重点：

1. var/let 作用域区别
2. for in/of区别
3. object可以自定义方法
4. object只能用字符串做key
5. class的method定义方法
6. java script的const变量不可以重新赋值
7. ==和===区别，==会隐式转换
8. js的private，public和inheritance

01. Intro to 669; JavaScript Primer

Intro to 669 (Slides)

JavaScript Primer

(coming from a Python background)

Learning Goals

- Understand how JavaScript came to be the most popular programming language in the world
- Apply knowledge about programming from previous Python experience to master basic concepts in JavaScript
- Understand how to use basic programming constructs in JavaScripts, including variables, data types, conditionals, loops, strings, arrays, objects, and classes

01.1 A Brief History of JavaScript

The history of JavaScript is closely tied to the **history of the Web.** It was created in the mid-90s at Netscape, which was the dominant browser company at the time, as a way to make web pages "more dynamic." Originally it was called "**Mocha**," which was intended as a riff on the "hottest" programming language at the time, **Java**. Mocha/JavaScript was partly inspired by Java and the syntax of JavaScript superficially resembles that of Java (which, in turn, was inspired by C++, which was inspired by C), but for the most part JavaScript and Java are very, very different.

One of the big differences was that Java was intended as a serious, industrial strength programming language whereas **JavaScript** (as it was named when it was released to the public) **was intended as a fairly lightweight "scripting" language** that, at least initially, most people did not consider suitable for serious application development. Netscape was thinking that JavaScript would support things like animations (think "rollovers") and simple interactions. Web pages would still be

mostly HTML with just a smidge of JavaScript thrown in. CSS was starting to be worked on around the same time, but very few people could envision how HTML, CSS, and JavaScript would end up becoming the dominant platform for the web, or how "dynamic" and interactive web pages (or what we now call "web applications") would become.

JavaScript has grown up and changed a lot in the last 20+ years. **It is now the most widely used programming language on the planet** and has been increasingly refined towards more "serious" uses. It is supported by everything that calls itself a "web browser," and more and more it is used for **creating applications that aren't intended for web browsers at all**. The `node.js` project started in 2009 as a browser-less JavaScript run-time environment and quickly became very, very popular. This allowed JavaScript to be used for backend programming (on servers) as well as for front end programming (usually in the browser). It also supported the use of JS and node for lots of other things, including the creation of development tools and mobile application development platforms like React Native, as we will see in this course.

When talking about modern JavaScript, you'll hear people talk about JavaScript and JS, but you'll also hear them talk about **ECMAScript**. For all intents and purposes, these are the same. Technically, J**avaScript is an implementation of the ECMAScript standard**. The ECMAScript (or ES) standard was created to deal with the fact that when JavaScript first got started, every browser development company implemented JavaScript in slightly (and not so slightly) different ways, largely as ways to differentiate themselves from each other (and also to try to get people to develop sites for their browser that didn't work on other people's browser—it was a mess). The current state of browser compliance and standardization is much better, but it can still be dizzying. The latest and greatest version of ECMAScript is ES2019, which is the 10th Edition. **A lot of big, important changes happened with the 6th edition, usually called ES6**, so you'll often see people writing about JavaScript that works "in ES6" (and, by implication, probably not in earlier versions). Mainly **it's important to know that there are different versions** so that if you find some code on the web and it's not working, you might check to see if the code was written for the same version of JavaScript that you're using. Writing ES6-compliant code is generally pretty safe these days. Newer versions might not be as widely available.

Most people who have a bit of experience with JavaScript have used it in web pages. This is a great place to start, but it can be challenging to understand what is really core to JavaScript and what is specific to the browser environment and to the standard browser DOM. If you've written or read code that looks like this:

```
<script>
document.getElementById("p1").innerHTML = "New text!";
</script>
```

Or the jQuery-ified version:

```
<script>
$("p1").html = "New text!";
</script>
```

You've written/read JavaScript alright, but this code is quite different from any code that you would write for the node.js environment or, as we shall see, for ReactNative. Once you get outside the browser, the DOM, **all of the browser-specific objects like `document` and `window`, and pretty much all of jQuery don't make much sense.**
DOM必须要有浏览器支持
So for our first task in SI 669, **we will take a look at JavaScript from pretty much the ground up as a programming language**, rather than as a browser scripting language. To do this, we'll take a step back from worrying about the particular environment we're operating in. Given that the prerequisites for this course, at least in Fall 2020, are a bunch of python and a little tiny smidge of JavaScript, **we'll start by comparing JavaScript to Python, step by step**. We'll do this pretty fast, so that by the end of the first two lessons you should be able to

- abstract the knowledge you have about python to see it as knowledge about *programming*. All programming languages do pretty much the same things, they just do it a bit differently.
- do most of what you already know how to do in Python in JavaScript, and have the scaffolding to add new knowledge about how to use JavaScript as you need it.

