1) Custom menu in Google Sheets

Does: Adds a menu item so users can click to run functions.
How it works: `onOpen()` runs automatically when the file opens (for editors), and `SpreadsheetApp.getUi()` adds UI.

```
function onOpen() {
  SpreadsheetApp.getUi()
    .createMenu('Tools')
    .addItem('Say Hello', 'sayHello')
    .addToUi();
}

function sayHello() {
  SpreadsheetApp.getUi().alert('Hello from Apps Script!');
}
```

---

2) Show a toast message

Does: Shows a small notification in Sheets.
How it works: `toast()` displays messages without interrupting users.

```
function showToast() {
  SpreadsheetApp.getActive().toast('Processing complete ✅', 'Status',
5);
}
```

---

3) Read active cell value

Does: Gets the current selected cell value.
How it works: `getActiveCell()` returns the cell object, `getValue()` returns its value.

```
function readActiveCell() {
  const cell = SpreadsheetApp.getActiveRange();
  Logger.log('Value: ' + cell.getValue());
}
```

---

4) Write value to a cell

Does: Writes text to A1 on the active sheet.
How it works: `getRange("A1").setValue(...)` updates the cell.

```
function writeToA1() {
  const sheet = SpreadsheetApp.getActiveSheet();
  sheet.getRange('A1').setValue('Updated at ' + new Date());
}
```

---

5) Append a row to a sheet

Does: Adds a row at the bottom with timestamp + user.
How it works: `appendRow()` auto-finds the next row.

```
function appendLogRow() {
```

```
  const ss = SpreadsheetApp.getActive();
  const sheet = ss.getSheetByName('Log') || ss.insertSheet('Log');
  sheet.appendRow([new Date(), Session.getActiveUser().getEmail(),
'Ran appendLogRow']);
}
```

---

6) Read a whole range into a 2D array

Does: Reads values from A1:D10.
How it works: `getValues()` returns a 2D array.

```
function readRangeArray() {
  const sheet = SpreadsheetApp.getActiveSheet();
  const data = sheet.getRange('A1:D10').getValues();
  Logger.log(JSON.stringify(data));
}
```

---

7) Write a 2D array into a range

Does: Writes a mini table starting at A1.
How it works: `setValues()` writes all cells at once (faster than loops).

```
function writeArrayTable() {
  const sheet = SpreadsheetApp.getActiveSheet();
  const values = [
    ['Name', 'Score'],
    ['Ava', 92],
    ['Noah', 87]
  ];
  sheet.getRange(1, 1, values.length,
values[0].length).setValues(values);
}
```

---

8) Clear a range (values + formats)

Does: Clears A1:D20 completely.
How it works: `clear()` removes content and formatting.

```
function clearBlock() {
  SpreadsheetApp.getActiveSheet().getRange('A1:D20').clear();
}
```

---

9) Clear only contents (keep formatting)

Does: Removes values but keeps styles.
How it works: `clearContent()` doesn't touch formats.

```
function clearOnlyContent() {
  SpreadsheetApp.getActiveSheet().getRange('A1:D20').clearContent();
}
```

---

10) Freeze header row

Does: Freezes first row.
How it works: Freeze helps keep headers visible while scrolling.

```
function freezeHeader() {
  SpreadsheetApp.getActiveSheet().setFrozenRows(1);
}
```

---

11) Auto-resize columns

Does: Resizes columns A–D to fit content.
How it works: `autoResizeColumns(start, num)`.

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

```
  sheet.autoResizeColumns(1, 4); // A=1, 4 columns => A:D
}
```

---

12) Add a filter to header row

Does: Creates a filter over A1:D.
How it works: Filters allow sorting/filtering in UI.

```
function addFilter() {
  const sheet = SpreadsheetApp.getActiveSheet();
  const lastRow = sheet.getLastRow();
  if (lastRow < 2) return;
  const range = sheet.getRange(1, 1, lastRow, 4);
  if (!range.getFilter()) range.createFilter();
}
```

