/\*\*

\* Programming With JavaScript - QAP2

\*

\*

\* Name: <Maryam Shaheen>

\* Date: <March 29, 2024>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 1: replace all internal whitespace in a string value with underscore

\* ('\_'), and makes it lowercase.

\*

\* We want to be able to convert a string to Lower Snake Case style, so that all

\* leading/trailing whitespace is removed, and any internal spaces, tabs, or dots,

\* are converted to '\_' and all letters are lower cased.

\*

\* The snake() function should work like this:

\*

\* snake('abc') --> returns 'abc'

\* snake(' ABC ') --> returns 'abc'

\* snake('ABC') --> returns 'abc'

\* snake('A BC') --> returns 'a\_bc'

\* snake(' A bC ') --> returns 'a-bc'

\* snake('A BC') --> returns 'a\_bc'

\* snake('A.BC') --> returns 'a\_bc'

\* snake(' A.. B C ') --> returns 'a\_b\_c'

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function snake(str) {

// Step 1: Remove leading and trailing whitespace, then convert to lowercase

str = str.trim().toLowerCase();

// Step 2: Replace internal whitespace, tabs, and dots with underscores

str = str.replace(/[ \t.]+/g, '\_');

// Step 3: Return the modified string

return str;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 2: create an HTML <video> element for the given url.

\*

\* In HTML, we use markup syntax to indicate how browsers should format elements

\* of a web page. For example, here is a URL to video:

\*

\* http://distribution.bbb3d.renderfarming.net/video/mp4/bbb\_sunflower\_1080p\_60fps\_normal.mp4

\*

\* Try navigating to it in your browser. In order for a web page to know how to

\* interpret this URL (e.g., should we show the text of the URL itself, or use

\* it to load an image? or a video?), we have to use special markup. The <video>

\* tag is how we specify that a URL is to be interpreted as a video, see:

\* https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video

\*

\* Here is how we might use HTML to markup this video:

\*

\* <video src="http://distribution.bbb3d.renderfarming.net/video/mp4/bbb\_sunflower\_1080p\_60fps\_normal.mp4" width="500"></video>

\*

\* Our <video> has two attributes:

\*

\* - src: the URL to a video file

\* - width: the width to use (in pixels) when showing the video

\*

\* Write the createVideo() function to accept a URL and width, and return the

\* properly formatted video element. For example:

\*

\* createVideo('http://distribution.bbb3d.renderfarming.net/video/mp4/bbb\_sunflower\_1080p\_60fps\_normal.mp4', 500)

\*

\* should return the following string of HTML:

\*

\* '<video src="http://distribution.bbb3d.renderfarming.net/video/mp4/bbb\_sunflower\_1080p\_60fps\_normal.mp4" width="500"></video>'

\*

\* A <video> can also optionally contain a `controls` attribute, which turns on the play/pause/etc controls. For example:

\*

\* <video src="http://distribution.bbb3d.renderfarming.net/video/mp4/bbb\_sunflower\_1080p\_60fps\_normal.mp4" width="500" controls></video>

\*

\* In this case, the <video> element should include the user controls. Therefore,

\* your createVideo() function should also accept a third argument, controls:

\*

\* // No controls

\* createVideo('http://distribution.bbb3d.renderfarming.net/video/mp4/bbb\_sunflower\_1080p\_60fps\_normal.mp4', 500)

\* // With controls

\* createVideo('http://distribution.bbb3d.renderfarming.net/video/mp4/bbb\_sunflower\_1080p\_60fps\_normal.mp4', 500, true)

\*

\* The returned <video> HTML string should be formatted as follows:

\*

\* - Remove leading/trailing whitespace from src before you use them

\* - The src and width attribute values should be wrapped in double-quotes (e.g., src="..." width="...")

\* - There should be a single space between the end of one attribute and start of the next (e.g., src="..." width="..." controls)

\* - The width attribute should only be added if a valid integer value (number or string) is included. Otherwise ignore it.