The rest of this lesson is intended to bring you up to speed on all of the basic features of browserless JavaScript, to make sure that when we jump into the world

of React Native you won't be confused by the JS basics—just the new and nutty stuff that React Native is adding on top of vanilla JavaScript!

So let's get started!

01.2 Using node.js to run browserless JavaScript

Before class, you should have completed all the steps in Week 1 Prep on Canvas and you should have created the file `<699dir>/week01/hello.js` with the contents `console.log('Hello World!');` and you should have been able to run this file like so:

```
$ node hello.js
Hello World!
```

Now let's get into some

01.2.1 Printing to the Console

You can't do anything unless you can see what you're doing. In Python, you often use the `print()` statement to see what's going on in your code. For example

```
x = 1
y = 2
z = x + y
print(z)
```

In JavaScript you accomplish the same thing using `console.log()`.

```
var x = 1;
var y = 2;
z = x + y;
console.log(z);
```

When running code in node.js, `console.log()` will output to the terminal (which is sometimes called a "console"). When running code in the browser, `console.log()` will output to the browser's "console," which is part of the browser's developer tools.

01.3 Statements and Syntax

01.3.1 Terminating Statements: `;` vs `newline`

Python statements end in a newline. This is three statements in Python:

```
# Python
x = 5
y = x + (6 / 2)
print(y)
```

In JS, statements are separated by a semicolon. This is three statements in JS (and totally invalid in Python).

Example 01.3.1A

```
// JS
x = 5; y =
x +
  (6/2
);
console.log(x);
// of course you'd never write ugly code like this!
// can you identify the 3 statements?
```

- Fun facts you may already know:
  - you can actually use `;` in python to separate statements on a single line
  - you can use `\` to create multiline statements in python
  - both of these are considered unconventional in python (generally)

01.3.2 Nesting Statements: `{}` vs. `:` and indentation

In Python, a *nested* code block is denoted by the indent level, and the *nesting* code has to end with a `:`

```
# Python
if a > b:
  print('a is bigger')
elif a == b:
  print('a and b are equal')
else:
  print('b is bigger')
```

In JS, newlines and indent level don't matter, but `{` and `}` very much do! This JS snippet is equivalent to the Python snippet above

Example 01.3.2A

```
// JS
if (a > b) {
console.log('a is bigger'); // no indent, but no biggie!
}
else if (a == b) { console.log('a and b are equal'); } // dont even need a new line
else { console.log('b is bigger' // and you can put newlines in weird places
                          );}
```

That's syntactically correct, but it's hideous. Even though you don't *need* indentation and newlines to make the code work, you should use them to make the code readable.

Example 01.3.2B

```
// JS
if (a > b) {
  console.log('a is bigger');
} else if (a == b) {
  console.log('a and b are equal');
```

```
} else {
  console.log('b is bigger');
}
```

01.3.3 The upshot: Whitespace matters in python, not in JS

In JS 'whitespace' (e.g., spaces, newlines, tabs) generally don't matter. You have to use `;` (to terminate statements) and `{}` (to nest code blocks inside a statement). In Python whitespace matters a lot. Indentation (tabs and spaces at the beginning of a line) and newlines define what counts as a statement and what is nested inside of what.

These differences are a major source of annoying errors for people transitioning from python to JS (but only for a little while).

There are other syntax differences, but these are the biggies. If you run into errors, check to make sure the syntax is right.

Usually syntax errors will be reported as `SyntaxError`s but not always. Because Python code doesn't always look all *that* different from JS code it might run OK but not in the way you expect. While you're still getting used to JS, keep syntax in mind as a prime suspect.

01.3.4 Comments

Python comments are denoted by `#`

```
# this is a comment
x = 5 # and this is a comment too, on a line of code

'''
There really isn't support for multiline comments in python,
but you can use triple quotes to create a multi-line string
that isn't assigned to anything.
```

JS single-line comments are denoted by `//`. Multi-line comments can must be contained by `/*` and `*/`.

Example 01.3.4A

```
// Example 1.7
// this is a comment
x = 5; // and this is one too!

/*
Multi-line comments are possible in JS, using this syntax.
*/
```

01.4 Variables and DataTypes

01.4.1 Assignment

Assignment in Python is straightforward.

```
x = 5
x += 1
s = 'Hello World'
```

The same kind of assignment (usually) works in JS (as long as you add the `;`), but there are several other options. In "strict" mode, which will be used by default in React Native, you will need to use a declaration keyword (i.e., `var`, `let`, or `const`). These keywords allow you to specify the *scope* and the *modifiability* of the variable (we'll talk more about the differences among these options later):

Example 01.4.1A

```
undelcared_var = 'I am global';
var x = 5;
let y = 7;
const z = 200;
x += 1; // x is now 6
```

```
y *= 5; // y is now 35
z -= 100; // TypeError; can't modify const
```

不可重新给const变量赋值

01.4.2 Basic Data Types

Python has a few more built-in data types than JS. In particular, JS has only one `number` type and lacks a built-in `dictionary` type, though JS Objects are essentially dictionaries (we'll get to this later).

