JavaScript for Python Programmers Documentation

Release 0.1c

Jonathan Fine

July 02, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

This provides an introduction of JavaScript for programmers who are already familiar with Python. Its focus is on JavaScript as a programming language, and not the HTML document object model (or DOM).

The author has over 10 years experience of Python and about 3 years of JavaScript. It contains examples and information that he wished he had when he started with JavaScript.

Even if you don't know Python, you may find this useful, particularly if you know another programming language (and you'll learn some Python on the way).

CHAPTER

TWO

EUROPYTHON 2011 TUTORIAL

2.1 Time, date and location

This tutorial will take place on Tuesday 21st June, from 14.30 to 18.30 (2.30pm to 6.30pm) in Training Pizza Napoli.

2.2 Installed software

Participants should bring with them a laptop computer with the following installed:

1. The Firefox web browser, with the Firebug add-on.
2. A programmer's editor, preferably with a JavaScript mode.

   I use emacs, together with js2-mode.
3. A command line JavaScript interpreter.
   - For Linux I suggest SpiderMonkey.

     ```
     $ sudo apt-get install spidermonkey-bin    # Ubuntu
     ```

     (No longer, I'm told, available for Ubuntu 10.4.)
   - For Windows and Mac I suggest JSDB, from http://www.jsdb.org/download.html. It also works on Linux.
4. This documentation, downloaded from Bitbucket downloads.
5. The Python documentation, download from Python docs site.

2.3 Timetable

We start at 14.30 prompt, with software already installed if possible.

I hope to run seven sessions, each about 30 minutes long. They'll be about 20-30 minutes for breaks, as we need then. I hope each session will be 15 minutes of me talking followed by 15 minutes of programming. The topics I'm intending to cover are

1. Demonstration and discussion of the counters example application.
2. JavaScript gotchas.
3. Objects and inheritance, class basics.
4. JavaScript's *this* pseudo-variable (including *call* and *apply*).

We'll certainly have a break here.

5. Closures, modules and memory leaks.
6. Delegation.
7. Review of counters example application.
8. JavaScript and Python objects and classes compared.

We finish at 18.30.

2.4 Is this tutorial for me?

This tutorial aimed at Python web developers who already know a bit of JavaScript, and who need to understand JavaScript better.

The tutorial has two related objectives. One is a good understanding of the counters example. The other is a good understanding of the things that make JavaScript so different from Python (apart from JavaScript being the only language supported by web browsers).

If you're thinking of taking this tutorial take the *Gotcha quiz* and read through the *Counters example*. If you understand what's there a bit, and would like to understand it more, then this tutorial is for you.

2.5 Let me know

If you're going to attend the tutorial, I'd appreciate an email from you that tells me a little bit about yourself and what you'd like to get from the tutorial. If you have specific questions about JavaScript, I'd like to here them also. (I can be contacted at Jonathan.Fine1 at gmail.com.)

You can also, I believe, just turn up on the day, if there is space. Don't forget to install the software on your laptop.

CHAPTER

THREE

GOTCHA QUIZ

On this page are examples of real or potential gotchas. You'll find the answers in *Gotcha answers*, but you'll learn better if you try to answer the questions yourself first.

A **gotcha** is a trap for the unwary. Here there should be fairly obvious, but in the wild they'll be disguised and will sneak up on you. You're focussing on the main thing and then you write a gotcha. **Equality**

```
if (a == b) {
    ...
}
```

Answer.

Addition

```
x = (a + b) + c
y = a + (b + c)
x == y                          // True or False?  When?
```

Answer.

Trailing comma

```
x = [
    'first value',
    'second value',
    'third value',
    ]
```

Answer.

Missing comma

```
x = [
   [1, 2, 3],
   [4, 5, 6]
   [7, 8, 9]
]
```

Answer.

Missing new

```
var Point = function(x, y){
    this.x = x;
    this.y = y;
}
pt = Point(2, 4)
```

Answer.

Bind is transient

```
fn = obj.method;                    // General form of the gotcha.

x = [0, 1, 2, 3, 4, 5];             // Example of gotcha.
a = x.slice(1, 3);                  // What happens behind the scenes?
tmp = x.slice;                      // What happens here?
b = tmp(1, 3);                      // What happens here?
```

Answer.

Missing var

```
var average = function(a, b){
    total = a + b;
    return total / 2;
}
```

Answer.

Missing closure

```
// Add handlers: click on item to show item number.
var make_and_add_handlers(items){

    for(var i=0; i < items.length; i++){

       items[i].onclick = function(){
           alert("This is item " + i);
       };
    }
}
```

Answer.

Missing that

```
// Return fn that does something (and logs the instance).
proto.fn_factory = function(...){

   var fn = function(...){
       ...
       log("Name = " + this.name);
       ...
   }
   return fn;
}

// Example of desired output.
js> instance.name
Charles
js> fn = instance.fn_factory(...)
js> fn(...)                         // Logging to go to console.
Name = Charles
```

Answer.

CHAPTER

FOUR

GOTCHA ANSWERS

This page contains the answers to the *Gotcha quiz*. **Equality**

```
if (a == b) {
    ...
}
```

Ordinary equality (==) can be surprising and so use **===** instead:

```
0 == '0'                    // True
0 == ''                     // True
'' == '0'                   // False.  Not transitive!
undefined == null           // True
```

I don't know of any situations where == is preferred. To compare two values after conversion to string (respectively number) make it explicit by writing respectively:

```
'' + x === '' + y
 +x === +y
```

Question.

Addition

```
x = (a + b) + c
y = a + (b + c)
x == y                              // True or False?  When?
```

In **str + num** the *num* is silently converted to a string.

```
js> typeof ('1' + 2)
string

js> ('1' + 2) + 3
123
js> '1' + (2 + 3)
15
```

In **num + str** the *num* is again silently converted to a string.

```
js> typeof (1 + '2')
string

js> (1 + 2) + '3'
33
js> 1 + (2 + '3')
123
```

If conversion to a string is required make it explicit by writing:

```
'' + a + b + c
```

Question.

Trailing comma

```
x = [
   'first value',
   'second value',
   'third value',
   ]
```

Doing this is good in Python and bad in JavaScript. In Python it makes it easier to reorder, insert and delete values. In JavaScript you'll get, **depending on the browser** a syntax error (IE) or a trailing *undefined* in the array (FF).

Question.

Missing comma

```
x = [
  [1, 2, 3],
  [4, 5, 6]
  [7, 8, 9]
]
```

In Python you'd get an TypeError from this code, as in:

```
py> [4, 5, 6][7, 8, 9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not tuple
```

In JavaScript, we have:

```
js> [4, 5, 6][7, 8, 9] === undefined
true
```

and so the assignment to *x* is equivalent to

```
x = [ [1, 2, 3], undefined ]
```

Question.

