Script Import JSON

```
/**
 * Retrieves all the rows in the active spreadsheet that contain data and logs the
 * values for each row.
 * For more information on using the Spreadsheet API, see
 * https://developers.google.com/apps-script/service_spreadsheet
 */
function readRows() {
  var sheet = SpreadsheetApp.getActiveSheet();
  var rows = sheet.getDataRange();
  var numRows = rows.getNumRows();
  var values = rows.getValues();

  for (var i = 0; i <= numRows - 1; i++) {
    var row = values[i];
    Logger.log(row);
  }
};

/**
 * Adds a custom menu to the active spreadsheet, containing a single menu item
 * for invoking the readRows() function specified above.
 * The onOpen() function, when defined, is automatically invoked whenever the
 * spreadsheet is opened.
 * For more information on using the Spreadsheet API, see
 * https://developers.google.com/apps-script/service_spreadsheet
 */
function onOpen() {
  var sheet = SpreadsheetApp.getActiveSpreadsheet();
  var entries = [{
    name : "Read Data",
    functionName : "readRows"
  }];
  sheet.addMenu("Script Center Menu", entries);
};

/*============================================================================================================
  ImportJSON by Trevor Lohrbeer (@FastFedora)
  ===========================================================================================================
  Version:      1.1
  Project Page: http://blog.fastfedora.com/projects/import-json
  Copyright:    (c) 2012 by Trevor Lohrbeer
  License:      GNU General Public License, version 3 (GPL-3.0)
                http://www.opensource.org/licenses/gpl-3.0.html
  -----------------------------------------------------------------------------------------------------------
  A library for importing JSON feeds into Google spreadsheets. Functions include:
     ImportJSON            For use by end users to import a JSON feed from a URL
     ImportJSONAdvanced    For use by script developers to easily extend the functionality of this library
  Future enhancements may include:
   - Support for a real XPath like syntax similar to ImportXML for the query parameter
   - Support for OAuth authenticated APIs
  Or feel free to write these and add on to the library yourself!
  -----------------------------------------------------------------------------------------------------------
  Changelog:

  1.1    Added support for the noHeaders option
  1.0    Initial release
 *===========================================================================================================
/**
 * Imports a JSON feed and returns the results to be inserted into a Google Spreadsheet. The JSON feed is flatt
 * a two-dimensional array. The first row contains the headers, with each column header indicating the path to
 * the JSON feed. The remaining rows contain the data.
 *
 * By default, data gets transformed so it looks more like a normal data import. Specifically:
 *
 *   - Data from parent JSON elements gets inherited to their child elements, so rows representing child elemen
 *      of the rows representing their parent elements.
 *   - Values longer than 256 characters get truncated.
 *   - Headers have slashes converted to spaces, common prefixes removed and the resulting text converted to ti
 *
 * To change this behavior, pass in one of these values in the options parameter:
 *
 *    noInherit:     Don't inherit values from parent elements
 *    noTruncate:    Don't truncate values
 *    rawHeaders:    Don't prettify headers
 *    noHeaders:     Don't include headers, only the data
 *    debugLocation: Prepend each value with the row & column it belongs in
 *
 * For example:
 *
 *   =ImportJSON("http://gdata.youtube.com/feeds/api/standardfeeds/most_popular?v=2&alt=json", "/feed/entry/tit
 *               "noInherit,noTruncate,rawHeaders")
 *
 * @param {url} the URL to a public JSON feed
 * @param {query} a comma-separated lists of paths to import. Any path starting with one of these paths gets im
 * @param {options} a comma-separated list of options that alter processing of the data
 *
 * @return a two-dimensional array containing the data, with the first row containing headers
 **/
function ImportJSON(url, query, options) {
  return ImportJSONAdvanced(url, query, options, includeXPath_, defaultTransform_);
}

/**
 * An advanced version of ImportJSON designed to be easily extended by a script. This version cannot be called
 * spreadsheet.
 *
 * Imports a JSON feed and returns the results to be inserted into a Google Spreadsheet. The JSON feed is flatt
 * a two-dimensional array. The first row contains the headers, with each column header indicating the path to
 * the JSON feed. The remaining rows contain the data.
 *
 * Use the include and transformation functions to determine what to include in the import and how to transform
 * imported.