Python	JS equivalent
Integer (e.g., 5)	Number (5 is the same as 5.0)
Floating Point (e.g., 5.3)	Number (in JS all numbers are floating point, more or less)
Complex (e.g., `2+3j`)	N/A
Boolean (e.g., `True` or `False`)	Boolean (e.g., `true` or `false`)
String	String
List	Array
Object	Object
Dictionary	Object
None	null (variable is explicitly set to `null`)
None	undefined (variable has not been set to any value)

Automatic Type Conversion

A few notes here:

- Automatic type conversion occurs when you try to use a variable of one type (say, string) where another type is expected (say, integer). This can happen, for example, when you try to add a string and an integer together. It can happen other times, too, like when you try to pass a string as an argument to a function that is expecting an integer as a parameter (or vice versa).
- Python refuses to do much in the way of automatic type conversion. If you try to use an unexpected type somewhere, it will result in an error.

```
a = 5
```

```
b = '6'
print (a + b)
```

This throws a TypeError in Python.

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

JS tries to do more, which can sometimes lead to confusing results. In JS, the following works fine, but might not do what you expect:

Example 01.4.2A

```
var a = 5;
var b = '6';
console.log(a + b); // prints '56' -- why?
```

隐式转换

There are lots of details about declaring numbers and strings. Many of these details are very esoteric, but the good news is they're mostly the same in Python and JS.

```
# Python
color = 0xF9A033 # declare a number using hexadecimal notation
str = 'Roses are red\nViolets are blue' # declare string with
a newline in the middle
```

Example 01.4.2B

```
// JS
var color = 0xF9A033; // declare a number using hexadecimal no
tation
var str = 'Roses are red\nViolets are blue'; // declare string
with a newline
```

01.4.3 The Upshot

- Basic data types are even more basic in JS than they are in Python, in that there are fewer of them.
- JS is more generous about trying to guess what you're trying to do and converting data types for you on the fly. Most of the time you will appreciate this. Sometimes you will not.

- We'll say a bit more about strings, arrays, and objects later in the lesson.

Announcements

Before Next Class

- Join Slack
- Complete Week 1 Module items if you didn't
  - Watch videos
  - Complete About Me questionnaire and slide (if you didn't already)
- See the Week 2 Module in Canvas
  - Complete the first Demonstrate Your Understanding (DYU #1)
    - A bit longer than the next 3 DYUs
    - Open Browser: refer to these notes & this week's readings
    - Open Editor: completely OK to type/paste code to get the answers, as long as you *think* about it
  - Look over Week 2 readings (No installation or prep this week)

Intermission (10 minutes of fresh air!)

01.5 Control Flow

01.5.1 If and Else ("Conditionals")

Python uses `if`, `elif`, and `else`

```
if a > b:
  print('a is bigger')
elif a == b:
```

```
  print('a and b are equal')
else:
  print('b is bigger')
```

JS uses `if`, `else if`, and `else`

Example 01.5.1A

```
// Example 1.15
if (a > b) {
  console.log('a is bigger');
} else if (a == b) {
  console.log('a and b are equal');
} else {
  console.log('b is bigger');
}
```

Most of JavaScript's comparison operators are the same as Python's. For example, all of these are the same:

```
>, <, >=, <=, !=
```

However, JavaScript has *two* equality operators, which differ as follows:

`==`	"abstract comparison": tries to convert both operands to the same type and then compare.
	`6 == 6` evaluates to `true`
	`6 == '6'` also evaluates to `true`

`===`	"strict comparison": does *not* perform type conversion. Only true if operands are of the same type *and* value.
	`6 == 6` evaluates to `true`
	`6 == '6'` evaluates to `false`

This is a bit confusing at first. Most of the time it doesn't matter which one you use, as you will mostly be comparing variables of the same type. All else being equal, though, it's generally better to use the strict version (`===`) to avoid nasty surprises.

The logical operators (and, or, not) for JS look different but behave the same way.

