$whoami

Farhaan Bukhsh

- Senior Software Engineer at OpenCraft
- Open edX Core Committer
- Open edX Release Manager

EUROPYTHON 2026 · KRAKÓW

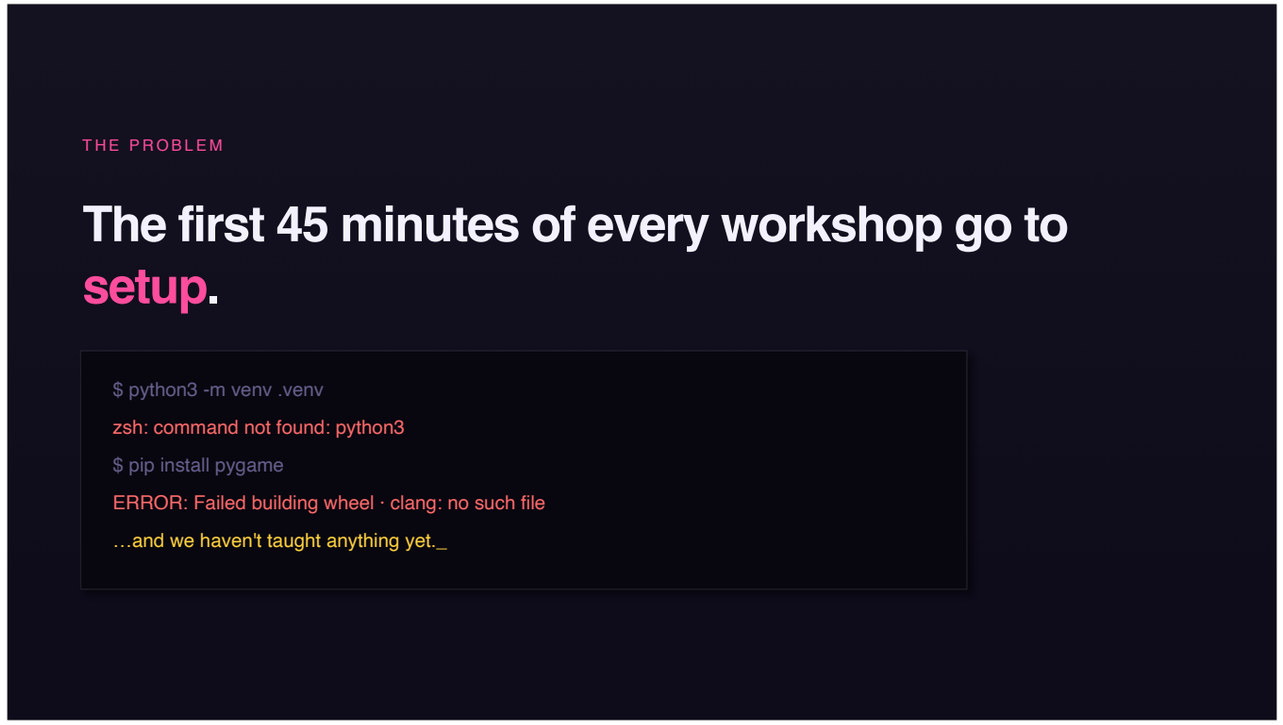
Python games in the browser

Teaching with WebAssembly — zero-install, safe, and built for how people actually learn.

Farhaan Bukhsh · open-source · Open edX core committer

PRESS START

Welcome. This is a talk about using WebAssembly to teach Python — by running games entirely in the browser.
- Games here do not mean the high-definition console games but the games that can be used for teaching



- [ACTION] Can I get a show of hands of how many people have tried teaching Python in schools or colleges?
- Every workshop starts the same way. Before a single line of Python gets written, we lose the room to setup: virtualenvs, versions, path issues, corporate laptops. The teaching hasn't even begun.
- You have faced these issues and so have I and this is a big turn-off for learners and for instructors as well.
- At the end we either run out of time or run out of patience. Or if everything in the universe aligns, we get it to work.

