```
/**
 * WEB222 – Assignment 1
 *
 * I declare that this assignment is my own work in accordance with
 * Seneca Academic Policy. No part of this assignment has been copied
 * manually or electronically from any other source (including web sites)
 * or distributed to other students.
 *
 * Please update the following with your information:
 *
 *      Name: <YOUR_NAME>
 *      Student ID: <YOUR_STUDENT_ID>
 *      Date: <SUBMISSION_DATE>
 *
 * Please see all unit tests in the files problem-0.test.js, problem-1.test.js,
etc.
 */

/*******************************************************************************
 * Problem 0: learn how to implement code to pass unit tests.
 *
 * Welcome to Assignment 1! In this assignment, you're going to be practicing
lots
 * of new JavaScript programming techniques.  Before you dive in let's spend a
 * minute helping you learn how to read this code, and how to understand the
 * tests that go with it.
 *
 * In addition to this file, please also open the src/problem-0.test.js file.
 * Start by reading the comment at the top of that file, then come back here and
 * continue.
 *
 * Make sure you have completed the necessary Setup (install node.js, run `npm
install`
 * before continuing).  The rest of the instructions assume that you have done
this.
 *
 * After you finish reading src/problem-0.test.js, it's time to try running
 * the tests.  To run the tests, go to your terminal and type the following
command:
 *
 *   npm test
 *
 * You have to run this command in the root of your project (i.e., in the same
 * directory as src/ and package.json).  When you do, you will see a lot of
failures.
 * That's expected, since we haven't written any code below.
 *
 * You can also run tests for only this problem instead of everything.  To do
that:
 *
 *   npm test problem-0
 *
 * This will look for tests that are part of the problem-0.test.js file, and
only
 * run those.  Doing so should result in 1 failed and 1 passed test.
 *
 * The first test passes:
 *
 *  ✓ greeting should be a function (2ms)
 *
 * But the second one failed:
 *
 * × greeting should  return the correct string output (3ms)
 *
 * ● Problem 0 - greeting() function › greeting should  return the correct
string output
 *
 *   expect(received).toBe(expected) // Object.is equality
 *
 *   Expected: "Hello WEB222 Student!"
 *   Received: "Hello WEB222 Student"
 *
 *     63 |   test('greeting should return the correct string output',
function() {
 *     64 |     let result = greeting('WEB222 Student');
 *   > 65 |     expect(result).toBe('Hello WEB222 Student!');
 *        |                    ^
 *     66 |   });
 *     67 |
 *     68 |   /**
 *
 * We can see that the test 'greeting should say "Hello Name!"' is failing.
 * It's failing on line 65 of src/problem-0.test.js.  In particular, it's
failing
 * because it expected to get the String "Hello WEB222 Student!" but instead
 * it actually received the String "Hello WEB222 Student".
 *
 * It looks like we have a small typo in our code below, and we are missing
 * the final ! character.  Try adding it below, save this file, and re-run the
 * tests again:
 *
 * npm test problem-0
 * PASS  src/problem-0.test.js
 *  Problem 0 - greeting() function
 *   ✓ greeting should be a function (2ms)
 *   ✓ greeting should return the correct string output
 *
 * Test Suites: 1 passed, 1 total
 * Tests:       2 passed, 2 total
 *
 * Excellent! At this point you're ready to move on to Problem 1. As you do,
 * read both the information in the Problem's comment, and also read the tests
 * to understand what they expect from your code's implementation.
 *
 * One final word about these comments.  You'll see comments like this one:
 *
 * @param {String} name - the name to greet in the message
 *
 * These are specially formatted comments that define parameters to functions,
 * and tell use the type {String} or {Number}, and also the parameter's name.
 * Finally, we also explain a bit about how the parameter is used, and what
 * data it will have, whether it's optional, etc.
 ******************************************************************************/

function greeting(name) {
  return `Hello ${name}!`;
}

/*******************************************************************************
 * Problem 1: remove all spaces and dashes from a String value, and make it
lowercase.
 *
 * We want to be able to "crush" a string so that it contains no spaces or
dashes,
 * and all letters are lower case.  The crush() function should work like this:
 *
 * crush('ABC') --> returns 'abc' (all lowercase)
 * crush('abc') --> returns 'abc' (the string was already correct, same value)
 * crush('A BC') --> returns 'abc' (lowercase, space removed)
 * crush('a b- c') --> returns 'abc' (lowercase, spaces removed, dash removed)
 *
 * @param {String} value - a string to be crushed
 ******************************************************************************/

function crush(value) {
  value = value.toLowerCase();
  value = value.match(/[a-za-za-z]/g).join('');
  return value;
}

/*******************************************************************************
 * Problem 2: create an email address for the given user info.