Python	JavaScript
`and`	`&&`
`or`	`||`
`not`	`!`

And finally, JavaScript doesn't have some of Python's handy operators like `in`, `not in` , `is`, or `is not`.

```
# Choose an activity to do based on the temperature
temp = 77
activity = ""
if (temp < 10 and temp > 100):
  activity = "stay inside"
elif (temp >= 10 and temp < 30):
  activity = "go skiing"
elif (temp >= 30 and temp < 60):
  activity = "go for a run"
elif (temp >= 60 and temp < 80):
  activity = "play golf"
else:
  activity = "go swimming"
print("It's " + str(temp) + " outside, you should " + activity
+ "!")
```

```
var temp = 77;
var activity = "";
if (temp < 10 || temp > 100) {
  activity = "stay inside";
} else if (temp >= 10 && temp < 30) {
  activity = "go skiing";
```

```
} else if (temp >= 30 && temp < 60) {
  activity = "go for a run";
} else if (temp >= 60 && temp < 80) {
  activity = "play golf";
} else {
  activity = "go swimming";
}
console.log("It's " + temp + " outside, you should " + activit
y + "!");
console.log(activity.length);
```

Now You Try #1

Translate the above Python program into a valid JavaScript program. Test it out to make sure it works.

Enter your response at https://app.sli.do/event/t1uqvv8s

01.5.2 While and Do/While ("Indefinite Iteration")

Python has one type of while loop. It continues until the specified condition is *False*

```
r = 1
while r > 0.1:
    print("Not this, it's", r)
    r = random.random()
print("finally!", r, "is less than 0.1!");
```

JS has two types of while loops, though the first one is used the vast majority of the time. The first one is pretty much exactly like the Python version, except with JS syntax of course.

Example 01.5.2A

```
var r = 1;
while (r > 0.1) {
  console.log("Looping! r = " + r);
```

```
  r = Math.random();
}
console.log('Finally! ' + r + ' is less than 0.1!');
```

The second one is the `do … while` style, which puts the condition at the end of the loop. The only difference is that with a normal `while` loop it's possible for the loop to execute **zero** times. With a `do … while` loop the loop **must** execute at least **one** time. If you know for sure that your loop needs to execute at least one time you can save yourself a line or two of code with the `do … while` syntax. But you don't need to use this form if you don't want to. It's a "nice-to-have."

Example 01.5.2B

```
do {
  var r2 = Math.random();
  console.log('Got ' + r2);
} while (r2 > 0.10);


console.log('Finally! ' + r2 + ' is less than 0.1!');
```

01.5.3 For Loops ("Definite Iteration")

Python `for` loops

In Python, there is only one kind of for loop: the kind that lets you iterate through a sequence (or, more generally, an *iterable*). Sequences include lists, tuples, and strings (which are sequences of characters), among other things.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

If you want to make a loop that executes a specified number of times, you need to make a list of numbers and then iterate through that (note that this is essentially the same as the previous example since `range` produces a list of numbers:

Example 01.5.3A

```
for x in range(6):
  print(x)

for y in range(4, 8):
  print (y - 4)
```

3 kinds of JS `for` loops

JavaScript has 3 (yes 3!) kinds of `for` loops! They are:

- `for…of`
- `for…in`
- `for( ; ; )`

注意of和in的区别，of和python in一样，in用来遍历对象的key

You'll notice that JS also has a `for…in` loop, just like Python. But be warned! They are different!

Use `for…of` to iterate through sequences (like lists)

The kind of JS `for` loop that is the most like the python `for…in` loop is actually the `for…of` loop.

Let me say that again.

The kind of JS loop that is the most like the python `for…in` loop is actually the `for…of` loop.

This loops through a sequence or iterable, just like the python `for…in` loop. For example:

Example 01.5.3B

```
var stoneFruits = ['cherry', 'plum', 'peach'];
for sf of stoneFruits {
  console.log("I'll have some " + sf + " pie, please.");
}
```

Use `for…in` to iterate through the properties of an object

The next kind of JS for loop is the `for…in` type, and this does **not** work like the Python `for … in` loop. Instead, the JS `for…in` loop iterates through the *properties of an object.* We'll talk more about objects in a moment, but for now you can think of them as something like Python's dictionaries in that they consist of name-value pairs.

When you iterate through an object using `for…in`, the iterating variable (`f` in the following example) is assigned to the *property name.* See the example below for a way to access the *value* of each of an object's properties.

Example 01.5.3C

```
var fruitObj = {
  best: "nectarine",
  worst: "red delicious",
  other: ["banana", "pear", "orange"]
};

for (f in fruitObj){
  console.log(f);
  console.log(fruitObj[f]);
}
```

遍历对象的key

Use `for( ; ; )` to iterate a specific number of times (or to iterate using numbers)
The final kind of `for` loop in JS is what I think of as the "classic" for loop (probably because it's the first kind I learned). This is the style that you'll find in most of the other popular languages *other* than Python. For example, you'll find this style in Java, C++, C, and C#. I'm guessing you'll come across it in other languages too but I don't really know any others.

