Life After JavaScript: A Tour of the Options

This is a post about running Python in the web browser - that is, using Python to build dynamic web UIs, rather than needing to use JavaScript – and the different approaches to making it possible.

I gave this as a talk at PyCon UK 2019. Here's a video:

A split brain

Let me tell you about me: I used to be a back-end Python developer. I was happy. And then I got a job writing front-end JavaScript code too. This was…less happy. I was using two languages with very different ways of thinking. Both my brain and my app logic were getting scrambled between the two, and I was constantly missing the features and usability of Python.

As a Python programmer, I want to use classes - especially if I'm building a UI, something that's inherently object-oriented. I want inheritance heirarchies. I want keyword arguments. And I want to know what `this` (or `self`) means at any given point in my code!

So I was thinking: "Why do I **have to** use JavaScript? Why am I constrained to this one language?" Imagine if Microsoft Windows only let you write Ruby. It would not be a very

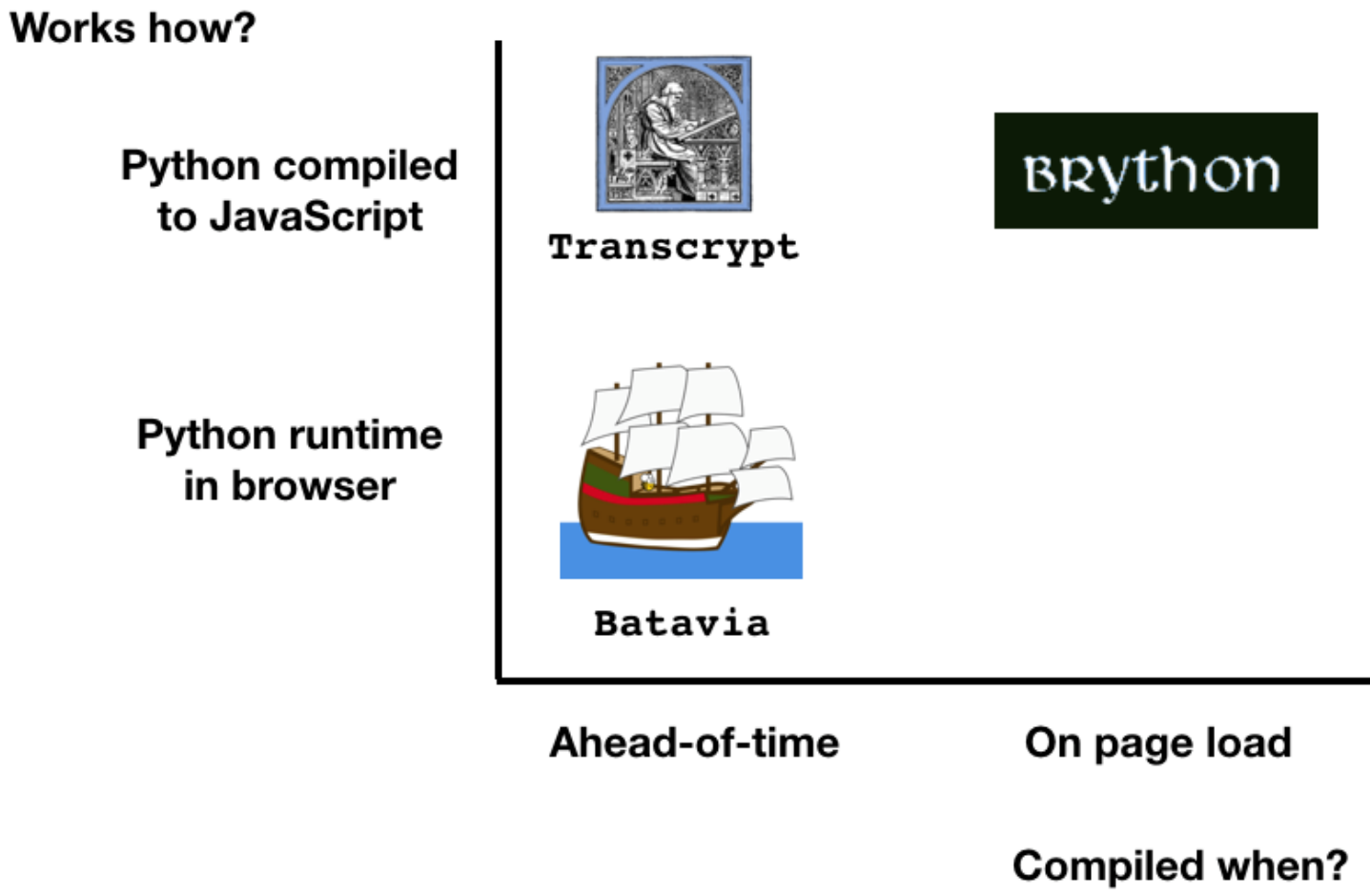
popular operating system! And yet web browsers *are* an application delivery platform that only supports one language.

