheetEdits(e) {
 const changeLogSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Change Log")
|| SpreadsheetApp.getActiveSpreadsheet().insertSheet("Change Log");
 const {range, oldValue, value} = e;
 const timestamp = new Date();
 const editedCell = range.getA1Notation();
 const user = Session.getActiveUser().getEmail();
 changeLogSheet.appendRow([timestamp, editedCell, oldValue, value,
user]);
}
```

Dynamic Resource Allocation Based on Skills

Objective: Create a script for allocating team members to projects in a Google Sheet based on their skills and project requirements, ensuring optimal team composition.

Explanation: Facilitates efficient team management and project staffing by dynamically matching team members' skills with project needs, automating the allocation process.

Code:

```
function allocateResourcesBasedOnSkills() {
 const teamSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Team
Members");
 const projectSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Projects");
 const allocationSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Allocations") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Allocations");
 const teamMembers = teamSheet.getDataRange().getValues();
```

```
 const projects = projectSheet.getDataRange().getValues();
 projects.forEach((project, pIndex) => {
 if (pIndex === 0) return; // Skip header row
 const [projectName, requiredSkill] = project;
 teamMembers.forEach((member, mIndex) => {
 if (mIndex === 0) return; // Skip header row
 const [memberName, memberSkills] = member;
 if (memberSkills.includes(requiredSkill)) {
 allocationSheet.appendRow([projectName, memberName]);
 }
 });
 });
}
```

Expense Tracking and Analysis

Objective: Build a script to track expenses entered into a Google Sheet, categorizing them and providing monthly summaries and insights into spending patterns.

Explanation: Offers a streamlined approach to personal or organizational expense tracking, enabling detailed analysis and budget management directly within Google Sheets.

Code:

```
function trackAndAnalyzeExpenses() {
 const expensesSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Expenses");
 const summarySheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Monthly
Summary") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Monthly Summary");
 const expenses = expensesSheet.getDataRange().getValues();
 const expenseCategories = {};
 expenses.forEach((expense, index) => {
 if (index === 0) return; // Skip header row
 const [date, category, amount] = expense;
```

```
 const month = date.toLocaleDateString("default", {month: "long"});
 expenseCategories[month] = expenseCategories[month] || {};
 expenseCategories[month][category] =
(expenseCategories[month][category] || 0) + amount;
 });
 summarySheet.clear();
 summarySheet.appendRow(["Month", "Category", "Total Spent"]);
 Object.keys(expenseCategories).forEach(month => {
 Object.keys(expenseCategories[month]).forEach(category => {
 const totalSpent = expenseCategories[month][category];
 summarySheet.appendRow([month, category, totalSpent]);
 });
 });
}
```

Automated Meeting Scheduler

Objective: Develop a script that schedules meetings in Google Calendar based on availability data from a Google Sheet, sending calendar invites to participants.

Explanation: Automates the process of finding suitable meeting times and scheduling them in Google Calendar, based on participant availability stored in a Google Sheet.

Code:

```
function scheduleMeetingsBasedOnAvailability() {
 const availabilitySheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Availability");
 const meetingDetailsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Meeting
Details");
 const participants = availabilitySheet.getDataRange().getValues();
 const meetings = meetingDetailsSheet.getDataRange().getValues();
 meetings.forEach((meeting, index) => {
 if (index === 0) return; // Skip header row
 const [meetingName, meetingTime] = meeting;
```

```
 const calendar = CalendarApp.getDefaultCalendar();
 const eventGuests = participants.filter(participant => {
 // Logic to determine if participant is available at meetingTime
 return participant.includes(meetingTime); // Simplified for example
 }).map(availableParticipant => availableParticipant[0]); // Assuming email
addresses are in the first column
 calendar.createEvent(meetingName, new Date(meetingTime), new
Date(meetingTime), {
 guests: eventGuests.join(','),
 sendInvites: true,
 });
 });
}
```

Custom Feedback Form Analysis

Objective: Create a script to analyze feedback submitted through a Google Forms and stored in a Sheet, identifying key themes, sentiment, and suggestions for improvement.

Explanation: Enables detailed analysis of collected feedback, utilizing text analysis techniques to extract meaningful insights and actionable recommendations.

Code:

```
function analyzeFeedbackForms() {
 const feedbackSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Feedback");
 const analysisSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Feedback
Analysis") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Feedback
Analysis");
 const feedbackResponses = feedbackSheet.getDataRange().getValues();
 // Placeholder for sentiment analysis and theme detection logic
 const feedbackAnalysis = feedbackResponses.map(response => {
 // Example: Analyze feedback text for sentiment and themes
```

```
 return {
 sentiment: "Positive", // Simplified example
 themes: ["Customer Service", "Product Quality"]
 };
 });
 feedbackAnalysis.forEach((analysis, index) => {
 analysisSheet.appendRow([`Response ${index + 1}`, analysis.sentiment,
...analysis.themes]);
 });
}
```

Project Risk Management Dashboard

Objective: Implement a script to create a risk management dashboard in Google Sheets, tracking project risks, their likelihood, impact, and mitigation strategies.

Explanation: Facilitates proactive project risk management by providing a visual dashboard in Google Sheets, enabling teams to monitor and address potential risks effectively.

Code:

```
function createRiskManagementDashboard() {
 const riskSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Project
Risks");
 const dashboardSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Risk
Dashboard") || SpreadsheetApp.getActiveSpreadsheet().insertSheet("Risk
Dashboard");
 const risks = riskSheet.getDataRange().getValues();
 dashboardSheet.clear();
 dashboardSheet.appendRow(["Risk", "Likelihood", "Impact", "Mitigation
Strategy"]);
 risks.forEach((risk, index) => {
 if (index === 0) return; // Skip header row
 const [riskDescription, likelihood, impact, mitigation] = risk;
```

```
 dashboardSheet.appendRow([riskDescription, likelihood, impact,
mitigation]);
 });
}
```

Multi-source Data Aggregation

Objective: Build a script to aggregate data from multiple external APIs into a Google Sheet, transforming and consolidating the data for analysis.

Explanation: Demonstrates how to programmatically fetch, process, and aggregate data from various sources, enabling comprehensive analysis within Google Sheets.

Code:

```
function aggregateExternalData() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Aggregated
Data");
 const sources = [
 'https://api.example.com/data1',
 'https://api.example2.com/data2'
 ];
 sources.forEach(url => {
 const response = UrlFetchApp.fetch(url);
 const data = JSON.parse(response.getContentText());
 // Transform data as needed before appending to the sheet
 const transformedData = data.map(item => [item.id, item.value]); //
Example transformation
 transformedData.forEach(row => sheet.appendRow(row));
 });
}
```

Invoice Generation and Distribution

Objective: Create a script that generates invoices as PDFs from data in a Google Sheet and automatically emails them to clients.

Explanation: Automates the invoicing process, leveraging Google Sheets to maintain client and transaction data, and efficiently distribute invoices via email.

Code:

```
function generateAndEmailInvoices() {
 const clientsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Clients");
 const clients = clientsSheet.getDataRange().getValues();
 clients.forEach((client, index) => {
 if (index === 0) return; // Skip header
 const [clientName, clientEmail, amountDue] = client;
 // Generate invoice PDF (placeholder for document creation and
conversion to PDF)
 const invoiceDoc = DocumentApp.create(`Invoice for ${clientName}`);
 invoiceDoc.getBody().appendParagraph(`Invoice total: $${amountDue}`);
 const pdfBlob = invoiceDoc.getAs(MimeType.PDF);
 // Email PDF invoice
 MailApp.sendEmail(clientEmail, "Your Invoice", "Attached is your invoice.",
{attachments: [pdfBlob]});
 // Clean up by removing the temporary document
 DriveApp.getFileById(invoiceDoc.getId()).setTrashed(true);
 });
}
```

Custom Project Tracking Dashboard

Objective: Develop a script to create a custom dashboard within Google Sheets that tracks project milestones, deadlines, and statuses, updating in real-time.

Explanation: Enhances project management by providing a dynamic visual dashboard in Google Sheets, enabling teams to monitor progress and address delays proactively.

Code:

```
function createProjectTrackingDashboard() {
 const projectsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Projects");
 const dashboardSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Dashboard") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Dashboard");
 const projects = projectsSheet.getDataRange().getValues();
 dashboardSheet.clear();
 dashboardSheet.appendRow(["Project", "Milestone", "Deadline", "Status"]);
 projects.forEach((project, index) => {
 if (index === 0) return; // Skip header
 // Assuming project structure: [Project Name, Milestone, Deadline, Status]
 dashboardSheet.appendRow(project);
 });
}
```

Dynamic Expense Approval System

Objective: Implement a script for a dynamic expense approval system in Google Sheets, where expense submissions trigger email notifications to approvers, and updates are tracked in real-time.

Explanation: Streamlines the expense approval process, using Google Sheets as an interactive platform for submitting, tracking, and approving expense requests.

Code:

```
function triggerExpenseApproval() {
 const expenseSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Expenses");
 const expenses = expenseSheet.getDataRange().getValues();
 expenses.forEach((expense, index) => {
 if (index === 0 || expense[4] === "Approved" || expense[4] === "Denied")
return; // Skip header and already processed expenses
```

```
 // Assuming expense structure: [Submitter, Amount, Description, Date,
Status]
 const [submitter, amount, description] = expense;
 // Placeholder for email notification logic to approver
 MailApp.sendEmail("approver@example.com", "Expense Approval
Request", `Please review the following expense request: \nSubmitter:
${submitter}\nAmount: ${amount}\nDescription: ${description}`);
 // Update the sheet or handle approvals as needed
 });
}
```

Automated Resource Onboarding

Objective: Create a script that automates the onboarding process for new resources in a Google Sheet, including setup tasks, documentation distribution, and email notifications.

Explanation: Facilitates efficient onboarding of new team members or resources, coordinating necessary steps and communications through Google Sheets.

Code:

```
function automateResourceOnboarding() {
 const onboardingSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Onboarding");
 const newResources = onboardingSheet.getDataRange().getValues();
 newResources.forEach((resource, index) => {
 if (index === 0) return; // Skip header
 const [name, email, role] = resource;
 // Placeholder for setup tasks and documentation distribution
 // Send welcome email with onboarding instructions
 MailApp.sendEmail(email, "Welcome to the Team!", `Hi ${name}, welcome
to the team as our new ${role}! Please find attached your onboarding
documents.`);
 // Additional onboarding logic here
 });
}
```

Feedback Collection and Analysis Tool

Objective: Develop a script that collects feedback from a Google Form into a Google Sheet, analyzes the feedback for trends or common themes, and generates a summary report.

Explanation: Enhances the utility of feedback by automating collection, analysis, and reporting, allowing for quick identification of actionable insights.

Code:

```
function collectAndAnalyzeFeedback() {
 const feedbackSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Feedback");
 const analysisSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Feedback
Analysis") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Feedback
Analysis");
 const feedback = feedbackSheet.getDataRange().getValues();
 const themeCounts = {};
 feedback.forEach((response, index) => {
 if (index === 0) return; // Skip header
 const themes = response[2].split(", "); // Assuming themes are listed in the
third column and separated by commas
 themes.forEach(theme => {
 themeCounts[theme] = (themeCounts[theme] || 0) + 1;
 });
 });
 analysisSheet.clear();
 analysisSheet.appendRow(["Theme", "Count"]);
 Object.keys(themeCounts).forEach(theme => {
 analysisSheet.appendRow([theme, themeCounts[theme]]);
 });
}
```

Inventory Optimization Model

Objective: Build a script to model inventory levels in Google Sheets, using historical sales data to predict future stock requirements and optimize inventory orders.

Explanation: Applies predictive modeling to manage inventory efficiently, reducing the risk of stockouts or excess inventory through data-driven decision-making.

Code:

```
function modelInventoryOptimization() {
 const salesSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sales Data");
 const inventorySheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Inventory
Planning");
 const salesData = salesSheet.getDataRange().getValues();
 // Placeholder for predictive modeling logic based on historical sales data
 const futureDemand = salesData.map(sale => {
 // Example: Simple moving average for demand prediction
 return sale[1]; // Assuming sales quantity is in the second column
 });
 inventorySheet.clear();
 inventorySheet.appendRow(["Product ID", "Predicted Future Demand"]);
 futureDemand.forEach((demand, index) => {
 if (index > 0) { // Skip header
 inventorySheet.appendRow([salesData[index][0], demand]); // Assuming
product ID is in the first column
 }
 });
}
```

Custom Order Processing System

Objective: Implement a script for a custom order processing system within Google Sheets, handling order entries, status tracking, and automatic notifications for order updates.

Explanation: Streamlines order management by leveraging Google Sheets as a central system for processing, tracking, and communicating order status updates.

Code:

```
function processOrders() {
 const orderSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Orders");
 const orders = orderSheet.getDataRange().getValues();
 orders.forEach((order, index) => {
 if (index === 0) return; // Skip header
 const [orderId, customerEmail, orderStatus] = order;
 // Placeholder for order processing logic
 if (orderStatus === "Shipped") {
 // Send notification email to customer
 MailApp.sendEmail(customerEmail, "Order Shipped", `Your order
${orderId} has been shipped.`);
 }
 // Additional order status handling as needed
 });
}
```

Automated Budget Tracking

Objective: Create a script to track budget allocations and expenditures in a Google Sheet, providing alerts when spending approaches or exceeds budget limits.

Explanation: Enhances financial management by using Google Sheets to monitor budgets in real-time, ensuring financial discipline and preventing overspending.

Code:

```
function trackBudgets() {
 const budgetSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Budgets");
 const expenditureSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Expenditures"
);
 const budgets = budgetSheet.getDataRange().getValues();
 const expenditures = expenditureSheet.getDataRange().getValues();
 const budgetMap = {};
 budgets.forEach((budget, index) => {
 if (index === 0) return; // Skip header
 const [category, allocatedAmount] = budget;
 budgetMap[category] = allocatedAmount;
 });
 const expenditureSummary = {};
 expenditures.forEach((expenditure, index) => {
 if (index === 0) return; // Skip header
 const [category, amount] = expenditure;
 expenditureSummary[category] = (expenditureSummary[category] || 0) +
amount;
 });
 // Compare expenditures against budgets
 Object.keys(expenditureSummary).forEach(category => {
 if (expenditureSummary[category] > budgetMap[category]) {
 // Alert on budget overrun
 Logger.log(`Budget alert for ${category}: expenditure exceeds budget.`);
 }
 });
}
```

Sales Forecasting Model

Objective: Develop a script to forecast future sales in Google Sheets based on historical sales data, applying linear regression or other statistical methods for prediction.

Explanation: Applies data analysis and predictive modeling within Google Sheets to forecast sales, supporting strategic planning and decision-making with data-driven insights.

Code:

```
function forecastSales() {
 const salesSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Historical
Sales");
 const forecastSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sales
Forecast");
 const salesData = salesSheet.getDataRange().getValues();
 // Placeholder for sales forecasting model (e.g., linear regression)
 const salesForecast = salesData.map((data, index) => {
 if (index === 0) { // Skip header
 // Example: Predict future sales
 return [data[0], data[1] * 1.05]; // Assuming a simple 5% growth projection
 }
 return data;
 });
 forecastSheet.clear();
 forecastSheet.getRange(1, 1, salesForecast.length,
salesForecast[0].length).setValues(salesForecast);
}
```

Custom Data Validation and Cleanup

Objective: Implement a script to perform custom data validation across a Google Sheet, automatically cleaning or flagging data that does not meet specified criteria.

Explanation: Automates the process of ensuring data integrity within a Google Sheet by checking for and correcting common data entry errors, or highlighting them for review.

Code:

```
function customDataValidationAndCleanup() {
 const sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data");
 const dataRange = sheet.getDataRange();
 const data = dataRange.getValues();
 data.forEach((row, rowIndex) => {
 row.forEach((cell, cellIndex) => {
 // Example validation: Check if cell contains a number in a text column
 if (cellIndex === 1 && !isNaN(cell)) { // Assuming column B should only
have text
 sheet.getRange(rowIndex + 1, cellIndex + 1).setBackground("yellow"); //
Flag cell
 }
 });
 });
}
```

Automated Project Timeline Updates

Objective: Create a script to automatically update project timelines in a Google Sheet based on progress entries, adjusting dates and highlighting any delays.

Explanation: Facilitates project management by dynamically updating project schedules within Google Sheets, ensuring timelines are current and visually indicating any project delays.

Code:

```
function updateProjectTimelines() {
 const projectsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Projects");
 const projects = projectsSheet.getDataRange().getValues();
 projects.forEach((project, index) => {
 if (index === 0) return; // Skip header row
 const [projectName, startDate, endDate, progress] = project;
 // Assuming 'progress' is a percentage
 if (progress < 100) {
```

```
 const today = new Date();
 const end = new Date(endDate);
 if (today > end) {
 // Highlight delayed projects
 projectsSheet.getRange(index + 1, 1, 1,
projectsSheet.getLastColumn()).setBackground("red");
 }
 }
 });
}
```

Sync Sheet Data with Cloud Storage

Objective: Develop a script to synchronize Google Sheets data with a cloud storage solution, enabling automatic backup and retrieval of sheet data.

Explanation: Demonstrates how to ensure data resilience and accessibility by maintaining a synchronized copy of Google Sheets data in an external cloud storage service.

Code:

```
function syncWithCloudStorage() {
 const sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 const data = sheet.getDataRange().getValues();
 const dataAsString = JSON.stringify(data);
 // Example using Google Drive as cloud storage
 const fileName = "SheetBackup_" + new Date().toISOString();
 const file = DriveApp.createFile(fileName, dataAsString, MimeType.JSON);
 Logger.log(`Backup created: ${file.getName()}`);
}
```

Dynamic Financial Modeling

Objective: Implement a script to create dynamic financial models within Google Sheets, allowing users to input different assumptions and instantly see projected outcomes.

Explanation: Enhances financial analysis by providing flexible modeling tools within Google Sheets, enabling quick adjustment of assumptions and immediate visualization of their impact.

Code:

```
function dynamicFinancialModeling() {
 const assumptionsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Assumptions")
;
 const modelSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Financial
Model");
 const assumptions = assumptionsSheet.getDataRange().getValues();
 // Reset model sheet
 modelSheet.clear();
 modelSheet.appendRow(["Year", "Revenue", "Expenses", "Net Income"]);
 let revenue = assumptions[1][1]; // Initial revenue assumption
 let growthRate = assumptions[2][1]; // Revenue growth rate
 let expenseRate = assumptions[3][1]; // Expense as % of revenue
 for (let year = 1; year <= 5; year++) {
 let expenses = revenue * expenseRate;
 let netIncome = revenue - expenses;
 modelSheet.appendRow([year, revenue, expenses, netIncome]);
 revenue *= (1 + growthRate); // Apply growth rate for next year
 }
}
```

Inventory Level Optimization

Objective: Build a script to optimize inventory levels based on sales velocity, reorder lead time, and safety stock levels, updating order recommendations in a Google Sheet.

Explanation: Automates inventory management by calculating optimal reorder points and quantities within Google Sheets, minimizing stockouts and overstock situations.

Code:

```
function optimizeInventoryLevels() {
 const inventorySheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Inventory");
 const salesSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sales Data");
 const recommendationsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Order
Recommendations");
 const salesData = salesSheet.getDataRange().getValues();
 const inventoryData = inventorySheet.getDataRange().getValues();
 // Placeholder for inventory optimization logic
 recommendationsSheet.clear();
 recommendationsSheet.appendRow(["Product ID", "Recommended Order
Quantity"]);
 inventoryData.forEach((item, index) => {
 if (index === 0) return; // Skip header
 const [productId, currentStock, leadTime, safetyStock] = item;
 // Calculate recommended order quantity based on sales velocity, lead
time, and safety stock
 // Placeholder calculation
 const recommendedOrderQty = (leadTime * averageSalesVelocity +
safetyStock) - currentStock;
 recommendationsSheet.appendRow([productId, recommendedOrderQty]);
 });
}
```

Custom CRM Dashboard

Objective: Create a script to build a custom CRM (Customer Relationship Management) dashboard within Google Sheets, displaying key customer metrics, recent interactions, and action items.

Explanation: Utilizes Google Sheets to manage customer relationships more effectively, aggregating and visualizing critical data to support sales and service activities.

Code:

```
function buildCustomCRMDashboard() {
 const interactionsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Customer
Interactions");
 const metricsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Customer
Metrics");
 const dashboardSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("CRM
Dashboard") || SpreadsheetApp.getActiveSpreadsheet().insertSheet("CRM
Dashboard");
 const interactions = interactionsSheet.getDataRange().getValues();
 const metrics = metricsSheet.getDataRange().getValues();
 dashboardSheet.clear();
 dashboardSheet.appendRow(["Customer", "Last Interaction", "Total
Spend", "Action Items"]);
 // Example aggregation logic
 metrics.forEach((metric, index) => {
 if (index === 0) return; // Skip header
 const [customerId, customerName, totalSpend] = metric;
 const lastInteraction = interactions.find(interaction => interaction[0] ===
customerId);
 const actionItems = "Follow-up call"; // Placeholder for dynamic action item
determination
 dashboardSheet.appendRow([customerName, lastInteraction ?
lastInteraction[2] : "N/A", totalSpend, actionItems]);
 });
}
```

Automated Data Anomaly Detection

Objective: Develop a script for detecting anomalies in a dataset within Google Sheets, flagging unusual entries for further investigation.

Explanation: Facilitates data quality control by automatically identifying outliers or anomalies in large datasets, highlighting potential errors or extraordinary events.

Code:

```
function detectDataAnomalies() {
 const dataSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data
Analysis");
 const data = dataSheet.getDataRange().getValues();
 data.forEach((row, rowIndex) => {
 row.forEach((cell, cellIndex) => {
 // Placeholder for anomaly detection logic
 const isAnomalous = Math.random() < 0.05; // Random placeholder for
demonstration
 if (isAnomalous) {
 // Flag the cell or row as anomalous
 dataSheet.getRange(rowIndex + 1, cellIndex +
1).setBackground("orange");
 }
 });
 });
}
```

Event Impact Analysis Tool

Objective: Implement a script to analyze the impact of specific events on key metrics within a Google Sheet, comparing pre-event and post-event data.

Explanation: Enables in-depth analysis of event impacts, such as marketing campaigns or operational changes, by measuring variations in relevant metrics before and after the event.

Code:

```
function analyzeEventImpact() {
```

```
 const metricsSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Metrics");
 const eventImpactSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Event Impact
Analysis") || SpreadsheetApp.getActiveSpreadsheet().insertSheet("Event
Impact Analysis");
 const metrics = metricsSheet.getDataRange().getValues();
 eventImpactSheet.clear();
 eventImpactSheet.appendRow(["Metric", "Pre-Event Average",
"Post-Event Average", "Change"]);
 // Placeholder for event impact analysis logic
 metrics.forEach((metric, index) => {
 if (index === 0) return; // Skip header
 const [metricName, preEventAvg, postEventAvg] = metric;
 const change = postEventAvg - preEventAvg;
 eventImpactSheet.appendRow([metricName, preEventAvg, postEventAvg,
change]);
 });
}
```

Multi-level Task Decomposition

Objective: Create a script to facilitate task decomposition in project management, allowing users to break down tasks into subtasks within a Google Sheet, tracking progress at multiple levels.

Explanation: Enhances task management by providing a structured approach to breaking down complex tasks into manageable subtasks, supporting detailed planning and progress tracking.

Code:

```
function decomposeTasks() {
 const tasksSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Tasks");
 const subtasksSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Subtasks") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Subtasks");
```

```
 const tasks = tasksSheet.getDataRange().getValues();
 subtasksSheet.clear();
 subtasksSheet.appendRow(["Task", "Subtask", "Status"]);
 tasks.forEach((task, index) => {
 if (index === 0) return; // Skip header
 const [taskName, taskStatus] = task;
 // Assuming function to retrieve subtasks for the given task
 const subtasks = getSubtasksForTask(taskName); // Placeholder function
 subtasks.forEach(subtask => {
 subtasksSheet.appendRow([taskName, subtask.name, subtask.status]);
 });
 });
}
```

Sentiment Analysis of Customer Feedback

Objective: Develop a script to perform sentiment analysis on customer feedback stored in a Google Sheet, categorizing feedback into positive, neutral, and negative sentiments.

Explanation: Applies natural language processing techniques to evaluate customer feedback, assisting in understanding customer sentiment and identifying areas for improvement.

Code:

```
function analyzeCustomerFeedbackSentiment() {
 const feedbackSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Customer
Feedback");
 const sentimentSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sentiment
Analysis") ||
SpreadsheetApp.getActiveSpreadsheet().insertSheet("Sentiment
Analysis");
 const feedback = feedbackSheet.getDataRange().getValues();
```

```
 sentimentSheet.clear();
 sentimentSheet.appendRow(["Feedback", "Sentiment"]);
 feedback.forEach((item, index) => {
 if (index === 0) return; // Skip header
 const feedbackText = item[1]; // Assuming feedback text is in the second
column
 // Placeholder for sentiment analysis (integration with a sentiment analysis
API or library)
 const sentiment = "Positive"; // Simplified for example purposes
 sentimentSheet.appendRow([feedbackText, sentiment]);
 });
}
```

These exercises are structured to inspire the development of sophisticated, practical solutions using Google Apps Script in Google Sheets. From enhancing project management practices to automating financial models, and from integrating with external data sources to applying data analysis techniques, these tasks aim to provide a solid foundation for leveraging Google Sheets as a powerful tool for a wide array of applications.