Teaching Python has an environment problem

01

Install hell

Versions, virtualenvs, compilers — different on every OS and locked-down laptop.

02

“Works on my machine”

The learner's environment never quite matches the tutorial's. Debugging setup, not code.

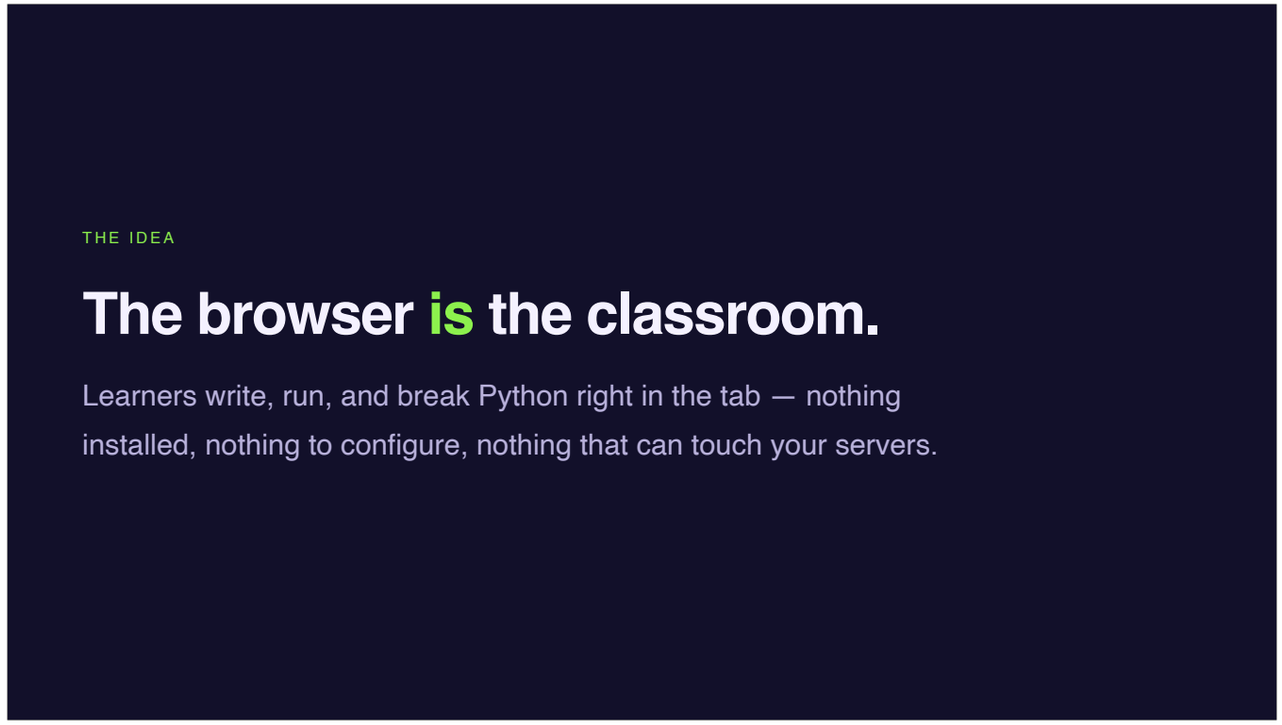
03

Running untrusted code

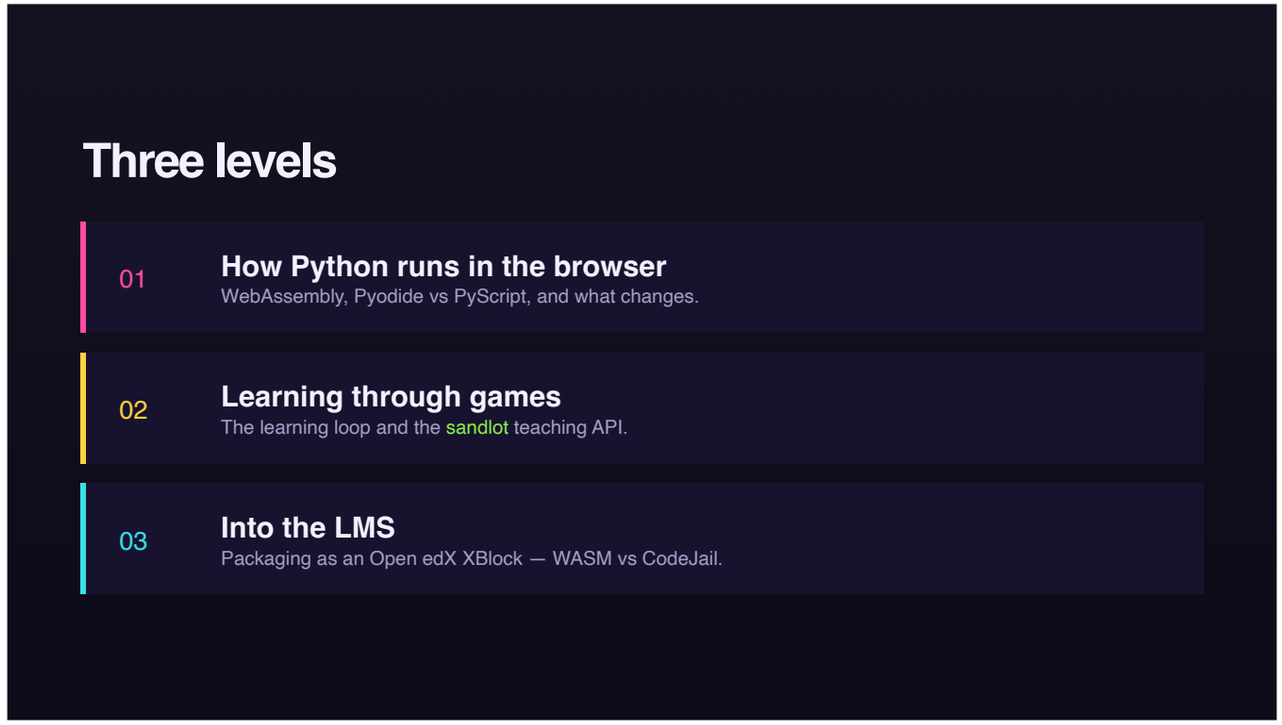
On shared or school machines, arbitrary learner code is a security headache.

Three recurring costs.
- Install hell across OSes. - (Should I use Anaconda, do I install mise, which venv should I create)
- (Problem that has led to so many wonderful solutions) The classic works-on-my-machine gap.
- And on shared or school machines, you can't safely let learners run arbitrary code. — (Permissions from authorities)