Missing new

```
var Point = function(x, y){
    this.x = x;
    this.y = y;
}
pt = Point(2, 4)
```

After running this code we have, at the command line:

```
js> x
2
js> y
4
```

In other words, we've assigned values to the **global object**! In addition, we have:

```
js> pt == undefined
true
```

Question.

Bind is transient

```
fn = obj.method;                    // General form of the gotcha.

x = [0, 1, 2, 3, 4, 5];             // Example of gotcha.
a = x.slice(1, 3);                  // What happens behind the scenes?
tmp = x.slice;                      // What happens here?
b = tmp(1, 3);                      // What happens here?
```

Python uses bound methods and so in Python *a.b* is a first class object.

```
class A(object):

    def b(self):
        pass

py> a = A()                         # Create an instance.
py> a.b
<bound method A.b of <__main__.A object at 0x21d7910>>
```

JavaScript uses what can be called *transient bound methods*. Thus

```
js> a = [1, 2, 3]
1,2,3
js> b = [4, 5, 6, 7, 8]
4,5,6,7,8
js> a.slice === Array.prototype.slice
true
js> a.slice === b.slice      // No implied reference to *this*.
true
js> a.slice(1, 3)            // Transient implied ref to *this*.
2,3
js> b.slice(1, 2)            // Difference transient implied ref.
5,6
```

The function call

```
b = tmp(1, 3);
```

is equivalent to calling Array's slice method on the global object, which is probably not what is wanted!

Question.

Missing var

```
var average = function(a, b){
    total = a + b;
    return total / 2;
}
```

The Python code

```
def average(a, b):
    total = a + b
    return total / 2.0
```

works because in Python *total* is recognised as being local to the function *average*.

In JavaScript variables are global unless explicitly declared as local.

The sample code has a interesting side effect. It assigns to the global variable *total*.

```
js> average(3, 7)
5
js> total
10
```

Question.

Missing closure

```
// Add handlers: click on item to show item number.
var make_and_add_handlers(items){

    for(var i=0; i < items.length; i++){

       items[i].onclick = function(){
           alert("This is item " + i);
       };
    }
}
```

This code, when run, gives all items the same number (namely number of items less one). This is because only one value of *i* is captured. (It is not because all items are getting the same onclick function. They are getting different functions, but with identical behaviour.)

Here's one solution. In the main function instead write

```
items[i].onclick = alert_n(i);
```

where we have:

```
var alert_n(n){
    return function(){
        alert("This is item " + n);
    };
};
```

When we do this, each onclick function captures its own copy of the argument *i* to *alert_n*. (Closure capture function parameters in the same way as it captures function variables.)

Question.

Missing that

```
// Return fn that does something (and logs the instance).
proto.fn_factory = function(...){

   var fn = function(...){
       ...
       log("Name = " + this.name);
       ...
   }
   return fn;
}

// Example of desired output.
js> instance.name
Charles
js> fn = instance.fn_factory(...)
```

```
js> fn(...)                          // Logging to go to console.
Name = Charles
```

The code above (provided the global object does not have a *name* attribute) produces:

```
js> fn(...)                          // Logging to go to console.
Name = undefined
```

When a JavaScript function executes, what *this* refers to depends on the calling context. In particular, in

```
var fn2 = function(...){

    this.attribute = '123';

    return function() {
       return this.attribute;
    }
```

the two occurences of *this* usually refer to completely different objects. To fix instead write:

```
var fn2 = function(...){

    this.attribute = '123';

    var that = this;
    return function() {
       return that.attribute;
    }
```

This works because *that* does not have the special role *this* has. In the returned function, *that* refers to the same object as *that* does in main part of *fn2*, which in turn refers to the same object as the *this* object in the body.

It is customary to write **that = that** to achieve this effect. (Following the custom means no further explanation is required.)

Question.

CHAPTER

FIVE

ANSWERS TO THE EASIEST QUIZ

5.1 Answer 2: Equality is transitive

Here's the question and answer in a concise form.

```
js> a = '0'; b = 0; c = '';
js> [a == b, b ==c, a == c]
true,true,false
```

Clearly, it's not sensible to say that the empty string **''** and the string **'0'** are equal. For a start, they have different lengths.

So perhaps the question should be: Why **a == b** and **b == c**? Given JavaScript's implicit conversion of numbers to strings and vice versa, $a == b$ is sensible. However, it's not clear why $b == c$. Perhaps there's a good reason, but it has an unpleasant consequence.

In any case, in JavaScript it generally (perhaps always) best not to use the == operator. Fortunately it has a === operator, whose behaviour is sensible.

5.2 Answer 4: Assignment assigns

Here's the question and answer.

```
js> words = 'Call me Ishmael.'
Call me Ishmael.
js> words.lang = 'en'
js> lang = words.lang       // What is lang?
js> lang == undefined
true
```

JavaScript has two sorts of strings, which we can call ordinary strings and object strings. Ordinary strings have no attributes or properties. Object strings are object wrappers around ordinary strings. The *String* constructor creates object strings.

5.2.1 Object strings

Object strings have some sensible behaviour:

```
js> typeof 'abc'
string
js> typeof new String('abc')
object
js> x = new String('abc')
abc
```

```
js> x.name = 'def'
def
js> x.name == 'def'
true
```

Object strings have some strange behavior:

```
js> typeof (x + x)
string
```

5.2.2 Ordinary strings

Ordinary strings don't have attributes. When attribute get or set is performed on an ordinary string, the interpreter substitutes an object string in place of the ordinary string. This replacement does not affect the value of any variables.

Thus

```
js> x = 'abc'
js> y = x.charAt(2)
```

is at run time interpreted as

```
js> x = 'abc'
js> y = (new String(x)).charAt(2)
```

which creates an object string from *x* and calls its *charAt()* method. This call gives *y* the expected value. Notice that after the call there are no references to the new object string, and so it is recycled (garbage collected).

We can now explain why in

```
js> words = 'Call me Ishmael.'
Call me Ishmael.
js> words.lang = 'en'
```

the assignment does not stick. It is because the assignment is equivalent to

```
js> (new String(words)).lang = 'en'
```

which assigns an attribute not to *words* but to an anonymous object string wrapper. Object wrapping occurs for both get and set.

The answer to the supplementary question

```
js> lang = words.lang = 'en'  // What is lang?
js> lang
en
```

follow because in JavaScript the value of an assignment expression is always the right-hand side of the assignment.