\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function createVideo(src, width, controls) {

// Remove leading/trailing whitespace from src

src = src.trim();

// Initialize the video HTML string with the src attribute

let videoHTML = `<video src="${src}"`;

// Add width attribute if valid width is provided

if (!isNaN(width)) {

videoHTML += ` width="${width}"`;

}

// Add controls attribute if controls is true

if (controls === true) {

videoHTML += ' controls';

}

// Close the video tag

videoHTML += '></video>';

return videoHTML;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 3: extract Date from date string

\*

\* A date string is expected to be formatted as follows:

\*

\* YYYY-MM-DD

\*

\* Meaning, Year (4 digits), Month (2 digits), Day (2 digits).

\*

\* January 1, 2021 would therefore be the following date string:

\*

\* 2021-01-01

\*

\* Similarly, September 29, 2021 would be:

\*

\* 2021-09-29

\*

\* Write a function, parseDateString() that accepts a date string of the format

\* specified above, and returns a JavaScript Date object, set to the correct day.

\* In your solution, you are encouraged to use the following Date methods:

\*

\* - setFullYear()

\* - setMonth()

\* - setDate()

\*

\* To help developers using your function, you are asked to provide detailed error

\* messages when the date string is formatted incorrectly. We will use the `throw`

\* statement to throw an Error object when a particular value is not what we expect, see:

\* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw

\*

\* For example: parseDateString('01-01-01') should fail, because the year is

\* not 4 digits.

\*

\* Similarly, parseDateString('2021-1-01') should fail because

\* the day is not 2 digits, and parseDateString('2021-01-1') should fail because

\* the month is not 2 digits.

\*

\* Also, a totally invalid date string should also

\* cause an exception to be thrown, for example parseDateString(null) or

\* parseDateString("this is totally wrong").

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function parseDateString(value) {

// Check if the input value is a string

if (typeof value !== 'string') {

throw new Error('Invalid date string: Input must be a string.');

}

// Parse the date string

const parts = value.split('-');

// Check if the date string has 3 parts (year, month, day)

if (parts.length !== 3) {

throw new Error('Invalid date string: Date must be in the format YYYY-MM-DD.');

}

const year = parseInt(parts[0]);

const month = parseInt(parts[1]);

const day = parseInt(parts[2]);

// Check if year, month, and day are valid numbers

if (isNaN(year) || isNaN(month) || isNaN(day)) {

throw new Error('Invalid date string: Year, month, and day must be numbers.');

}

// Check if year, month, and day have correct lengths

if (parts[0].length !== 4 || parts[1].length !== 2 || parts[2].length !== 2) {

throw new Error('Invalid date string: Year must be 4 digits, and month/day must be 2 digits.');

}

// Check if month is between 1 and 12

if (month < 1 || month > 12) {

throw new Error('Invalid date string: Month must be between 01 and 12.');

}

// Check if day is valid for the given month and year

const lastDayOfMonth = new Date(year, month, 0).getDate();

if (day < 1 || day > lastDayOfMonth) {

throw new Error(`Invalid date string: Day must be between 01 and ${lastDayOfMonth}.`);

}

// Create a Date object with the provided year, month, and day

const date = new Date(year, month - 1, day); // Month is 0-based in Date objects

return date;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 4: convert Date to date string with specified format.

\*

\* As above, a date string is expected to be formatted as follows:

\*

\* YYYY-MM-DD

\*

\* Meaning, Year (4 digits), Month (2 digits), Day (2 digits).

\*

\* Write a function, toDateString() that accepts a Date object, and returns a

\* date string formatted according to the specification above. Make sure your

\* day and month values are padded with a leading '0' if necessary (e.g., 03 vs. 3).

\*

\* In your solution, you are encouraged to use the following Date methods:

\*

\* - setFullYear()

\* - setMonth()

\* - setDate()

\*

\* NOTE: it should be possible to use parseDateString() from the previous question

\* and toDateString() to reverse each other. For example:

\*

\* toDateString(parseDateString('2021-01-29)) should return '2021-01-29'

\*

\* If an invalid Date is passed, throw an Error object with an appropriate error message.

