50 Exercises
55 Quiz Questions

Getting started with Google Apps Script

Apps Script Coding Exercises

Objective: Write a simple script to log a custom message in the Google Apps Script log.

Code:

```
function logMessage() {
 var message = "Hello, Google Apps Script!";
 Logger.log(message);
}
```

Instructions:

- Open the Google Apps Script editor by navigating to script.google.com.
- Create a new project and paste the code into the script editor.
- Save and name your project.
- Run the logMessage function.
- View the log by clicking on View > Logs.

Explanation: This script defines a function logMessage that creates a variable message containing a string. It then uses Logger.log() to print this message to the Google Apps Script log.

Objective: Use Google Apps Script to send an email to a specific recipient.

Code:

```
function sendEmail() {
 var recipient = "recipient@example.com";
 var subject = "Test Email from Google Apps Script";
 var body = "This is a test email sent from Google Apps Script.";
 MailApp.sendEmail(recipient, subject, body);
}
```

Instructions:

- Follow the same initial steps as above to create a new Google Apps Script project.
- Paste the code into the script editor, replacing "recipient@example.com" with your actual recipient's email address.
- Save and run the sendEmail function.
- Check the recipient's email inbox for the email.

Explanation: This script uses the MailApp.sendEmail() method to send an email. It requires the recipient's email address, the subject of the email, and the body of the email as parameters.

Objective: Programmatically create a new Google Doc and insert text into it.

Code:

```
function createGoogleDoc() {
 var doc = DocumentApp.create('New Google Doc');
 var body = doc.getBody();
 body.appendParagraph('This is a new document created by Google Apps Script.');
}
```

Instructions:

- Create a new Google Apps Script project and paste the code into the editor.
- Save and run the createGoogleDoc function.
- Check your Google Drive for the new document named "New Google Doc".

Explanation: This script creates a new Google Doc titled "New Google Doc" and then inserts a paragraph of text into the document.

Objective: Read and log data from the first cell in a Google Sheets spreadsheet.

Code:

```
function readSpreadsheetData() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var range = sheet.getRange("A1");
 var value = range.getValue();
 Logger.log(value);
}
```

Instructions:

- Open a Google Sheet or create a new one and input some data into cell A1.
- Open the script editor from the sheet (Extensions > Apps Script).
- Paste the code into the script editor.
- Save and run the readSpreadsheetData function.
- View the log to see the data from cell A1.

Explanation: This script gets the active sheet of the current spreadsheet, accesses cell A1, reads its value, and logs it.

Objective: Write a specific piece of data to cell B1 in a Google Sheets spreadsheet.

Code:

```
function writeSpreadsheetData() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.getRange("B1").setValue("Hello, Sheets!");
}
```

Instructions:

- Use the same setup as in the reading data exercise.
- Replace the script with the code provided above.
- Save and run the writeSpreadsheetData function.

Explanation: This script writes the text "Hello, Sheets!" to cell B1 of the active sheet.

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

Code:

```
function appendRowToSheet() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var rowData = ["New", "Row", "Data"];
 sheet.appendRow(rowData);
}
```

Instructions:

- Follow the general script setup instructions.
- Paste this code into the script editor.
- Save and run the appendRowToSheet function.

Explanation: This script appends a new row to the bottom of the active sheet with the data provided in the rowData array.

Objective: Add a custom menu to the Google Sheets UI.

Code:

```
function onOpen() {
```

```
 var ui = SpreadsheetApp.getUi();
 ui.createMenu('Custom Menu')
 .addItem('Say Hello', 'showAlert')
 .addToUi();
}
function showAlert() {
 SpreadsheetApp.getUi().alert('Hello, World!');
}
```

Instructions:

- Open the Google Sheets script editor and paste the code.
- Save the project. The onOpen function automatically runs when the spreadsheet is opened.
- Reload your spreadsheet and look for the "Custom Menu" in the menu bar.
- Click on "Custom Menu" and select "Say Hello" to see the alert.

Explanation: This script creates a custom menu item in Google Sheets that, when clicked, shows an alert box with "Hello, World!".

Objective: Change the background color of a range in Google Sheets.

Code:

```
function updateFormatting() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 sheet.getRange("A1:C3").setBackground("yellow");
}
```

Instructions:

- Use the same setup process as previous spreadsheet-related exercises.
- Paste this code into the script editor.
- Save and run the updateFormatting function.

Explanation: This script changes the background color of the cells in the range A1 to C3 to yellow.

Objective: Use Google Apps Script to fetch data from an external API and log it.

Code:

```
function fetchDataFromAPI() {
 var response = UrlFetchApp.fetch("https://api.example.com/data");
 var jsonData = response.getContentText();
 var data = JSON.parse(jsonData);
 Logger.log(data);
}
```

Instructions:

- Create a new Google Apps Script project.
- Replace "https://api.example.com/data" with a valid API URL.
- Paste the code into the script editor.
- Save and run the fetchDataFromAPI function.
- View the log for the fetched data.

Explanation: This script uses the UrlFetchApp.fetch() method to make a GET request to an API URL, retrieves the response as text, parses the JSON response, and logs it.

Objective: Create a Google Doc and share it with a specific email address.

Code:

```
function createAndShareDocument() {
 var doc = DocumentApp.create('Shared Google Doc');
 var docId = doc.getId();
 DriveApp.getFileById(docId).addEditor('recipient@example.com');
 doc.getBody().appendParagraph('This document is shared with you.');
}
```

Instructions:

- Create a new Google Apps Script project.
- Replace 'recipient@example.com' with the email address of the person you want to share the document with.
- Paste the code into the script editor.
- Save and run the createAndShareDocument function.

Explanation: This script creates a new Google Doc, shares it with the specified email address as an editor, and adds a paragraph of text to the document.

Objective: Send email reminders based on dates and details listed in a Google Sheets spreadsheet.

Code:

```
function sendEmailReminders() {
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Reminders");
```

```
 var dataRange = sheet.getDataRange();
 var data = dataRange.getValues();
 for (var i = 1; i < data.length; i++) {
 var row = data[i];
 var date = new Date(row[0]);
 var today = new Date();
 if (date.setHours(0,0,0,0) === today.setHours(0,0,0,0)) {
 var emailAddress = row[1];
 var subject = "Reminder: " + row[2];
 var message = row[3];
 MailApp.sendEmail(emailAddress, subject, message);
 }
 }
}
```

Instructions:

- Ensure your spreadsheet named "Reminders" has the following columns: Date, Email Address, Subject, Message.
- Paste the code into the Google Apps Script editor linked to the spreadsheet.
- Set a time-driven trigger for this function to run daily.

Explanation: This script checks a "Reminders" spreadsheet for any rows with today's date and sends an email reminder based on the details in those rows.

Objective: Convert data from a Google Sheets spreadsheet into JSON format and log it.

Code:

```
function convertSheetDataToJson() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var data = sheet.getDataRange().getValues();
 var headers = data.shift();
 var jsonData = data.map(function(row) {
 var obj = {};
 headers.forEach(function(header, i) {
 obj[header] = row[i];
 });
 return obj;
 });
 Logger.log(JSON.stringify(jsonData));
}
```

Instructions:

- Use a spreadsheet with the first row as headers.
- Paste this script into the linked Google Apps Script editor.
- Run the function and view the log for the JSON output.

Explanation: This script converts the active sheet's data into an array of JSON objects, with the first row serving as keys.

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

Code:

```
function createCalendarEvents() {
```

```
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Events");
 var data = sheet.getDataRange().getValues();
 var calendar = CalendarApp.getDefaultCalendar();
 data.forEach(function(row, index) {
 if (index > 0 && row[0] && row[1] && row[2]) { // Skip header and empty
rows
 var title = row[0];
 var startTime = new Date(row[1]);
 var endTime = new Date(row[2]);
 calendar.createEvent(title, startTime, endTime);
 }
 });
}
```

Instructions:

- Ensure your spreadsheet named "Events" has columns for Event Title, Start Time, and End Time.
- Paste and run the script, which will create events in your default Google Calendar.

Explanation: This script iterates through each row in the "Events" sheet, creating a calendar event for each row with the specified title, start time, and end time.

Objective: Open a Google Doc, and replace specific placeholder text with new text.

Code:

```
function updateGoogleDoc() {
 var docId = 'YOUR_DOCUMENT_ID';
 var doc = DocumentApp.openById(docId);
 var body = doc.getBody();
 body.replaceText('{{placeholder}}', 'New Text');
 doc.saveAndClose();
}
```

Instructions:

- Replace 'YOUR_DOCUMENT_ID' with the ID of your Google Doc.
- Ensure your document contains the text {{placeholder}}.
- Run the function to replace {{placeholder}} with "New Text".

Explanation: This script opens a Google Doc by ID, searches for a specific placeholder text, replaces it with new text, and then saves the document.

Objective: Aggregate Google Forms responses in a Google Sheet to generate a summary report on a new sheet.