In Python, assignments are statements, not expressions. In Python, multiple assignment is arises from the parsing rules and not from assignment as an expression. This allow the multiple assignment statement and prohibits:

```
if (a = b):
    pass
```

CHAPTER

SIX

COUNTERS EXAMPLE

6.1 Goal

The goal is to create a web page which contains several independent counters. Each time a counter is clicked, it is incremented. Here's you can try out a working example of what's wanted.

Below is the complete code of this example. To simplify the matter, it is completely self-contained. It does not use any library code.

In general library code is a good idea. This example is designed to teach you the basics of JavaScript, and not the use of a library. We hope that what you learn here will help you choose (and contribute to) a library.

6.2 counters.html

```
<html>
<head>
<script src="counters.js"></script>
<link rel="stylesheet" type="text/css" href="counters.css" />
<title>JS for Python: example: counters</title>
</head>
<body>
<h1>Counters</h1>

<p>Click on a counter to increment its value.</p>

<div id="example">
<p>This will disappear if JavaScript is working properly.</p>
</div>

<p>Return to documentation of <a href="../counters-example.html">Counters example</a>.</p>
</body>
```

6.3 counters.css

```
body {
    background: #DDD;
    font-family: sans-serif;
}

#example {
    padding: 20px;
}
```

```
#example span {
    padding: 10px;
    margin: 10px;
    border: 10px solid blue;
    background: #DDF;
    foreground: blue;
    font-weight: bold;
}
```

6.4 counters.js

```
(function()
{

    var _create_fn = function(){};
    var create = function(parent){

        _create_fn.prototype = parent;
        var instance = new _create_fn();
        return instance;
    };

    var global = (function(){return this;})();

    var counter = {};           // Prototype object for Counter.

    var Counter = function(){

        var instance = create(counter);
        instance.__init__.apply(instance, arguments);
        return instance;
    };

    counter.__init__ = function(name){

        this.name = name;
        this.count = 0;
    };

    counter.onclick = function(event){

        this.count ++;
    };

    counter.html = function(parent_id){

        return this.name + ' ' + this.count;
    };

    global.onload = function(){

        var models = [
            Counter('apple'),
            Counter('banana'),
            Counter('cherry'),
            Counter('date')
        ];

        var element = document.getElementById('example');
```

```
        element.innerHTML = (
            '<span id="a0">apple 0</span>'
            + '<span id="a1">banana 0</span>'
            + '<span id="a2">cherry 0</span>'
            + '<span id="a3">date 0</span>'
        );

        element.onclick = onclick_factory(models);
        element = undefined;    // Avoid IE memory leak.
    };

    var onclick_factory = function(models){

        var onclick = function(event){

            event = event || global.event; // For IE event handling.
            var target = event.target || event.srcElement;
            var id = target.id;
            if (id) {
                var id_num = +id.slice(1);
                var model = models[id_num];
                model.onclick();
                var html = model.html(id);
                if (html){
                    document.getElementById(id).innerHTML = html;
                }
            }
        };
        return onclick;
    };

})();
```

CHAPTER

SEVEN

COUNTERS DISCUSSION

Here we go though the file *counter.js* broken into small pieces, explaining what's going on.

7.1 Namespace

We define and execute an anonymous function in order to create a private namespace. Just like any other function, variables declared and used in the function are not accessible from outside, unless we explicitly provide access. Returning a value is one way to do this. Adding an attribute to the global object, or something accessible from the global object, is another.

```
(function()
{
```

This use of *function* to provide a namespace is similar in some ways to the module concept in Python. Some people call this use of *function* in JavaScript the *module pattern*.

7.2 Create

From one point of view, the *create* function should be, but is not, a built-in function in JavaScript. Instead we have to create it ourselves out of the *new* function, which is but should not be part of JavaScript.

That point of view is that JavaScript's prototype inheritance is best understood for what it is, rather than presented as if *new* in JavaScript is similar to Java's *new*. That point of view also say's that code should be written to use *create* when required, but that use of *new* is considered harmful (except to define *create*, as below).

```
    var _create_fn = function(){};
    var create = function(parent){

        _create_fn.prototype = parent;
        var instance = new _create_fn();
        return instance;
    };
```

The *create* function creates a new object whose parent in the inheritance tree is the *parent* argument. What could be simpler than that? The *create* function is similar to Python's *__new__*, in that it gives the class/parent of an object.

The *new* operator has semantics such that the above code provides an implementation of *create*. However, we don't for now need the somewhat complex semantics of *new*.

7.3 Global object

JavaScript has a global object, whereas Python does not. (Python's main module is similar but in many important ways different.) JavaScript's global object can be the cause of many obscure and hard-to-diagnose problems, and so it's generally best to take care when using it.

Therefore, it is best to make explicit any use of the global object, and a good way to do this is to introduce a variable, called *global* of course, whose value is the global object. That way, when you access the global object or its properties, you can see that's what you're doing.

```
var global = (function(){return this;})();
```

This line of code executes an anonymous function which returns the value of *this* during the execution of the anonymous function. Due to the semantics of JavaScript and the way the function is called, in the anonymous function *this* is the global object. Thus, the anonymous function returns the global object, which we store in the *global* variable (which confusingly is local to the function).

In some situations, such as above, the global object is available as the value of *this*, but sometimes *this* refers to something else. So using *this* to refer to the global object is not a good idea.

In browsers the global object has an attribute called *window* whose value is the global object. It is as if we had written

```
global.window = global;
```

It is better to write *window* rather than *this*, but writing *global* for the global object is best of all. (Command line JavaScript interpreters don't start with *window* as the global object, and in the browser *window* has many special properties.)

7.4 Counter class

Every class needs a prototype object. We'll call it *counter*. Don't confuse the *counter* prototype with a *Counter* instance, which might also be called *counter*. Fortunately, in well organised code the prototype object of a class is in one namespace/module, and instances are in different namespaces.

Here's the implementation of *Counter*. It relies on a function *__init__*, similar to Python's *__init__*, which we haven't defined yet.

```
var counter = {};            // Prototype object for Counter.

var Counter = function(){

    var instance = create(counter);
    instance.__init__.apply(instance, arguments);
    return instance;
};
```

A word about the use of *apply*. In Python we would write something a bit like

```
def Counter(*args, **kwargs):

    instance = object.__new__(counter)
    instance.__init__(*args, **kwargs)
    return instance
```

In JavaScript we don't have keyword arguments, and instead of *args* we have a keyword *arguments* which has special properties. These properties, together with those of *apply*, cause the JavaScript code above to have the same general effect as the Python code.

