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.

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.

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`.

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.

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 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.

By default, the `.sort()` method converts the elements of an array into strings, then sorts them alphabetically. 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`.

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.

```
const testArr1 = [1, 2, 3, 4, 5];
const testArr2 = [1, 2, 3, 4, 5, 6];
const isEven = testArr2.length % 2 === 0;
console.log(isEven);
const oddListMedian = testArr1[Math.floor(testArr1.length
 / 2)];
console.log(oddListMedian);
const evenListMedian = getMean([testArr2[testArr2.length
/ 2 - 1], testArr2[testArr2.length / 2]]);
console.log(evenListMedian);
```

Step 29

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 36

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.

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 43

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.

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 54

To calculate a root exponent, such as $x\sqrt[n]{}$xn, you can use an inverted exponent $x^{1/n}$x1/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 $variance^{1/2}$variance1/2.