 *
 * For example:
 *
 *   =ImportJSON("http://gdata.youtube.com/feeds/api/standardfeeds/most_popular?v=2&alt=json",
 *               "/feed/entry",
 *                function (query, path) { return path.indexOf(query) == 0; },
 *                function (data, row, column) { data[row][column] = data[row][column].toString().substr(0, 100
 *
 * In this example, the import function checks to see if the path to the data being imported starts with the qu
 * function takes the data and truncates it. For more robust versions of these functions, see the internal code
 *
 * @param {url}           the URL to a public JSON feed
 * @param {query}         the query passed to the include function
 * @param {options}       a comma-separated list of options that may alter processing of the data
 * @param {includeFunc}   a function with the signature func(query, path, options) that returns true if the dat
 *                        should be included or false otherwise.
 * @param {transformFunc} a function with the signature func(data, row, column, options) where data is a 2-dime
 *                        and row & column are the current row and column being processed. Any return value is
 *                        contains the headers for the data, so test for row==0 to process headers only.
 *
 * @return a two-dimensional array containing the data, with the first row containing headers
 **/
function ImportJSONAdvanced(url, query, options, includeFunc, transformFunc) {
  var jsondata = UrlFetchApp.fetch(url);
  var object   = JSON.parse(jsondata.getContentText());

  return parseJSONObject_(object, query, options, includeFunc, transformFunc);
}

/**
 * Encodes the given value to use within a URL.
 *
 * @param {value} the value to be encoded
 *
 * @return the value encoded using URL percent-encoding
 */
function URLEncode(value) {
  return encodeURIComponent(value.toString());
}

/**
 * Parses a JSON object and returns a two-dimensional array containing the data of that object.
 */
function parseJSONObject_(object, query, options, includeFunc, transformFunc) {
  var headers = new Array();
  var data    = new Array();

  if (query && !Array.isArray(query) && query.toString().indexOf(",") != -1) {
    query = query.toString().split(",");
  }

  if (options) {
    options = options.toString().split(",");
  }

  parseData_(headers, data, "", 1, object, query, options, includeFunc);
  parseHeaders_(headers, data);
  transformData_(data, options, transformFunc);

  return hasOption_(options, "noHeaders") ? (data.length > 1 ? data.slice(1) : new Array()) : data;
}

/**
 * Parses the data contained within the given value and inserts it into the data two-dimensional array starting
 * If the data is to be inserted into a new column, a new header is added to the headers array. The value can b
 * array or scalar value.
 *
 * If the value is an object, it's properties are iterated through and passed back into this function with the
 * property extending the path. For instance, if the object contains the property "entry" and the path passed i
 * this function is called with the value of the entry property and the path "/feed/entry".
 *
 * If the value is an array containing other arrays or objects, each element in the array is passed into this f
 * the rowIndex incremeneted for each element.
 *
 * If the value is an array containing only scalar values, those values are joined together and inserted into t
 * a single value.
 *
 * If the value is a scalar, the value is inserted directly into the data array.
 */
function parseData_(headers, data, path, rowIndex, value, query, options, includeFunc) {
  var dataInserted = false;

  if (isObject_(value)) {
    for (key in value) {
      if (parseData_(headers, data, path + "/" + key, rowIndex, value[key], query, options, includeFunc)) {
        dataInserted = true;
      }
    }
  } else if (Array.isArray(value) && isObjectArray_(value)) {
    for (var i = 0; i < value.length; i++) {
      if (parseData_(headers, data, path, rowIndex, value[i], query, options, includeFunc)) {
        dataInserted = true;
        rowIndex++;
      }
    }
  } else if (!includeFunc || includeFunc(query, path, options)) {
    // Handle arrays containing only scalar values
    if (Array.isArray(value)) {
      value = value.join();
    }

    // Insert new row if one doesn't already exist
    if (!data[rowIndex]) {
      data[rowIndex] = new Array();
    }

    // Add a new header if one doesn't exist
    if (!headers[path] && headers[path] != 0) {
      headers[path] = Object.keys(headers).length;
    }

    // Insert the data
    data[rowIndex][headers[path]] = value;
    dataInserted = true;
  }

  return dataInserted;
}

/**
 * Parses the headers array and inserts it into the first row of the data array.