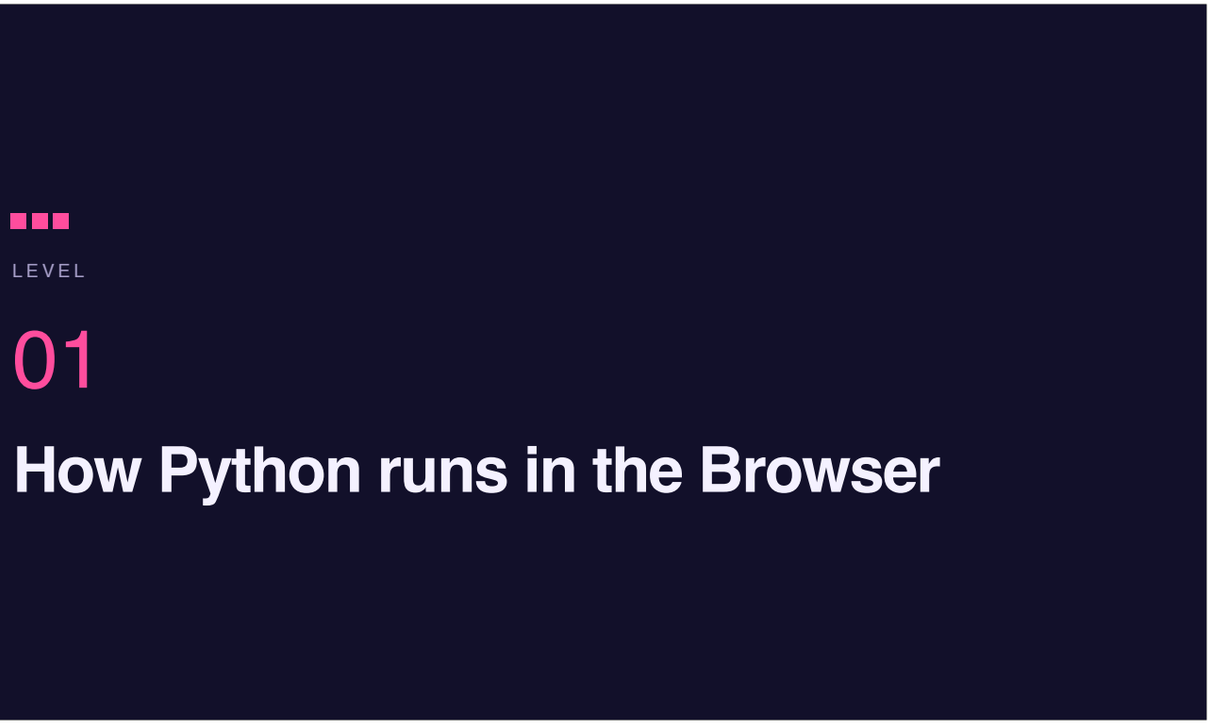
- These aren't edge cases — they're the default.



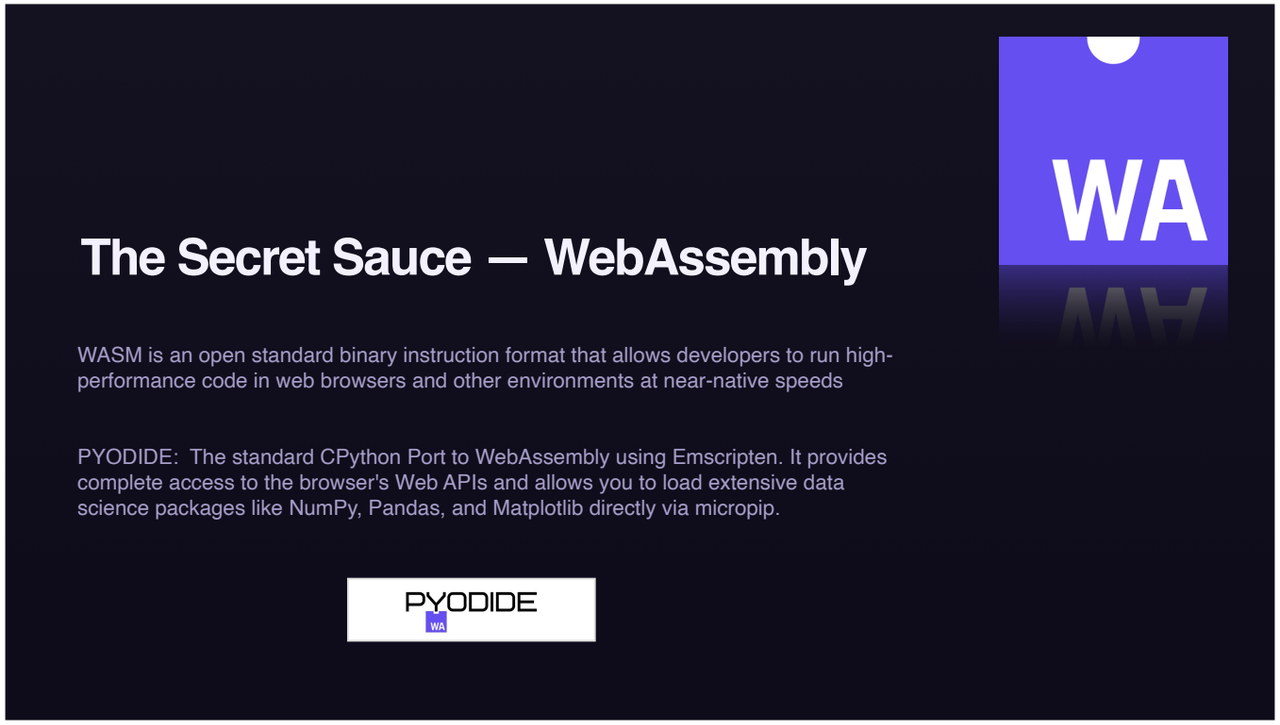
So here's the reframe. Don't treat the browser as somewhere you ship the finished thing. Treat it as the room where learning happens — where they write, run, and break Python with nothing installed.



