LEARN ADVANCED ARRAY METHODS BY BUILDING A STATISTICS CALCULATOR

Introduction:

As you expand your JavaScript skills, you'll want to get comfortable with array manipulation methods, such as `map()`, `reduce()`, and `filter()`.

In this statistics calculator project, you'll gain experience with handling user input, DOM manipulation, and method chaining. You'll get practice by performing statistical calculations like mean, median, mode, variance, and standard deviation.

Step 1:

Statistics is a way of using math to make sense of data. It helps us understand patterns and trends in information, so we can make predictions and decisions based on that information.

In this challenge, you will build a statistics calculator that takes a set of numbers and returns the mean, median, mode, standard deviation, and variance.

The HTML and CSS have been provided for you. Feel free to explore the code – you may notice that the `calculate` function is called when the form is submitted. When you are ready, declare a `calculate` variable and assign it an empty function in the `script.js` file.

Step 2:

To begin, the `calculate` function needs to find the number that was entered in the `#numbers` input field. To do this, use a `.querySelector`

to locate the input field and then use the `.value` property to get the number entered.

Store this in a `value` variable.

Step 3:

Now that you have the value of the input, you need to split it into an array of numbers. Use the `.split()` method to do this.

The `.split()` method takes a string and splits it into an array of strings. You can pass it a string of characters or a RegEx to use as a separator. For example, `string.split(",")` would split the string at each comma and return an array of strings.

Use the `/,\s*/g` regex to split the `value` string by commas. You can tweak it based on the number of spaces separating your values. Store the array in an `array` variable.

Step 4:

The value of an input element is always a string, even if the input type is `number`. You need to convert this array of strings into an array of numbers. To do this, you can use the `.map()` method.

Create a `numbers` variable and assign it the value of `array.map()`. Remember that `.map()` creates a new array, instead of mutating the original array.

Step 5:

The `.map()` method takes a callback function as its first argument. This callback function takes a few arguments, but the first one is the current element being processed. Here is an example:

Example Code:

```
array.map(el => {

})
```

The callback function needs to return a value. In this case, you want to return the value of each element converted to a number. You can do this by using the `Number()` constructor, passing the element as an argument.

Add a callback function to your `.map()` method that converts each element to a number.

Step 6:

A user could put any text they want into the input box. You want to make sure that you are only working with numbers. The `Number()` constructor will return `NaN` (which stands for "not a number") if the value passed to it cannot be converted to a number.

You need to filter these values out – thankfully, arrays have a method specifically for this. The `.filter()` method will allow you to filter elements out of an array, creating a new array in the process.

Declare a `filtered` variable and assign `numbers.filter()` to it.

Step 7:

Much like the `.map()` method, the `.filter()` method takes a callback function. The callback function takes the current element as its first argument.

Example Code:

```
array.filter(el => {

})
```

The callback function needs to return a Boolean value, which indicates whether the element should be included in the new array. In this case, you want to return `true` if the element is not `NaN` (not a number).

However, you cannot check for equality here, because `NaN` is not equal to itself. Instead, you can use the `isNaN()` method, which returns `true` if the argument is `NaN`.

Add a callback function to your `.filter()` method that returns `true` if the element is not `NaN`.

Step 8:

Array methods can often be chained together to perform multiple operations at once. As an example:

Example Code:

```
array.map().filter();
```

The `.map()` method is called on the array, and then the `.filter()` method is called on the result of the `.map()` method. This is called method chaining.

Following that example, remove your `filtered` variable, and chain your `.filter()` call to your `.map()` call above. Do not remove either of the callback functions.

Step 9:

That is as far as you can get with the calculate function for now. It is time to write your mean logic.

Create an empty function called `getMean`. It should take a single parameter `array`.

Step 10:

The mean is the average value of all numbers in a list. The first step in calculating the mean is to take the sum of all numbers in the list. Arrays have another method, called `.reduce()`, which is perfect for this situation. The `.reduce()` method takes an array and applies a callback function to condense the array into a single value.

Declare a `sum` variable and assign `array.reduce()` to it.

Step 11:

Like the other methods, `.reduce()` takes a callback. This callback, however, takes at least two parameters. The first is the accumulator, and the second is the current element in the array. The return value for the callback becomes the value of the accumulator on the next iteration.

Example Code:

```
array.reduce((acc, el) => {

});
```

For your `sum` variable, pass a callback to `.reduce()` that takes the accumulator and the current element as parameters. The callback should return the sum of the accumulator and the current element.

Step 12:

The `.reduce()` method takes a second argument that is used as the initial value of the accumulator. Without a second argument, the `.reduce()` method uses the first element of the array as the accumulator, which can lead to unexpected results.