\* HINT: use try/catch, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function toDateString(value) {

try {

// Check if the input value is a Date object

if (!(value instanceof Date) || isNaN(value.getTime())) {

throw new Error('Invalid Date object: Input must be a valid Date object.');

}

// Get year, month, and day from the Date object

const year = value.getFullYear();

const month = String(value.getMonth() + 1).padStart(2, '0'); // Month is zero-based, so we add 1

const day = String(value.getDate()).padStart(2, '0');

// Construct the date string in the format 'YYYY-MM-DD'

const dateString = `${year}-${month}-${day}`;

return dateString;

} catch (error) {

throw new Error('Invalid Date object: Input must be a valid Date object.');

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 5: parse a geographic coordinate

\*

\* Coordinates are defined as numeric, decimal values of Longitude and Latitude.

\* A example, let's suppose the Keyin College St.John's campus is located at:

\*

\* Longitude: -77.4369 (negative number means West)

\* Latitude: 42.9755 (positive number means North)

\*

\* A dataset includes thousands of geographic coordinates, stored as strings.

\* However, over the years, different authors have used slightly different formats.

\* All of the following are valid and need to be parsed:

\*

\* 1. "42.9755,-77.4369" NOTE: no space

\* 4. "[-77.4369, 42.9755]" NOTE: the space, as well as lat/lng values are reversed

\*

\* Valid Longitude values are decimal numbers between -180 and 180.

\*

\* Valid Latitude values are decimal numbers between -90 and 90.

\*

\* Parse the value and reformat it into the form: "(lat, lng)"

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function normalizeCoord(value) {

// Regular expression to match latitude and longitude in different formats

const regex = /\[?\s\*(-?\d+\.\d+)\s\*,\s\*(-?\d+\.\d+)\s\*\]?/;

// Extract latitude and longitude using regular expression

const match = value.match(regex);

// Check if the match is found and has valid latitude and longitude

if (!match || isNaN(match[1]) || isNaN(match[2])) {

// Throw an error if the coordinate format is invalid or latitude/longitude cannot be parsed

throw new Error('Invalid coordinate format: Unable to parse latitude and longitude.');

}

// Extract latitude and longitude from the match

const latitude = parseFloat(match[2]);

const longitude = parseFloat(match[1]);

// Validate latitude and longitude ranges

if (latitude < -90 || latitude > 90 || longitude < -180 || longitude > 180) {

// Throw an error if latitude or longitude is out of range

throw new Error('Invalid coordinate values: Latitude must be between -90 and 90, and Longitude must be between -180 and 180.');

}

// Reformat the coordinate into the form "(lat, lng)"

const formattedCoordinate = `(${latitude}, ${longitude})`;

return formattedCoordinate;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 6: format any number of coordinates as a list in a string

\*

\* You are asked to format geographic lat, lng coordinates in a list using your

\* normalizeCoord() function from problem 5.

\*

\* Where normalizeCoord() takes a single geographic coord, the formatCoords()

\* function takes a list of any number of geographic coordinates, parses them,

\* filters out any invalid coords, and creates a list.

\*

\* For example: given the following coords, "42.9755,-77.4369" and "[-62.1234, 42.9755]",

\* a new list would be created of the following form "((42.9755, -77.4369), (42.9755, -62.1234))".

\*

\* Notice how the list has been enclosed in an extra set of (...) braces, and each

\* formatted geographic coordinate is separated by a comma and space.

\*

\* The formatCoords() function can take any number of arguments, but they must all

\* be strings. If any of the values can't be parsed by normalizeCoord() (i.e., if

\* an Error is thrown), skip the value. For example:

\*

\* formatCoords("42.9755,-77.4369", "[-62.1234, 42.9755]", "300,-9000") should return

\* "((42.9755, -77.4369), (42.9755, -62.1234))" and skip the invalid coordinate.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function formatCoords(...values) {

// Initializing an empty array to store valid formatted coordinates

const formattedCoordinates = [];

// Starting iteration through each value passed as arguments

for (const value of values) {

try {

// Parsing the coordinate using the normalizeCoord function

const coord = normalizeCoord(value);

// If parsing is successful, push the formatted coordinate to the array

formattedCoordinates.push(coord);

} catch (error) {

// If an error is thrown (invalid coordinate), skip this value

continue;

}

}

// Formatting the array of valid coordinates into a list string

const listString = `(${formattedCoordinates.join(', ')})`;

return listString;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 7: determine MIME type from filename extension

\*

\* Web browsers and servers exchange streams of bytes, which must be interpreted

\* by the receiver based on their type. For example, an HTML web page is

\* plain text, while a JPG image is a binary sequence.