---

13) Sort by a column

Does: Sorts data range A2:D by column B ascending.
How it works: `sort(colIndex)` sorts by that column.

```
function sortByColumnB() {
  const sheet = SpreadsheetApp.getActiveSheet();
  const lastRow = sheet.getLastRow();
  if (lastRow < 3) return;
  sheet.getRange(2, 1, lastRow - 1, 4).sort(2); // column B is 2
}
```

---

14) Find duplicates and highlight them

Does: Highlights duplicates in column A (red).
How it works: Uses a `Set` to track seen values.

```
function highlightDuplicatesInColA() {
```

```
  const sheet = SpreadsheetApp.getActiveSheet();
  const range = sheet.getRange('A1:A' + sheet.getLastRow());
  const values = range.getValues().flat();
  const seen = new Set();

  const backgrounds = values.map(v => {
    const key = String(v).trim();
    if (!key) return ['white'];
    if (seen.has(key)) return ['#ffcccc'];
    seen.add(key);
    return ['white'];
  });

  range.setBackgrounds(backgrounds);
}
```

---

15) Data validation dropdown

Does: Creates dropdown options in B2:B20.
How it works: `DataValidationBuilder` builds rules.

```
function addDropdownValidation() {
  const sheet = SpreadsheetApp.getActiveSheet();
  const rule = SpreadsheetApp.newDataValidation()
    .requireValueInList(['Low', 'Medium', 'High'], true)
    .setAllowInvalid(false)
    .build();

  sheet.getRange('B2:B20').setDataValidation(rule);
}
```

---

16) Conditional formatting (greater than)

Does: Colors values > 90 in green in column B.
How it works: Adds a conditional format rule.

```
function conditionalFormatScores() {
  const sheet = SpreadsheetApp.getActiveSheet();
  const range = sheet.getRange('B2:B');
  const rules = sheet.getConditionalFormatRules();

  const rule = SpreadsheetApp.newConditionalFormatRule()
    .whenNumberGreaterThan(90)
    .setRanges([range])
    .setBackground('#ccffcc')
    .build();

  sheet.setConditionalFormatRules([...rules, rule]);
}
```

---

17) Protect a range (warning only)

Does: Warns users when editing A1:D1.
How it works: `setWarningOnly(true)` shows a warning instead of blocking.

```
function protectHeaderWarning() {
  const sheet = SpreadsheetApp.getActiveSheet();
  const protection = sheet.getRange('A1:D1').protect();
  protection.setWarningOnly(true);
}
```

---

18) Create a new sheet (if missing)

Does: Ensures a sheet named “Data” exists.
How it works: Checks by name, creates if not found.

```
function ensureSheetData() {
  const ss = SpreadsheetApp.getActive();
  const name = 'Data';
  const sheet = ss.getSheetByName(name) || ss.insertSheet(name);
  sheet.activate();
}
```

---

19) Duplicate a sheet

Does: Copies active sheet and renames it.
How it works: `copyTo()` duplicates into same spreadsheet.

```
function duplicateActiveSheet() {
  const ss = SpreadsheetApp.getActive();
  const sheet = ss.getActiveSheet();
  const copy = sheet.copyTo(ss);
  copy.setName(sheet.getName() + ' Copy ' + Utilities.formatDate(new
Date(), ss.getSpreadsheetTimeZone(), 'yyyyMMdd_HHmm'));
}
```

---

20) Export a sheet as PDF and email it

Does: Generates a PDF of the spreadsheet and emails it.
How it works: Builds an export URL + uses OAuth token to fetch the blob.

```
function emailSpreadsheetAsPdf() {
  const ss = SpreadsheetApp.getActive();
  const fileId = ss.getId();
  const url =
`https://docs.google.com/spreadsheets/d/${fileId}/export?format=pdf&po
rtrait=true&fitw=true`;

  const token = ScriptApp.getOAuthToken();
  const response = UrlFetchApp.fetch(url, { headers: { Authorization:
'Bearer ' + token } });
  const pdfBlob = response.getBlob().setName(ss.getName() + '.pdf');

  const recipient = Session.getActiveUser().getEmail();
  GmailApp.sendEmail(recipient, 'PDF Export: ' + ss.getName(),
'Attached is the PDF export.', { attachments: [pdfBlob] });
}
```