Code:

```
function generateFormResponsesReport() {
 var spreadsheet = SpreadsheetApp.getActiveSpreadsheet();
 var formResponsesSheet = spreadsheet.getSheetByName("Form
Responses 1");
 var reportSheet = spreadsheet.getSheetByName("Report") ||
spreadsheet.insertSheet("Report");
 var responses = formResponsesSheet.getDataRange().getValues();
```

```
// Assume first row is headers and data starts from second row
var summary = {}; // To hold the summary of responses
for (var i = 1; i < responses.length; i++) {
var response = responses[i];
var question = response[1]; // Assuming question is in the second column
if (!summary[question]) {
summary[question] = 1;
} else {
summary[question]++;
}
}
// Clear existing data
reportSheet.clear();
var row = 1;
for (var question in summary) {
reportSheet.getRange(row, 1).setValue(question);
reportSheet.getRange(row, 2).setValue(summary[question]);
row++;
}
}
```

Instructions:

- Ensure your form responses are being collected in a sheet named "Form Responses 1".
- Run the function to generate a summary report in a sheet named "Report".

Explanation: This script tallies responses to a specific question from a Google Form, creating or updating a "Report" sheet with the summarized data.

Objective: Synchronize events listed in a Google Sheets spreadsheet with Google Calendar, creating or updating events as needed.

Code:

```
function syncWithGoogleCalendar() {
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Events");
 var eventsRange = sheet.getDataRange();
 var events = eventsRange.getValues();
 var calendar = CalendarApp.getDefaultCalendar();
 events.forEach(function(event, index) {
 if (index === 0) return; // Skip header row
 var title = event[0];
 var startTime = new Date(event[1]);
 var endTime = new Date(event[2]);
 var options = {description: event[3], location: event[4]};
 var existingEvents = calendar.getEvents(startTime, endTime, {search:
title});
 if (existingEvents.length === 0) {
 calendar.createEvent(title, startTime, endTime, options);
 }
```

```
 });
}
```

Instructions:

- Your "Events" sheet should have columns for Title, Start Time, End Time, Description, and Location.
- Running this function will check for existing events in the specified time range and create new ones if none exist.

Explanation: This script ensures your Google Calendar reflects the events listed in your spreadsheet, avoiding duplicates by checking for existing events with the same title and time.

Objective: Insert data from a Google Sheets spreadsheet into a Google Doc as a formatted list.

Code:

```
function addSheetDataToDoc() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var data = sheet.getDataRange().getValues();
 var docId = 'YOUR_DOCUMENT_ID';
 var doc = DocumentApp.openById(docId);
 var body = doc.getBody();
 data.forEach(function(row) {
 body.appendParagraph(row.join(", "));
 });
 doc.saveAndClose();
}
```

Instructions:

- Replace 'YOUR_DOCUMENT_ID' with the ID of your target Google Doc.
- Run the function to append each row of spreadsheet data to the document as a comma-separated list.

Explanation: This script reads each row of data from the active sheet and appends it to a Google Doc, converting the row data into a comma-separated string.

Objective: Send an email notification when changes are made to a specific Google Sheet.

Code:

```
function onEdit(e) {
 var range = e.range;
 var sheet = range.getSheet();
 if (sheet.getName() === "Monitored Sheet") {
 var userEmail = "your_email@example.com";
 var subject = "Google Sheet Edited";
 var body = "Changes were made to the Monitored Sheet.";
 MailApp.sendEmail(userEmail, subject, body);
 }
}
```

Instructions:

- Replace "Monitored Sheet" with the name of the sheet you want to monitor.
- Replace "your_email@example.com" with your email address.

- This script uses a simple trigger onEdit to run automatically when any cell in the monitored sheet is edited, sending an email notification.

Explanation: The onEdit simple trigger automatically executes when a user edits any cell in a specified sheet, sending a predefined email notification to alert of the change.

Objective: Fetch JSON data from an external API and import it into a Google Sheets spreadsheet.

Code:

```
function importJsonData() {
 var response = UrlFetchApp.fetch("https://api.example.com/data");
 var data = JSON.parse(response.getContentText());
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var rows = data.map(function(item) {
 return [item.id, item.name, item.value]; // Adjust based on JSON structure
 });
 sheet.getRange(1, 1, rows.length, rows[0].length).setValues(rows);
}
```

Instructions:

- Replace "https://api.example.com/data" with the actual URL of the JSON API you're using.
- Adjust the map function to match the structure of your JSON data.
- Run the function to import the data into the active sheet.

Explanation: This script fetches JSON data from a specified API, parses it, and then writes the data to the active spreadsheet, converting each item in the JSON array to a row in the sheet.

Objective: Create a custom function in Google Sheets that can be used as a formula within the spreadsheet.

Code:

```
function DOUBLE(value) {
 return value * 2;
}
```

Instructions:

- Paste this code into the Google Apps Script editor linked to your spreadsheet.
- In your spreadsheet, use the custom function as a formula, e.g., =DOUBLE(A1).

Explanation: This custom function, DOUBLE, takes a numerical value as input and returns the value doubled. Once defined in Apps Script, it can be used directly in your spreadsheet as a formula.

Objective: Perform a batch update to change multiple cell values at once in a Google Sheets spreadsheet.

Code:

```
function batchUpdateCells() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var values = [
```

```
 ["Batch", "Update", "Example"],
 ["Row 2", "Data", "Here"]
 ];
 var range = sheet.getRange("A1:C2");
 range.setValues(values);
}
```

Instructions:

- Open the script editor from your Google Sheets and paste the code.
- Run the batchUpdateCells function to update cells A1:C2 with the data provided in the values array.

Explanation: This script specifies a range within the active sheet and uses the setValues() method to update that range with a 2D array of new values, demonstrating how to efficiently update multiple cells.

Objective: Analyze responses from a Google Form (stored in a Google Sheet) to generate a summary report on another sheet.

Code:

```
function generateReportFromForm() {
 var ss = SpreadsheetApp.getActiveSpreadsheet();
 var formSheet = ss.getSheetByName("Form Responses 1");
 var reportSheet = ss.getSheetByName("Summary Report") ||
ss.insertSheet("Summary Report");
 var responses = formSheet.getDataRange().getValues();
 var summary = {};
```

```
 responses.shift(); // Remove header row
 responses.forEach(function(response) {
 var status = response[2]; // Assuming status is in column C
 if (!summary[status]) {
 summary[status] = 0;
 }
 summary[status]++;
 });
 var row = 1;
 for (var key in summary) {
 reportSheet.getRange(row, 1).setValue(key);
 reportSheet.getRange(row, 2).setValue(summary[key]);
 row++;
 }
}
```

Instructions:

- Ensure your form responses are stored in a sheet named "Form Responses 1" and that one of the columns contains a status or category for summarization.
- Replace "Form Responses 1" and column references as necessary.
- Run the function to populate the "Summary Report" sheet with the aggregated data.

Explanation: This script aggregates data from form responses based on a specific column (status/category) and generates a summary report in a

separate sheet, showcasing data processing and report generation capabilities.

Objective: Fetch data from an external database via a mock API and update a Google Sheets spreadsheet with the data.

Code:

```
function syncWithExternalDatabase() {
 var url = "https://api.example.com/database"; // Mock API endpoint
 var response = UrlFetchApp.fetch(url);
 var data = JSON.parse(response.getContentText());
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var rows = [];
 data.forEach(function(record) {
 rows.push([record.id, record.name, record.value]); // Adjust based on your database structure
 });
 sheet.getRange(1, 1, rows.length, 3).setValues(rows); // Update range based on data size
}
```

Instructions:

- Replace "https://api.example.com/database" with your actual API endpoint.
- Adjust the array push line to match the structure of your database records.
- Run the function to pull data from the external database and update the sheet.

Explanation: This script demonstrates how to integrate Google Sheets with external databases by fetching records via an API and updating the sheet with the retrieved data, simulating a basic data sync operation.

Objective: Send personalized emails to a list of recipients defined in a Google Sheets spreadsheet, including custom messages based on spreadsheet data.

Code:

```
function sendEmailCampaign() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var data = sheet.getDataRange().getValues();
 var subject = "Custom Email Campaign";
 data.forEach(function(row, index) {
 if (index === 0) return; // Skip headers
 var emailAddress = row[0]; // Assuming email addresses are in column A
 var customMessage = row[1]; // Custom message in column B
 var body = "Hello,\n\n" + customMessage + "\n\nBest, Your Team";
 MailApp.sendEmail(emailAddress, subject, body);
 });
}
```

Instructions:

- Ensure your sheet includes email addresses in column A and custom messages in column B.
- Replace the subject and body text as needed.
- Run the function to send personalized emails based on your sheet data.

Explanation: This script showcases how to automate personalized email campaigns using Google Sheets data, iterating over rows to send custom emails to each recipient.

Objective: Generate a Google Docs report that includes dynamic content from a Google Sheets spreadsheet.

Code:

```
function generateDocsReportFromSheets() {
 var sheetData =
SpreadsheetApp.getActiveSpreadsheet().getActiveSheet().getDataRange()
.getValues();
 var doc = DocumentApp.create('Dynamic Report from Sheets');
 var body = doc.getBody();
 sheetData.forEach(function(row, index) {
 if (index === 0) {
 body.appendParagraph(row.join(" |
")).setHeading(DocumentApp.ParagraphHeading.HEADING1);
 } else {
 body.appendParagraph(row.join(", "));
 }
 });
 Logger.log('Report created: ' + doc.getUrl());
}
```

Instructions:

- Run the script to create a new Google Docs report containing data from the active sheet of your Google Spreadsheet.
- The first row is formatted as a heading.