 *
 * For example, Jeff Sickel uses the email host myseneca.ca for his email, and
 * we'd like to format his email address like this:
 *
 * createEmailAddress('Jeff', 'Sickel', 'myseneca.ca')
 *
 * This should return the following string:
 *
 *   Jeff Sickel <jsickel@myseneca.ca>
 *
 * The returned email address should be formatted as follows:
 *
 * - Combine the lowercase first letter of the first name with the crushed last
name
 *   (i.e., use the crush() function on the last name).
 * - Add an '@' symbol and the host. If the host is missing, use 'gmail.com' by
default
 *
 * @param {String} firstName - the person's first name
 * @param {String} lastName - the peron's last name
 * @param {String} host - (optional) the email host.  If missing, use
'gmail.com'
 ******************************************************************************/

function createEmailAddress(firstName, lastName, host) {
  let firstA = firstName.charAt(0);
  firstA = firstA.toLowerCase();
  let lastNa = lastName.toLowerCase();
  lastNa = lastNa.match(/[a-za-za-za-za-za-za-za-za-za-za-za-za-za-za-
z]/g).join('');
  if (host == undefined) {
    host = 'gmail.com';
  }
  let at = '@';
  let email = firstA + lastNa + at + host;
  return firstName + ' ' + lastName + ' <' + email + '>';
}

/*******************************************************************************
 * Problem 3: extract the Forward Sortation Area (FSA) from a Postal Code
 *
 * The first three letters of a Canadian Postal Code are known as a forward
 * sortation area, and designate a geographic unit.  See details at:
 *
 * https://www.ic.gc.ca/eic/site/bsf-osb.nsf/eng/br03396.html and
 *
https://en.wikipedia.org/wiki/Postal_codes_in_Canada#Components_of_a_postal_code
 *
 * Given a postal code, you are asked to extract the FSA.  If the resulting FSA
 * is invalid, you should throw a new Error, otherwise return the extracted FSA.
 *
 * For example:
 *
 * extractFSA('M2J 2X5') --> returns 'M2J' (valid FSA for Newnham Campus)
 * extractFSA('2MJ X25') --> throws an Error with the message, 'invalid FSA'
 *
 * See https://rollbar.com/guides/javascript-throwing-exceptions/ for examples
 * of how to throw a generic Error.
 *
 * A valid FSA is defined as:
 *
 * - first character is a letter, one of A, B, C, E, G, H, J, K, L, M, N, P, R,
S, T, V, X, Y (not D, F, I, O, Q, U, W)
 * - second character is a digit, one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
 * - third character is any letter (A to Z)
 *
 * @param {String} postalCode - a postal code
 ******************************************************************************/

function extractFSA(postalCode) {
  if (postalCode.slice(0, 3).match(/[ABCEGHJKLMNPRSTVXY]\d[A-Z]/gim)) {
    return postalCode.slice(0, 3);
  } else {
    throw 'invalid FSA';
  }
}

/*******************************************************************************
 * Problem 4: convert an Age (number) to an Age Range Category (string)
 *
 * For statistical reasons, we are interested in taking literal ages like 21,
23, 26, etc
 * and converting them to the string '20 to 29 years' so that we can do
grouping.
 *
 * The ageToRange() function takes an age as a Number and returns the
appropriate
 * age range category string:
 *
 * ageToRange(21) --> returns '20 to 29 years'
 * ageToRange(29) --> returns '20 to 29 years'
 * ageToRange(30) --> returns '30 to 39 years'
 *
 * The age range categories you need to support are as follows:
 *
 * '19 and younger',
 * '20 to 29 Years',
 * '30 to 39 Years',
 * '40 to 49 Years',
 * '50 to 59 Years',
 * '60 to 69 Years',
 * '70 to 79 Years',
 * '80 to 89 Years',
 * '90 and older'
 *
 * Your ageToRange() function should accept age Numbers from 0 to 125.  If the
 * age passed to your function is outside 0 to 125, throw an Error with the
message
 * 'invalid age'.