To be safe, it's best to set an initial value. Here is an example of setting the initial value to an empty string:

Example Code:

```
array.reduce((acc, el) => acc + el.toLowerCase(), "");
```

Set the initial value of the accumulator to `0`.

Step 13:

The next step in calculating the mean is to divide the sum of numbers by the count of numbers in the list.

Declare a `mean` variable and assign it the value of `sum` divided by the length of `array`

Step 14:

Finally, you need to return the value of `mean`.

Step 15:

You can actually clean this logic up a bit. Using the implicit return of an arrow function, you can directly return the value of the `.reduce()` method divided by the length of the array, without having to assign any variables.

Update your `getMean` function as described above.

Step 16:

Now you need to use your new `getMean` function. In your `calculate` function, declare a `mean` variable and assign it the value of `getMean(numbers)`.

Step 17:

To display the value of `mean`, your app has a `#mean` element ready to go.

Use a `.querySelector` to find that element, and then set its `.textContent` to the value of `mean`.

Step 18:

If you test your form with a list of numbers, you should see the mean display on the page. However, this only works because freeCodeCamp's iframe has special settings. Normally, when a form is submitted, the event triggers a page refresh.

To resolve this, add `return false;` after your `calculate();` call in the `onsubmit` attribute.

Step 19:

Time to start working on the median calculation. The median is the midpoint of a set of numbers.

Begin with an empty function called `getMedian`, which should take an `array` parameter.

Step 20:

The first step in calculating the median is to ensure the list of numbers is sorted from least to greatest. Once again, there is an array method ideal for this – the `.sort()` method.

Declare a `sorted` variable and assign `array.sort()` to it.

Step 21:

By default, the `.sort()` method converts the elements of an array into strings, then sorts them alphabetically. The `.sort()` method mutates the original array. This works well for strings, but not so well for numbers. For example, `10` comes before `2` when sorted as strings, but `2` comes before `10` when sorted as numbers.

To fix this, you can pass in a callback function to the `.sort()` method. This function takes two arguments, which represent the two elements being compared. The function should return a value less than `0` if the first element should come before the second element, a value greater than `0` if the first element should come after the second element, and `0` if the two elements should remain in their current positions.

To sort your numbers from smallest to largest, pass a callback function that takes parameters `a` and `b`, and returns the result of subtracting `b` from `a`.

Step 22:

In the next few steps, you'll learn how to determine if an array's length is even or odd, as well as how to find the median. You will then be able to apply what you learned to the `getMedian` function.

To check if a number is even or odd, you can use the modulus operator `%`. The modulus operator returns the remainder of the division of two numbers.

Here is an example checking if an array length is even or odd:

Example Code:

```
// check if array length is even

arr.length % 2 === 0;

// check if array length is odd

arr.length % 2 === 1;
```

If the remainder is `0`, the number is even. If the remainder is `1`, the number is odd.

Create a variable called `isEven`. Then use the modulus operator to check if the length of the `testArr2` array is even. Assign that expression to the `isEven` variable.

Below your `isEven` variable, log out the `isEven` variable to the console.

Open up the console to see the result.

Step 23:

To get the median of an array with an odd number of elements, you will need to find and return the middle number.

Here is how to find the middle number of an array with an odd number of elements:

Example Code:

```
arr[Math.floor(arr.length / 2)];
```

Here is a longer example finding the middle number of an array with `5` elements:

Example Code:

```
const numbers = [1, 2, 3, 4, 5];

const middleNumber = numbers[Math.floor(numbers.length / 2)];

console.log(middleNumber); // 3
```

The reason why you use `Math.floor` is because the result of dividing an odd number by `2` will be a decimal. `Math.floor` will round down to the nearest whole number.

Declare an `oddListMedian` variable and assign it the result of finding the middle number of the `testArr1`. Then log the `oddListMedian` variable to the console.

Open up the console to see the result.

Step 24:

To find the median of an even list of numbers, you need to find the two middle numbers and calculate the mean of those numbers.

Here is how to find the two middle numbers of an even list of items:

Example Code:

```
// first middle number

arr[arr.length / 2];

// second middle number

arr[(arr.length / 2) - 1];
```

To find the median, you can use the `getMean` function which adds the middle numbers and divides the sum by `2`.

Example Code:

```
const numbers = [1, 2, 3, 4];

const firstMiddleNumber = numbers[numbers.length / 2];

const secondMiddleNumber = numbers[(numbers.length / 2) - 1];

// result is 2.5

getMean([firstMiddleNumber, secondMiddleNumber]);
```

Create an `evenListMedian` variable and assign it the result of finding the median of the `testArr2`.

Then, log the `evenListMedian` variable to the console.