This question had already been playing on my mind when I came to PyCon UK two years ago, so I was immediately drawn to the Anvil stand, with its promise of writing Python in the web browser. (I ended up working here, so that went well!) But Anvil is a whole integrated platform for building web apps, and the library Anvil uses (Skulpt) isn't the answer for everyone. So what are the other options?

I'm going to take you on a tour of the design space, by comparing six different ways to run Python code in the browser. **There will be demos!**

A tour of six options

These six technologies that are representative examples of what's out there. They're all relatively mature, and they cover a lot of the design space. I've put them on two axes:



Our first big design choice is on the Y-axis: **How is the Python being run by the browser?** You have two options: either translate the Python code to Javascript and then run that, or implement a Python interpreter in the browser and pass your Python through it.

The second choice, on the X-axis, is "**When does the Python get compiled?**". You can compile ahead-of-time, or after the page has loaded. This is a tricky one - you can trade off development convenience, on-page performance, and dynamism. (Interactive REPLs in the browser!)

1. Transcrypt: compiling ahead-of-time

The first we'll talk about is Transcrypt. This is a compiler that translates Python into JavaScript ahead-of-time. I'll show you a quick example:

I have two files: `hello.html` and `hello.py`.

```
~/python-browser/transcrypt$ ls
hello.html      hello.py
```

`hello.html` is an incredibly simple HTML file: it has an input box and a button.

```
<head>
  <script type="module">import * as hello from
"./__target__/hello.js";</script>
</head>

<body>
  <input id="name-box" placeholder="Enter your name">
  <button id="greet-button">Say Hello</button>
</body>
```

When I type my name into the input box, I'm going to get an alert. The dynamic behaviour is implemented by the script I'm importing at the top, and the script is compiled from `hello.py`:

```
def greet():
    alert("Hello " + document.getElementById("name-box").value + "!")

document.getElementById("greet-button").addEventListener('click', greet)
```

This is approximately the simplest web app you've ever seen. The python code isn't *quite* the simplest, because it has to access the DOM. There's a `greet` function that uses `getElementById` to get the `name-box` value. Then I'm binding that to the `click` handler of the button with another call to `document.getElementById`.

This is a very JavaScript-y way of interacting with my UI. This is a deliberate design choice by Transcrypt; they've imported the same objects and methods you'd use in JavaScript for accessing the Document Object Model (the DOM). This will be familiar if you've used

JavaScript before, and allows you to use the JavaScript documentation as a reference. The downside, of course, is that you've still got all the complexity of the JS APIs!

I compile that Python to JavaScript by running `transcrypt -n hello.py`:

```
~/python-browser/transcrypt$ transcrypt -n hello.py

Transcrypt (TM) Python to JavaScript Small Sane Subset Transpiler Version
3.7.16
Copyright (C) Geatec Engineering. License: Apache 2.0


Saving target code in: /Users/stm/python-
browser/transcrypt/__target__/org.transcrypt.__runtime__.js
Saving target code in: /Users/stm/python-
browser/transcrypt/__target__/hello.js

Ready
```

This has created a directory called `__target__`, which contains my compiled JavaScript file, `hello.js`:

```
~/python-browser/transcrypt/__target__$ ls
hello.js                hello.project           org.transcrypt.__runtime__.js
```

`hello.js` is actually fairly readable - it's very similar to my original Python. I've got the `alert`, I've got the `document.getElementById`…the main thing that's changed is the function definition is very JavaScript-y (and, in my opinion, not as pretty).

```
// Transcrypted from Python, 2019-09-18 12:04:42
import {AssertionError, AttributeError, BaseException, DeprecationWarning,
Exception, IndexError, IterableError, KeyError, NotImplementedError,
RuntimeWarning, StopIteration, UserWarning, ValueError, Warning,
__JsIterator__, __PyIterator__, __Terminal__, __add__, __and__, __call__,
__class__, __envir__, __eq__, __floordiv__, __ge__, __get__, __getcm__,
__getitem__, __getslice__, __getsm__, __gt__, __i__, __iadd__, __iand__,
__idiv__, __ijsmod__, __ilshift__, __imatmul__, __imod__, __imul__, __in__,
__init__, __ior__, __ipow__, __irshift__, __isub__, __ixor__,
__jsUsePyNext__, __jsmod__, __k__, __kwargtrans__, __le__, __lshift__,
__lt__, __matmul__, __mergefields__, __mergekwargtrans__, __mod__, __mul__,
__ne__, __neg__, __nest__, __or__, __pow__, __pragma__, __proxy__,
__pyUseJsNext__, __rshift__, __setitem__, __setproperty__, __setslice__,
__sort__, __specialattrib__, __sub__, __super__, __t__, __terminal__,
__truediv__, __withblock__, __xor__, abs, all, any, assert, bool, bytearray,
bytes, callable, chr, copy, deepcopy, delattr, dict, dir, divmod, enumerate,
filter, float, getattr, hasattr, input, int, isinstance, issubclass, len,
list, map, max, min, object, ord, pow, print, property, py_TypeError,
py_iter, py_metatype, py_next, py_reversed, py_typeof, range, repr, round,
set, setattr, sorted, str, sum, tuple, zip} from
'./org.transcrypt.__runtime__.js';
var __name__ = '__main__';
export var greet = function () {
```

```
        alert (('Hello ' + document.getElementById ('name-box').value) + '!');
};
document.getElementById ('greet-button').addEventListener ('click', greet);

//# sourceMappingURL=hello.map
```