 *
 * @param {Number} age - the age of the person
 ******************************************************************************/

function ageToRange(age) {
  if (age >= 0 && age <= 125) {
    for (let i = 0; i <= 19; i++) {
      if (age === i) {
        return '19 and younger';
      }
    }
    for (let i = 20; i <= 29; i++) {
      if (age === i) {
        return '20 to 29 Years';
      }
    }
    for (let i = 30; i <= 39; i++) {
      if (age === i) {
        return '30 to 39 Years';
      }
    }
    for (let i = 40; i <= 49; i++) {
      if (age === i) {
        return '40 to 49 Years';
      }
    }
    for (let i = 50; i <= 59; i++) {
      if (age === i) {
        return '50 to 59 Years';
      }
    }
    for (let i = 60; i <= 69; i++) {
      if (age === i) {
        return '60 to 69 Years';
      }
    }
    for (let i = 70; i <= 79; i++) {
      if (age === i) {
        return '70 to 79 Years';
      }
    }
    for (let i = 80; i <= 89; i++) {
      if (age === i) {
        return '80 to 89 Years';
      }
    }
    for (let i = 90; i <= 125; i++) {
      if (age === i) {
        return '90 and older';
      }
    }
  } else {
    throw 'invalid age';
  }
}

/*******************************************************************************
 * Problem 5: convert and validate Date Strings
 *
 * A dataset contains dates stored as Strings in the following form:
 *
 * YYYY-MM-DD
 *
 * For example, March 12, 2020 would be "2020-03-12".
 *
 * The toDate() function takes a string argument, and tries to return a Date.
 * It first checks to see if the date string is in the proper YYYY-MM-DD format.
 * If it is not, an Error is thrown with the error message 'invalid date'.
 *
 * Otherwise, the year, month, and day are extracted from the string, and a new
 * Date Object is created.  Use the Date Object's setFullYear(), setMonth(), etc
 * methods to set the appropriate values for your date before returning it.
 *
 * NOTE: if the date string value is null or undefined, return the current date.
 *
 * @param {String} value - a date string
 ******************************************************************************/

function toDate(value) {
  var regEx = /^\d{4}-\d{2}-\d{2}$/;
  if (regEx.test(value) || !value) {
    if (regEx.test(value)) {
      //var newDate = new Date(value);
      let result = value.split('-');
      var d = new Date();
      d.setFullYear(parseInt(result[0]));
      d.setMonth(parseInt(result[1]) - 1);
      d.setDate(parseInt(result[2]));
      return d;
    } else {
      var newDate = new Date();
      return newDate;
    }
  } else {
    throw 'invalid date';
  }
}

/*******************************************************************************
 * Problem 6: moving average calculations
 *
 * A dataset contains fluctuating data points, and we'd like to be able to
smooth
 * out the variation by showing trends for a period.  To do that, we need to
 * calculate a moving average.  There are various types of moving averages, but
 * we'll focus on Simple Moving Average (SMA):
 *
 * simpleMovingAverage(20, 22, 24, 25, 23) --> returns 22.8
 * simpleMovingAverage(20, 22) --> returns 21
 * simpleMovingAverage(20) --> returns 20
 *
 * The simpleMovingAverage() function accepts any number of arguments and
 * calculates their average.
 *
 * Further, the simpleMovingAverage() function can accept both Numbers and
 * numbers formatted as Strings:
 *
 * simpleMovingAverage('20', '22', '24', '25', '23') --> returns 22.8
 * simpleMovingAverage(20, '22') --> returns 21
 * simpleMovingAverage(20) --> returns 20
 *
 * Passing in an argument that is NOT a Number or a number formatted as a String
 * should cause an error to be thrown with the error message, 'invalid
argument'.
 *
 * simpleMovingAverage('20', '', '24', '25', '23') --> throws, since '' is
invalid
 * simpleMovingAverage(20, true) --> throws, since true is invalid
 * simpleMovingAverage(20, 'twenty') --> throws, since 'twenty' is invalid
 *
 * @params {Number|String} - any number of arguments are accepted.
