I am

Jakub Trąd

Software Developer at Anixe

working primarily with Rust

WebAssembly

A practical introduction

The plan

1. Using **wasm-bindgen** for fun and profit
2. Digging into WASM
    a. How much work is **wasm-bindgen** and **wasm-pack** doing for us
    b. WASM syntax, format and structure
3. Python and WASM interop
    a. Writing our own bindings
    b. WASM as a portable, safe binary format
    c. briefly Wasmtime, WASI, etc.

If you want to follow along

Repo:
https://github.com/Dzejkop/rust-wroclaw-wasm-talk-01-2020

https://webassembly.studio/

https://rustwasm.github.io/wasm-pack/

+

https://crates.io/crates/wasm-bindgen

“I’m not an expert, I’m just a dude.”
~ Scott Shurr, CppCon 2015

Using **wasm-bindgen** for fun and profit

Conway’s Game Of Life

<code time>

So what is actually going on?

#[wasm_bindgen]

wasm-pack

```
steps.extend(steps![
    step_build_wasm,
    step_create_dir,
    step_copy_readme,
    step_copy_license,
    step_install_wasm_bindgen,
    step_run_wasm_bindgen,
    step_run_wasm_opt,
    step_create_json,
]);
```

```
cargo build --release --target wasm32-unknown-unknown
```

Produces the .wasm file

`wasm-bindgen ...`

Generates JavaScript bindings. And optionally TypeScript type definitions.

`wasm-opt...`

Optimizes the resulting WASM by removing dead code, etc.
(part of the Binaryen toolchain)

Let’s look at some WASM!

If you want to follow along

https://wasmtime.dev/

```
> wasmtime wasm/wasm_fib.wat --invoke fib 4
```

<wasm time>

Can we have it running in Python?

<python time>

~fin~