And then there's this **massive** import statement at the top. This is importing a bunch of things you're probably familiar with from the Python builtins. They're not actually Python, though: they're hand-written JavaScript implementations of those things, courtesy of `org.transcrypt.__runtime__.js`. For example, Python has a `callable` function that tells you if an object can be called. The JavaScript implementation of that is a single boolean expression that checks if it's an object, then checks if it has a `__call__` method. This has been handwritten to replicate the behaviour of Python:

```
export function callable (anObject) {
    return anObject && typeof anObject == 'object' && '__call__' in anObject
? true : typeof anObject === 'function';
};
```

The other file, `hello.project`, contains some metadata that the compiler likes to have.

Here's that example, live on this page. You can enter a name into the input box, and you get an alert when you click the button:

So I've written some Python, and we've got a dynamic web application - albeit the simplest one you can imagine. (If you don't believe me, you can inspect this page in your browser tools and see my auto-generated JavaScript.)

But Transcrypt is not exactly a one-to-one switch-in for JavaScript yet. I still can't write a `<script>` tag that contains some Python.

2. Brython: Python script tags

If I want a Python `<script>` tag, I can use Brython. Brython is a compiler that's written in JavaScript, so I can run it with my web browser. Let's look at the same example, but this time written in Brython.

```
~/python-browser/brython$ ls
hello.html
```

Now I only have one file, `hello.html`. I still have my input box and button as before, but now I have a script whose type is `text/python`.

```
<body onload=brython()>
    <script type="text/python">
        from browser import document, alert
```

```
        def greet(event):
            alert("Hello " + document["name-box"].value + "!")
        document["greet-button"].bind("click", greet)
    </script>

    <input id="name-box" placeholder="Enter your name">
    <button id="greet-button">Say Hello</button>
</body>
```

My Python is very similar to what I had before, but a bit more comfortable. The way I access the DOM has a bit more syntactic sugar: I'm accessing `document` like a dictionary, using square-bracket notation, and my `bind` method is nice and succinct too. I imported `document` and `alert`, rather than having them as magic Javascript-y globals. It all feels a bit more Pythonic.

To actually run this code, I have a `<script>` tag that includes the Brython compiler. It defines a global `brython()` function that I call on page load. When the page loads, Brython will find all the Python `<script>`s, turn them into JavaScript, and run them.

```
<head>
    <script type="text/javascript"

src="https://cdnjs.cloudflare.com/ajax/libs/brython/3.7.5/brython.min.js">
    </script>
</head>

<body onload=brython()>
  <!-- ... and the rest of the body as before ... -->
```

Now, my example is going to look very similar - in fact, identical - but when I inspect it I actually have my Python in the page source.

```
<body onload="brython()">
  <script type="text/python">
        from browser import document, alert
        def greet(event):
            alert("Hello " + document["name-box"].value + "!")
        document["greet-button"].bind("click", greet)
    </script>
    <input id="name-box" placeholder="Enter your name">
    <button id="greet-button">Say Hello</button> event
    <script></script>
  </body>
```

Python in the browser - literally!

You can modify that Python script and re-run the compiler (by running `brython()` in the browser console). But if you do that, you'll actually have two click handlers, because it has bound both and they're both still in memory.

So, now I have essentially Python as a direct replacement for JavaScript, but dynamically editing code doesn't go so well – I still need to write my Python before the page loads. What if I want to build an interactive Python coding environment? What if I want a REPL? That's what Skulpt was built for.

3. Skulpt: interactive Python in the browser

Skulpt was originally written for education, to provide an interactive Python environment in the browser.

If you're just getting someone started with Python, you really don't want their first experience to be downloading and installing it! They might ruin their system Python in the process, you've got as many different environments as you have students, and you have to debug them all before anyone can write their first `print` statement. But if they have an environment in the browser, they can just go ahead and write some code.

A typical Skulpt example looks like this:

There's a box at the top to write some Python, and somewhere where the output arrives. When you hit the Run button, it compiles and runs your code, all in the browser (no network traffic!).

You can edit that code – try it! You'll get some output from Python code that didn't exist when the page loaded.

There's a really cool project called trinket.io that gives students an interpreter in the browser, and that's based on Skulpt.