Step 25:

Now that you have a better understanding of how to find the median for odd and even lists of numbers, you can remove all your test code from the previous steps.

Step 26:

The `.sort()` method mutates the original array - in other words, it modifies the order of the elements directly. This is generally considered bad practice, as it can result in unexpected side effects.

Instead, you should use the `.toSorted()` method, which creates a new array. Change your `.sort()` call to `.toSorted()`. Do not modify the callback function.

Step 27:

Now it is time to apply what you have learned to the `getMedian` function.

Inside your `getMedian` function, check if the length of `sorted` is even. If it is, find the middle two numbers, calculate their mean, and return the result. If the length of `sorted` is odd, return the middle number.

Make sure to work with the `sorted` array to find the middle numbers.

Also if you need help, refer back to the previous few steps to see how to find the median for an array.

Step 28:

Like you did with your `getMean` function, you need to add your `getMedian` function to your `calculate` logic.

Declare a variable `median` and assign it the value of `getMedian(numbers)`. Then, query the DOM for the `#median` element and set the `textContent` to `median`.

Step 29:

Your next calculation is the mode, which is the number that appears most often in the list. To get started, declare a `getMode` function that takes the same `array` parameter you have been using.

Step 30:

To calculate the occurrence you can use the following approach:

Example Code:

```
const numbersArr = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4];
```

```
const counts = {};

numbersArr.forEach((el) => {

  if (counts[el]) {

    counts[el] += 1;

  } else {

    counts[el] = 1;

  }

});
```

Check if the current number is already in the `counts` object. If it is, increment it by `1`. If it is not, set it to `1`.

Resulting object. The keys are the numbers from the array and the values are the number of times each number appears in the list:

Example Code:

```
{ 1: 3, 2: 3, 3: 3, 4: 3, 5: 2 }
```

For this step, start by declaring an empty `counts` object. Later on in the project, you will use this object to calculate the mode of the list of numbers.

Step 31:

To better understand how the `getMode` function is going to work, you will need to print out its contents. This will allow you to see what is happening as you build out the function. But first you will need to return the `array` so it can be tested properly.

Inside your getMode function return your array parameter.

Step 32:

Inside the calculate function, you have already called the getMean and getMedian functions.

Below those function calls, add a console.log(getMode(numbers)).

To see the result, enter the numbers 4, 4, 2, 5 and click on the "Calculate" button. Open up the console to see the following array:

Example Code:

```
[ 4, 4, 2, 5 ]
```

Step 33:

Inside your getMode function, on the empty line above your return statement, call forEach on array. Your .forEach() method should have an empty callback function that takes an el parameter.

In the next few steps, you will use this loop to count the frequency of occurrences of each number in the array.

Step 34:

Inside the array.forEach() callback function, check if the current element is inside the counts object. If the element can be found, increment the value of counts[el] by 1. Otherwise, assign the number 1 to counts[el].

Change your return statement to return counts instead of array.

To test this, enter the numbers 4, 4, 2, 5 and click Calculate. You should see the following in the console:

Example Code:

```
{ '2': 1, '4': 2, '5': 1 }
```

Step 35:

There is another way to write the `forEach`. Instead of using a block body `() => {}` for the callback, you can use an expression body `() =>`.

You will have to convert the `if...else` statements into an expression. Write the expression as a ternary and use a single assignment for the ternary.

Example Code:

```
assignment = condition ? exprIfTrue : exprIfFalse
```

Convert the `forEach` callback to use an expression body and replace the statements with a ternary.

Step 36:

Now that you have a better understanding of how the `getMode` function works, you can remove the `console.log(getMode(numbers))` statement from the `calculate` function.

Step 37:

Returning the `counts` variable was only for testing purposes. Now that you are done testing, remove the `return counts` line from the `getMode` function.

Step 38:

There are a few edge cases to account for when calculating the mode of a dataset. First, if every value appears the same number of times, there is no mode.

To calculate this, you will use a `Set`. A `Set` is a data structure that only allows unique values. If you pass an array into the `Set` constructor, it will remove any duplicate values.

Start by creating an `if` statement. In the condition, create a `Set` with `new Set()` and pass it the `Object.values()` of your `counts` object. If the `size` property of this `Set` is equal to `1`, that tells you every value appears the same number of times. In this case, return `null` from your function.

Step 39:

Now you need to find the value that occurs with the highest frequency. You'll use the `Object.keys()` method for this.

Start by declaring a `highest` variable, and assigning it the value of the `counts` object's `Object.keys()` method.

Step 40:

Now you need to sort the values properly. Chain the `.sort()` method to your `Object.keys()` call.

For the callback, you'll need to use the `counts` object to compare the values of each key. You can use the `a` and `b` parameters to access the keys. Then, return the value of `counts[b]` minus the value of `counts[a]`.