Explanation: This script dynamically creates a Google Docs report from Google Sheets data, demonstrating how to bridge data between Sheets and Docs for reporting purposes.

Objective: Use Google Translate to automatically translate content in a Google Sheets spreadsheet into another language.

Code:

```
function translateSpreadsheetContent() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var range = sheet.getDataRange();
 var values = range.getValues();
 values.forEach(function(row, rowIndex) {
 row.forEach(function(cell, colIndex) {
 var translatedText = LanguageApp.translate(cell, 'en', 'es'); // Translates from English to Spanish
 sheet.getRange(rowIndex + 1, colIndex + 1).setValue(translatedText);
 });
 });
}
```

Instructions:

- Adjust the translate function parameters to match your source and target languages.
- Run the function to translate the content of your active sheet.

Explanation: This script uses the LanguageApp service to translate each cell's content from one language to another, showcasing how to automate language translation within Sheets.

Objective: Set up a script to monitor a specific Google Drive folder for new files and log the names of newly added files.

Code:

```
function monitorDriveFolder() {
 var folderId = 'YOUR_FOLDER_ID';
 var folder = DriveApp.getFolderById(folderId);
 var files = folder.getFiles();
 while (files.hasNext()) {
 var file = files.next();
 Logger.log(file.getName());
 }
}
```

Instructions:

- Replace 'YOUR_FOLDER_ID' with the ID of the Google Drive folder you want to monitor.
- Run the function to log the names of files in the specified folder.

Explanation: This script iterates through all files in a specified Google Drive folder, logging their names, which can be the basis for monitoring changes or additions to the folder.

Objective: Create a new Google Slides presentation and add slides with content from a Google Sheets spreadsheet.

Code:

```
function createSlidesFromSheet() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var data = sheet.getDataRange().getValues();
 var presentation = SlidesApp.create('Presentation from Sheet');
 data.forEach(function(row) {
 var slide =
presentation.appendSlide(SlidesApp.PredefinedLayout.TITLE_AND_BODY
);
 slide.getShapes()[0].getText().setText(row[0]); // Set title
 slide.getShapes()[1].getText().setText(row[1]); // Set body
 });
 Logger.log('Presentation created: ' + presentation.getUrl());
}
```

Instructions:

- Structure your sheet with titles in column A and body text in column B.
- Run the function to create a Google Slides presentation, with each row in the sheet adding a new slide.

Explanation: This script bridges data from Sheets to Slides, creating a presentation where each sheet row generates a new slide with a title and body text, demonstrating automation between different Google Workspace apps.

Objective: Automatically insert a timestamp in a specific column whenever a row in a spreadsheet is edited.

Code:

```
function onEdit(e) {
 var range = e.range;
 var sheet = range.getSheet();
 if (sheet.getName() === "Track Edits") { // Specify the sheet to track
 var row = range.getRow();
 var timestampColumn = 3; // Change as needed
 sheet.getRange(row, timestampColumn).setValue(new Date());
 }
}
```

Instructions:

- Replace "Track Edits" with the name of the sheet you want to track edits on.
- Adjust timestampColumn to the column where you want to insert timestamps.
- This script uses a simple trigger (onEdit) to run automatically after any cell in the specified sheet is edited.

Explanation: The onEdit function checks if the edit occurred in the specified sheet and inserts the current date and time as a timestamp in the designated column, providing an automated way to track edits.

Objective: Review and automatically approve time-off requests submitted via a Google Form, logged in a Google Sheet, based on custom criteria (e.g., request does not exceed the maximum weekly allowance).

Code:

```
function approveTimeOffRequests() {
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Time-Off
Requests");
 var data = sheet.getDataRange().getValues();
 var maxDaysAllowed = 5; // Example criteria
 data.forEach(function(row, index) {
 if (index === 0) return; // Skip header
 var daysRequested = row[2]; // Assuming days requested is in column C
 var statusColumn = 4; // Assuming status is in column D
 var status = daysRequested <= maxDaysAllowed ? "Approved" : "Denied";
 sheet.getRange(index + 1, statusColumn).setValue(status);
 });
}
```

Instructions:

- Ensure your "Time-Off Requests" sheet has columns for request details, including the number of days requested and a column for the approval status.
- Adjust column references as needed based on your sheet's layout.
- Run the function to automatically approve or deny requests based on the specified criteria.

Explanation: This script automates the process of reviewing time-off requests by applying a simple rule (e.g., not exceeding a maximum number of days) to approve or deny requests, demonstrating how to automate workflow decisions based on spreadsheet data.

Objective: Summarize data in a Google Sheet and create a chart based on the aggregated data.

Code:

```
function aggregateDataAndCreateChart() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var dataRange = sheet.getDataRange();
 var data = dataRange.getValues();
 // Assume data has headers and the first column is the category
 var summary = {};
 for (var i = 1; i < data.length; i++) {
 var category = data[i][0];
 var value = data[i][1]; // Assuming values to aggregate are in the second column
 if (summary[category]) {
```

```
 summary[category] += value;
 } else {
 summary[category] = value;
 }
 }
 var chartSheet =
SpreadsheetApp.getActiveSpreadsheet().insertSheet('Chart Sheet');
 var chartData = [['Category', 'Total']];
 for (var key in summary) {
 chartData.push([key, summary[key]]);
 }
 var chartRange = chartSheet.getRange(1, 1, chartData.length, 2);
 chartRange.setValues(chartData);
 var chart = chartSheet.newChart()
 .setChartType(Charts.ChartType.BAR)
 .addRange(chartRange)
 .setPosition(5, 5, 0, 0)
 .build();
 chartSheet.insertChart(chart);
}
```

Instructions:

- Ensure your data is structured with categories in the first column and numeric values in the second column.
- Run the function to aggregate data by category, generate a summary in a new sheet, and create a bar chart visualizing the aggregated data.

Explanation: This script demonstrates how to process and summarize data in Google Sheets, and then use the Google Sheets Chart Service to create a visual representation of that summary.

Objective: Create a custom Google Sheets function that calculates age from a date of birth.

Code:

```
/**
 * Calculates age from the given date of birth.
 *
 * @param {date} dob The date of birth in format "YYYY-MM-DD".
 * @customfunction
 */
function CALCULATEAGE(dob) {
 var today = new Date();
 var birthDate = new Date(dob);
 var age = today.getFullYear() - birthDate.getFullYear();
 var m = today.getMonth() - birthDate.getMonth();
 if (m < 0 || (m === 0 && today.getDate() < birthDate.getDate())) {
 age--;
 }
 return age;
}
```

Instructions:

- Paste this script into the Apps Script editor linked to your Google Sheet.

- Use the function in your sheet like any other formula, e.g., =CALCULATEAGE(A2) where A2 contains a date of birth.

Explanation: This custom function, CALCULATEAGE, takes a date of birth as input and calculates the age based on the current date, showcasing how to create and use custom functions in Google Sheets.

Objective: Automatically share Google Docs with email addresses listed in a Google Sheets spreadsheet.

Code:

```
function autoShareDocuments() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var range = sheet.getDataRange();
 var data = range.getValues();
 data.forEach(function(row, index) {
 if (index === 0) return; // Skip header row
 var docId = row[0]; // Assuming the first column contains document IDs
 var emailAddress = row[1]; // Assuming the second column contains email addresses
 DriveApp.getFileById(docId).addEditor(emailAddress);
 });
}
```

Instructions:

- Structure your sheet with Google Docs IDs in the first column and email addresses in the second column.

- Run the function to share each document with the corresponding email address.

Explanation: This script iterates through a list of document IDs and email addresses, automatically sharing each document with the specified email, demonstrating how to automate document sharing based on spreadsheet data.

Objective: Create or update Google Contacts based on information stored in a Google Sheets spreadsheet.

Code:

```
function syncContactsFromSheet() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var contactsData = sheet.getDataRange().getValues();
 contactsData.shift(); // Skip headers
 contactsData.forEach(function(contact) {
 var fullName = contact[0];
 var email = contact[1];
 var phone = contact[2]; // Adjust the indices based on your data structure
 var contacts = ContactsApp.getContactsByName(fullName);
 var contact;
 if (contacts.length === 0) {
 // Create new contact if not found
 contact = ContactsApp.createContact(fullName, '', email);
 contact.addPhone(ContactsApp.Field.MOBILE_PHONE, phone);
 } else {
```

```
 // Update existing contact
 contact = contacts[0];
 contact.setEmails([]);
 contact.addEmail(email);
 contact.setPhones([]);
 contact.addPhone(ContactsApp.Field.MOBILE_PHONE, phone);
 }
 });
}
```

Instructions:

- Ensure your sheet includes columns for Full Name, Email, and Phone Number.
- Run the script to synchronize each row of your spreadsheet with Google Contacts, creating or updating contacts as necessary.

Explanation: Leveraging the Google Contacts API, this script demonstrates how to either create new contacts or update existing ones with information pulled from a Google Sheets spreadsheet, effectively syncing contact information.

Objective: Use a Google Docs template to generate personalized certificates, filling in names and other details from a Google Sheets spreadsheet.