Let me give an example and then explain how it works:

Example 01.5.3D

注意var和let区别
var可以在block外使用，可以重定义
let不可以，且只能在subblock中重定义

```
// Example 1. X
for (var i = 0; i < 10; i++) {
```

```
  console.log(i);
}
```

The general form of this style of `for` loop is

```
for (statement 1; expression 2; statement 3) {
  // code to be executed each iteration
}
```

- statement 1 is the *initializer.* It is executed exactly once, before the first time the loop is executed.
- statement 2 is the *condition* that determines whether the loop will execute again. The condition is tested *at the beginning of each loop.*
- statement 3 is the *incrementer.* It executes *after* the loop's code block is executed and, get this, *before* the condition is tested. Why is it listed after the condition instead of before it? I'm sure there's a good, historical reason.

The clever reader may realize that, in fact, there aren't really too many restrictions on what can go into statements 1, 2, and 3. Usually they follow the pattern shown in the example above but they don't have to. The statements don't even need to be filled in! The following is equivalent to a `while(True){}` loop:

Example 01.5.3E

```
for ( ; ; ) {
  // do something forever
}
```

But mostly you will see the `for( ; ; )` loop used according convention, where the first statement declares a temporary variable and initializes it to 0, the second statement tests to see if the temporary variable is bigger than some threshold, and the third statement increments the variable. Also, by convention, the variable used in a for loop like this is called `i` (and if there is another for loop nested inside of it, that one is called `j`, and the next one is `k` and after that you're on your own). Here's a nested example, just to drive all of this home:

Example 01.5.3F

```
for (var i = 0; i < 5; i++) {
  for (var j = 0; j < 5; j++) {
    for (var k = 0; k < 5; k++) {
      console.log(i + ' ' + j + ' ' + k);
    }
  }
} // how many lines will print out?
```

The Upshot (for `for` loops)

- JavaScript has several different styles of `for` loop, unlike Python which essentially has one.
- You can iterate
  - through properties of an object (`for…in`)
  - through elements in a sequence (`for…of`)
  - until a threshold is reached

01.5.4 Break and Continue

Both Python and JS have the `break` and `continue` keywords to control how loops execute. In both

- `break` ends the loop immediately. Execution resumes at the line immediately after the loop.
- `continue` ends the *current iteration* of the loop immediately. Execution resumes at the beginning of the (next iteration of) the loop.

Both `break` and `continue` work with `while`, `do/while`, and `for` loops (all 3 kinds). Here's just one simple example, which is a slightly different and less elegant version of some code we saw earlier (Examples 01.5.2A & 01.5.2B) :

Example 01.5.4A

```
while (true) {
  var r = Math.random();
  if (r < 0.1) {
    console.log(r + " < 0.1. We're done!");
    break;
```

```
  }
  if (r > 0.1 && r < 0.2) {
    console.log(r + " is just a little bigger than 0.1");
    continue;
  }
  console.log(r + " is way bigger than 0.1")
}
```

01.6 Basic Sequences: Strings and Arrays

01.6.1 Strings

In both Python and JavaScript, strings are sequences of characters. Dealing with strings as sequences works pretty much the same in both languages.

```
# in Python
h = "hello"
for c in h:
    print(c) # prints each character -- not sure why you'd eve
r do this
print("The second letter of " + h + " is " + h[1])
```

Example 01.6.1A

```
// in JS
var h = "hello";
for (c of h) {
  console.log(c); // prints each character -- not sure why yo
u'd ever do this
}
console.log("The second letter of " + h + " is " + h[1]);
```

JavaScript does *not* have Python's syntax for slicing, so while you can do this in Python:

```
print(h[2:4]) # prints 'll'
```

You *can't* do this in JS:

```
console.log(h[2:4]); // nope
```

Instead, you'd do it like this:

Example 01.6.1B

```
console.log(h.slice(2, 4)); // prints 'll'
```

or like this

```
console.log(h.substring(2, 4)); // prints 'll'
```

The difference between `slice()` and `substring()` is that `slice()` can accept negative indices. So I guess there's really no reason to use `substring()`.

In both languages there are a zillion things you can do with strings using built-in functions. Here are a few of the most useful. For all of the following, assume that `s` is the string `'hello world!'`.

Example 01.6.1C

```
s = 'hello world!'
```

Python	JS
`s.split(' ') # returns ['hello', 'world!']`	`s.split(' ') // returns ['hello', 'world!']`
`s.replace('h', 'j') # returns "jello world!"`	`s.replace('h', 'j') // returns "jello world!"`
`len(s) # returns 5`	`s.length // returns 5` note that length is a *property,* not a function
`'ll' in s # returns true`	`s.includes('ll') // returns true`
`'xyz' not in s # returns true`	`!s.includes('xyz') // returns true`
`s.strip() # strips whitespace`	`s.trim() // strips whitespace`

`s.upper() # returns 'HELLO WORLD!'`	`s.toUpperCase() # returns 'HELLO WORLD!'`

Of course there are dozens more functions and operations you can do on strings in both languages. For anything not listed here, look up the docs when you need to.