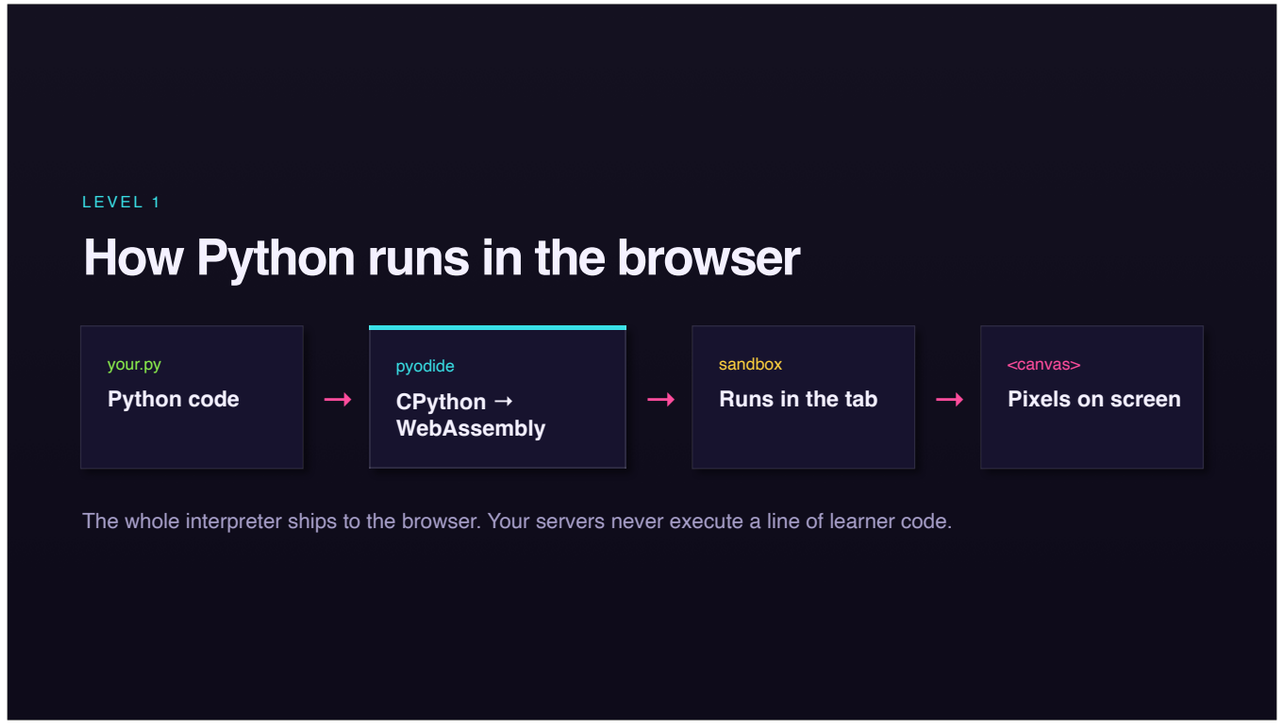
Three levels.
- First, how Python even runs in a browser.
- Second, why we wrap it in games and how the sandlot library works.
- Third, how this drops into a real LMS as an Open edX XBlock.