Code:

```
function generateCertificates() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
```

```
 var data = sheet.getDataRange().getValues();
 var templateId = 'YOUR_TEMPLATE_DOCUMENT_ID';
 data.forEach(function(row, index) {
 if (index === 0) return; // Skip header
 var docCopy = DriveApp.getFileById(templateId).makeCopy();
 var docCopyId = docCopy.getId();
 var doc = DocumentApp.openById(docCopyId);
 var body = doc.getBody();
 // Assuming first column is the name and second column is the date
 body.replaceText('{{Name}}', row[0]);
 body.replaceText('{{Date}}', row[1]);
 doc.saveAndClose();
 // Rename the document
 docCopy.setName('Certificate for ' + row[0]);
 });
}
```

Instructions:

- Replace 'YOUR_TEMPLATE_DOCUMENT_ID' with your Google Docs template ID.
- Ensure your template contains the placeholders {{Name}} and {{Date}}.
- Run the function to generate personalized certificates for each row in your spreadsheet.

Explanation: This script creates personalized certificates for each entry in a Google Sheets spreadsheet by copying a Google Docs template and

replacing placeholders with actual data, demonstrating how to automate document personalization.

Objective: Validate data entries in real-time in a Google Sheet and mark invalid entries for review.

Code:

```
function onEdit(e) {
 var range = e.range;
 var value = range.getValue();
 var sheet = range.getSheet();
 if (sheet.getName() === "Data Entry") { // Specify your sheet name
 var columnOfInterest = 2; // Specify the column to monitor (e.g., column B
= 2)
 if (range.getColumn() === columnOfInterest) {
 var isValid = validateData(value); // Implement your validation logic
 if (!isValid) {
 range.setBackground('red'); // Highlight invalid data entries
 } else {
 range.setBackground(null); // Clear background if data is valid
 }
 }
 }
}
function validateData(value) {
 // Example validation: check if the value is a number
```

```
 return !isNaN(value);
}
```

Instructions:

- Implement your validation logic within the validateData function based on your criteria.
- Adjust the columnOfInterest and sheet name as needed.
- This script uses a simple trigger (onEdit) to validate data in real-time as it's entered into the specified column, highlighting invalid entries.

Explanation: The script automatically validates data as it's entered into a specified column of a sheet, using conditional logic to highlight entries that don't meet predefined validation criteria, illustrating real-time data validation.

Objective: Generate custom reports based on Google Sheets data and email them as PDF attachments.

Code:

```
function generateAndEmailReports() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var dataRange = sheet.getDataRange();
 var data = dataRange.getValues();
 var reportContent = createReportContent(data); // Implement this function based on your reporting logic
 var tempDoc = DocumentApp.create('Temporary Report');
 tempDoc.getBody().appendParagraph(reportContent);
 DocumentApp.getActiveDocument().saveAndClose();
```

```
 var pdf = DriveApp.getFileById(tempDoc.getId()).getAs('application/pdf');
 var recipients = 'recipient@example.com'; // Specify recipients
 var subject = 'Custom Report';
 var body = 'Find attached the custom report.';
 MailApp.sendEmail(recipients, subject, body, {attachments: [pdf]});
 // Clean up by deleting the temporary document
 DriveApp.getFileById(tempDoc.getId()).setTrashed(true);
}
function createReportContent(data) {
 // Example content creation logic
 var content = "Report Title\n\n";
 data.forEach(function(row, index) {
 if (index === 0) return; // Skip headers
 content += row.join(", ") + "\n";
 });
 return content;
}
```

Instructions:

- Customize the createReportContent function to format your report content based on the data in your sheet.
- Specify the recipients, subject, and body for the email.
- Run the function to generate a report as a Google Doc, convert it to PDF, and email it as an attachment.

Explanation: This script illustrates how to programmatically generate custom reports from spreadsheet data, convert those reports to PDFs, and

then email the PDFs, showcasing end-to-end automation of report generation and distribution.

Objective: Fetch data from an external API periodically and update a Google Sheets spreadsheet with the latest data.

Code:

```
function updateSheetWithAPIData() {
 var apiURL = "https://api.example.com/data"; // Use your API endpoint
 var response = UrlFetchApp.fetch(apiURL);
 var jsonData = JSON.parse(response.getContentText());
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("API Data");
 // Clear existing data
 sheet.clear();
 // Assuming JSON data is an array of objects
 var headers = Object.keys(jsonData[0]);
 sheet.appendRow(headers); // Append headers
 jsonData.forEach(function(item) {
 var row = headers.map(function(header) {
 return item[header];
 });
 sheet.appendRow(row);
 });
}
```

Instructions:

- Replace "https://api.example.com/data" with your actual API URL.
- Ensure your Google Sheet has a sheet named "API Data".
- Run the function to fetch and update your sheet with fresh data from the API.

Explanation: This script demonstrates how to pull data from an external API and use it to refresh or update a Google Sheets spreadsheet, showing how to automate data synchronization tasks.

Objective: Automatically create Google Calendar events based on form submissions logged in a Google Sheets spreadsheet.

Code:

```
function createEventsFromFormSubmissions() {
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Form
Responses");
 var data = sheet.getDataRange().getValues();
 var calendar = CalendarApp.getDefaultCalendar();
 data.forEach(function(row, index) {
 if (index === 0) return; // Skip headers
 var eventName = row[1]; // Adjust based on your form structure
 var startTime = new Date(row[2]);
 var endTime = new Date(row[3]);
 // Check if event already exists to avoid duplicates
 var events = calendar.getEventsForDay(startTime);
```

```
 var exists = events.some(function(event) {
 return event.getTitle() === eventName && event.getStartTime().getTime()
=== startTime.getTime();
 });
 if (!exists) {
 calendar.createEvent(eventName, startTime, endTime);
 }
 });
}
```

Instructions:

- Adjust column references to match where event names and start/end times are stored in your "Form Responses" sheet.
- Run the function to create calendar events for new form submissions, ensuring no duplicates are created.

Explanation: This script checks for new form submissions in a Google Sheet and creates corresponding events in Google Calendar, demonstrating how to automate event creation from structured data inputs while avoiding duplicate entries.

Objective: Generate a Gantt chart in Google Sheets based on project timelines and milestones data.

Code:

```
function createGanttChart() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
```

```
 var chartRange = sheet.getRange("A1:C10"); // Adjust range based on
your data
 var chartBuilder = sheet.newChart();
 chartBuilder
 .setChartType(Charts.ChartType.BAR)
 .addRange(chartRange)
 .setPosition(5, 5, 0, 0)
 .setOption('title', 'Project Gantt Chart')
 .setOption('legend', { position: 'none' })
 .setOption('hAxis', { format: 'yyyy-MM-dd' })
 .setOption('direction', -1);
 var chart = chartBuilder.build();
 sheet.insertChart(chart);
}
```

Instructions:

- Adjust the chartRange to include your project timeline data, typically structured with task names in column A, start dates in column B, and end dates or durations in column C.
- Run the function to create and insert a Gantt-like chart in your Google Sheet, visualizing your project timelines.

Explanation: This script utilizes Google Sheets' charting capabilities to create a Gantt-like chart for visualizing project schedules, illustrating how to dynamically generate visual project management tools within spreadsheets.

Objective: Update a large dataset in a Google Sheet efficiently using batch processing to minimize execution time and API calls.

Code:

```
function batchUpdateSpreadsheet() {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var data = []; // Assuming we have a 2D array of data to write
 for (let i = 0; i < 1000; i++) {
 data.push(["Row " + (i + 1), Math.random() * 100]);
 }
 // Perform a batch update instead of updating cells one by one
 var range = sheet.getRange(1, 1, data.length, 2);
 range.setValues(data);
}
```

Instructions:

- Prepare a 2D array of data you wish to write to the spreadsheet.
- Replace the loop with your data preparation logic if necessary.
- Run the function to perform a batch update of the spreadsheet.

Explanation: This script demonstrates the efficient update of a Google Sheet with a large amount of data using batch processing, which significantly reduces the execution time and the number of API calls compared to updating cells individually.

Objective: Automatically parse and extract specific information from Gmail messages and log it in a Google Sheets spreadsheet.

Code:

```
function extractEmailDataToSheet() {
 var query = 'subject:"Order Confirmation"'; // Customize your Gmail search
 var threads = GmailApp.search(query, 0, 10);
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 threads.forEach(function(thread) {
 var messages = thread.getMessages();
 messages.forEach(function(message) {
 var content = message.getPlainBody();
 // Example parsing logic (customize as needed)
 var orderIdMatch = content.match(/Order ID: (\d+)/);
 var amountMatch = content.match(/Amount: \$(\d+(\.\d+)?)/);
 if (orderIdMatch && amountMatch) {
 sheet.appendRow([message.getDate(), orderIdMatch[1],
amountMatch[1]]);
 }
 });
 });
}
```

Instructions:

- Customize the query to target specific emails in your Gmail inbox.
- Adjust the parsing logic based on the format of your emails.

- Run the function to extract and log the desired information in your sheet.

Explanation: This script searches for emails in Gmail based on a specified query, extracts information using regular expressions, and appends it to a Google Sheet, showcasing how to automate data extraction from emails.

Objective: Generate customized documents by assembling text from a Google Sheets spreadsheet into a Google Docs template.