(We won't sweat the details now. Most of the time use of *call*, *apply*, *create* and *arguments* can and should be hidden behind the scenes. But you need to know that this can be done so that when the time come, **you** can provide an efficient and elegant interface by using them to refactor complexity into something that is put behind the scenes.)

The gist of the above code is that

```
var my_counter = Counter(arg1, ...)
```

causes *my_counter* to be an object, whose parent is the *counter* prototype, and which has been initialised by the *__init__* function.

7.5 Counter properties

The *__init__* method is called, of course, when a newly created child of the *counter* prototype needs to be initialised. Each counter has a *name*, the thing being counted, and its *count*.

```
counter.__init__ = function(name){

    this.name = name;
    this.count = 0;
};
```

When a counter is clicked, its count is increased by one. We'll come to the display of counters next.

```
counter.onclick = function(event){

    this.count ++;
};
```

Often, the most efficient way of changing the content of a DOM node is to use its *innerHTML* property. This method works well with delegation, as with delegation we don't have to add (or remove) handlers from nodes created (or destroyed) in this way.

This implementation is very simple. In a production system *name* would be escaped. The argument *parent_id* is provided in case the *html* method wants to create IDs for its subnodes. This would happen, for example, if the returned HTML was for a slideshow with its controls.

```
counter.html = function(parent_id){

    return this.name + ' ' + this.count;
};
```

7.6 Onload

This code is specific to a particular page or perhaps group of pages. It creates *Counter* instances (sensible, as we want to count) and sets up links between the DOM and the JavaScript.

We can't change the DOM until the nodes we wish to change have been constructed. In browsers the global object has an *onload* event that can be used to solve this timing problem.

Here, when the page loads we create an array of counters.

```
global.onload = function(){

    var models = [
        Counter('apple'),
```

```
        Counter('banana'),
        Counter('cherry'),
        Counter('date')
    ];
```

We also need to link the counters, which are JavaScript objects, to DOM nodes and events. Let's suppose we're to display the counters in an element whose ID is *example*.

This will do the job. Notice the inelegant way in which we initialise the display of the counters. Clearly this won't do if we're inserting say a slideshow into the page. It's an exercise to write something better.

```
    var element = document.getElementById('example');

    element.innerHTML = (
        '<span id="a0">apple 0</span>'
        + '<span id="a1">banana 0</span>'
        + '<span id="a2">cherry 0</span>'
        + '<span id="a3">date 0</span>'
    );
```

We now need to link DOM events to the counters we created. The counters are stored in the *models* variable (which is local to the onload function). Each counter element on the DOM corresponds, via its ID, to a counter instance in the models array. This was done on purpose.

When a counter element on the page is clicked we can, from its ID, find the correspond counter instance in the models and, so to speak, click it. We can also ask the counter instance to generate new HTML for the refreshing of the counter element.

In short, everything has been set up to make delegation as easy as possible. We'll assume that the generic function *onclick_factory* will handle the delegation (and cross browser issues) for us.

```
    element.onclick = onclick_factory(models);
```

This line of code helps prevent a memory leak in Internet Explorer, prior to IE8.

```
    element = undefined;     // Avoid IE memory leak.
```

And finally we can finish the onload function.

```
};
```

7.7 Delegation

In an ideal production environment this code would come from a well-supported standard library. However, we're not there yet.

The *onclick_factory* is an example of a closure. The returned function has a hidden reference to the models argument that is passed to the *onclick_factory*. Each execution of *onclick_factory* refers to the *models* argument that was passed to it.

```
var onclick_factory = function(models){

    var onclick = function(event){

        event = event || global.event; // For IE event handling.
        var target = event.target || event.srcElement;
        var id = target.id;
        if (id) {
            var id_num = +id.slice(1);
```

```
                var model = models[id_num];
                model.onclick();
                var html = model.html(id);
                if (html){
                    document.getElementById(id).innerHTML = html;
                }
            }
        };
        return onclick;
    };
```

7.8 Closing namespace

```
})();
```

This little piece of line noise does three things.

1. It uses **}** to close the function definition.
2. It then uses **)** to close the function expresssion.
3. Finally, it uses **()** to execute the function.

The final semi-colon **;** closes the statement. (It's generally best when writing code to put in the semicolons yourself, rather then let JavaScript put them in.)

CHAPTER

EIGHT

COUNTERS MEMORY

8.1 Memory

Understanding garbage collection and memory leaks is an advanced topic, but even beginners should know that there can be problems here. Correct use of a suitable framework is the simplest way to avoid memory leaks.

The code in the counters example indicates how to construct and use such a framework.

8.2 IE memory leaks

Internet Explorer, prior to IE8 (check) had **two memory heaps**, one for JavaScript and the other for the DOM. Each heap had its own independent garbage collector.

This means that if a DOM node *d* held a reference to a JavaScript object *j* and also that *j* held a reference to *d* then neither garbage collector could collect *j* or *d*. What's worse, even when the page was unloaded IE did not reclaim this memory.

Thus, prior to IE8, JavaScript could cause Internet Explorer to leak memory. This lost memory could be reclaimed only by closing the browser! Closing the page was not enough.

8.3 Garbage collection

The JavaScript garbage collection deletes objects provide it can discover that they can never be used again. The simplest case is:

```
x = [1, 2, 3, 4, 5, 6, 7];
x = null;
```

The second assignment to *x* ensures that the original array (created by the array literal) can no longer be accessed, and so it can be garbage collected.

After the counters namespace anonyomous function has executed there are no global objects holding even indirect references to the function. Therefore it can and will be garbage collected. However, as we shall see, the execution context of the function continues.

The whole of the discussion here reduces to two things

- The execution of the context of the namespace anonymous function continues to exist after the function has completed execution.
- If the line

  were replaced by

  then after the execution of the namespace anonymous function all the JavaScript objects it created would be garbage collected.

The trick is to allow garbage collection to take place by releasing, when the time comes, all DOM reference to JavaScript objects.

8.4 Residue

The *element* DOM node continues to exist, as it can be reached from the *document* node. It has an *onclick* attribute, which is the function created by the *onclick_factory*, with *models* as the argument. So that function is not garbage collected and neither is *models*.

This is just as it should be. Anything that can be reached from the rendered web page is not garbage collected. It has to be there so that click has the desired effect.

The function *element.onclick* retains a reference to *models* just as surely as executing

```
var f = function(arg){ return [1, 2, 3, arg] };
var y = f(x);
```

causes *y* to retain a reference to the value of *x*.

Finally, *models* contains references to the *Counter* instances and thus, by the hidden prototype reference, to the *counter* prototype object.