 ******************************************************************************/
```

```
function simpleMovingAverage() {
  let count;
  let total = 0;
  for (let j = 0; j < arguments.length; j++) {
    if (parseFloat(arguments[j]).toString() == 'NaN') {
      throw new Error('invalid argument');
    }
  }
  for (let i = 0; i < arguments.length; i++) {
    total += Number(arguments[i]);
  }
  let average = total / arguments.length;
  return average;
}

/*******************************************************************************
 * Problem 7: determine MIME type from file 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:
 *
 * mimeFromExt('.txt') --> returns 'text/plain'
 * mimeFromExt('txt') --> also returns 'text/plain' (note: leading . is missing)
 *
 * Your mimeFromExt() 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
 *
 * NOTE: any other extension should use the `application/octet-stream` type,
 * to indicate that it is an unknown stream of bytes.
 *
 * @params {String} - a file extension, either with or without the leading '.'
 ******************************************************************************/

function mimeFromExt(ext) {
  switch (ext) {
    case '.html':
    case '.htm':
    case 'html':
    case 'htm':
      return 'text/html';
    case '.css':
    case 'css':
      return 'text/css';
    case '.js':
    case 'js':
      return 'text/javascript';
    case '.jpg':
    case 'jpg':
    case '.jpeg':
    case 'jpeg':
      return 'image/jpeg';
    case '.gif':
    case 'gif':
      return 'image/gif';
    case '.bmp':
    case 'bmp':
      return 'image/bmp';
    case '.ico':
    case 'ico':
    case '.cur':
    case 'cur':
      return 'image/x-icon';
    case '.png':
    case 'png':
      return 'image/png';
    case '.svg':
    case 'svg':
      return 'image/svg+xml';
    case '.webp':
    case 'webp':
      return 'image/webp';
    case '.mp3':
    case 'mp3':
      return 'audio/mp3';
    case '.wav':
    case 'wav':
      return 'audio/wav';
    case '.mp4':
    case 'mp4':
      return 'video/mp4';
    case '.webm':
    case 'webm':
      return 'video/webm';
    case '.json':
    case 'json':
      return 'application/json';
    default:
      return 'application/octet-stream';
  }
}

/*******************************************************************************
 * Problem 8 Part 1: quote paths with spaces
 *
 * When passing a filepath to a command line tool, if it contains spaces, it
 * usually needs to be quoted:
 *
 * /path/with/no/spaces --> doesn't need to be quoted
 * /path/with/a bit/of/space --> needs to be quoted due to the space
 *
 * Write a quotePath() function that takes a filepath, and returns the path
 * wrapped in double-quotes if it contains spaces, or unmodified if not.
 *
 * quotePath('/path/with/no/spaces' --> returns the String /path/with/no/spaces
 * quotePath('/path/with/a bit/of/space') --> returns the String "/path/with/a
bit/of/space"
 *
 * @params {String} path - a filepath, possibly with spaces
 ******************************************************************************/
function quotePath(path) {
  if (path.indexOf(' ') == -1) {
    return path;
  } else {
    return '"' + path + '"';
  }
}

/*******************************************************************************
 * Problem 8 Part 2: quote multiple paths
 *
 * quotePaths() takes any number of path arguments and calls quotePath() on
each,
 * returning a string of all the quoted paths joined by a space:
 *
 * quotePaths('/one', '/two', '/three hundred') --> returns '/one /two "/three
hundred"'
 ******************************************************************************/
function quotePaths() {
  let s;
  s = quotePath(arguments[0]);
  for (let i = 1; i < arguments.length; i++) {
    if (i <= arguments.length - 1) {
      s += ' ';
    }
    s += quotePath(arguments[i]);
  }
  return s;
}

/*******************************************************************************
 * Problem 9 Part 1: normalizing True/False data in a dataset
 *
 * A dataset contains fields that indicate a value is True or False.  However,
 * users have entered data in various formats, for example:
 *
 * Yes, yes, YES, Y, t, TRUE, true, True, 1
 * No, no, NO, N, n, f, FALSE, false, False, 0, null, undefined, -1
 *
 * Write a function normalizeBoolean() which takes a String, Number, Boolean,
 * null, or undefined and attempts to convert it into a pure Boolean value.
 *
 * If the value is not one of the values above, throw an error with the error
 * message, 'invalid boolean value'.