Code:

```
function automateDocumentAssembly() {
 var dataSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Data");
 var rows = dataSheet.getDataRange().getValues();
 var templateId = "TEMPLATE_DOCUMENT_ID"; // Place your template
document ID here
 rows.forEach(function(row, index) {
 if (index === 0) return; // Skip header row
 var documentCopy = DriveApp.getFileById(templateId).makeCopy();
 var copyId = documentCopy.getId();
 var doc = DocumentApp.openById(copyId);
 var body = doc.getBody();
 // Assuming first column is the recipient's name, second column is a date,
etc.
 body.replaceText("{{RecipientName}}", row[0]);
 body.replaceText("{{Date}}", row[1]);
```

```
 // Add more replacements as needed
 doc.saveAndClose();
 // Optionally rename the document
 documentCopy.setName("Document for " + row[0]);
 });
}
```

Instructions:

- Ensure your sheet named "Data" contains the necessary information for document customization.
- Replace "TEMPLATE_DOCUMENT_ID" with the ID of your Google Docs template.
- Run the function to generate customized documents based on your spreadsheet data.

Explanation: Leveraging a Google Docs template, this script creates personalized documents for each row in a Google Sheets spreadsheet by replacing placeholder text with actual data, demonstrating automated document assembly.

Objective: Automatically send email reports at scheduled intervals containing aggregated data from a Google Sheets spreadsheet.

Code:

```
function scheduleEmailReports() {
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Sales Data");
```

```
 var data = sheet.getDataRange().getValues();
 var report = aggregateSalesData(data); // Implement this function based
on your needs
 var recipient = "manager@example.com";
 var subject = "Weekly Sales Report";
 var body = "Here is the latest sales report:\n\n" + report;
 MailApp.sendEmail(recipient, subject, body);
}
function aggregateSalesData(data) {
 // Example aggregation logic
 var totalSales = 0;
 data.forEach(function(row, index) {
 if (index === 0) return; // Skip headers
 totalSales += row[2]; // Assuming sales figures are in the third column
 });
 return "Total Sales: $" + totalSales;
}
```

Instructions:

- Customize the aggregateSalesData function to generate a report based on your spreadsheet's data structure.
- Set a time-driven trigger for the scheduleEmailReports function to send reports automatically.

Explanation: This script aggregates data from a Google Sheets spreadsheet to create a report and uses Google Apps Script's email

capabilities to send this report to a specified recipient at scheduled intervals, automating the reporting process.

Objective: Publish a Google Sheets spreadsheet as a web app, allowing users to view and interact with the data in a custom interface.

Code:

```
function doGet(e) {
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
 var data = sheet.getDataRange().getValues();
 var output = HtmlService.createHtmlOutput("<h1>Data List</h1>");
 data.forEach(function(row) {
 output.append("<p>" + row.join(", ") + "</p>");
 });
 return output;
}
```

Instructions:

- Deploy this script as a web app from the Google Apps Script editor by selecting Publish > Deploy as web app.
- Visit the provided URL to view your spreadsheet data in a simple web interface.

Explanation: This script uses the HtmlService to generate and serve a simple HTML page that displays data from a Google Sheets spreadsheet, illustrating how to make spreadsheet data accessible via a custom web interface.

Objective: Automatically categorize and analyze survey responses stored in a Google Sheets spreadsheet to identify common themes or keywords.

Code:

```
function analyzeSurveyResponses() {
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Survey
Responses");
 var responses = sheet.getRange("B2:B" +
sheet.getLastRow()).getValues(); // Assuming responses are in column B
 var analysis = {};
 responses.forEach(function(response) {
 var words = response[0].split(/\s+/);
 words.forEach(function(word) {
 if (!analysis[word]) {
 analysis[word] = 0;
 }
 analysis[word]++;
 });
 });
 // Log the frequency of each word for analysis
 Logger.log(analysis);
}
```

Instructions:

- Ensure your survey responses are collected in the specified column of your sheet.
- Run the function to parse responses and count the occurrence of each word.

Explanation: This script demonstrates a basic approach to text analysis, parsing survey responses to calculate the frequency of word occurrences, which can help in identifying common themes or concerns among respondents.

Objective: Distribute tasks listed in a Google Sheets spreadsheet among team members via automated emails, ensuring an even workload distribution.

Code:

```
function distributeTasksAndSendEmails() {
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Tasks");
 var tasks = sheet.getDataRange().getValues();
 var teamMembers = ["member1@example.com",
"member2@example.com", "member3@example.com"]; // Add your team
members
 tasks.forEach(function(task, index) {
 if (index === 0) return; // Skip header row
 var assignedMember = teamMembers[index % teamMembers.length];
 var subject = "New Task Assigned: " + task[0]; // Assuming task name is in
the first column
```

```
 var body = "You have been assigned a new task: " + task[0] +
"\nDescription: " + task[1]; // Assuming task description is in the second
column
 MailApp.sendEmail(assignedMember, subject, body);
 });
}
```

Instructions:

- Adjust the teamMembers array to include the email addresses of your team members.
- Ensure your tasks are listed in the specified columns of your "Tasks" sheet.
- Run the function to automatically assign tasks and send notification emails to team members.

Explanation: This script cycles through tasks in a Google Sheets spreadsheet, assigning each task to a team member in a round-robin fashion and sending an email to notify the member of their new task, demonstrating how to automate task distribution.

Objective: Aggregate data from multiple sheets within a Google Sheets spreadsheet to create a comprehensive dashboard on a master sheet.

Code:

```
function createDashboard() {
 var spreadsheet = SpreadsheetApp.getActiveSpreadsheet();
 var dashboardSheet = spreadsheet.getSheetByName("Dashboard") ||
spreadsheet.insertSheet("Dashboard");
```

```
 var dataSources = ["Sales", "Expenses", "Inventory"]; // Names of your
data source sheets
 var dashboardContent = [];
 dataSources.forEach(function(sheetName) {
 var sheet = spreadsheet.getSheetByName(sheetName);
 var data = sheet.getDataRange().getValues();
 // Example: Aggregate data or extract key metrics
 var summaryMetric = data.reduce(function(total, row) {
 return total + row[1]; // Assuming relevant data is in the second column
 }, 0);
 dashboardContent.push([sheetName, summaryMetric]);
 });
 // Clear existing content and update dashboard
 dashboardSheet.clear();
 dashboardSheet.getRange(1, 1, dashboardContent.length,
2).setValues(dashboardContent);
}
```

Instructions:

- Customize dataSources with the names of your actual data source sheets.
- Adjust the aggregation logic to suit your data structure and analysis needs.
- Run the function to compile key metrics from multiple sources onto the "Dashboard" sheet.

Explanation: This script consolidates key data from multiple sheets into a single "Dashboard" sheet, providing a unified view of critical metrics, illustrating how to build a simple dashboard within Google Sheets.

Objective: Send out feedback forms via email and automatically collect and summarize responses in a Google Sheet.

Code:

```
function sendFeedbackForms() {
 var recipients = ["user1@example.com", "user2@example.com"]; // Add recipients
 var formUrl = "https://docs.google.com/forms/d/e/YOUR_FORM_ID/viewform"; // Replace with your Google Form URL
 var subject = "We value your feedback!";
 var body = "Please take a moment to fill out this feedback form: " + formUrl;
 recipients.forEach(function(email) {
 MailApp.sendEmail(email, subject, body);
 });
}
function collectFeedbackResponses() {
 var form = FormApp.openById("YOUR_FORM_ID"); // Replace with your form ID
 var formResponses = form.getResponses();
 var sheet = SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Feedback Summary");
 formResponses.forEach(function(formResponse) {
```

```
 var itemResponses = formResponse.getItemResponses();
 var responseData = [];
 itemResponses.forEach(function(itemResponse) {
 responseData.push(itemResponse.getResponse());
 });
 // Append the response data to the summary sheet
 sheet.appendRow(responseData);
 });
}
```

Instructions:

- Replace "YOUR_FORM_ID" and formUrl with your actual Google Form ID and URL.
- Customize the recipient list for the feedback forms.
- Run sendFeedbackForms to email the feedback forms, and collectFeedbackResponses to aggregate responses into a "Feedback Summary" sheet.

Explanation: The first script automates the sending of feedback forms to a list of recipients, while the second script collects responses from the feedback form and summarizes them in a Google Sheet, demonstrating an end-to-end feedback collection and analysis process.

Objective: Use Google Sheets to track project milestones and visualize progress with conditional formatting and charts.

Code:

```
function trackAndVisualizeMilestones() {
 var sheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName("Project
Milestones");
 var milestonesRange = sheet.getDataRange();
 var milestonesData = milestonesRange.getValues();
 // Apply conditional formatting to visualize progress
 var rules = sheet.getConditionalFormatRules();
 rules.push(SpreadsheetApp.newConditionalFormatRule()
 .whenCellNotEmpty()
 .setBackground("#4CAF50")
 .setRanges([sheet.getRange("B2:B" + milestonesData.length)]) //
Assuming status is in column B
 .build());
 sheet.setConditionalFormatRules(rules);
 // Generate a pie chart to show the distribution of milestone statuses
 var chart = sheet.newChart()
 .setChartType(Charts.ChartType.PIE)
 .addRange(sheet.getRange("B1:B" + milestonesData.length))
 .setPosition(5, 5, 0, 0)
 .build();
 sheet.insertChart(chart);
}
```

Instructions:

- Ensure your "Project Milestones" sheet includes milestone statuses in column B.
- Adjust the range in the script based on the actual location of your milestone data.
- Run the function to apply conditional formatting and insert a pie chart visualizing the distribution of milestone statuses.