The *counter* prototype object lies in the execution context of the anonymous function, and as it happend that keeps alive the whole of the execution context.

8.5 Reclaiming memory

Notice that the JavaScript holds only two reference to DOM nodes, namely *document* and *example*. However, the line of code

```
element = undefined;    // Release reference to DOM node.
```

means that the JavaScript no longer holds a reference to *element*.

The DOM node *example* holds a reference, via its *onclick* function, to the JavaScript on the page, and in particular to the *Counter* instances. This cannot be avoided, because we want certain DOM nodes to change the state of the counters.

However, the converse is not true. We can write the JavaScript code so that it has but one reference to a DOM object, namely the *document* node held for example as a property of the *global* object.

8.6 Summary

To avoid the IE memory leaks the trick is to ensure that, when the page unloads, there is nothing that is keeping the garbage is alive. If we execute

```
element.onclick = undefined
```

then there is nothing on the page that is keeping our JavaScript code alive. And once all the JavaScript is garbage collected then there's nothing to prevent all the DOM notde being garbage collected.

The key principle in this strategy is that the page unload event should unbind all JavaScript event handlers and other data attached to the page. This will ensure that the JavaScript is garbage collected, and hence there is nothing to obstruct garbage collection of the DOM nodes.

Clearly, this strategy requires us to keep track of what we add to the DOM. Delegation makes this much easier to do.

Reference:

CHAPTER

NINE

WHAT DO YOU NEED TO KNOW IN PYTHON

9.1 Global and local scopes

Whenever you reference a name in Python, it looks for it in several scopes. First it tries to find it in the *local* scope, which is usually the body of the function that is being currently executed. If the name is not there, Python tries the *global* scope, which is the body of the module. When that fails, it tries the *buildins*.

```
import os
# The name *os* gets defined in the *global* scope.

g = 3
# Define variable *g* in the *global* scope with value 3.

def my_function(x):
    # The name *x* gets defined in *local* scope of *my_function*

    # *x* is found in the *local* scope and its value is
    # assigned to new variable *y* created in *local* scope.
    y = x

    # The name *os* is not found in the *local* scope, so the *global*
    # scope is tried. A new variable *os* is created in the *local* scope
    # with value of the *os* from the *global* scope.
    os = os

    # The name *g* from *global* scope is imported into the *local* scope.
    global g

    # The *global* variable *g* is assigned the value of *local*
    # variable *x*
    g = x

my_function(3)
```

In JavaScript you have a whole hierarchy of scopes – each new nested function creates a new scope. The scopes are searched beginning from the innermost up to the global object.

9.2 Default scope

In Python, when you create a new name, it goes by default to the *local* scope (when you are at the module scope, it's equal to the *global* scope). You don't need to do anything special to make that happen.

In JavaScript, variables go to the global object by default. You have to use the *var* keyword to have them defined locally.

9.3 Functions passed as parameters

Everything is an object in Python. In particular, a function is an object. This means that you can assign a function to a variable or pass it as a function argument. This may be useful when you want to have some generic algorithms. This is used in the *sort* method so that you can, for example, sort strings in case-insensitive way:

```
def lowercase(s):
    return s.lower()

my_list = ['a', 'B', 'c']
my_list.sort(key=lowercase)
print my_list
```

You can pass functions as parameters and assign them to variables in JavaScript too, although there is one important detail that is different about methods.

9.4 Method binding

Methods in Python are bound to object instances. This means that you can pass them around as shown above, and they will still remember what object they belong to.

```
def call_with_a(method):
    method('a')

my_list = []
call_with_a(my_list.append)
print my_list
```

This is not true for JavaScript. In JavaScript the functions and methods don't know where they came from, and they get the reference to their object at the moment when they are called.

9.5 Closures

You can define a function inside other function in Python. That new function then has access to all the variables in the local scope of the initial function. Moreover, if you return that new function from the original function, all the variables in the original function's scope are still alive – because the new function still has access to them. This saved scope is called a closure.

An example use of a closure is a "private" variable that can be only accessed by *get* and *set* functions.

```
def create_get_set():
    """
    Get a pair of functions for getting and setting a hidden value.
    Using a closure.
    """

    value = None

    def get():
        return value

    def set(new_value):
```

```
        value = new_value

    return get, set
```

The *value* variable stays in the memory even after *create_get_set* finishes, because the *get* and *set* functions still hold a reference to it. Because they are the only objects holding reference to it, the variable is effectively hidden from other code. You could have a similar effect by using an object:

```
class GetSet(object):
    def __init__(self):
        self.value = None

    def get(self):
        return self.value

    def set(self, new_value):
        self.value = new_value

def create_get_set():
    """
    Get a pair of functions for getting and setting a hidden value.
    Using a closure.
    """

    getset = GetSet()
    return getset.get, getset.set
```

Here instead of a closure we have an explicit *getset* object. Again, this object stays in memory because the bound methods *get* and *set* hold a reference to it. This couldn't be done in JavaScript.

9.6 Star arguments

You can define a function that takes a variable number of arguments:

```
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return float(total)/len(args)
```

And you can even have a function that takes keyword arguments:

```
def get_arguments(*args, **kwargs):
    return args, kwargs

get_argumenst(1, 2, 3, four=5)
```

You can also call a function and pass it a dictionary or list as the argument:

```
my_list = [1, 2, 3]
my_dict = {'four': 5}

get_arguments(*my_list, **my_dict)
```

In JavaScript there are other mechanisms for doing that: the *arguments* keyword has the list of all parameters passed to a function. These are all positional arguments, JavaScript doesn't have keyword arguments. The *apply* method of a function lets you pass an array of parameters in a function call.

CHAPTER

TEN

OBJECTS

JavaScript and Python have rather different object models and here we give a broad view of the differences. The simplest way to create a JavaScript object **jso** is to write:

```
jso = {}
```

The closest builtin equivalent to jso in Python is the dictionary such as **pyo**, which is created using:

```
pyo = {}
```

The two objects jso and pyo have important similarities and differences (mostly differences).

10.1 Late news

Not incorporated in this document is a recent discovery, that

> **JavaScript objects are like Python classes with custom item methods (on the metaclass) that are never instantiated.**

I've put this material in a page on *JavaScript objects*.

10.2 Similarities

Both objects can be used to store data. Here's a lightly edited command line session (using the SpiderMonkey JavaScript interpreter):

```
js> jso = {}
js> jso['A'] = 'apple'
js> jso['A']
apple
```

Here's the corresponding Python session (with *sys.ps1 = 'py> '*):

```
py> pyo = {}
py> pyo['A'] = 'apple'
py> pyo['A']
'apple'
```

In the examples above *A* is the **key** and **apple** is the value. In both cases we are able to store values against keys. In both cases the value can be any object. In most other respects, jso and pyo are different.

