
Slice notation

This repository contains a proposal for adding slice notation syntax to JavaScript. This is currently at stage 1 of the TC39 process.

Champions:

- Sathya Gunasekaran (@gsathya)
- HE Shi-Jun (@hax)

Introduction

The slice notation provides an ergonomic alternative to the various slice methods present on `Array.prototype`, `TypedArray.prototype`, etc.

```
1 const arr = ['a', 'b', 'c', 'd'];
2
3 arr[1:3];
4 // → ['b', 'c']
5
6 arr.slice(1, 3);
7 // → ['b', 'c']
```

This notation can be used for slice operations on primitives like `Array` and `TypedArray`.

Motivation

```
1 const arr = ['a', 'b', 'c', 'd'];
2 arr.slice(3);
3 // → ['a', 'b', 'c'] or ['d'] ?
```

In the above example, it's not immediately clear if the newly created array is a slice from the range 0 to 3 or from 3 to `len(arr)`.

```
1 const arr = ['a', 'b', 'c', 'd'];
2 arr.slice(1, 3);
3 // → ['b', 'c'] or ['b', 'c', 'd'] ?
```

Adding a second argument is also ambiguous since it's not clear if the second argument specifies an upper bound or the length of the new slice.

Programming language like Ruby and C++ take the length of the new slice as the second argument, but JavaScript's slice methods take the upper bound as the second argument.

```
1 const arr = ['a', 'b', 'c', 'd'];
2 arr[3:];
3 // → ['d']
4
5 arr[1:3];
6 // → ['b', 'c']
```

With the new slice syntax, it's immediately clear that the lower bound is 3 and the upper bound is `len(arr)`. It makes the intent explicit.

The syntax is also much shorter and more ergonomic than a function call.

Examples

In the following text, 'length of the object' refers to the `length` property of the object.

Default values

The lower bound and upper bound are optional.

The default value for the lower bound is 0.

```
1 const arr = ['a', 'b', 'c', 'd'];
2
3 arr[:3];
4 // → ['a', 'b', 'c']
```

The default value for the upper bound is the length of the object.

```
1 const arr = ['a', 'b', 'c', 'd'];
2 arr[1:];
3 // → ['b', 'c', 'd']
```

Omitting all lower bound and upper bound value, produces a new copy of the object.

```
1 const arr = ['a', 'b', 'c', 'd'];
2
3 arr[:];
4 // → ['a', 'b', 'c', 'd']
```

Negative indices

If the lower bound is negative, then the start index is computed as follows:

```
1 start = max(lowerBound + len, 0)
```

where `len` is the length of the object.

```
1 const arr = ['a', 'b', 'c', 'd'];
2
3 arr[-2:];
4 // → ['c', 'd']
```

In the above example, `start = max((-2 + 4), 0) = max(2, 0) = 2`.

```
1 const arr = ['a', 'b', 'c', 'd'];
2
3 arr[-10:];
4 // → ['a', 'b', 'c', 'd']
```

In the above example, `start = max((-10 + 4), 0) = max(-6, 0) = 0`.

Similarly, if the upper bound is negative, the end index is computed as follows:

```
1 end = max(upperBound + len, 0)
```

```
1 const arr = ['a', 'b', 'c', 'd'];
2
3 arr[: -2];
4 // → ['a', 'b']
5
6 arr[: -10];
7 // → []
```

These semantics exactly match the behavior of existing slice operations.

Out of bounds indices

Both the lower and upper bounds are capped at the length of the object.

```
1 const arr = ['a', 'b', 'c', 'd'];
2
3 arr[100:];
4 // → []
5
6 arr[: 100];
7 // → ['a', 'b', 'c', 'd']
```

These semantics exactly match the behavior of existing slice operations.

Prior art

Python

This proposal is highly inspired by Python. Unsurprisingly, the Python syntax for slice notation is strikingly similar:

```
1 slicing      ::= primary "[" slice_list "]"
2 slice_list  ::= slice_item ("," slice_item)* [","]
3 slice_item  ::= expression | proper_slice
4 proper_slice ::= [lower_bound] ":" [upper_bound] [ ":" [stride] ]
5 lower_bound ::= expression
6 upper_bound ::= expression
7 stride      ::= expression
```

Examples:

```
1 arr = [1, 2, 3, 4];
2
3 arr[1:3];
4 // → [2, 3]
5
6 arr[1:4:2]
7 // → [2, 4]
```

CoffeeScript

CoffeeScript provides a Range operator that is *inclusive* with respect to the upper bound.

```
1 arr = [1, 2, 3, 4];
2 arr[1..3];
3 // → [2, 3, 4]
```

CoffeeScript also provides another form the Range operator that is *exclusive* with respect to the upper bound.

```
1 arr = [1, 2, 3, 4];
2 arr[1...3];
3 // → [2, 3]
```

Go

Go offers slices:

```
1 arr := []int{1,2,3,4};
```

```
2 arr[1:3]
3 // → [2, 3]
```

There is also ability to *not* provide lower or upper bound:

```
1 arr := []int{1,2,3,4};
2 arr[1:]
3 // → [2, 3, 4]
4
5 arr := []int{1,2,3,4};
6 arr[:3]
7 // → [1, 2, 3]
```

Ruby

Ruby seems to have two different ways to get a slice:

- Using a Range:

```
1 arr = [1, 2, 3, 4];
2 arr[1..3];
3 // → [2, 3, 4]
```

This is similar to CoffeeScript. The `1..3` produces a Range object which defines the set of indices to be sliced out.