Explanation: This script applies conditional formatting to easily visualize the completion status of project milestones in a Google Sheet and generates a pie chart to provide a visual summary of the project's progress, demonstrating the use of Google Sheets for project management and reporting.

Google Apps Script Quiz Questions

What is Google Apps Script?

A) A JavaScript library for building web applications
B) A scripting language provided by Google to automate tasks across Google products
C) A standalone IDE for developing Google Play applications
D) A cloud service for hosting Google applications

Correct Answer: B) A scripting language provided by Google to automate tasks across Google products
Explanation: Google Apps Script is a cloud-based scripting language for

light-weight application development within the Google Workspace platform. It allows users to automate tasks, manipulate data across Google products (like Sheets, Docs, Drive, Calendar, and Gmail), and connect to external APIs, all with a JavaScript-like syntax.

Which service in Google Apps Script is used to send emails?

A) GmailApp
B) MailService
C) EmailApp
D) GoogleMail

Correct Answer: A) GmailApp
Explanation: The GmailApp service in Google Apps Script provides methods to send emails, access and modify Gmail messages, labels, and threads. It is the preferred service for interacting with Gmail.

How can you trigger a Google Apps Script to run at a specific time?

A) Using an onEdit trigger
B) By setting a time-driven trigger
C) With a manual trigger setup
D) Through an external API call

Correct Answer: B) By setting a time-driven trigger
Explanation: Time-driven triggers are used to execute Google Apps Script

functions automatically at a specified date, time, or at regular intervals. These triggers can be configured in the script's Triggers page in the Apps Script editor.

Which of the following is NOT a valid trigger type in Google Apps Script?

A) onOpen
B) beforeClose
C) onEdit
D) onChange

Correct Answer: B) beforeClose
Explanation: beforeClose is not a valid trigger type in Google Apps Script. Valid simple trigger types include onOpen, which runs when a document is opened; onEdit, which runs when a user edits a document; and onChange, which runs when a change is made to a document. There is no trigger that runs before a document is closed.

What is the purpose of the Lock Service in Google Apps Script?

A) To encrypt script code
B) To prevent simultaneous execution of specific sections of code
C) To lock a Google Sheet from being edited
D) To manage access permissions for Google Drive files

Correct Answer: B) To prevent simultaneous execution of specific sections of code

Explanation: The Lock Service in Google Apps Script is used to prevent concurrent access to sections of code that cannot be simultaneously executed by multiple instances. This is particularly useful in scenarios where scripts are accessing or modifying shared resources and data consistency is critical.

Which method is used to fetch data from an external API in Google Apps Script?

A) HttpClient.get()

B) UrlFetchApp.fetch()

C) Ajax.get()

D) WebApp.fetchData()

Correct Answer: B) UrlFetchApp.fetch()

Explanation: UrlFetchApp.fetch() is the method used in Google Apps Script to make HTTP requests to external URLs, allowing scripts to fetch data from APIs, submit forms, or access other resources over the internet.

How can you access and manipulate the contents of a Google Doc through Google Apps Script?

A) Using the DriveApp service

B) By employing the DocumentApp service

C) Through the GoogleDocs service
D) With the DocsApp service

Correct Answer: B) By employing the DocumentApp service
Explanation: The DocumentApp service provides classes and methods to access and modify Google Docs documents. This service allows scripts to create, access, and modify Google Docs content programmatically.

In Google Apps Script, which object is typically used to interact with the active spreadsheet in a Google Sheets add-on?

A) SpreadsheetApp.getActiveSpreadsheet()
B) GoogleSheets.getActive()
C) SheetsApp.getCurrent()
D) Workbook.getActiveInstance()

Correct Answer: A) SpreadsheetApp.getActiveSpreadsheet()
Explanation: SpreadsheetApp.getActiveSpreadsheet() is used to get the currently active spreadsheet where the script is running. This method is commonly used in scripts associated with Google Sheets add-ons or custom functions to interact with the spreadsheet data.

What is the purpose of the Cache Service in Google Apps Script?

A) To temporarily store data for fast access
B) To cache script files for quicker execution
C) To save user preferences between script executions
D) To archive script execution logs

Correct Answer: A) To temporarily store data for fast access
Explanation: The Cache Service in Google Apps Script is used to temporarily store data in key-value pairs for fast access across script executions. This can significantly improve performance by reducing the need to fetch or compute data repeatedly.

Which of the following is a way to execute a function within a Google Apps Script project?

A) Right-clicking the function name and selecting "Run"
B) Using a custom keyboard shortcut defined in the script
C) Selecting the function in the Apps Script editor and clicking the "Run" button
D) Sending an email to the script's project address with the function name in the subject

Correct Answer: C) Selecting the function in the Apps Script editor and clicking the "Run" button
Explanation: In the Google Apps Script editor, you can execute a specific function by selecting it from the dropdown menu near the "Run" button, then clicking "Run". This is the standard way to manually execute functions during development and testing.

Which service in Google Apps Script can be used to create a new file in Google Drive?

A) DriveService
B) GoogleDrive
C) DriveApp
D) FileManager

Correct Answer: C) DriveApp
Explanation: The DriveApp service in Google Apps Script provides methods to create, find, and manipulate files and folders in Google Drive. It is the correct service to use for interacting with Google Drive within Apps Script projects.

How can you access the properties of a Google Form from Google Apps Script?

A) FormsService
B) FormApp
C) GoogleForms
D) SurveyApp

Correct Answer: B) FormApp
Explanation: The FormApp service allows scripts to create, access, and modify Google Forms. It provides a wide range of functionalities for manipulating form properties, items, and responses.

Which of the following is NOT a valid way to trigger Google Apps Script code to run?

A) On form submission
B) On document load
C) On spreadsheet cell value change
D) On external web service call

Correct Answer: D) On external web service call
Explanation: While Google Apps Script can trigger functions to run on various internal events (like form submissions, document openings, or cell value changes in Sheets), there is no built-in trigger for executing script code directly in response to an external web service call. However, web apps or executions via the Apps Script API can be set up to respond to web requests.

In Google Apps Script, what is the purpose of the PropertiesService?

A) To set properties of Google Workspace documents
B) To manage script and user properties for storing and retrieving data
C) To change system properties of the Google Apps Script environment
D) To configure property rules for Google Forms

Correct Answer: B) To manage script and user properties for storing and retrieving data
Explanation: The PropertiesService allows scripts to store and retrieve data

in the form of properties. It provides access to three different types of property stores: script, user, and document (for Add-ons), which can be used to save data needed across executions or for configuration purposes.

What does the Session service in Google Apps Script provide?

A) Management of user sessions for Web Apps
B) Access to information about the current script's execution session
C) Creation of parallel execution threads
D) Session storage similar to web session cookies

Correct Answer: B) Access to information about the current script's execution session
Explanation: The Session service provides access to session information, such as the user's email address and the script's running environment. It does not manage web sessions or parallel execution threads but rather offers information about the current user and script execution context.

Which method can be used to append a new row to a Google Sheet using Google Apps Script?

A) SpreadsheetApp.appendRow()
B) Sheet.appendRow()
C) GoogleSheets.addRow()
D) ActiveSheet.addRow()

Correct Answer: B) Sheet.appendRow()
Explanation: The appendRow(rowContents) method belongs to the Sheet class in Google Apps Script, allowing a single array of values to be appended as a new row at the bottom of the sheet. This method is used on a specific sheet object, typically obtained through SpreadsheetApp.

What is the return type of UrlFetchApp.fetch(url) in Google Apps Script?

A) JSON
B) HttpResponse
C) String
D) WebContent

Correct Answer: B) HttpResponse
Explanation: UrlFetchApp.fetch(url) returns an HTTPResponse object, not the raw content directly. This object provides methods to further access the response content, headers, and status codes, allowing for more detailed handling of the HTTP response.

How can you make a Google Apps Script available as a web service?

A) By publishing the script as a web app
B) By enabling the "Web Service" advanced Google service
C) Through configuring the script's manifest file for web deployment
D) By deploying the script as an API executable

Correct Answer: A) By publishing the script as a web app
Explanation: Google Apps Script can be made available as a web service by publishing it as a web app. This process involves setting specific access permissions and executing the script as the user accessing the web app or the developer. The web app URL can then be accessed by users to interact with the script via HTTP requests.

Which statement about custom functions in Google Sheets created with Google Apps Script is TRUE?

A) They can make calls to UrlFetchApp services.
B) They run only when the spreadsheet is open.
C) They cannot access other cells besides those they are directly called with.
D) They automatically recalculate only when the script is manually executed.

Correct Answer: C) They cannot access other cells besides those they are directly called with.
Explanation: Custom functions in Google Sheets are designed to operate based on the parameters passed to them and cannot call services that require authorization, such as UrlFetchApp, or access other cells not included in their parameters. They recalculate automatically based on changes to their input cells, not just when the script is executed.