10.3 Differences

10.3.1 Keys

In JavaScript all keys are converted to strings.

```
js> jso = {}
js> jso['1'] = 'one'
js> jso[1]
one
```

Here's an even more surprising example:

```
js> jso = {}
js> a = {} ; b = {};
js> jso[a] = 'apple'
apple
js> jso[b]
apple
```

What's going on here? Well, objects *a* and *b* are converted to strings, and they have the same string representation, namely *[object Object]*. (We'll see later how we can change this.) So let's try this out (continuing the previous example):

```
js> jso['[object Object]']
apple
```

Here, for comparison, is what happens with Python:

```
py> pyo = {}
py> pyo['1'] = 'one'
py> pyo[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
py> a = {}; b = {}
py> pyo[a] = 'apple'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Here JavaScript, faced with input of the wrong type, has converted it to the required type, which is a string.

Note: In JavaScript anything can be a key, and all keys are converted to strings before lookup. In Python, keys are unchanged, and have to be hashable.

Note: Python built-in objects throw an error when given incorrect input. JavaScript converts the input to something that might work.

10.3.2 Missing keys

The notation *a[b]* is called **indexing**. In Python, indexing with a key that is not present in the dictionary raises an exception.

```
py> pyo = {}
py> pyo['dne']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'dne'
```

In JavaScript, we get *undefined*, which is a special JavaScript object (which is diferent from the *null* JavaScript object).

```
js> jso = {}
[object Object]
js> jso['dne']
js> jso['dne'] === undefined
true
```

(SpiderMonkey does not echo *undefined* at the command prompt, just as Python does not echo *None*. Therefore we use equality to show that we truly get undefined.)

Note: In JavaScript you won't get a KeyError (or IndexError). Instead, you get **undefined**.

In practice, JavaScript left to itself rarely throws an error. It most commonly happens with code such as

where respectively *obj_a.method* and *obj_name* are undefined.

10.3.3 Attributes

In JavaScript an object has a single set of properties, which can be accessed both by indexing and as attributes. Here is an example:

```
js> jso = {}
js> jso['a'] = 'apple'
js> jso.a
apple
js> jso.b = 'banana'
banana
js> jso['b']
banana
```

In Python a dictionary has a set of key-value pairs and in addition it has dictionary attributes, which are methods.

Note: In JavaScript the three statements below are completely equivalent (except for assignment to *tmp* as a side effect).

```
x = a.b
x = a['b']
tmp = 'b'; x = a[tmp]
```

In addition, when *a.b* is a function then

```
x = a.b(arg1, arg2, arg3)
tmp = 'b'; x = a[tmp](arg1, arg2, arg3)
```

are equivalent.

But note also that

```
x = a.b(arg1, arg2, arg3)
tmp = a.b; x = tmp(arg1, arg2, arg3
```

are sometimes completely different.

CHAPTER

ELEVEN

JAVASCRIPT OBJECTS

11.1 Like Python classes

In JavaScript all objects are part of an inheritance tree. The **create** function adds a node to the inheritance tree.

```
// A JavaScript object.
js> root = {}

// Prototype inheritance.
js> create = function (obj) {
        var f = function () {return this;};
        f.prototype = obj;
        return new f;
    }

js> a = create(root)
js> b = create(a)

js> a.name = 5
js> a.name
5
js> b.name
5
```

In Python classes inherit in the same way.

```
>>> root = type              # Most classes are instance of type.
>>> class a(root): pass
>>> class b(a): pass         # Class inheritance.

>>> a.name = 5               # Just like JavaScript.
>>> a.name
5
>>> b.name
5
```

11.1.1 class explanation

In Python we can subclass anything whose type is **type** (or a subclass of type). A subclass (and its instances) inherits properties from the super-class.

```
>>> type(root) == type(a) == type(b) == type
True
```

11.2 Custom item methods

In JavaScript attribute and item access are the same.

```
js> a = create(root)

js> a.name = 5
js> a['name']
5

js> a['key'] = 6
js> a.key
6

js> a[1] = 6
js> a['1']
6
```

In Python we can defined our own item methods. (The programmer owns the dot.)

```
>>> class A(object):
...
...     def __getitem__(self, key):
...         return getattr(self, str(key))
...     def __setitem__(self, key, value):
...         return setattr(self, str(key), value)

>>> a = A()
>>> a.name = 5

>>> a['name']
5

>>> a['key'] = 6
>>> a.key
6

>>> a[1] = 6
>>> a['1']
6
```

Because **type(a)** is **A**, which has the special item methods, we get the special item behaviour.

```
>>> type(a) is A
True
```

11.3 On metaclass

Using previous definition, we cannot subclass **a** to create **b**.

```
>>> class b(a): pass
Traceback (most recent call last):
    class b(a): pass
TypeError: Error when calling the metaclass bases
    object.__new__() takes no parameters
```

This is because **a** is not a type. The solution involves Python metaclasses (an advanced topic).

```
>>> isinstance(a, type)
False
```

11.3.1 metaclass construction

We will subclass type, not object, and add to it the special item methods.

```
>>> class ObjectType(type):
...
...     def __getitem__(self, key):
...         return getattr(self, str(key))
...
...     def __setitem__(self, key, value):
...         return setattr(self, str(key), value)
```

Here is a fancy way of calling **ObjectType**.

```
>>> class root(object):
...     __metaclass__ = ObjectType
```

Here is a more direct (and equivalent) construction (create an instance of **ObjectType**, whose instances are objects).

```
>>> root = ObjectType('root', (object,), {})
>>> isinstance(root(), object)
True
```

11.3.2 metaclass demonstration

```
>>> class a(root): pass
>>> class b(a): pass

>>> a.name = 5
>>> a.name
5
>>> b.name
5
>>> a['name']
5
>>> b['name']
5

>>> a[1] = 6
>>> a['1']
6
```

11.3.3 metaclass explanation

Because **type(root)** is a subclass of type we can subclass root.

```
>>> issubclass(type(root), type)
True
```

Because the **type(root)** is **ObjectType**, which has special item methods, we get the special item behaviour.

```
>>> type(root) == type(a) == type(b) == ObjectType
True
```

11.4 Never instantiated

We can't call JavaScript objects (unless they are a function). But **create** creates ordinary JavaScript objects.

```
js> a = create(root)
js> a(1, 2, 3)
TypeError: a is not a function
```

We will monkey-patch the previous Python class, to provide custom behaviour when called.