Here's how that example works. Here's the HTML that defines the textareas:

```
<body>
  <textarea id="yourcode" cols="40" rows="10" style="width: 100%; height:
60px;">
print("Hello World")
print(4**2.)
  </textarea>
  <textarea id="output"  style="width: 100%; height: 60px;"></textarea>
  <button type="button" onclick="runit()" style="width: 100%; height:
10px;">Run</button>
</body>
```

And here's how we run these lines of code when you click the button:

```
    var prog = document.getElementById("yourcode").value;
    var myPromise = Sk.misceval.asyncToPromise(function() {
        return Sk.importMainWithBody("<stdin>", false, prog, true);
    });
```

You get that `Sk` object in the global scope when you import Skulpt:

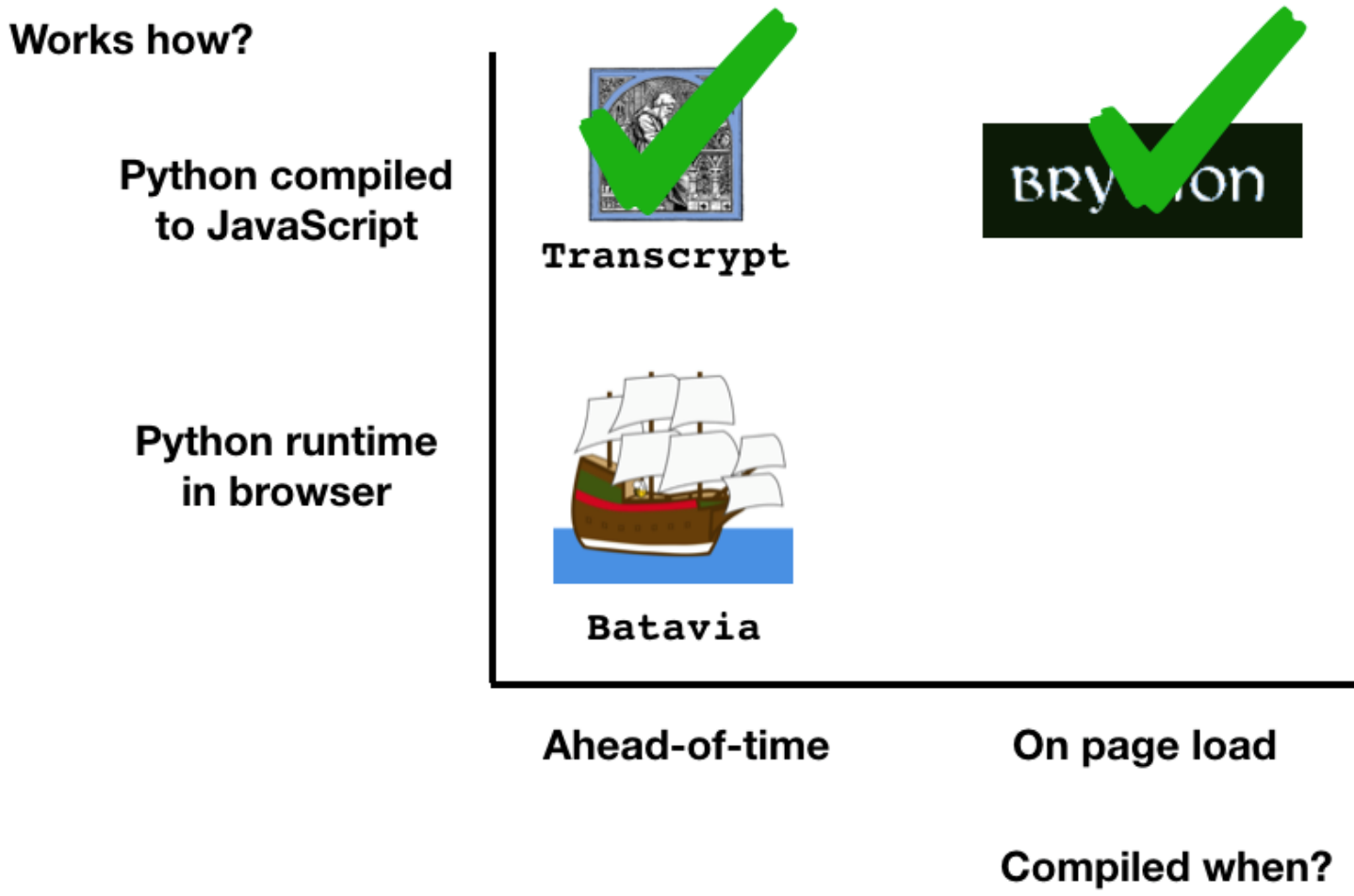
```
<head>
  <script src="http://www.skulpt.org/js/skulpt.min.js" type="text/javascript"
/>
  <script src="http://www.skulpt.org/js/skulpt-stdlib.js"
type="text/javascript" />
</head>
```

You can just pass a string of Python code to `Sk`, and it will compile it and run it as JavaScript, as many times as you like. I'm getting my Python code from the `yourcode` textarea in JavaScript - and then I pass that into Skulpt.