Level one. The foundation: how do you get real Python running inside a browser tab at all?



The secret sauce to this dish is WebAssembly.
- CPython is compiled to WASM — that's Pyodide.
- Wasm is a low-level binary instruction format designed to execute code at near-native speeds across different platforms, most notably inside modern web browsers
- It loads in the tab, runs your Python inside the browser's sandbox, and talks to the DOM and canvas through a JS bridge. No server executes anything.



- It loads in the tab, runs your Python inside the browser's sandbox, and talks to the DOM and canvas through a JS bridge. No server executes anything.

Different Ways To Run Python on Browser

Pyodide

- CPython compiled to WASM — the runtime itself
- Low-level Python ↔ JS bridge
- You own loading, packages, workers
- Full control over isolation

best for → embedding a custom runtime

PyScript

- Framework built on top of Pyodide
- HTML tags: <script type="py">, config
- Batteries included, less to wire up
- More magic, heavier, less control

best for → quick starts & prototypes

For a classroom, we chose Pyodide + a custom worker host — control and isolation matter.

Two ways to do this. Pyodide is the runtime — CPython in WASM with a low-level JS bridge;

CPython is compiled to WASM — that's Pyodide.

You own the wiring.

PyScript sits on top, gives you HTML tags and batteries included, but more magic and less control. For a classroom runtime we picked Pyodide plus a custom worker host.

I also worked on a similar implementation called — waspy

We are working on another way to run python on browser.

Running client-side flips three assumptions. Isolation:
- the tab is the sandbox, no shared server to harden. Visual representation of a sandbox
- Infrastructure: nothing to scale per learner — static hosting is enough.
- Security: untrusted code never reaches your machines.

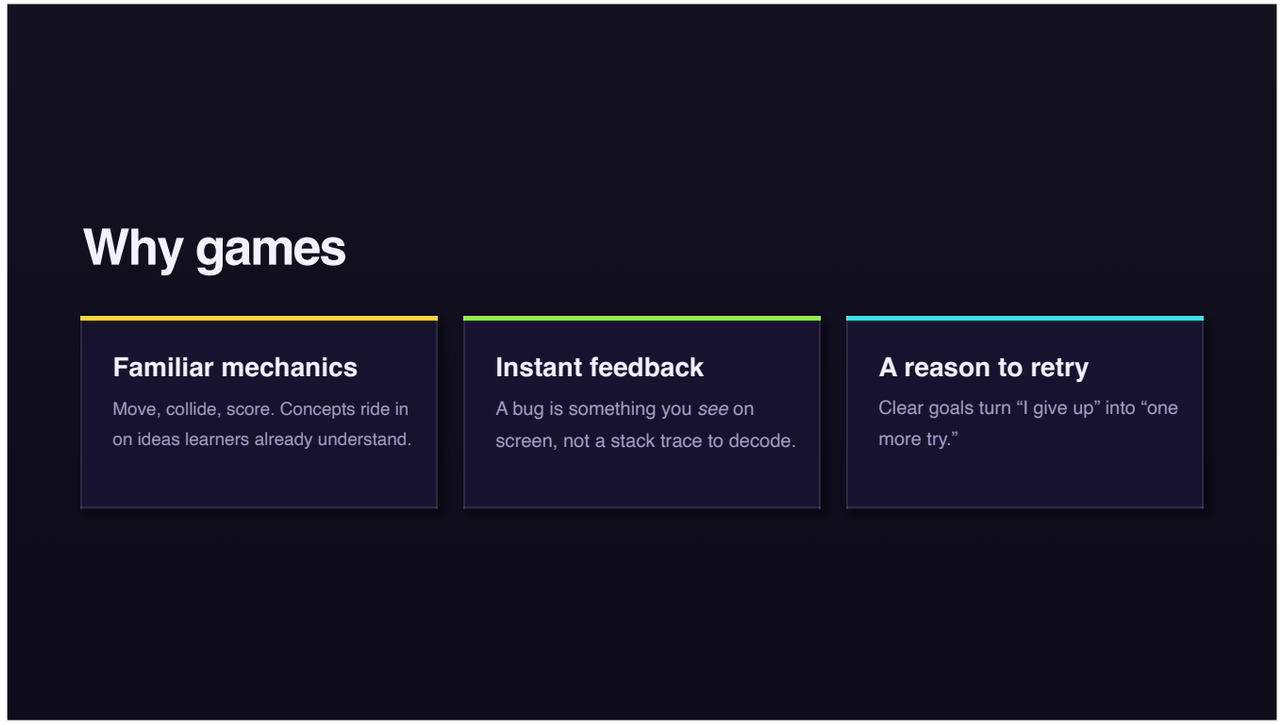
Browser tabs makes the best learning platform