---

21) Create a Google Doc and write content

Does: Makes a new Doc and inserts text.
How it works: `DocumentApp.create()` returns a Doc; body holds paragraphs.

```
function createDocWithText() {
  const doc = DocumentApp.create('Auto Doc ' + new Date());
  doc.getBody().appendParagraph('Hello! This document was created by
Apps Script.');
  Logger.log('Doc URL: ' + doc.getUrl());
}
```

---

22) Find and replace in a Google Doc

Does: Replaces “foo” with “bar” in the active doc.
How it works: `replaceText(pattern, replacement)` uses regex-like patterns.

```
function replaceInDoc() {
  const doc = DocumentApp.getActiveDocument();
  const body = doc.getBody();
  body.replaceText('foo', 'bar');
}
```

---

23) Add a table to a Google Doc

Does: Inserts a simple 3×2 table.
How it works: `appendTable()` with a 2D array.

```
function addTableToDoc() {
  const doc = DocumentApp.getActiveDocument();
  const body = doc.getBody();
  body.appendTable([
    ['Item', 'Qty'],
    ['Pencils', '10'],
```

```
    ['Notebooks', '5']
  ]);
}
```

---

24) Create a Docs sidebar

Does: Opens a sidebar UI in Google Docs.
How it works: HTMLService returns UI HTML.

```
function onOpen() {
  DocumentApp.getUi()
    .createMenu('Doc Tools')
    .addItem('Open Sidebar', 'openSidebar')
    .addToUi();
}

function openSidebar() {
  const html = HtmlService.createHtmlOutput('<div
style="font-family:Arial;padding:12px">Hi from Sidebar!</div>')
    .setTitle('My Sidebar');
  DocumentApp.getUi().showSidebar(html);
}
```

---

25) Insert current date at cursor in Google Docs

Does: Inserts date where the cursor is.
How it works: Uses `getCursor()` and inserts into the element.

```
function insertDateAtCursor() {
  const doc = DocumentApp.getActiveDocument();
  const cursor = doc.getCursor();
  if (!cursor) throw new Error('Place the cursor in the document
first.');
  cursor.insertText(Utilities.formatDate(new Date(),
Session.getScriptTimeZone(), 'yyyy-MM-dd'));
}
```

---

26) Create a Google Form with questions

Does: Creates a form with a short answer + multiple choice.
How it works: `FormApp.create()` then `addTextItem()`, `addMultipleChoiceItem()`.

```
function createSampleForm() {
  const form = FormApp.create('Feedback ' + new Date());
  form.addTextItem().setTitle('Your name').setRequired(true);
  form.addMultipleChoiceItem()
    .setTitle('Rate the session')
    .setChoiceValues(['1', '2', '3', '4', '5'])
    .setRequired(true);

  Logger.log('Form URL: ' + form.getEditUrl());
}
```

---

27) Read latest form responses

Does: Logs the latest response values.
How it works: `getResponses()` returns response objects.

```
function logLatestFormResponse() {
  const form = FormApp.getActiveForm();
  const responses = form.getResponses();
  if (!responses.length) return;

  const latest = responses[responses.length - 1];
  const items = latest.getItemResponses().map(r => ({
    title: r.getItem().getTitle(),
    answer: r.getResponse()
  }));

  Logger.log(JSON.stringify(items, null, 2));
}
```

---

28) Send an email (simple)

Does: Emails yourself a test message.
How it works: `GmailApp.sendEmail(to, subject, body)`.

```
function sendTestEmail() {
  const to = Session.getActiveUser().getEmail();
  GmailApp.sendEmail(to, 'Apps Script Test', 'Hello from Apps Script
at ' + new Date());
}
```

---

29) Search Gmail and list subject lines

Does: Finds last 5 emails matching a query.
How it works: `GmailApp.search()` uses Gmail search operators.

```
function searchGmailSubjects() {
  const threads = GmailApp.search('newer_than:7d', 0, 5);
  threads.forEach(t => {
    const msg = t.getMessages()[0];
    Logger.log(msg.getDate() + ' - ' + msg.getSubject());
  });
}
```