```
>>> def raise_not_a_function(obj, *argv, **kwargs):
...     raise TypeError, obj.__name__ + ' is not a function'

>>> ObjectType.__call__ = raise_not_a_function

>>> a(1, 2, 3)
Traceback (most recent call last):
     a(1, 2, 3)
TypeError: a is not a function
```

11.5 Conclusion

JavaScript objects are like Python classes (because they inherit like Python classes).

For JavaScript attribute and item access are the same. This is achieved in Python by providing custom item methods.

In Python the custom item methods must be placed on the type of the object (or a superclass of its type).

Ordinary JavaScript objects are not functions and cannot be called. A Python class can be called (to create an instance of the object). But we can override this behaviour by supplying a custom method for call.

To summarize: ..

> **JavaScript objects are like Python classes with custom item methods (on the metaclass) that are never instantiated.**

It's worth saying again:

> **JavaScript objects are like Python classes with custom item methods (on the metaclass) that are never instantiated.**

CHAPTER

TWELVE

INHERITANCE

Objects have attributes. In both JavaScript and Python the statements

```
value = a.name                  # Get the name of 'a'.
a.name = value                  # Set the name of 'a'.
```

respectively **set** and **get** the *name* attribute of the object *a*. Inheritance is where an object gets some its attributes from one or other more objects.

JavaScript and Python handle inheritance differently. Here we describe what JavaScript does, and later [where] we compare with Python.

12.1 Tree

In JavaScript all objects are part of an inheritance tree. Each object in the tree has a parent object, which is also called the prototype object (of the child). There is a single exception to this rule, which is the **root** of the tree. The root of the tree does not have a parent object.

You can't get far in JavaScript without understanding the inheritance tree.

12.1.1 Get

When JavaScript needs to get an attribute value of an object, it first looks up the name of the attribute in the object's dictionary. If the name is a key in the dictionary, the associated value is returned.

If the name is not a key, then the process is repeated using the object's parent, grandparent, and so on until the key is found. If the key is not found in this way then (see *Missing keys*) *undefined* is returned.

12.1.2 Set

When JavaScript needs to set an attribute value of an object it ignores the inheritance tree. It simply sets that value in the object's dictionary. With Python its just the same. (However, JavaScript's `pseudo-objects` are don't behave in this way.)

12.2 Root

When the interpreter starts up, the **root** of the tree is placed at *Object.prototype*.

Every object inherits from the **root**, although perhaps not directly. Here's an example:

```
js> root = Object.prototype
js> a = {}
js> a.name === undefined
true
js> root.name = 'gotcha'
js> a.name
gotcha
```

Once we give *root* a *name* attribute every other object, including those already created and those not yet created, also has a *name* attribute with the same value.

However, this does not apply if *name* is found earlier in the tree. We continue the previous example to show this, and the behaviour of set.

```
js> a.name = 'fixed'
js> a.name
fixed
js> root.name
gotcha
```

12.3 Using create

Any tree can be constructed from its root, together with a command **create(parent)** that returns a new **child** of the given parent node.

In JavaScript the *create* function is not built in (although perhaps it should be). However, it's easy to write one [link], once you know enough JavaScript.

Here's an example of its use:

```
js> a = {}
js> b = create(a)
js> a.name = 'apple'
apple
js> b.name
apple
```

And a continuation of the example:

```
js> c = create(b)
js> c.name
apple
js> b.name = 'banana'
banana
js> c.name
banana
```

Note: JavaScript uses an inheritance tree. By using *create*, we can create any inheritance tree. All JavaScript objects are in this tree.

CHAPTER

THIRTEEN

FUNCTIONS

A function contains code, possibly with parameters, that is stored for later use. In both JavaScript and Python, functions are first-class objects. This means that identifiers can have a function as their value, and be used as the parameter or return value of a function.

It's common, when writing JavaScript for web pages, to create a large number of anonymous (nameless) and similar event-handing functions, one for each node. Often, `delegation` allows us replace a many similar functions by a single function.

13.1 Defining a function

When defining a function in JavaScript use

```
var my_fn = function(arg1, arg2, arg3){
    //  Body of the function.
};
```

as the template, where of course you get to choose the number and names of the parameters. Unlike Python, you can't provide default values for the arguments, nor can you provide ***args** and ****kwargs**. However, as we will see, there is the pseudo-variable *arguments* in JavaScript.

This template can be adapted

```
obj.my_fn = function(arg1, arg2, arg3){
    //  Body of the function.
};
```

to define an attribute or method of an object. This is most commonly done when *obj* is a prototype.

You can also put a function in an object literal, as in

```
obj = {

    'my_fn': function(arg1, arg2, arg3){
    //  Body of the function.
    }
};
```

but this is often best avoided.

JavaScript provides another method (called function declaration)

```
function my_fn(arg1, arg2, arg3){
    //  Body of the function.
};
```

which should (in the author's view) never be used. It has strange properties [link].

13.2 Calling a function

There are four ways of calling a function in JavaScript. The difference involves *Functions and this*, which we've not covered yet. In brief, the four forms are:

```
x = my_fn(a, b, c);                // Simple function call.
x = obj.my_fn(a, b, c);            // Method  call.
x = my_fn.call(obj, a, b, c);      // Explicit this call.
x = my_fn.apply(obj, [a, b, c]);   // Explicit this apply.
```

For all forms of the function call, supplying the wrong number of arguments **does not raise an error**. Instead, excess arguments are ignored, and excess parameters are initialised to *undefined*.

13.2.1 Function call

This is the simplest form of function call.

```
x  = my_fn(a, b, c);
```

The this-object is set to the `global-object`, which causes a problem only if the body of the function refers to *this*. Many functions are intended to be used in this way.

13.2.2 Method call

This is the usual, but not the most general, way of setting the this-object.