What is the correct way to log messages for debugging in Google Apps Script?

A) console.log("Message")
B) Logger.log("Message")
C) Debug.print("Message")
D) System.out.println("Message")

Correct Answer: B) Logger.log("Message")
Explanation: In Google Apps Script, the Logger.log(message) method is used to log debugging messages. These messages can be viewed in the Apps Script editor under the "Logs" section after script execution. While console.log() is also supported, especially for modern development and Stackdriver Logging, Logger.log() is the traditional method for logging in Apps Script.

What is the primary purpose of the trigger in Google Apps Script?

A) To change script permissions
B) To schedule script execution in response to specific events
C) To debug script errors
D) To compile the script into an executable format

Correct Answer: B) To schedule script execution in response to specific events
Explanation: Triggers in Google Apps Script allow scripts to be executed

automatically in response to specified events, such as time-based triggers, edits to a Google Sheet, or receiving an email, among others. They are essential for automating workflows without manual intervention.

Which of the following is NOT a valid event object for a Google Sheet trigger function in Google Apps Script?

A) edit
B) change
C) submit
D) open

Correct Answer: C) submit
Explanation: In the context of Google Sheets, the event objects associated with triggers include edit, change, and open. The submit event object is not valid for Google Sheets triggers but is associated with Google Forms to capture form submissions.

How can you programmatically create a new Google Calendar event with Google Apps Script?

A) CalendarApp.createEvent(title, startTime, endTime)
B) GoogleCalendar.newEvent(title, startTime, endTime)
C) Calendar.newEvent(title, start, end)
D) GCalendar.createEvent(title, startDate, endDate)

Correct Answer: A) CalendarApp.createEvent(title, startTime, endTime)
Explanation: The CalendarApp.createEvent(title, startTime, endTime) method is used in Google Apps Script to create a new event in the user's default calendar, with specified title, start time, and end time.

In Google Apps Script, which method would you use to retrieve values from a range in a Google Sheet?

A) Sheet.getValues(range)
B) Range.pull()
C) Range.getValues()
D) SpreadsheetApp.fetch(range)

Correct Answer: C) Range.getValues()
Explanation: The Range.getValues() method is used to retrieve the values of cells within a specified range in a Google Sheet. This method returns a two-dimensional array of values, corresponding to the rows and columns of the range.

What does the ContentService in Google Apps Script allow you to do?

A) Manage file contents in Google Drive
B) Serve JSON or XML content for web apps
C) Modify the content of Google Docs documents
D) Encrypt script content

Correct Answer: B) Serve JSON or XML content for web apps
Explanation: The ContentService in Google Apps Script is used to create and serve JSON, XML, or plain text content to clients. It is often used in the context of publishing a script as a web app, enabling the script to return data in a format that can be easily consumed by web clients or other applications.

Which of the following is true about library use in Google Apps Script?

A) Libraries can only be used within the same Google Workspace domain.
B) Libraries are deprecated and no longer supported in Apps Script.
C) You can use libraries to share and reuse code across different scripts.
D) Using libraries significantly slows down the execution of scripts.

Correct Answer: C) You can use libraries to share and reuse code across different scripts.
Explanation: Libraries in Google Apps Script allow developers to share and reuse code across multiple projects. By creating a library, developers can include common functionalities and services in their scripts without duplicating code, making development more efficient and manageable.

How would you describe the execution of an onEdit(e) trigger function in Google Apps Script?

A) It runs before any edit is made to a document.
B) It executes automatically in response to every edit made in a Google

Sheet.
C) It requires manual activation each time an edit occurs.
D) It only runs when a script editor makes an edit.

Correct Answer: B) It executes automatically in response to every edit made in a Google Sheet.
Explanation: The onEdit(e) trigger function in Google Apps Script is designed to run automatically every time a user edits a cell in a Google Sheet. This simple trigger helps automate tasks or actions immediately following an edit, enhancing interactivity and efficiency within spreadsheets.

What is the maximum execution time for a Google Apps Script bound to a Google Sheet, Doc, or Form?

A) 1 minute
B) 6 minutes
C) 30 minutes
D) There is no time limit

Correct Answer: B) 6 minutes
Explanation: For scripts that are bound to a Google Sheet, Doc, or Form, Google Apps Script imposes a maximum execution time limit of 6 minutes per execution. This limit helps manage resources on Google's servers and ensures fair usage among all users.

Which service allows you to interact with Google Sheets in Google Apps Script?

A) GoogleSheets
B) SpreadsheetService
C) SheetsAPI
D) SpreadsheetApp

Correct Answer: D) SpreadsheetApp
Explanation: The SpreadsheetApp service in Google Apps Script provides the ability to create, access, and modify Google Sheets files. It offers a wide range of functionalities for interacting with spreadsheet data, such as reading and writing data, managing sheets, and configuring properties.

When deploying a Google Apps Script as a Web App, which access levels can you choose for "Who has access to the app"?

A) Only myself
B) Anyone with Google account
C) Anyone, even anonymous
D) All of the above

Correct Answer: D) All of the above
Explanation: When deploying a Google Apps Script as a Web App, developers can specify who has access to the app by selecting from several options: "Only myself," "Anyone with Google account," and "Anyone, even anonymous." This flexibility allows developers to control the

visibility and accessibility of their web apps based on the intended audience and use case.

What is the purpose of the Logger class in Google Apps Script?

A) To log errors and exceptions in the script execution
B) To track user interactions within a web app
C) To write debug information to the Google Apps Script log
D) To log data changes in Google Sheets

Correct Answer: C) To write debug information to the Google Apps Script log
Explanation: The Logger class in Google Apps Script is used primarily for debugging purposes, allowing developers to write informational messages to the log, which can be viewed in the Apps Script editor. This aids in troubleshooting and refining scripts.

Which object is passed as an argument to the doGet(e) and doPost(e) functions in a Web App script?

A) A URL query string
B) An event object containing information about the request
C) A JSON object with the user's credentials
D) A response object to send data back to the client

Correct Answer: B) An event object containing information about the request

Explanation: The doGet(e) and doPost(e) functions in Web App scripts receive an event object e as an argument, which contains information about the HTTP request, such as query string parameters for doGet or posted data for doPost. This allows the script to process incoming requests and respond accordingly.

How can you make a script run when a Google Form is submitted?

A) By setting a time-driven trigger on the form
B) By manually executing the script after each submission
C) By attaching an onFormSubmit trigger to the form
D) By configuring the form to send a POST request to the script

Correct Answer: C) By attaching an onFormSubmit trigger to the form

Explanation: An onFormSubmit trigger can be attached to a Google Form to automatically execute a script function when the form is submitted. This trigger can be set up from the Google Apps Script editor under the current project's triggers.

What method can be used to create a menu in Google Sheets with Google Apps Script?

A) SpreadsheetApp.getUi().createMenu('My Menu')
B) Sheet.createMenu('My Menu')

C) UiApp.createApplication().createMenu('My Menu')
D) DocumentApp.createMenu('My Menu')

Correct Answer: A) SpreadsheetApp.getUi().createMenu('My Menu')
Explanation: To create a custom menu in Google Sheets using Google Apps Script, you use the createMenu(menuName) method on the Ui class obtained from SpreadsheetApp.getUi(). This allows you to add custom menu items to the Sheets UI.

Which method is used to save changes to a document in Google Docs Script?

A) DocumentApp.getActiveDocument().save()
B) Changes are automatically saved
C) DocumentApp.getActiveDocument().commitChanges()
D) GoogleDocs.saveActiveDocument()

Correct Answer: B) Changes are automatically saved
Explanation: In Google Docs Script (part of Google Apps Script), changes made to a document through a script are automatically saved. There is no need for a specific method call to save changes, as the document is automatically updated in real-time.

In Google Apps Script, how can you continue script execution even after encountering an error?

A) By using the try...catch statement
B) By setting the script's failure tolerance property
C) By configuring an onError trigger
D) By enabling advanced error handling features

Correct Answer: A) By using the try...catch statement
Explanation: The try...catch statement in Google Apps Script (and JavaScript in general) allows you to catch errors that occur in the try block and handle them in the catch block, enabling the script to continue running even if an error is encountered.

What is a correct way to access a specific sheet by name in a Google Sheets document using Google Apps Script?

A) SpreadsheetApp.openByName('SheetName')
B) SpreadsheetApp.getActiveSpreadsheet().getSheetByName('SheetName')
C) Sheets.getSheet('SheetName')
D) GoogleSheets.find('SheetName')

Correct Answer: B) SpreadsheetApp.getActiveSpreadsheet().getSheetByName('SheetName')
Explanation: To access a specific sheet by name within the active Google Sheets document, you use the getSheetByName('SheetName') method on the Spreadsheet object, which can be obtained with SpreadsheetApp.getActiveSpreadsheet().

Which of the following scopes is required in the manifest file to send emails using Google Apps Script?

A) https://www.googleapis.com/auth/gmail.compose
B) https://www.googleapis.com/auth/gmail.send
C) https://www.googleapis.com/auth/gmail.readonly
D) https://www.googleapis.com/auth/email

Correct Answer: B) https://www.googleapis.com/auth/gmail.send
Explanation: The scope https://www.googleapis.com/auth/gmail.send is required to send emails through Gmail using Google Apps Script. This scope allows the script to send emails on behalf of the user but does not grant access to read or modify emails.