---

30) Create a Calendar event

Does: Adds a 30-minute event to your primary calendar.
How it works: CalendarApp creates events with start/end dates.

```
function createCalendarEvent() {
  const cal = CalendarApp.getDefaultCalendar();
  const start = new Date();
  const end = new Date(start.getTime() + 30 * 60 * 1000);
```

```
  cal.createEvent('Quick Meeting', start, end, { description: 'Created
by Apps Script' });
}
```

---

31) List upcoming Calendar events

Does: Logs next 10 events for the next 7 days.
How it works: `getEvents(start, end)` returns event objects.

```
function listUpcomingEvents() {
  const cal = CalendarApp.getDefaultCalendar();
  const start = new Date();
  const end = new Date(start.getTime() + 7 * 24 * 60 * 60 * 1000);
  const events = cal.getEvents(start, end).slice(0, 10);

  events.forEach(e => Logger.log(e.getStartTime() + ' | ' +
e.getTitle()));
}
```

---

32) Create a Drive folder

Does: Makes a folder in your Drive.
How it works: DriveApp creates folders, returns a Folder object.

```
function createDriveFolder() {
  const folder = DriveApp.createFolder('My Folder ' + new
Date().toISOString());
  Logger.log(folder.getUrl());
}
```

---

33) List files in a folder by ID

Does: Lists file names from a specific folder.
How it works: Uses `getFiles()` iterator.

```
function listFilesInFolder() {
  const folderId = 'PASTE_FOLDER_ID_HERE';
  const folder = DriveApp.getFolderById(folderId);
  const files = folder.getFiles();

  while (files.hasNext()) {
    const f = files.next();
    Logger.log(f.getName() + ' | ' + f.getUrl());
  }
}
```

---

34) Copy a file

Does: Copies a Drive file and renames it.
How it works: `makeCopy(newName)` duplicates.

```
function copyDriveFile() {
  const fileId = 'PASTE_FILE_ID_HERE';
  const file = DriveApp.getFileById(fileId);
  const copy = file.makeCopy(file.getName() + ' (Copy)');
  Logger.log(copy.getUrl());
}
```

---

35) Convert a Google Doc to PDF in Drive

Does: Creates a PDF file version in Drive.
How it works: `getAs(MimeType.PDF)` returns a blob you can save.

```
function saveDocAsPdf() {
  const docId = 'PASTE_DOC_ID_HERE';
  const file = DriveApp.getFileById(docId);
  const pdfBlob = file.getAs(MimeType.PDF).setName(file.getName() +
'.pdf');
  const pdfFile = DriveApp.createFile(pdfBlob);
  Logger.log(pdfFile.getUrl());
}
```

---

36) Fetch JSON from a public API

Does: Calls an API and parses JSON.
How it works: UrlFetch gets text; `JSON.parse()` makes an object.

```
function fetchJsonExample() {
  const url = 'https://api.github.com/';
  const res = UrlFetchApp.fetch(url, { muteHttpExceptions: true });
  const data = JSON.parse(res.getContentText());
  Logger.log(data);
}
```

---

37) POST JSON to a webhook

Does: Sends JSON payload to a webhook URL.
How it works: `UrlFetchApp.fetch()` with method + JSON body.

```
function postJsonWebhook() {
  const webhookUrl = 'PASTE_WEBHOOK_URL_HERE';
  const payload = { event: 'test', time: new Date().toISOString() };

  const res = UrlFetchApp.fetch(webhookUrl, {
    method: 'post',
    contentType: 'application/json',
    payload: JSON.stringify(payload),
    muteHttpExceptions: true
  });

  Logger.log(res.getResponseCode() + ' ' + res.getContentText());
}
```

---

38) Generate a UUID

Does: Creates a unique ID for records.
How it works: `Utilities.getUuid()` generates a random UUID.

```
function makeUuid() {
  Logger.log(Utilities.getUuid());
}
```

---

39) Format dates consistently

Does: Logs a date formatted for spreadsheets.
How it works: Uses timezone-aware formatting.

```
function formatDateExample() {
  const tz = Session.getScriptTimeZone();
  Logger.log(Utilities.formatDate(new Date(), tz, 'yyyy-MM-dd
HH:mm:ss'));
}
```

