Title

So you're writing a website, right? And you're just thinking to yourself "Wow, javascript really sucks, I wish I could be using anything else right now." Well guess what, you can, and it's called webassembly, and I'm going to teach you all about it in tonights session.

What am I talking about?

So what will I be talking about?

- What is webassembly?
- Rust
  - Mandelbrot set
- Raw
  - for crazy people or you want to learn fundamentals
  - AoC day 4

What am I not talking about?

First, what I won't be talking about are...

- Batteries included frameworks
  - Compiles your code into webassembly
  - Abstract away webassembly into a technical detail
- I'm teaching webassembly, not a framework
- Super useful if you want to make a whole website in webassembly

How do websites work?

So first, how do websites work?

1. Put an address into your browser
2. Browser makes HTTP request
3. Returns an HTML file
   - Content of the webpage
   - Example on the right
   - Instructs the browser to download additional files
- CSS
  - How your website looks
- Javascript
  - Programming language
  - What the website should do
  - Ex: Making scrolling do something other than actually scrolling

What problems does Webassembly solve?

So where does webassembly fit into this? Before that I need to explain what problems webassembly attempts to solve.

- Javascript is cursed
  - You can compare strings and numbers and it might return true
  - Added a triple equals operator when people figured out that made no sense
  - No type checking
- Scripting language; Slow as heck
  - No rendering fractals

- Running non-web stuff on the web
  - FFMPEG
  - Your browser doesn't understand assembly

So for these reasons, developers invented Webassembly.

What is Webassembly?

So now, what is webassembly?

- Standardized machine code
  - Compilation target for other languages
  - You're not meant to code in it directly, but you totally can and we will
  - You can write code for the web in any language you like
  - Compiling desktop programs for the web
- Can only interface with javascript
  - Importing and exporting functions from/to javascript
- Can't send or receive complex data structures, only numbers
  - You need javascript to encode and decoded data directly in your webassembly module's memory
  - "Glue code"
    - We'll use Rust for our first demo and we'll see that it takes care of this for us.
- No direct access to DOM or standard library
  - You need to import from Javascript
  - Also taken care of by Rust

Demo 1!

So now we can start with the first demo!

- Mandelbrot set
  - I'm assuming that if you wanted to follow along you set everything up in accordance with the README on the github page.

First, we need to set up the HTML for our page.

HTML

index.html

```
<!doctype html>

<html>
  <head>
    <title>Mandelbrot</title>
  </head>

  <body>
    <canvas id="canvas" width="800" height="600"></canvas>
    <script type="module" src="sketch.js"></script>
  </body>
</html>
```

JS

Now lets write our Javascript code

I'm implementing this in a roundabout way

- JS asks WASM to render a frame
- WASM renders frame
- WASM asks JS to display the rendered frame
- JS asks WASM through a callback to copy the frame data into the canvas buffer
- More straightforward to have WASM return the rendered frame and have JS copy it into the buffer but I want to demo the cool JS interop

sketch.js

```
import init, { render_data } from "./pkg/mandelbrot.js";

(async () => {
  await init()

  showThingy(1);
})()

—

function showThingy(scale) {
  render_data(scale)

  requestAnimationFrame(() => showThingy(scale * 0.99))
}

—

export function set_canvas_data(callback) {
  let canvas = document.getElementById("canvas");
  let ctx = canvas.getContext("2d");

  let imageData = ctx.getImageData(0, 0, 800, 600);

  callback(imageData.data)

  ctx.putImageData(imageData, 0, 0);
}
```

Rust

Now we can write our Rust code that will actually generate the mandelbrot set.

Cargo.toml

```
[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2.74"
js-sys = "0.3.68"

—
```

lib.rs

```
fn mandelbrot_shade_at_pos(c: (f64, f64), count: usize) -> f64 {
    let mut z = (0., 0.);

    for i in 0..count {
        if z.0 * z.0 + z.1 * z.1 > 4. {
            return i as f64 / count as f64;
```

```
        }

        z = (z.0 * z.0 - z.1 * z.1 + c.0, z.0 * z.1 * 2. + c.1);
    }

    return 1.;
}
```

—

```
#[wasm_bindgen]
pub fn render_data(scale: f64) {
    let mut data = vec![0; 800 * 600 * 4];

    let count = (10. * (1. / scale).ln()) as usize + 30;

    for x in 0..800 {
        for y in 0..600 {
            let shade = mandelbrot_shade_at_pos(
                (
                    (x as f64 - 400.) / 200. * scale - 1.5,
                    (y as f64 - 300.) / 200. * scale,
                ),
                count,
            );

            data[4 * (x + y * 800) + 0] = (shade * 256.) as u8;
            data[4 * (x + y * 800) + 1] = (shade * 256.) as u8;
            data[4 * (x + y * 800) + 2] = (shade * 256.) as u8;
            data[4 * (x + y * 800) + 3] = 255;
        }
    }

    set_canvas_data(&Closure::once(move |array: Uint8ClampedArray| {
        array.copy_from(&data)
    }));
}
```

—

```
#[wasm_bindgen(raw_module = "/sketch.js")]
extern "C" {
    fn set_canvas_data(s: &Closure<dyn FnMut(Uint8ClampedArray) -> ()>);
}
```

Compiling and running

Now we can actually run it!

```
wasm-pack build --target web
python3 -m http.server
```

WAT

Now observe this meme, it's very important...

Now I know what you're thinking...

WAT?

And you'd be right!