\*

\* The Content-Type header contains information about a resource's MIME type, see:

\* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type

\*

\* The MIME type is made-up of a `type` and a `subtype` separated by a `/` character.

\* The type is general, for example, 'text' or 'video'. The subtype is more

\* specific, indicating the specific type of text, image, video, etc. See:

\* https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics\_of\_HTTP/MIME\_types

\*

\* A number of common types are used in web development:

\*

\* mimeFromFilename('/User/Documents/readme.txt') --> returns 'text/plain'

\*

\* Your mimeFromFilename() function should support all of the following extensions

\* with and without the leading '.':

\*

\* - .html, .htm --> text/html

\* - .css --> text/css

\* - .js --> text/javascript

\* - .jpg, .jpeg --> image/jpeg

\* - .gif --> image/gif

\* - .bmp --> image/bmp

\* - .ico, .cur --> image/x-icon

\* - .png --> image/png

\* - .svg --> image/svg+xml

\* - .webp --> image/webp

\* - .mp3 --> audio/mp3

\* - .wav --> audio/wav

\* - .mp4 --> video/mp4

\* - .webm --> video/webm

\* - .json --> application/json

\* - .mpeg --> video/mpeg

\* - .csv --> text/csv

\* - .ttf --> font/ttf

\* - .woff --> font/woff

\* - .zip --> application/zip

\* - .avi --> video/x-msvideo

\*

\*

\* NOTE: any other extension should use the `application/octet-stream` MIME type,

\* to indicate that it is an unknown stream of bytes (e.g., binary file). You should

\* also use `application/octet-stream` if the file has no extension.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function mimeFromFilename(filename) {

// Extract the file extension from the filename

const extension = filename.split('.').pop().toLowerCase();

// Use a switch statement to handle different file extensions

switch (extension) {

case 'html':

case 'htm':

return 'text/html';

case 'css':

return 'text/css';

case 'js':

return 'text/javascript';

case 'jpg':

case 'jpeg':

return 'image/jpeg';

case 'gif':

return 'image/gif';

case 'bmp':

return 'image/bmp';

case 'ico':

case 'cur':

return 'image/x-icon';

case 'png':

return 'image/png';

case 'svg':

return 'image/svg+xml';

case 'webp':

return 'image/webp';

case 'mp3':

return 'audio/mp3';

case 'wav':

return 'audio/wav';

case 'mp4':

return 'video/mp4';

case 'webm':

return 'video/webm';

case 'json':

return 'application/json';

case 'mpeg':

return 'video/mpeg';

case 'csv':

return 'text/csv';

case 'ttf':

return 'font/ttf';

case 'woff':

return 'font/woff';

case 'zip':

return 'application/zip';

case 'avi':

return 'video/x-msvideo';

default:

// For unknown extensions or files with no extension, return 'application/octet-stream'

return 'application/octet-stream';

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 8, Part 1: generate license text and link from license code.

\*

\* Images, videos, and other resources on the web are governed by copyright.

\* Everything you find on the web is copyright to its creator automatically, and

\* you cannot reuse it unless you are granted a license to do so.

\*

\* Different licenses exist to allow creators to share their work. For example,

\* the Creative Commons licenses are a popular way to allow people to reuse

\* copyright material, see https://creativecommons.org/licenses/.

\*

\* Below is a list of license codes, and the associated license text explaining the code:

\*

\* CC-BY: Creative Commons Attribution License

\* CC-BY-NC: Creative Commons Attribution-NonCommercial License

\* CC-BY-SA: Creative Commons Attribution-ShareAlike License

\* CC-BY-ND: Creative Commons Attribution-NoDerivs License

\* CC-BY-NC-SA: Creative Commons Attribution-NonCommercial-ShareAlike License

\* CC-BY-NC-ND: Creative Commons Attribution-NonCommercial-NoDerivs License

\*

\* NOTE: any other licenseCode should use the URL https://choosealicense.com/no-permission/

\* and the explanation text, "All Rights Reserved"

\*

\* Write a function, generateLicenseLink(), which takes a license code, and returns

\* an HTML link to the appropriate license URL, and including the explanation text.

