Bootstrap

Iteration with Functions

Iteration with Functions

```
//While Loop
function power_iter(base, exponent){
  var result = 1;
  var i = 0;
  while (i < exponent) {
    result = result * base;
    var i = i + 1;
  }
  return result;
}

//For Loop
function power_iter(base, exponent){
  var result = 1;
  for   (var i = 0; i < exponent; i = i + 1) {
    result = result * base ;
  }
  return result;
}
```

Remember when we created a parametric power function using the while loop and for loop. Is there another way to achieve iteration.

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * base * base;
}
```

Let’s see our previous example. Here are three functions - each one raising a number to a certain power. We’ve written `power2` and `power3` before, though we called them `square` and cube at the time.

Iteration with Functions

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * base * base;
}

function power4(base) {
  return base * base * base * base;
}
```

If we wanted to add a function that raises a number to the power of 4, it would look like this...

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * base * base;
}

function power4(base) {
  return base * base * base * base;
}
```

Using this pattern we can imagine what additional power functions would look like: each subsequent “powerN” function simply adds one more set of “ `* base` ” to its return statement.

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * base * base;
}

function power4(base) {
  return base * base * base * base;
}
```

Q: How can we rewrite `power4` to avoid having to type out “**`base * base * base * base`**?”

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * base * base;
}

function power4(base) {
  return base * base * base * base;
}
```

A: Let’s reframe the problem. To raise base to the power of 4, multiply base by *the result of raising base to the power of 3.*

Iteration with Functions

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * base * base;
}

function power4(base) {
  return base * base * base * base;
}
```

Here’s another way to think about this: we have two very similar expressions with a lot of repetition. Functions can be used in expressions and are often used in place of repetitive calculations.

Iteration with Functions

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * base * base;
}

function power4(base) {
  return base * power3(base);
}
```

Since we already have a function that calculates raising x to the power of 3, we can substitute it in the power of 4 expression instead of typing out all those repetitive characters.

CodingLab Powered by RBK

Iteration with Functions

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * base * base;
}

function power4(base) {
  return base * power3(base);
}
```

We can reframe `power3` in a similar way: it multiplies base by *whatever base to the power of 2 is.*

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}
```

Therefore, we can rewrite power3 using our power2 function.

Iteration with Functions

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * base;
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}
```

Now we can see that `power2` can also be reframed...

Iteration with Functions

```
function power1(base) {
  return base;
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}
```

...to use `power1`. But we're not done yet! For reasons that we'll see in just a moment, writing `power1` to look like all of our other power functions will be very helpful. **Q:** Can you think of how to do that?

CodingLab Powered by RBK

Iteration with Functions

```
function power0(base) {
  return 1;
}

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}
```

A: Let's rewrite `power1` to make use of `power0` -- a function that always returns 1, since any number raised to the power of 0 is always 1. Notice the symmetry in all of the powerN functions, except of course, `power0`.

```
function power0(base) {
  return 1;
}

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}
```

We’ll stop here for now. We’ve saved ourselves from needing to type out an ever-lengthening series of “ `* base` ” with each new `powerN` function we write, but we still have a lot of repetition in our code.

Iteration with Functions

```
function power0(base) {
  return 1;
}

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}
```

Every time we want to calculate raising a number to a new power, we must write another function. What if we could create **one** function that will raise any number to any power? How might we write such a function?

Let’s put our many `powerN` functions aside and explore this new possibility.

```
function power(                  ) {




}
```

Let’s call this function power, since we want it to be able to calculate any exponent.

Iteration with Functions

```
function power(base, exponent) {



}
```

We’ll take two arguments: a base number, and an exponent.

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }

}
```

Here is the special case we mentioned earlier: raising a number to the power of `0` always results in 1. Let’s handle that case with a simple conditional statement.

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }

}
```

Now for the interesting part - how can we calculate the exponential value of a number without writing a function for every power along the way?

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }

}
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

Let’s peek at our previous solution and see if we can discover any clues.

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }

}
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$base^n = base * base^{n-1}$

For each **power of *n***, we’re returning the result of multiplying the base by the **power of *n* - 1**. We have just rephrased the power of *n* problem in terms of itself. Let’s make a note to ourselves summarizing this discovery.

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base *          ???           ;
}
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$base^n = base * base^{n-1}$

Let’s apply what we’ve just observed to our power function. We know we’re going to multiply **base** by whatever **base**$^{\text{exponent-1}}$ is. How can we determine that value?