LEVEL

02

Learning through games

Level two. Running Python in the browser is the enabler. The teaching idea is to wrap it in a game.



Why games?

The mechanics are already familiar — move, jump, collide, score.

Feedback is instant and visual, so a bug is something you see, not a stack trace.

And there's an actual goal, which keeps people trying one more time.

THE TOOL

sandlot

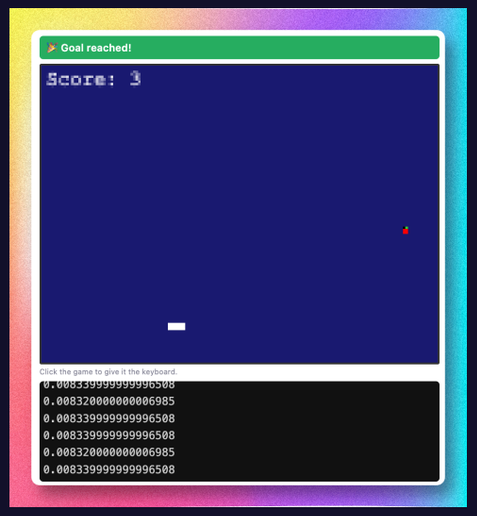
(Sandbox + axolot)

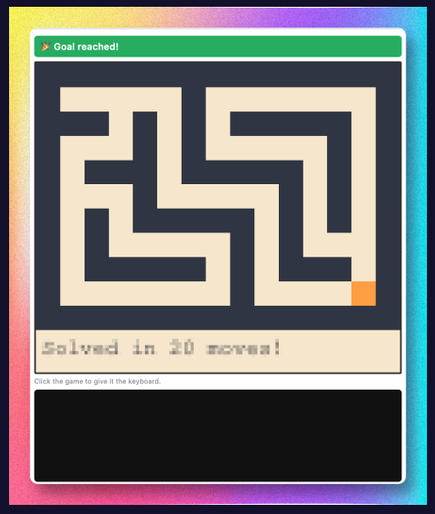
A small, dependency-free teaching library. Runs in Pyodide, tests in plain CPython. The curated API *is* the curriculum — what it exposes is what you teach.

The API they learn against is sandlot — a small, dependency-free Python library. It runs inside Pyodide, but it's plain CPython too, so it's testable and portable. The library itself is the curriculum: what it exposes is what you teach.