This is another design tradeoff. Skulpt doesn't have a built-in DOM API, so I have to write Javascript to interface with the page. But that can be a good thing: it allows us to build an environment to run Python without having to deal with the DOM. Our example above could be used in teaching: we wouldn't need to teach our students the DOM before we could do math and `print` statements.

At a more advanced level, Skulpt is what we use for Anvil, the platform for building full-stack web apps with nothing but Python. We *didn't want* our users to have to use the DOM, and HTML, and CSS – so instead, we built a component based UI library. In Anvil, you're not *querying a document* to find a textbox; you're instantiating a `TextBox` component and using its properties. We didn't want to expose our users to the DOM!

So we're halfway there. We've looked at all the technologies where Python is being compiled straight to JavaScript. Now I'm going to talk about a different approach: implementing a whole Python interpreter in the web browser.



Halfway there. The next three are full Python interpreters running in a web browser.

4. PyPy.js: a full Python interpreter in your web browser

Writing a Python interpreter and compiler in JavaScript sounds like a lot of work, but PyPy.js found a way around that. Instead of rewriting Python in JavaScript, which would take years, they recompiled PyPy to JavaScript using emscripten. That gives you a JavaScript version of PyPy. They didn't stop there – they also built a just-in-time Python-to-JavaScript compiler for compiling commonly-used code!

So it's quite a sophisticated, heavyweight tool… and the download size reflects that. The interpreter alone is 12 MB of JavaScript that your browser has to download and parse. (Even if the download is cached, your browser has to parse and compile all that JavaScript every time the page loads!) So the tradeoff for having this fully-featured Python interpreter in the browser is that you have to wait for it to fire up.

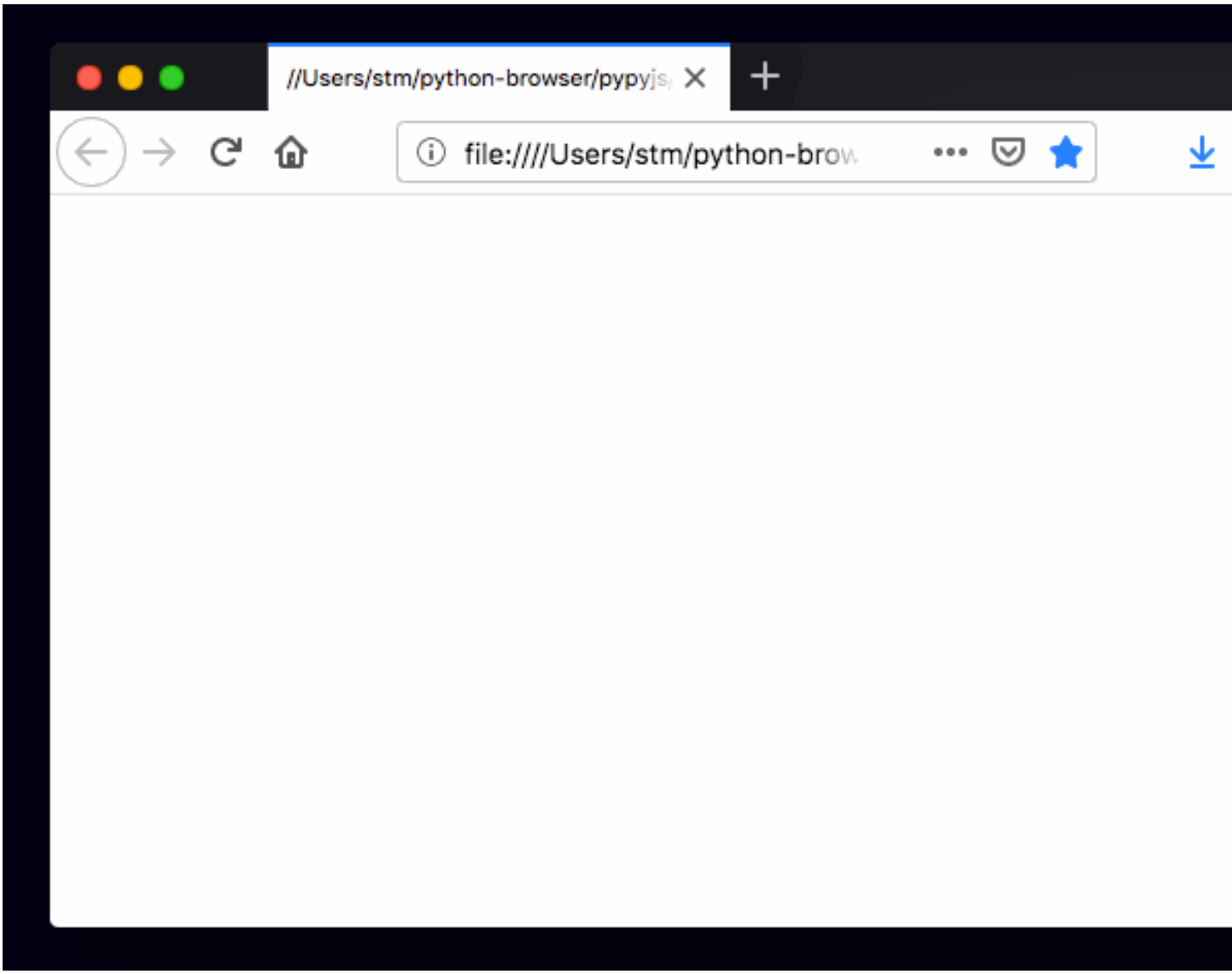
Let me show you an example and you'll see what I mean. You import the interpreter using script tags, as before. When those are imported, the interpreter is downloaded and loaded into the browser's memory.

```
<head>
  <script src="http://pypyjs.org/pypyjs-release/lib/Promise.min.js"></script>
  <script src="http://pypyjs.org/pypyjs-
release/lib/FunctionPromise.js"></script>
  <script src="http://pypyjs.org/pypyjs-release/lib/pypyjs.js"></script>
</head>
```