Finally, access the first element in the array using bracket notation to complete your `highest` variable.

Step 41:

If multiple numbers in a series occur at the same highest frequency, they are all considered the mode. Otherwise, the mode is the number that occurs most often, that single number is the mode.

Thankfully, you can handle both of these cases at once with the `.filter()` method. Start by declaring a `mode` variable and assigning it the value of `Object.keys(counts)`.

Step 42:

Now chain the filter method to your latest `Object.keys()` call. The callback function should return whether the value of `counts[el]` is equal to your `counts[highest]`.

Step 43:

Time to return your `mode` variable.

`mode` is an array, so return it as a string with the `.join()` method. Separate the elements with a comma followed by a space.

Step 44:

Add your `getMode()` function to your `calculate` logic, and update the respective HTML element.

Step 45:

Your next calculation is the range, which is the difference between the largest and smallest numbers in the list.

You previously learned about the global `Math` object. `Math` has a `.min()` method to get the smallest number from a series of numbers, and the

`.max()` method to get the largest number. Here's an example that gets the smallest number from an array:

Example Code:
```
const numbersArr = [2, 3, 1];

console.log(Math.min(...numbersArr));
// Expected output: 1
```

Declare a `getRange` function that takes the same `array` parameter you have been using. Using `Math.min()`, `Math.max()`, and the spread operator, return the difference between the largest and smallest numbers in the list.

Step 46:

Add the logic for calculating and displaying the range to your `calculate` function.

Step 47:

The variance of a series represents how much the data deviates from the mean, and can be used to determine how spread out the data are. The variance is calculated in a few steps.

Start by declaring a `getVariance` function that takes an `array` parameter. Within that function, declare a `mean` variable and assign it the value of the `getMean` function, passing `array` as the argument.

Step 48:

The next step is to calculate how far each element is from the mean. Declare a `differences` variable, and assign it the value of `array.map()`. For the callback, return the value of `el` minus `mean`.

Step 49:

The next step is to square each of the differences. To square a value, you can use the `**` operator. For example, `3 ** 2` would return `9`.

Declare a `squaredDifferences` variable, and assign it the value of `differences.map()`. For the callback, return the value of `el` squared.

Step 50:

Next, you need to take the sum of the squared differences.

Declare a `sumSquaredDifferences` variable, and assign it the value of `squaredDifferences.reduce()`. For the callback, return the sum of `acc` and `el`. Remember to set the initial value to `0`.

Step 51:

With two `.map()` calls and a `.reduce()` call, you're creating extra arrays and iterating more times than needed. You should move all of the logic into the `.reduce()` call to save time and memory.

Remove the `differences`, `squaredDifferences`, and `sumSquaredDifferences` variables (and their values). Declare a `variance` variable, and assign it the value of `array.reduce()`. For the callback, pass in your standard `acc` and `el` parameters, but leave the function body empty for now. Don't forget to set the initial value to `0`.

Step 52:

Within your empty `.reduce()` callback, declare a variable `difference` and set it to the value of `el` minus `mean`. Then declare a `squared` variable, and set it to the value of `difference` to the power of `2`. Finally, return the value of `acc` plus `squared`.

Step 53:

The final step in calculating the variance is to divide the sum of the squared differences by the count of numbers.

Divide your `.reduce()` call by the length of the array (in your `variance` declaration). Then, return `variance`.

Step 54:

Add your new `getVariance` function to the `calculate` function, and update the respective HTML element.

Step 55:

Your final calculation is the standard deviation, which is the square root of the variance.

Begin by declaring a `getStandardDeviation` function, with the `array` parameter. In the function body, declare a `variance` variable and assign it the variance of the `array`.

Step 56:

To calculate a root exponent, such as $\sqrt[n]{x}$, you can use an inverted exponent $x^{1/n}$. JavaScript has a built-in `Math.pow()` function that can be used to calculate exponents.

Here is the basic syntax for the `Math.pow()` function:

Example Code:

```
Math.pow(base, exponent);
```

Here is an example of how to calculate the square root of 4:

Example Code:

```
const base = 4;

const exponent = 0.5;

// returns 2

Math.pow(base, exponent);
```

Declare a `standardDeviation` variable, and use the `Math.pow()` function to assign it the value of $variance^{1/2}$.

Step 57:

The `Math` object has a `.sqrt()` method specifically for finding the square root of a number.

Change your `standardDeviation` variable to use this method instead of `Math.pow()`.

Step 58:

Return your `standardDeviation` variable.

Step 59:

Lastly update the `calculate` function to include the standard deviation logic, like you did with your other functions.

Congratulations! Your project is complete.