01.6.2 Arrays

What Python calls *lists,* JavaScript calls *arrays.* They are conceptually equivalent, and basic indexing operations work pretty much the same on both.

```
# In Python
fruits = ['apple', 'banana', 'grape']
print(fruits[1]) # print the item at index 1
fruits[0] = 'grapefruit' # modify the first item
print(fruits) // print modified array
```

Example 01.6.2A

```
// In JS
var fruits = ['apple', 'banana', 'grape'];
console.log(fruits[1]); // print the item at index 1
fruits[0] = 'grapefruit'; // replace the first item
console.log(fruits); // print modified array
```

However, after this promising beginning things start to go sour. Many of the specific operations on arrays in JS are different from the operations in Python.

Example 01.6.2B

Python	JS
`# access a sub-list` `print(fruits[1:3])`	`// access a sub-list` `console.log(fruits.slice(1, 3));`

`# add to end`
`fruits.append('kiwi')`
`print(fruits)`
	`// add to end`
	`fruits.push('kiwi');`
	`console.log(fruits);`

# add to front 	
fruits.insert(0, 'mango')
print(fruits)
	// add to front
	fruits.unshift('mango');
	console.log(fruits);

# add to the middle 	
fruits.insert(2, 'cherry')
print(fruits)
	// add to the middle
	fruits.splice(2, 0, 'cherry');
	console.log(fruits);

# remove by index 	
fruits.pop(2)
print(fruits)
	// remove by index
	fruits.splice(2, 1);
	console.log(fruits);

# remove from front 	
fruits.pop(0)
print(fruits)
	// remove from front
	fruits.shift();
	console.log(fruits);

# remove from end 	
fruits.pop()
print(fruits)
	// remove from end
	fruits.pop();
	console.log(fruits);

# get index of a list item 	
print(fruits.index('banana'))
	// get index of a list item
	console.log(fruits.indexOf('banana'));

# see if a list contains an item 	
print('banana' in fruits)
	// see if a list contains an item
	console.log(fruits.includes('banana'));

# get the length of a list 	
print(len(fruits))
	// get the length of a list
	console.log(fruits.length);

# concatenate lists 	
veggies = ['peas', 'broccoli']
foods = fruits + veggies
print(foods)
	// concatenate lists
	veggies = ['peas', 'broccoli'];
	foods = fruits.concat(veggies);
	console.log(foods);

# sort the list alphabetically 	
foods.sort()
print(foods)
	// sort the list alphabetically
	foods.sort();
	console.log(foods);
	
	// sort the list in reverse (?!)

splice有2个以上参数，可以用来插入和删除

`# sort the list in reverse`
`foods.sort(reverse=True)`
`print(foods)`
	`foods = foods.sort(function(a,b) {`
	`  if (b > a) {`
	`  return 1;`
	`  } else if (b == a) {`
	`  return 0;`
	`  } else {`
	`  return -1;`
	`  }`
	`  }`
	`);`
	`console.log(foods);`

`# sort a list of numbers`
`nums = [10, 2, 25, 3, -5]`
`nums.sort()`
`print(nums)`
	`// sort a list of numbers, WCGW?`
	`nums = [10, 2, 25, 3, -5];`
	`nums.sort();`
	`console.log(nums); // ?!?!`
	
	`// force sort() to treat them as numbers`
	`nums.sort(function(a,b) {return a-b;});`
	`console.log(nums);`

传入类似lambda来排序

As with strings, there are lots of other operations you can do on arrays. Consult the docs or do a search if you want to try to do something that isn't on this list. Odds are there's a (simple) way.

01.6.3 The Upshot

Strings and Lists/Arrays are pretty similar in Python and JavaScript. For the most part, anything you can do with either datatype in Python you can also do in JavaScript. Sometimes the syntax is very similar, sometimes it's very different. Whenever you are trying to do something and it isn't working, search for an example.

01.7 Objects (aka "Dictionaries", kinda)

01.7.1 Two ways to think about objects

A central feature of Python is the dictionary or `dict`. JavaScript has no such data type built-in—there is no basic data type called a "dictionary" or anything similar.

Looked at another way, though, JavaScript has dictionary-like functionality built in at an even more fundamental level. *JavaScript Objects are essentially dictionaries*.

According to the JavaScript Object tutorial at W3Schools: "JavaScript objects are containers for **named values** called properties or methods." Well, whaddya know! That's exactly what Python dictionaries are!