It starts running, and you get this `pypyjs` object which allows you to interface with that interpreter. I'm just calculating the first 10 square numbers using the `pypyjs.exec` method. It's got `get` and `set` methods where you can pass variables back-and-forth between JavaScript and Python. I'm getting the `y` value, which is an Array in JavaScript, and I'm printing that in an `alert`. It's got this nice promise-based way of doing things asynchronously:

```
<body>
<script type="text/javascript">
  pypyjs.exec(
    // Run some Python
    'y = [x**2. for x in range(10)]'
  ).then(function() {
    // Transfer the value of y from Python to JavaScript
    return pypyjs.get('y');
  }).then(function(result) {
    // Display an alert box with the value of y in it
    alert(result);
  });
</script>
</body>
```

Here's a GIF of it loading, in real time:



PyPy.js takes a while to fire up, but you get a full PyPy interpreter when it does.

PyPy is one of the best-tested Python implementations out there, and PyPy.js is pretty fast once it's loaded. If you're building something where your users are happy to wait for things to load up - perhaps an interactive REPL, or data visualisations - and you want a no-compromises Python interpreter, check it out. It even has an in-memory filesystem, so you can actually write to files that are stored in your browser's memory and you can read from them. You can also run multiple interpreters. So it's got loads of features, but it is very heavy!

So, what if we were willing to sacrifice some of these features to reduce the download size?

5. Batavia: A lightweight bytecode interpreter

It's called Batavia and it's from Beeware. They're an open-source project who make lots of really cool tools that make Python even more portable that it already is.

Batavia is their Python-in-the-browser solution and they *have* literally hand-written a Python interpreter in JavaScript. They haven't written the compiler – it just interprets precompiled Python bytecode. The result is a memory-efficient interpreter that's only 400 kB to download. The disadvantage is that you have to compile your script ahead of time and put bytecode in your HTML. Here we have a script tag whose type is `application/python-bytecode`:

```
<script id="batavia-helloworld" type="application/python-bytecode">
Â¬àÂ¬àÂ¬àÂ¬à7gwNCkIUE1cWAAAA4wAAAAAAAAAAAAAAAAIAAABAAAAAcw4AAABlAABk
AACDAQABZAEAUykCegtIÂ¬àÂ¬àÂ¬àÂ¬àZWxsbyBXb3JsZE4pAdoFcHJpbnSpAHICAAAA
cgIAAAD6PC92YXIvZm9sZGVycy85cC9uenY0MGxfÂ¬àÂ¬àÂ¬àÂ¬àOTc0ZGRocDFoZnJj
Y2JwdzgwMDAwZ24vVC90bXB4amMzZXJyddoIPG1vZHVsZT4BAAAAcwAAAA=
</script>
```

Their Getting Started guide actually suggests that you set up a server to do the compilation for you. So you could have an interactive REPL – but every time you pressed 'run', it would make an HTTP request to compile the new code, then send the bytecode back to run in your browser.