How do you publish a Google Apps Script as an API executable?

A) By deploying it as a web app
B) Through the "Deploy as API executable" option in the editor
C) By sharing the script with users who want to execute it
D) API executables are not supported in Google Apps Script

Correct Answer: B) Through the "Deploy as API executable" option in the editor
Explanation: To publish a Google Apps Script as an API executable, you use the "Deploy as API executable" option in the new Apps Script editor.

This allows other applications to directly call functions in the script via the Apps Script API, provided they have appropriate permissions.

What is the best practice for managing sensitive data, such as API keys, in Google Apps Script?

A) Storing them directly in the script code
B) Using the PropertiesService to store them as script or user properties
C) Hard-coding them into function calls
D) Saving them in a Google Sheet accessed by the script

Correct Answer: B) Using the PropertiesService to store them as script or user properties
Explanation: The best practice for managing sensitive data, like API keys, in Google Apps Script is to use the PropertiesService. This service allows you to store data as script or user properties, keeping it separate from the script code and thereby enhancing security and maintainability.

Which service allows you to programmatically manipulate charts in a Google Sheets spreadsheet?

A) ChartsService
B) SpreadsheetApp
C) GoogleCharts
D) ChartApp

Correct Answer: B) SpreadsheetApp
Explanation: The SpreadsheetApp service in Google Apps Script provides functionalities to manipulate spreadsheets, including creating and modifying charts within Google Sheets. While there is a Charts Service for more advanced chart operations in Apps Script, basic chart manipulations are handled through methods associated with sheet objects obtained via SpreadsheetApp.

What is the primary method for debugging Google Apps Script code?

A) Using console.log() statements
B) Running the script in debug mode using the Apps Script editor
C) Writing errors to a Google Sheet
D) Emailing error logs to the script owner

Correct Answer: B) Running the script in debug mode using the Apps Script editor
Explanation: The Apps Script editor provides a debug mode that allows developers to step through their code, inspect variables, and identify where errors occur. This is the primary method for debugging Google Apps Script code. While console.log() and Logger.log() can also be used to print out debugging information, using the debug mode offers a more interactive way to troubleshoot issues.

How can a Google Apps Script interact with Google Calendar?

A) By using the Calendar API
B) Through direct database access
C) By embedding calendar events in script
D) Using the CalendarApp service

Correct Answer: D) Using the CalendarApp service
Explanation: The CalendarApp service in Google Apps Script provides methods to create, access, and manipulate events in Google Calendar. This service allows scripts to interact directly with a user's calendar, offering a straightforward way to automate calendar-related tasks without needing to use the lower-level Calendar API directly.

Which of the following best describes how to use Google Apps Script to add custom functionality to Google Docs?

A) By inserting JavaScript directly into the Google Doc
B) Using the DocumentApp service to create bound scripts
C) Through external APIs that interact with Google Docs
D) By embedding HTML and CSS in the document

Correct Answer: B) Using the DocumentApp service to create bound scripts
Explanation: The DocumentApp service in Google Apps Script allows developers to create scripts that interact directly with Google Docs, adding

custom functionality. These scripts can be bound to the document, meaning they are associated directly with a specific Google Doc and can manipulate its content programmatically.

What is the correct method to retrieve the value from a cell in A1 notation in Google Sheets using Google Apps Script?

A) SpreadsheetApp.getActiveSpreadsheet().getValue("A1")
B) SpreadsheetApp.getActiveSpreadsheet().getActiveSheet().getCell("A1").getValue()
C) SpreadsheetApp.getActiveSpreadsheet().getActiveSheet().getRange("A1").getValue()
D) SheetsApp.getCurrentCell("A1").getValue()

Correct Answer: C) SpreadsheetApp.getActiveSpreadsheet().getActiveSheet().getRange("A1").getValue()
Explanation: The method getRange("A1").getValue() is the correct way to retrieve the value of a cell specified in A1 notation in Google Sheets using Google Apps Script. This method is called on a Sheet object, which is accessed via getActiveSheet() of the active Spreadsheet object.

In Google Apps Script, how can you publish a script to run as a web application?

A) Convert the script to a Google Cloud Platform project
B) Use the Publish > Deploy as web app menu option in the script editor
C) Export the script as an HTML file and host it on a web server
D) Google Apps Script cannot be published as web applications

Correct Answer: B) Use the Publish > Deploy as web app menu option in the script editor
Explanation: To publish a Google Apps Script as a web application, you use the Publish > Deploy as web app option in the Google Apps Script editor. This allows you to deploy your script so it can be accessed over the web, either publicly or within your organization, depending on the access settings you choose.

Which of the following cannot be achieved with Google Apps Script?

A) Automatically sending emails based on Google Sheets data
B) Creating a custom function in Google Sheets that calls external APIs
C) Directly manipulating the HTML content of a Google Site
D) Scheduling calendar events based on responses to a Google Form

Correct Answer: C) Directly manipulating the HTML content of a Google Site
Explanation: While Google Apps Script can interact with Google Sites through the Google Sites API to create or update pages and content, it cannot directly manipulate the HTML content of a Google Site in the same

way it can manipulate the DOM of a custom web app created with HTML Service.

What feature does Google Apps Script provide for managing script properties and user preferences?

A) The Configuration API
B) The Properties Service
C) DataStore Service
D) Preferences Library

Correct Answer: B) The Properties Service
Explanation: The Properties Service in Google Apps Script allows scripts to store and access data across different executions. This service can be used to manage script properties, user properties, and document-specific properties, making it ideal for storing configuration settings or user preferences.

How can you access the active user's email address in Google Apps Script?

A) Session.getActiveUser().getEmail()
B) GoogleApps.getUser().email
C) UserProperties.getActiveUserEmail()
D) ScriptApp.getUserEmail()

Correct Answer: A) Session.getActiveUser().getEmail()
Explanation: The Session.getActiveUser().getEmail() method is used to retrieve the email address of the user currently executing the script, assuming the script has the necessary permissions and the user's identity can be determined in the current context.

What is the correct approach to append data to an existing sheet in Google Sheets using Google Apps Script?

A) sheet.appendData([dataArray])
B) sheet.appendRow([dataArray])
C) spreadsheet.addRows(sheetName, [dataArray])
D) range.setValues([dataArray]) after selecting the last row

Correct Answer: B) sheet.appendRow([dataArray])
Explanation: The appendRow(rowContents) method is used to append a single row of data to the bottom of an existing sheet in Google Sheets using Google Apps Script. This method takes an array of values that correspond to the cells in the new row.

Which Google Apps Script service allows for the manipulation of Google Slides?

A) SlidesApp
B) PresentationApp
C) GoogleSlides
D) SlideService

Correct Answer: A) SlidesApp
Explanation: The SlidesApp service in Google Apps Script is used to create, access, and modify Google Slides presentations. This service provides a wide range of functionalities for interacting with slides, including adding content, modifying properties, and managing slides within a presentation.

In Google Apps Script, what does the method createTrigger(functionName) of the ScriptApp class do?

A) Creates a new document in Google Docs
B) Sends a request to an external API
C) Creates a new trigger programmatically
D) Starts a background web worker

Correct Answer: C) Creates a new trigger programmatically
Explanation: The createTrigger(functionName) method of the ScriptApp class in Google Apps Script is used to create a new trigger programmatically. This allows scripts to automatically execute a specified function in response to specified events, such as time-based triggers or event-driven triggers like form submissions or document edits.

What is the main use of the HtmlService in Google Apps Script?

A) To scrape web pages
B) To create HTML-formatted emails

C) To serve web pages as part of a web app
D) To convert Google Docs to HTML

Correct Answer: C) To serve web pages as part of a web app
Explanation: The HtmlService in Google Apps Script is primarily used to create and serve HTML content for web apps. It allows developers to serve custom HTML, CSS, and JavaScript to users, enabling the creation of dynamic web interfaces that can interact with Google Apps Script functions.

Which of the following is true about deploying Google Apps Script as an API executable?

A) It allows the script to be executed from any standard HTML form.
B) It enables the script to be called from other Google Apps Script projects using the Apps Script API.
C) It publishes the script to the Google Cloud Platform for external access.
D) It converts the script into a standalone web service with a public endpoint.

Correct Answer: B) It enables the script to be called from other Google Apps Script projects using the Apps Script API.
Explanation: Deploying Google Apps Script as an API executable allows the script to be called programmatically from other Google Apps Script projects or external applications using the Apps Script API, facilitating integration and automation across different platforms.

How can you customize the behavior of a Google Form with Google Apps Script?

A) By embedding custom JavaScript directly into the form
B) Using the FormApp service to modify the form and handle responses
C) Through direct manipulation of the form's HTML and CSS
D) By deploying a custom form as a web app

Correct Answer: B) Using the FormApp service to modify the form and handle responses
Explanation: The FormApp service in Google Apps Script allows for the creation, modification, and handling of Google Forms programmatically. This service enables developers to customize the behavior of forms, add or modify questions, set response validations, and manage form responses, among other functionalities.