1. 对象的key必须是字符串 2. 可以给对象添加自己定义的方法

It turns out that there are a few differences. For one, the syntax is different (sort of). For another, JavaScript property names (which are essentially keys) have to be strings, whereas in Python keys can be "any immutable object," which includes numbers, strings, tuples, and other things. And finally, while JavaScript Objects *can* be seen and used as dictionaries, they can also be seen and used as conventional objects such as the kind you have seen in Python that are intended to represent a single coherent entity. In this regard, JavaScript objects can also contain *functions* that define the behavior of that object—this is not something you would normally do with a dictionary (though in fact you could, but this would be unusual).

The following examples illustrate these two conceptualizations of objects

Example 01.7.1A

```
// Object as a "dictionary"; mapping student IDs to names
var classRoster = {
  id1123: "Jane Doe",
  id3467: "Liz Lemon",
  id7801: "Peter Parker",
};
for (p in classRoster) {
  console.log(p + ": " + classRoster[p]);
}
```

Example 01.7.1B

```
// Object as a coherent entity with different properties
var student = {
  id: "id1123",
```

```
  firstName: "Jane",
  lastName: "Doe",
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
}
console.log(student.fullName());
```

Of course you could combine these conceptualizations, like so:

Example 01.7.1C

```
// classRoster is a dictionary-type object whose values are an
entity-type object
// these kinds of object definitions can be difficult to read
sometimes.

var classRoster = {
  id1123: {
    id: "id1123",
    firstName: "Jane",
    lastName: "Doe",
    fullName: function() {
      return this.firstName + ' ' +
             this.lastName;
    }
  },
  id3467: {
    id: "id3467",
    firstName: "Liz",
    lastName: "Lemon",
    fullName: function() {
```

```
      return this.firstName + ' ' +
             this.lastName;
    }
  }
}

for (k in classRoster) {
  console.log(k + ": " +
              classRoster[k].fullName());
}
```

01.7.2 Object syntax

The examples above introduce the syntax for defining and interacting with objects. Much of this will be familiar to anyone with experience using Python dictionaries and/or objects.

Similar to Python objects and dictionaries

- The beginning and end of the object definition are denoted by curly braces `{ }`

Similar to Python objects

- To reference the properties and/or functions of an object, use the dot `.` As in `object.property` or `object.function()`.

Similar to Python dictionaries

- The key-value pairs are linked together with a colon `:`, in the following format: `key: value`
- The key-value pairs are separated from each other with commas `,` as in `key1: value1, key2: value2`
- The examples above didn't show this, but you can actually reference object properties/functions using square braces `[ ]` as well, just like you do in Python dictionaries. For example:

Example 01.7.2A

```
var student = {
  id: "id1123",
  firstName: "Jane",
  lastName: "Doe",
  fullName: function() {
    return this.firstName + ' ' +
           this.lastName;
  }
}


console.log(student['id']); // this is the same as saying stud
ent.id
console.log(student['fullName']())' // this is the same as stu
dent.fullName()
```

Which style you use is mostly up to you, but there are some cases where one makes more sense than the other or is more elegant. For example, when accessing functions `student.fullName()` is much more elegant than `student["fullName"]()`. On the other hand, sometimes you don't know the specific property name at the time of writing the code and need to use a variable for the key. In that case you have no choice but to use the `[ ]` syntax. For example:

Example 01.7.2B

```
var classRoster = {
  id1123: "Jane Doe",
  id3467: "Liz Lemon",
  id7801: "Peter Parker",
};


var idToLookUp = someFunctionThatGetsUserInput();
var studentName = classRoster[idToLookUp];
console.log("Here's your student: " + studentName);
```

Pause and think

1. The above code won't run because there is a placeholder function that doesn't exist. Replace the function call with an actual ID and run the code. What prints to the console?
- (Assuming line 7 is fixed) Would it work to replace line 8 with `var studentName = classRoster.idToLookup` ? Why or why not? No, because it's string

Adding Properties to Objects

Using JavaScript Objects like dictionaries wouldn't be very useful if you could only create properties (aka 'keys') via hardcoding. You need to be able to create new key-value pairs on the fly. JavaScript can do this, because you can add or delete properties (aka "keys") to/from objects at any time. You can do this using either the `.` or the `[ ]` syntax as well.

Example 01.7.2C

```
// assume classRoster has been defined as in the previous exam
ple

classRoster['id2244'] = "Mr. T";
```

If you now print out the contents of `classRoster` you will see that Mr. T has been added.

Deleting Properties from Objects

You can also delete properties from objects. If you do so, the property along with its associated value will be permanently deleted. For obvious reasons you should not delete properties from any JS Built-in objects, like say `Array` or `String`. Here's how you delete:

Example 01.7.2D

```
delete classRoster['id1123'];
```

Note that this syntax is somewhat different from most of the other things we've seen in JavaScript. Why isn't `delete` a built-in function

(`delete(classRoster['id1123']);`? Or a method on Object (`classRoster.delete('id1123');`)? Instead it's structured like a command that you might see in bash or cmd. I don't know what the story is there, but I'm guessing there is one.