\*

\* For example:

\*

\* generateLicenseLink('CC-BY-NC') should return the following HTML string:

\*

\* '<a href="https://creativecommons.org/licenses/by-nc/4.0/">Creative Commons Attribution-NonCommercial License</a>'

\*

\* The URL is generated based on the license code:

\*

\* - remove the `CC-` prefix

\* - convert to lower case

\* - place formatted license code within URL https://creativecommons.org/licenses/[...here]/4.0/

\*

\* Your generateLicenseLink() function should also accept an optional second argument,

\* `targetBlank`. If `targetBlank` is true, add ` target="\_blank"` to your link

\* so that the URL opens in a blank tab/window.

\*

\* You can read more about HTML links at https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function generateLicenseLink(licenseCode, targetBlank = false) {

// Define an object mapping license codes to their respective license texts

const licenseMap = {

'CC-BY': 'Creative Commons Attribution License',

'CC-BY-NC': 'Creative Commons Attribution-NonCommercial License',

'CC-BY-SA': 'Creative Commons Attribution-ShareAlike License',

'CC-BY-ND': 'Creative Commons Attribution-NoDerivs License',

'CC-BY-NC-SA': 'Creative Commons Attribution-NonCommercial-ShareAlike License',

'CC-BY-NC-ND': 'Creative Commons Attribution-NonCommercial-NoDerivs License'

};

// Generate the URL based on the license code

const baseUrl = 'https://creativecommons.org/licenses/';

const url = baseUrl + licenseCode.toLowerCase().replace('cc-', '') + '/4.0/';

// Get the license text based on the license code

const licenseText = licenseMap[licenseCode] || 'All Rights Reserved';

// Create the HTML link

const link = `<a href="${url}"${targetBlank ? ' target="\_blank"' : ''}>${licenseText}</a>`;

return link;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 9 Part 1: convert a value to a Boolean (true or false)

\*

\* A dataset contains fields that indicate a value is true or false. However,

\* users have entered data in various formats and languages (English and French)

\* over the years, and the data is a mess. For example, the dataset contains all

\* of the following values:

\*

\* Yes, yes, YES, Y, Oui, oui, OUI, O, t, TRUE, true, True, vrai, V, VRAI, 1, 2, ...any positive number

\* No, no, NO, Non, non, NON, N, n, f, FALSE, false, False, FAUX, faux, Faux, 0, -1, -2, ...any negative number

\*

\* Write a function pureBool() which takes a String, Number, or Boolean,

\* and attempts to convert it into a pure Boolean value, according to these rules:

\*

\* 1. If the value is already a Boolean (true or false) return the value without conversion

\* 2. If the value is one of the "true" type values, return `true` (Boolean)

\* 3. If the value is one of the "false" type values, return `false` (Boolean)

\* 4. If the value is none of the "true" or "false" values, throw an error with the error

\* message, 'invalid value'.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function pureBool(value) {

// If the value is already a Boolean, return it without conversion

if (typeof value === 'boolean') {

return value;

}

// Convert the value to lowercase if it's a string

if (typeof value === 'string') {

value = value.toLowerCase();

}

// List of true type values

const trueValues = ['yes', 'oui', 'o', 't', 'true', 'vrai', 'v', '1'];

// List of false type values

const falseValues = ['no', 'non', 'n', 'f', 'false', 'faux', '0'];

// Check if the value is in the trueValues list

if (trueValues.includes(value)) {

return true;

}

// Check if the value is in the falseValues list

if (falseValues.includes(value)) {

return false;

}

// If the value is neither true nor false, throw an error

throw new Error('invalid value');

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 9 Part 2: checking for all True or all False values in a normalized list

\*

\* Using your pureBool() function above, create three new functions to check

\* if a list of arguments are all "true", partially "true", or all "false" values:

\*

\* every('Y', 'yes', 1) --> returns true

\* any('Y', 'no', 1) --> returns true

\* none('Y', 'invalid', 1) --> returns false

\*

\* Use try/catch syntax to avoid having your functions throw errors when pureBool()