---

40) Create an installable time trigger

Does: Runs `dailyJob()` every day at ~9am.
How it works: Installable triggers run under your account.

```
function createDailyTrigger() {
  ScriptApp.newTrigger('dailyJob')
    .timeBased()
    .everyDays(1)
    .atHour(9)
    .create();
}

function dailyJob() {
  Logger.log('Daily job ran at ' + new Date());
}
```

---

41) Remove all triggers for this project

Does: Cleans up triggers (useful while testing).
How it works: Lists triggers then deletes them.

```
function deleteAllTriggers() {
  ScriptApp.getProjectTriggers().forEach(t =>
ScriptApp.deleteTrigger(t));
}
```

---

42) Simple web app (GET)

Does: Returns a plain text response from a deployed web app.
How it works: `doGet(e)` is the entrypoint for GET requests.

```
function doGet(e) {
  return ContentService
    .createTextOutput('Hello from Apps Script web app! Time: ' + new
Date())
    .setMimeType(ContentService.MimeType.TEXT);
}
```

---

43) Web app returning JSON

Does: Returns JSON data for frontend use.
How it works: `setMimeType(JSON)` makes it API-like.

```
function doGet(e) {
  const obj = { ok: true, now: new Date().toISOString() };
  return ContentService
    .createTextOutput(JSON.stringify(obj))
    .setMimeType(ContentService.MimeType.JSON);
}
```

---

44) HTML web app page

Does: Serves an HTML page.
How it works: `HtmlService.createHtmlOutput()` returns HTML.

```
function doGet() {
  const html = `
    <html>
      <body style="font-family:Arial;padding:20px">
        <h2>Apps Script Web App</h2>
        <p>Loaded at ${new Date().toISOString()}</p>
      </body>
    </html>
  `;
  return HtmlService.createHtmlOutput(html).setTitle('Web App');
}
```

---

45) Use PropertiesService (store settings)

Does: Saves and reads a key/value setting.
How it works: Script properties persist across runs.

```
function saveSetting() {
  PropertiesService.getScriptProperties().setProperty('API_KEY',
'demo-key');
}

function readSetting() {
  const key =
PropertiesService.getScriptProperties().getProperty('API_KEY');
  Logger.log('API_KEY: ' + key);
}
```

---

46) Locking to prevent race conditions

Does: Prevents two runs from writing at once.
How it works: `LockService` provides mutex-style locks.

```
function safeAppendWithLock() {
  const lock = LockService.getScriptLock();
  lock.waitLock(20000);

  try {
    const sheet = SpreadsheetApp.getActiveSheet();
    sheet.appendRow([new Date(), 'Safe write']);
  } finally {
    lock.releaseLock();
  }
}
```

---

47) Create a custom function for Sheets

Does: `=DOUBLE(A1)` returns A1*2.
How it works: Custom functions return values and can be used in cells.

```
/**
 * Doubles a number.
 * @param {number} n input number
 * @return {number} doubled
 */
function DOUBLE(n) {
  return Number(n) * 2;
}
```

---

48) Custom function: flatten + unique

Does: `=UNIQUE_FLAT(A1:C10)` returns unique values from a range.
How it works: Takes 2D array input, flattens, filters.

```
/**
 * Returns unique non-empty values from a range.
```

```
 * @param {any[][]} range input range
 * @return {any[]} unique values
 */
function UNIQUE_FLAT(range) {
  const flat = range.flat().map(v => String(v).trim()).filter(v => v);
  return [...new Set(flat)];
}
```

---

49) Add comments to cells

Does: Adds a comment to A1.
How it works: Comments help provide context for collaborators.

```
function addCellComment() {
  const sheet = SpreadsheetApp.getActiveSheet();
  sheet.getRange('A1').setComment('This cell was updated by Apps
Script.');
}
```

---

50) Read / write named ranges

Does: Sets a named range and reads it later.
How it works: Named ranges are stable references even if layout changes.

```
function setAndUseNamedRange() {
  const ss = SpreadsheetApp.getActive();
  const sheet = ss.getActiveSheet();

  const range = sheet.getRange('C2:C5');
  ss.setNamedRange('MyRange', range);

  const named = ss.getRangeByName('MyRange');
  named.setValue('Named range write ✅');
}
```