- Using the comma operator:

```
1 arr = [1, 2, 3, 4];
2 arr[1, 3];
3 // → [2, 3, 4]
```

The difference here is that the second argument is actually the length of the new slice, not the upper bound index.

This is currently valid ECMAScript syntax which makes this a non starter.

```
1 const s = 'foobar'
2 s[1, 3]
3 // → 'b'
```

FAQ

Why pick the Python syntax over the Ruby/CoffeeScript syntax?

The Python syntax which excludes the upper bound index is similar to the existing slice methods in JavaScript.

We could use exclusive Range operator (`...`) from CoffeeScript, but that doesn't quite work for all cases because it's ambiguous with the spread syntax. Example code from getify:

```
1 Object.defineProperty(Number.prototype, Symbol.iterator, {
2   *value({ start = 0, step = 1 } = {}) {
3     var inc = this > 0 ? step : -step;
4     for (let i = start; Math.abs(i) <= Math.abs(this); i += inc) {
5       yield i;
6     }
7   },
8   enumerable: false,
9   writable: true,
10  configurable: true
11 });
12
13 const range = [ ...8 ];
14 // → [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Why does this not use the iterator protocol?

The iterator protocol isn't restricted to index lookup making it incompatible with this slice notation which works only on indices.

For example, Map and Sets have iterators but we shouldn't be able to slice them as they don't have indices.

What about splice?

CoffeeScript allows similar syntax to be used on the left hand side of an [AssignmentExpression](#) leading to splice operation.

```
1 numbers = [1, 2, 3, 4]
2 numbers[2..4] = [7, 8]
3 // → [1, 2, 7, 8]
```

This feature is currently omitted to limit the scope of the proposal, but can be incorporated in a follow on proposal.

Why doesn't this include a step argument like Python does?

The step argument makes the slice notation ambiguous with the bind operator.

```
1 const x = [2];
2 const arr = [1, 2, 3, 4];
3 arr[::x[0]];
```

Is the above creating a new array with values [1, 3] or is it creating a bound method?

Should this create a view over the array, instead of a creating new array?

Go creates a [slice](#) over the underlying array, instead of allocating a new array.

```
1 arr := []int{1,2,3,4};
2 v = arr[1:3];
3 // → [2, 3]
```

Here, v is just descriptor that holds a reference to the original array `arr`. No new array allocation is performed. See this [blog post](#) for more details.

This doesn't map to any existing construct in JavaScript and this would be a step away from how methods work in JavaScript. To make this syntax work well within the JavaScript model, such a [view](#) data structure is not included in this proposal.

Should slice notation work on strings?

The `String.prototype.slice` method doesn't work well with unicode characters. This [blog post](#) by Mathias Bynens, explains the problem.

Given that the existing method doesn't work well, this proposal does not add `@@slice` to `String.prototype`.

How about combining this with + for append?

```
1 const arr = [1, 2, 3, 4] + [5, 6];
2 // → [1, 2, 3, 4, 5, 6]
```

This is not included in order to keep the proposal's scope maximally minimal.

The operator overloading proposal may be a better fit for this.

Can you create a Range object using this syntax?

The slice notation only provides an ergonomic syntax for performing a slice operation.

The current slice notation doesn't preclude creating a range primitive in the future.

A new Range primitive is being discussed here: <https://github.com/tc39/proposal-Number.range/issues/22>

Isn't it confusing that this isn't doing property lookup?

This is actually doing a property lookup using `[[Get]]` on the underlying object. For example,

```
1 const arr = [1, 2, 3, 4];
2
3 arr[1:3];
4 // → [2, 3]
```

This is doing a property lookup for the keys 1 and 2.

But, shouldn't it do a lookup for the string `'1:3'`?

```
1 const arr = [1, 2, 3, 4];
2
3 arr['1:3'];
4 // → undefined
```

No. The slice notation makes it analogous with how keyed lookup works. The key is first evaluated to a value and then the lookup happens using this value.

```
1 const arr = [1, 2, 3, 4];
2 const x = 0;
3
4 arr[x] !== arr['x'];
5 // → true
```

The slice notation works similarly. The notation is first evaluated to a range of values and then each of the values are looked up.

There are already many modes where `'.'` mean different things. Isn't this confusing?

Depending on context `a:b`, can mean:

- `LabelledStatement` with `a` as the label
- Property `a` with value `b` in an object literal: `{a: b }`
- `ConditionalExpression`: `confused ? a : b`

-
- Potential type systems (like TypeScript and Flow) that might make it to JavaScript in the future.

Is it a lot of overhead to disambiguate between modes with context? Major mainstream programming languages like Python have all these modes and are being used as a primary tool for teaching programming.