\* throws on invalid data.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function every() {

try {

return args.every(arg => pureBool(arg) === true);

} catch (error) {

return false; // Return false if any error occurs

}

}

function any() {

try {

// Check if any argument, when converted to Boolean, evaluates to true

return args.any(arg => pureBool(arg) === true);

} catch (error) {

return false; // Return false if any error occurs

}

}

function none() {

try {

// Check if none of the arguments, when converted to Boolean, evaluate to true

return args.every(arg => pureBool(arg) === false);

} catch (error) {

return false; // Return false if any error occurs

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Problem 10 - build a URL

\*

\* Querying the iNaturalist Web API (https://api.inaturalist.org/v2/observations)

\* for data involves formatting a URL in a particular way. As we know might know, a URL can contain optional name=value pairs. For reference: read query strings on web :)

\*

\* For example:

\*

\* https://www.store.com/search?q=dog includes q=dog

\*

\* https://www.store.com?\_encoding=UTF8&node=18521080011 includes

\* both \_encoding=UTF8 and also node=18521080011, separated by &

\*

\* We will write a buildUrl() function to build a URL for the iNaturalist API

\* based on arguments passed by the caller.

\*

\* The buildUrl() function accepts the following arguments:

\*

\* - query: a URI encoded search string, for example "butterfly" or "Horse-chestnut"

\* - order: a string indicating sort order, with possible values of `ascending` or `descending`

\* - count: a number from 1 to 50, indicating how many results to return per page

\* - license: a string indicating which items to return (e.g., which license they use). Possible

\* values include: none, any, cc-by, cc-by-nc, cc-by-sa, cc-by-nd, cc-by-nc-sa, cc-by-nc-nd

\*

\* Write an implementation of buildUrl() that accepts arguments for all of the above

\* parameters, validates them (e.g., count must be between 1 and 50, etc), and returns

\* a properly formatted URL.

\*

\* For example:

\*

\* buildUrl('Monarch Butterfly', 'ascending', 25, 'cc-by') would return the following URL:

\*

\* https://api.inaturalist.org/v2/observations?query='Monarch%20Butterfly'&count=25&order=ascending&license=cc-by

\*

\* NOTE: if any of the values passed to buildUrl() are invalid, an Error should be thrown.

\*

\* NOTE: make sure you properly encode the query value, since URLs can't contain

\* spaces or other special characters. HINT: use the encodeURIComponent() function

\* to do this, see:

\*

\* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/encodeURIComponent

\*

\* The following might be the parameters

\*

\* "query" the query to use. Must be properly URI encoded

\* "order" the sort order to use, must be one of `ascending` or `descending`

\* "count" the number of results per page, must be 1-50

\* "license" the license to use, must be one of none, any, cc-by, cc-by-nc, cc-by-sa, cc-by-nd, cc-by-nc-sa, cc-by-nc-nd

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function buildUrl(query, order, count, license) {

function buildUrl(query, order, count, license) {

// Validate count parameter

if (count < 1 || count > 50 || !Number.isInteger(count)) {

throw new Error('Count must be an integer between 1 and 50');

}

// Validate order parameter

if (order !== 'ascending' && order !== 'descending') {

throw new Error('Order must be either "ascending" or "descending"');

}

// Validate license parameter

const validLicenses = ['none', 'any', 'cc-by', 'cc-by-nc', 'cc-by-sa', 'cc-by-nd', 'cc-by-nc-sa', 'cc-by-nc-nd'];

if (!validLicenses.includes(license)) {

throw new Error('Invalid license');

}

// Encode the query parameter

const encodedQuery = encodeURIComponent(query);

// Construct the URL

const baseUrl = 'https://api.inaturalist.org/v2/observations';

const queryString = `query=${encodedQuery}&count=${count}&order=${order}&license=${license}`;

const url = `${baseUrl}?${queryString}`;

return url;

}

}

//returns the properly formatted iNaturlist URL

// Tested the buildUrl function below:

try {

const url = buildUrl('Monarch Butterfly', 'ascending', 25, 'cc-by');

console.log(url);

} catch (error) {

console.error(error.message);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/