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);

}
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$base^n = base * base^{n-1}$

We can use our power function!

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$$base^n = base * base^{n-1}$$

Let’s verify our approach by calling power with a base of 2 and an exponent of 4. We’ll track each step along the way in comments.

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$$base^n = base * base^{n-1}$$

To calculate 2^4, we must invoke power with an exponent of 3...

CodingLab Powered by RBK

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
// => 2 * 2 * power(2, 2)
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//.
```

$base^n = base * base^{n-1}$

... and again with an exponent of 2...

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
// => 2 * 2 * power(2, 2)
// => 2 * 2 * 2 * power(2, 1)
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$$base^n = base * base^{n-1}$$

... and once again with an exponent of 1...

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
// => 2 * 2 * power(2, 2)
// => 2 * 2 * 2 * power(2, 1)
// => 2 * 2 * 2 * 2 * power(2, 0)
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$base^n = base * base^{n-1}$

... and one last time with an exponent of `0`. Remember that an exponent of `0` is a special case: no matter what, raising any number to the power of `0` will result in the value 1.

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
// => 2 * 2 * power(2, 2)
// => 2 * 2 * 2 * power(2, 1)
// => 2 * 2 * 2 * 2 * power(2, 0)
// => 2 * 2 * 2 * 2 * 1
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$$base^n = base * base^{n-1}$$

Consequently, power(2, 0) will return 1. We have no more function invocations, so we can begin evaluating this entire expression. JavaScript will look at the expression from right-to-left.

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
// => 2 * 2 * power(2, 2)
// => 2 * 2 * 2 * power(2, 1)
// => 2 * 2 * 2 * 2 * power(2, 0)
// => 2 * 2 * 2 * 2 * 1
// => 2 * 2 * 2 * 2
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$base^n = base * base^{n-1}$

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
// => 2 * 2 * power(2, 2)
// => 2 * 2 * 2 * power(2, 1)
// => 2 * 2 * 2 * 2 * power(2, 0)
// => 2 * 2 * 2 * 2 * 1
// => 2 * 2 * 2 * 2
// => 2 * 2 * 4
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//...
```

$$base^n = base * base^{n-1}$$

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
// => 2 * 2 * power(2, 2)
// => 2 * 2 * 2 * power(2, 1)
// => 2 * 2 * 2 * 2 * power(2, 0)
// => 2 * 2 * 2 * 2 * 1
// => 2 * 2 * 2 * 2
// => 2 * 2 * 4
// => 2 * 8
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  return base * power3(base);
}

//.
```

$$base^n = base * base^{n-1}$$

Iteration with Functions

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}

power(2, 4);
// => 2 * power(2, 3)
// => 2 * 2 * power(2, 2)
// => 2 * 2 * 2 * power(2, 1)
// => 2 * 2 * 2 * 2 * power(2, 0)
// => 2 * 2 * 2 * 2 * 1
// => 2 * 2 * 2 * 2
// => 2 * 2 * 4
// => 2 * 8
// => 16
```

```
//...

function power1(base) {
  return base * power0(base);
}

function power2(base) {
  return base * power1(base);
}

function power3(base) {
  return base * power2(base);
}

function power4(base) {
  r         base * power3(base);
}

//.
```

$$base^n = base * base^{n-1}$$

We have our result!

Iteration with Functions

```
function power_iter(base, exponent){
  var result = 1;
  while (exponent > 0) {
    result = result * base;
    exponent = exponent - 1;
  }
  return result;
}
```

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}
```

Let’s look at our old while loop function and our new power function.

Iteration with Functions

```
function power_iter(base, exponent){
  var result = 1;
  while (exponent > 0) {
    result = result * base;
    exponent = exponent - 1;
  }
  return result;
}
```

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}
```

For one, they both use the same case to determine when it's time to stop repeating.

Iteration with Functions

```
function power_iter(base, exponent){
  var result = 1;
  while (exponent > 0) {
    result = result * base;
    exponent = exponent - 1;
  }
  return result;
}
```

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}
```

They also have conditions which will gradually move us toward that case.

Iteration with Functions

```
function power_iter(base, exponent){
  var result = 1;
  while (exponent > 0) {
    result = result * base;
    exponent = exponent - 1;
  }
  return result;
}
```

```
function power(base, exponent) {
  if (exponent === 0) {
    return 1;
  }
  return base * power(base, exponent - 1);
}
```

Most importantly, they are both used in the same way.

That’s it

For Iteration with Functions