 *
 * @params {String|Number|Boolean} value - a true/false value needing to be
normalized
 ******************************************************************************/

function normalizeBoolean(value) {
  switch (value) {
    case true:
    case 1:
      return true;
    case false:
    case -1:
    case 0:
      return false;
  }
  if (typeof value != 'string') {
    return false;
  }
  var valueNew = value.toString().toUpperCase();
  switch (valueNew) {
    case 'YES':
    case 'Y':
    case 'T':
    case 'TRUE':
      return true;
    case 'FALSE':
    case 'F':
    case 'NO':
    case 'N':
      return false;
    default:
      return 'invalid boolean value';
  }
}

/*******************************************************************************
 * Problem 9 Part 2: checking for all True or all False values in a normalized
list
 *
 * Using your normalizeBoolean() function, create two new functions to check
 * if a list of arguments are all normalized True or normalized False values:
 *
 * allTrue('Y', 'yes', 1) --> returns true
 * allTrue('Y', 'no', 1) --> returns false
 * allTrue('Y', 'invalid', 1) --> returns false (i.e., does not throw)
 *
 * allFalse('N', 'no', 0) --> returns true
 * allFalse('N', 'no', 0) --> returns false
 * allFalse('Y', 'invalid', 1) --> returns false (i.e., does not throw)
 *
 * Use try/catch syntax to avoid having allTrue() and allFalse() throw errors
 * when normalizeBoolean() throws on invalid data.
 ******************************************************************************/
function allTrue() {
  //let count = [];
  for (let i = 0; i < arguments.length; i++) {
    if (normalizeBoolean(arguments[i]) != true) {
      return false;
    }
  }
  return true;
}
function allFalse() {
  for (let i = 0; i < arguments.length; i++) {
    if (normalizeBoolean(arguments[i]) != false) {
      return false;
    }
  }
  return true;
}

/*******************************************************************************
 * Problem 10 Part 1 - format name=value pairs on a URL's query string
 *
 * A URL can contain optional name=value pairs at the end. See:
 * https://web222.ca/weeks/week01/#urls
 *
 * 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 &
 *
 * Given an ID (Number) and a neighbourhood name (String), build and return a
 * query string like this:
 *
 * formatQS(687134, 'Islington-City Centre West') --> returns
 *   ?id=687134&n=Islington-City%20Centre%20West (note: neighbourhood is
encoded)
 *
 * If the neighbourhood name is missing (null, empty string), don't include the
 * `&n=...` portion of the query string result:
 *
 * formatQS(687134) --> returns ?id=687134
 *
 * NOTE: make sure you properly encode the neighbourhood name, since URLs can't
 * contain spaces or certain other characters.  Hint use the
encodeURIComponent()
 * function to do this, see:
 *
 * https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent
 *
 * @param {Number} id - an id value
 * @param {String} neighbourhood - (optional) the name of the neighbourhood
 ******************************************************************************/

function formatQS(id, neighbourhood) {
  if (typeof neighbourhood == 'string' && neighbourhood.length != 0) {
    return '?id=' + id + '&n=' + encodeURIComponent(neighbourhood);
  } else {
    return '?id=' + id;
  }
}

/*******************************************************************************
 * Problem 10 Part 2 - look for an ID in a URL's query string
 *
 * The containsID() function takes an id (Number) and a query string formatted
 * with formatQS, and returns true if the given id is found in the query string:
 *
 * containsID(687134, '?id=687134') --> returns true
 * containsID(687134, '?id=687134&n=Islington-City%20Centre%20West') --> returns
true
 * containsID(123, '?id=687134') --> returns false
 * containsID(123, '?id=687134&n=Islington-City%20Centre%20West') --> returns
false
 *
 * Make sure you match the whole id and not just the initial digits (e.g.,
looking
 * for 10 in id=10000).
 ******************************************************************************/

function containsID(id, qs) {
  if (qs.indexOf(id) !== -1) {
    if (
      qs.slice(id.length + 3, id.length + 4) == '&' ||
      qs.slice(id.length + 3, id.length + 4) == null
    ) {
      return true;
    }
    return false;
  }

  return false;
}

// Our unit test files need to access the functions we defined
// above, so we export them here.
exports.greeting = greeting;
exports.crush = crush;
exports.createEmailAddress = createEmailAddress;
exports.extractFSA = extractFSA;
exports.ageToRange = ageToRange;
exports.toDate = toDate;
exports.simpleMovingAverage = simpleMovingAverage;
exports.mimeFromExt = mimeFromExt;
exports.quotePath = quotePath;
exports.quotePaths = quotePaths;
exports.normalizeBoolean = normalizeBoolean;
exports.allTrue = allTrue;
exports.allFalse = allFalse;
exports.formatQS = formatQS;
exports.containsID = containsID;
```