 */
function parseHeaders_(headers, data) {
  data[0] = new Array();

  for (key in headers) {
    data[0][headers[key]] = key;
  }
}

/**
 * Applies the transform function for each element in the data array, going through each column of each row.
 */
function transformData_(data, options, transformFunc) {
  for (var i = 0; i < data.length; i++) {
    for (var j = 0; j < data[i].length; j++) {
      transformFunc(data, i, j, options);
    }
  }
}

/**
 * Returns true if the given test value is an object; false otherwise.
 */
function isObject_(test) {
  return Object.prototype.toString.call(test) === '[object Object]';
}

/**
 * Returns true if the given test value is an array containing at least one object; false otherwise.
 */
function isObjectArray_(test) {
  for (var i = 0; i < test.length; i++) {
    if (isObject_(test[i])) {
      return true;
    }
  }

  return false;
}

/**
 * Returns true if the given query applies to the given path.
 */
function includeXPath_(query, path, options) {
  if (!query) {
    return true;
  } else if (Array.isArray(query)) {
    for (var i = 0; i < query.length; i++) {
      if (applyXPathRule_(query[i], path, options)) {
        return true;
      }
    }
  } else {
    return applyXPathRule_(query, path, options);
  }

  return false;
};

/**
```

```
 * Returns true if the rule applies to the given path.
 */
function applyXPathRule_(rule, path, options) {
  return path.indexOf(rule) == 0;
}

/**
 * By default, this function transforms the value at the given row & column so it looks more like a normal data
 *
 *   - Data from parent JSON elements gets inherited to their child elements, so rows representing child elemen
 *     of the rows representing their parent elements.
 *   - Values longer than 256 characters get truncated.
 *   - Values in row 0 (headers) have slashes converted to spaces, common prefixes removed and the resulting te
*      case.
 *
 * To change this behavior, pass in one of these values in the options parameter:
 *
 *    noInherit:     Don't inherit values from parent elements
 *    noTruncate:    Don't truncate values
 *    rawHeaders:    Don't prettify headers
 *    debugLocation: Prepend each value with the row & column it belongs in
 */
function defaultTransform_(data, row, column, options) {
  if (!data[row][column]) {
    if (row < 2 || hasOption_(options, "noInherit")) {
      data[row][column] = "";
    } else {
      data[row][column] = data[row-1][column];
    }
  }

  if (!hasOption_(options, "rawHeaders") && row == 0) {
    if (column == 0 && data[row].length > 1) {
      removeCommonPrefixes_(data, row);
    }

    data[row][column] = toTitleCase_(data[row][column].toString().replace(/[\/\_]/g, " "));
  }

  if (!hasOption_(options, "noTruncate") && data[row][column]) {
    data[row][column] = data[row][column].toString().substr(0, 256);
  }

  if (hasOption_(options, "debugLocation")) {
    data[row][column] = "[" + row + "," + column + "]" + data[row][column];
  }
}

/**
 * If all the values in the given row share the same prefix, remove that prefix.
 */
function removeCommonPrefixes_(data, row) {
  var matchIndex = data[row][0].length;

  for (var i = 1; i < data[row].length; i++) {
    matchIndex = findEqualityEndpoint_(data[row][i-1], data[row][i], matchIndex);

    if (matchIndex == 0) {
      return;
    }
  }

  for (var i = 0; i < data[row].length; i++) {
    data[row][i] = data[row][i].substring(matchIndex, data[row][i].length);
  }
}

/**
 * Locates the index where the two strings values stop being equal, stopping automatically at the stopAt index.
 */
function findEqualityEndpoint_(string1, string2, stopAt) {
  if (!string1 || !string2) {
    return -1;
  }

  var maxEndpoint = Math.min(stopAt, string1.length, string2.length);

  for (var i = 0; i < maxEndpoint; i++) {
    if (string1.charAt(i) != string2.charAt(i)) {
      return i;
    }
  }

  return maxEndpoint;
}


/**
 * Converts the text to title case.
 */
function toTitleCase_(text) {
  if (text == null) {
    return null;
  }

  return text.replace(/\w\S*/g, function(word) { return word.charAt(0).toUpperCase() + word.substr(1).toLowerCa
}

/**
 * Returns true if the given set of options contains the given option.
 */
function hasOption_(options, option) {
  return options && options.indexOf(option) >= 0;
}
```