01.7.3 Classes

Let's return to an earlier example (01.7.1C):

```
var classRoster = {
  id1123: {
    id: id1123,
    firstName: "Jane",
    lastName: "Doe",
    fullName: function() {
      return this.firstName + ' ' +
             this.lastName;
    }
  },
  id3467: {
    id: id3467,
    firstName: "Liz",
    lastName: "Lemon",
    fullName: function() {
      return this.firstName + ' ' +
             this.lastName;
    }
  }
}
```

You may have noticed that this was pretty unwieldy and also contained some repetition (both objects define exactly the same function) that seems like it ought to be unnecessary. JavaScript *classes* are intended to help get rid of that redundancy.

There are a lot of powerful and confusing features of JavaScript classes and we won't get too deep into them. Be forewarned, though, that if you start looking into how JS classes work in more detail you should be prepared for a bumpy ride. **Another thing that makes classes in JS slightly tricky is that they're fairly recent**—they were first introduced in ES6, also known as ECMAScript 2015, so it's not hard to find example JS code that does class-like things in even weirder ways, including "prototypes" and using functions to do what classes are now used to do. Many programmers that learned JS a long time ago still use these techniques since they got used to them and since they still work, leading to even more potential for confusion. The good news for us is that React Native sticks with post-ES6 constructs (like classes) so we won't need to deal with the weird old ways ourselves—only sometimes when we're trying to figure stuff out on Stack Overflow and the like (in which case it's often your best bet to just keep searching until you find an example that uses a more modern approach).

For our purposes, we will focus on the most important and most common use of JavaScript classes, which is to serve as a template for a *type of object* that you want to use in your code. To continue the above example, we could define a `Student` class like so:

Example 01.7.3A

```
class Student {
    constructor(fname, lname, id) {
      this.firstName = fname;
      this.lastName = lname;
      this.id = id;
    }
    fullName() {
      return this.firstName + ' ' + this.lastName;
    }
}
```

And then you can use the class to define new `Student`s like so:

Example 01.7.3B

```
var classRoster2 = {
  id1123: new Student("Jane", "Doe", 1123),
  id3467: new Student("Liz", "Lemon", 3467),
  id7801: new Student("Peter", "Parker", 7801),
}
```

Now we can use the objects, for example to print out each student's full name:

Example 01.7.1C

```
for (k in classRoster2) {
   console.log(classRoster2[k].fullName());
}
```

Class syntax

A few notes about some new syntax you just saw:

- you use the `class` keyword to define a class. The word after `class` is the class name. By convention, class name should start with a capital letter. The class definition is enclosed in `{ }`. 不要分号
- the `this` keyword is used to refer to the current object within class *methods* (a "method" is just a function that's part of a class or object). `this` in JavaScript is essentially the same as `self` in Python.
- a class's `constructor` is a special method that is called when you call `new ClassName( )`. It is used to initialize an object of that particular class. As you can see, you can pass parameters into the constructor to initialize particular object instances.
- when you define class methods you don't need to use the `function` keyword. If you just have a function name followed by `( )` at the right place within the class definition, JS knows it's a function (or, more specifically, a class method). 不要function关键字

Now You Try

1. Refer to the definition of `classRoster2` above. What would print when this line executes: `console.log(classRoster2['id3467'].lastName);`?
2. What class function from Python serves the same purpose as JavaScript's `constructor(…)`?

3. What class variable from Python serves (more or less) the same purpose as JavaScript's `this`?

Enter your response at https://app.sli.do/event/t1uqvv8s

Other stuff

There are a few other more "advanced" features of classes that we may need to cover later, including inheritance, `static` methods, and `public` vs `private` methods and variables. But we will cross those bridges when we come to them, if we come to them.

Conclusion

This lesson has introduced you to and/or refreshed your knowledge of basic JavaScript programming syntax and constructs. We've covered a lot of ground and you're not expected to take it all in at once. The DYUs are intended to focus your attention on key aspects of the Lesson to make sure your understanding is reasonably solid. Many of the details are not things you'd be expected to memorize. Most programmers have to look up many of the details covered here regularly.

For example, I can NEVER remember the specific parameters for `splice`, so I pretty much always have to look it up. But what I *do* remember is that there's a way to remove things from strings and arrays, and there's a way to insert things into the middle of them. I *might* remember that the way to do this is called `splice()` and that it does both of these things, but it's much more likely that I'll google "JavaScript delete array elements" and fairly quickly find my way to documentation or an example that reminds me how to use `splice()`. Having a mental model of what you can do with a programming language and knowing how to track down the details is much more important than memorizing a bunch of details that you're going to forget anyway unless you use them ALL THE TIME.