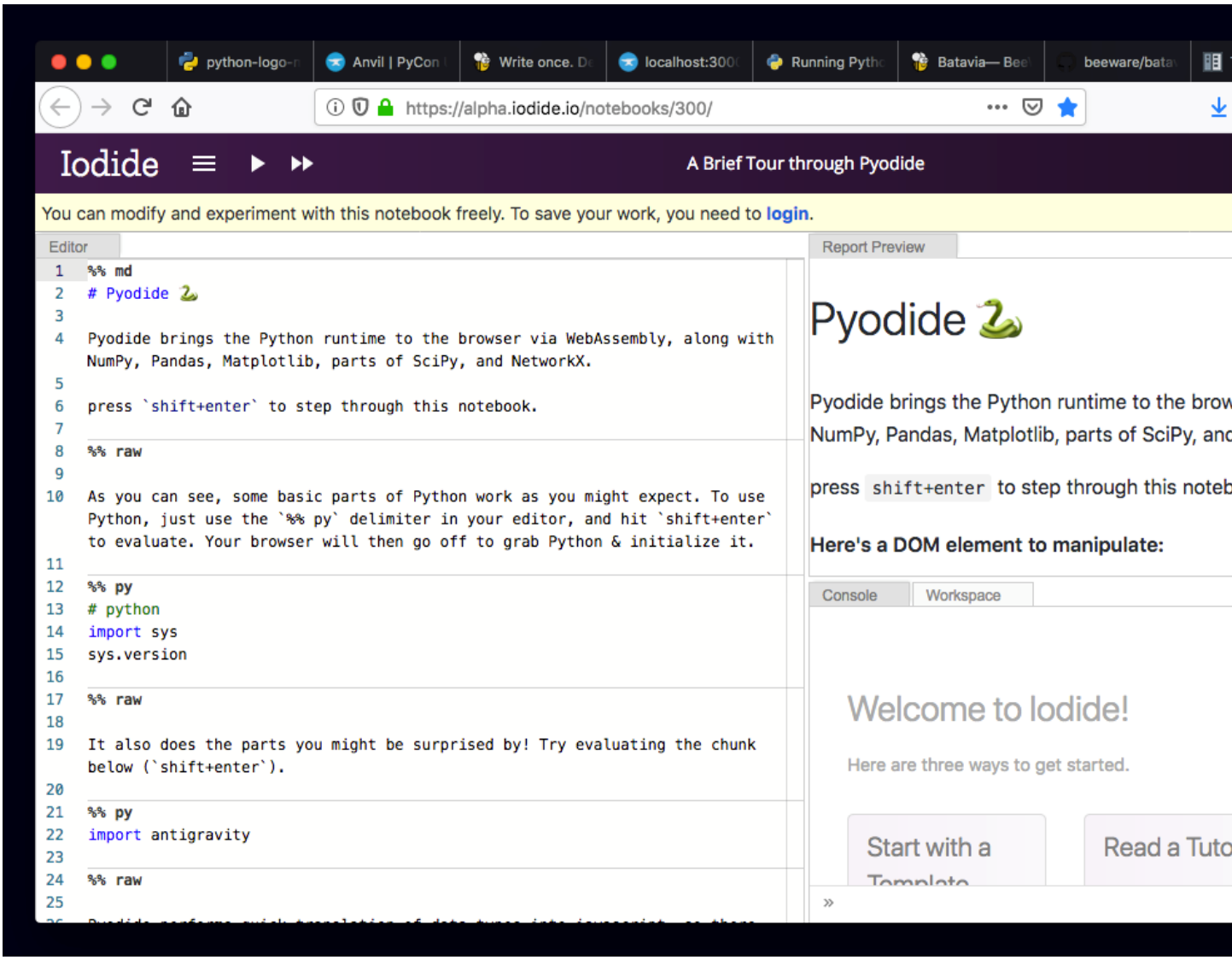
The other drawback is that, of the technologies I've talked about, this is probably the most immature. On their GitHub page they say "this is in early alpha and if it breaks you get to keep all the shiny pieces". So if this is something that fills a niche that you want filled, you'd do well to get involved and help them bring it to a more complete state.

6. Pyodide: Like a Jupyter Notebook in your web browser

Last but not least, I want to talk about Pyodide. It's a full environment, a bit like a Jupyter Notebook, but it runs entirely in your browser. You can have data science pipelines fully in the client, so there's no roundtrips and everything's a lot faster.

It's run by Mozilla, and it works by compiling the standard CPython interpreter to WebAssembly. WebAssembly is a bytecode that loads and runs in your browser (usually compiled to native code). Mozilla found an existing CPython-in-WebASM project, took charge of it, and recompiled things like Numpy, SciPy, and Pandas – all the data science libraries – into WebAssembly. The result is essentially the holy grail of data science.

Here's their example notebook. Pyodide is part of Iodide, a bigger project that has JavaScript and lots of other features. As with PyPy, the first time you run Python code you have to wait for the interpreter to download and initialise. But once you've done that, you have a full-strength Python environment.



Iodide: like Jupyter Notebooks in the browser.

Once it's loaded, subsequent Python steps execute pretty much instantly. You can also write HTML or MarkDown, run JavaScript, and do all sorts of things. Importantly it's all in the browser - there are no round-trips to a server, and you're using your users' compute so it scales brilliantly.

But there *is* this delay for the thing to start up, so it's not something you could use for an ecommerce site. And it's tied into the Iodide project, so you'd need to do quite a bit of hacking to use it with anything else.