```
x  = obj.my_fn(a, b, c);        // Usual form.
x  = obj['my_fn'](a, b, c);     // Variant form.
```

The two forms are completely equivalent. The this-object is set to the *obj*.

The two following calls are always equivalent:

```
x  = a.b.c.d.e.my_fn(a, b, c);
tmp = a.b.c.d.e
x  = tmp.my_fn(a, b, c);
```

13.2.3 Method call gotcha

In JavaScript the two following calls are rarely equivalent:

```
x  = obj.my_fn(a, b, c);
tmp = obj.my_fn
x  = tmp(a, b, c);
```

whereas in Python they always are.

For JavaScript, in the first the this-object is *obj* while in the second it is the global-object. For explanation see [link].

13.2.4 Explicit this-call

Suppose *obj* is a JavaScript object. Then

```
x = my_fn.call(obj, a, b, c);
```

is equivalent to

```
x = my_fn(a, b, c);
```

except that the this-object is set to *obj* (rather than the global object). For more see [link].

13.2.5 Explicit this-apply

Suppose *obj* is a JavaScript object. Then

```
x = my_fn.apply(obj, [a, b, c]);
```

is equivalent to

```
x = my_fn(a, b, c);
```

except that the this-object is set to *obj*. For more see [link].

13.2.6 Explicit this gotcha

If *obj* is not a JavaScript object then in

```
x = my_fn.call(obj, a, b, c);
x = my_fn.apply(obj, [a, b, c]);
```

the this-object is set to the global object (which is probably not what you want). For more see [link].

CHAPTER

FOURTEEN

FUNCTIONS AND *THIS*

Understanding *this* in JavaScript is hard, but also vital. It corresponds, roughly, to the use of **self** in Python. But there are big differences. We start with the pitfalls, then give the valuable use, and then give another pitfall.

We need *this* to do efficient object oriented programming in JavaScript. But there are potential pitfalls, which we describe first. We then describe the useful applications of *this*.

Note: It's not possible to get far in JavaScript without understanding **this**. But understanding **this** is helps you become a confident expert. Please persevere until you get it.

14.1 Pitfalls

Here we describe some of the traps that await the unwary.

14.1.1 *this* is a keyword

Unlike Python's *self*, in JavaScript *this* is a keyword. It is not possible to explicitly assign a value to *this*.

```
js> this = 1
typein:2: SyntaxError: invalid assignment left-hand side:
typein:2: this = 1
typein:2: .....^
```

We have here a syntax error. As in Python, each identifier in JavaScript, potentially, refers to an object. We can make the identifier refer to a different object by writing:

```
a = new_value;                // Sometimes called 'rebinding'.
```

But *this* is not an identifier. It is a keyword, and

```
this = new_value;
```

is a syntax error. Good editors highlight *this* to point out its special role.

14.1.2 Mutating *this*

Even though we cannot assign to *this*, we can mutate the this-object. It is best to think of *this* as quasi-fixed but mutable object. (There are ways of changing what *this* refers to, but not in the body of an executing function.)

```
js> a = this;
js> a.name = 'gotcha'         // Mutating the *this* object.
js> b = this;
js> a === b && a === this     // a, b and *this* are the same object.
```

```
true
js> this.name                  // We have mutated *this*.
gotcha
```

The keyword *this* is **bound** to the same object as before, but that object has been **mutated**. (There are two ways to 'change <identifier>', namely rebinding and mutation. But rebinding of *this* is not allowed.)

14.1.3 Global *this* gotcha

JavaScript has a global object, which roughly corresponds to the __main__ module in Python. However, unlike __main__, we should try to avoid using and changing JavaScript's global object. [link]

Let's continue the previous example. We just mutated the *this* object, by adding a name to it. But what object did we change? In fact, we've just added a new global value.

```
js> name
gotcha
```

Even if you don't understand why this happened (and it's not been explained yet [link]), it is something that quality code must avoid.

Note: Whenever you mutate **this**, be sure you know what object **this** is.

14.2 Methods and *this*

There are two ways to set *this*, namely implicit and explict. Here we describe the implicit method. The rule is quite simple.

Suppose we execute

```
result = obj.method()
```

where *obj.method* is a function. (If *obj.method* is not a function we get a runtime error.)

In this situation, *this* is bound to *obj* during this execution of *method*. Here is an example:

```
js> var obj = {}
js> obj.name = 'apple'
apple
js> obj.method = function(){return this.name}
js> obj.method()
apple
```

Here's a similar example, involving inheritance:

```
js> parent = {}
js> parent.method = function(){return this.name}
js> a = create(parent)
js> a.name = 'apple'
apple
js> a.method()
apple
```

14.3 Explicit this

For a summary, see the earlier section *Calling a function*. For the details, see [link] and for examples see [link].

CHAPTER

FIFTEEN

CLASSES

In Python, *class* is a language primitive. JavaScript has no built-in concept of class. Therefore, if we are to have classes in JavaScript then we (or some library) must implement them. Here we give an outline implementation of classes in JavaScript, so that the basic language features can be understood.

Recall that each object has a parent, which in turn has a parent, all the way up to the root object (which has no parent). In Python (and other languages), classes are a way for instances to share data and methods. As a first approximation, let us say *two objects belong to the same class if they have the same parent*.

As well as sharing class data, each instance of a class has its own data, which is added to the instance as part of its creation. In Python, this is usually done using the __init__ method (although sometimes __new__ is also used).

15.1 Point in Python

In Python we might write:

```
class Point(object):

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, dx, dy):
       '''Move point by changes dx and dy.'''
        self.x += dx
        self.y += dy

mypoint = Point(2, 3)
mypoint.move(1, 1)
```

15.2 Point in JavaScript

To implement a similar class in JavaScript we need a function Point that returns a point, and an object that is a parent for all the points created by the Point function. In Python we had a single object, namely the class Point, that performed both roles. In JavaScript we need a constructor function and a parent object.

15.2.1 Parent object

We'll start with the parent object:

```
var base = {};                // Grandparent for all instances.
var point = create(base);     // Parent to all points.

point.__init__ = function(x, y){
    this.x = x;
    this.y = y;
};

point.move = function(dx, dy){
    this.x += dx;
    this.y += dy;
};
```

See [link] for why we use this rather than self.

15.2.2 Constructor

We also need a constructor function

```
var Point = function(x, y){

    var instance = create(point);
    instance.__init__(x, y);
    return instance;
};
```

In production we would use *arguments* and *apply* rather than directly calling *__init__*. Do you know why?

15.3 Advanced features

This can be omitted at a first reading.

15.3.1 Improved toString

In Python we have, for example:

```
py> from point import Point
py> Point(2, 3)
<point.Point object at 0x20ead90>
```

We can get something similar in JavaScript by writing:

```
point.__classname__ = 'Point';
base.toString = function(){
    return '<' + this.__classname__ + ' object>'
};
```

15.3.2 Constructor factory

Here is a factory function for producing constructor functions, such as Point above.

```
var constructor_factory(parent){
    return function(){
        var instance = create(parent);
        instance.__init__.apply(instance, arguments);
```

```
        return instance;
    };
};

var Point = constructor_factory(point);
```

Resources:

CHAPTER

SIXTEEN

DOWNLOADS

- **/examples/counters.html**.
- **/examples/counters.css**.
- **/examples/counters.js**.
- *Search Page*