Summary table

So this concludes our tour. Here's a summary table of all the technologies I talked about (also available in the accompanying Python Tips blog post):

System	When compiled	How Python is run	Extra features	Typical use case	Built-i manip
Transcrypt	Ahead-of-time	Transpiled to JS		Replacement for JS	Yes
Brython	On page load	Transpiled to JS		Replacement for JS	Yes
Skulpt	Just-in-time	Transpiled to JS		Python environment in browser	No
PyPy.js	Just-in-time	PyPy interpreter in JavaScript	Multiple interpreters, virtual filesystem, calling JavaScript from Python	Python environment in browser	Yes
Batavia	Ahead-of-time	Custom bytecode interpreter in JavaScript		Python environment in browser	Yes
Pyodide	Just-in-time	CPython interpreter in WebAssembly	NumPy, SciPy, Pandas, Matplotlib	Data science pipelines in browser	Yes

Make everything Python!

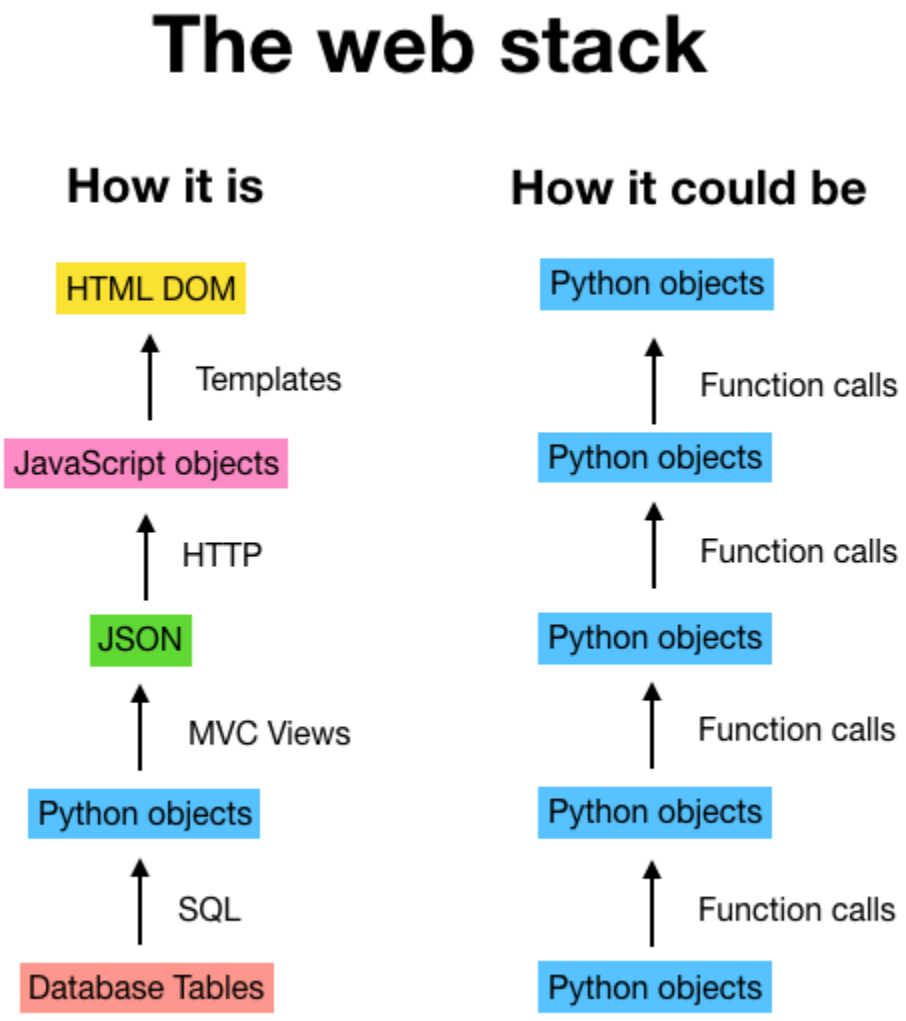
I want to make a broader point. As I said at the beginning:

> "Why do I **have to** use JavaScript? Why am I constrained to this one language?".

- me, earlier.

I think we need to be asking that question about the whole web stack.

The web stack started out as the thing on the left of this diagram, where you're using SQL for your data access, you have to convert things to JSON, you have to write HTML templates, you have to use the Document Object Model to access your UI.



We're beginning to move to something where we can use Python everywhere. We have ORMs like SQLAlchemy to write Python to access our databases. Obviously we have Python on the server. We now have Python in the browser. And to top it all off, with Anvil we now have access to a more Pythonic way of manipulating the UI.

We can access our data in one way on the server and have exactly the same representation, the same syntax in the browser. The average web app is, say, 1000 lines of code to translate data, and 40 lines of code that actually specifies the business logic. If there were no translation necessary, web development could be **way** simpler.

This is why we built Anvil. With Anvil, you can do *everything* in Python – from UI design to the database – without translating into five other representations. It’s free to use, and it hugely simplifies the web stack