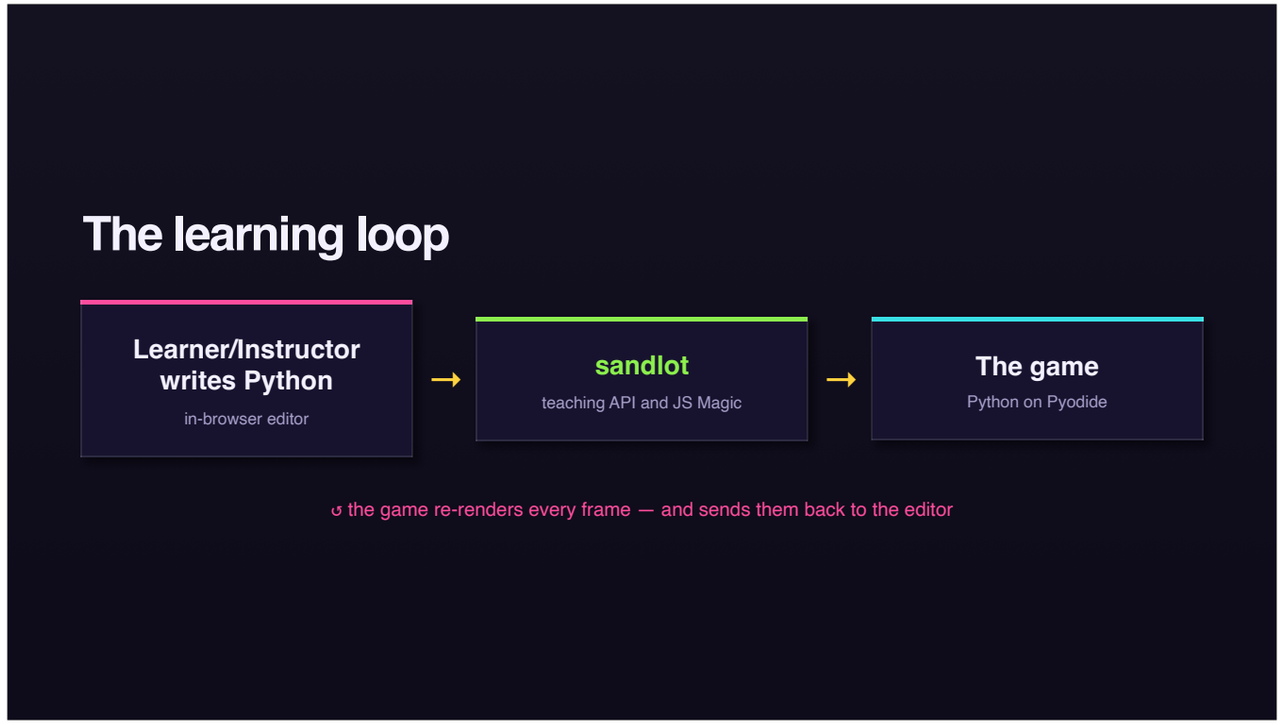
Games Built with Sandlot

▶ PLAY_





How the game looks like, look at the basket and fruite. You can look at maze — BTW all them are generated using ASCII.



Here's the loop that makes it work. The learner writes Python in an in-browser editor. That code calls the sandlot API, which drives a game that is itself written in Python on Pyodide. The code written in the browser is being called through a JavaScript bridge. The JavaScript renderer then renders the frame of the game.

The game re-renders every frame — and what they see sends them back to the editor. Both sides are Python.

Scenery · Sprites · Goal

Scenery (Scaffolding)

The world: the stage, boundaries, and goals learners build inside.

Sprites (Learner Code)

The actors: things that move, collide, and respond to their code.

Goals

Base case where the Learner Code helps to win the game.

The API is built from three primitives that map to how you'd describe a game to a kid. Scenery is the world. Sprites are the things that move. Swatches are the palettes and looks. Small vocabulary, lots of exercises.

A sandlot exercise

exercise_01.py

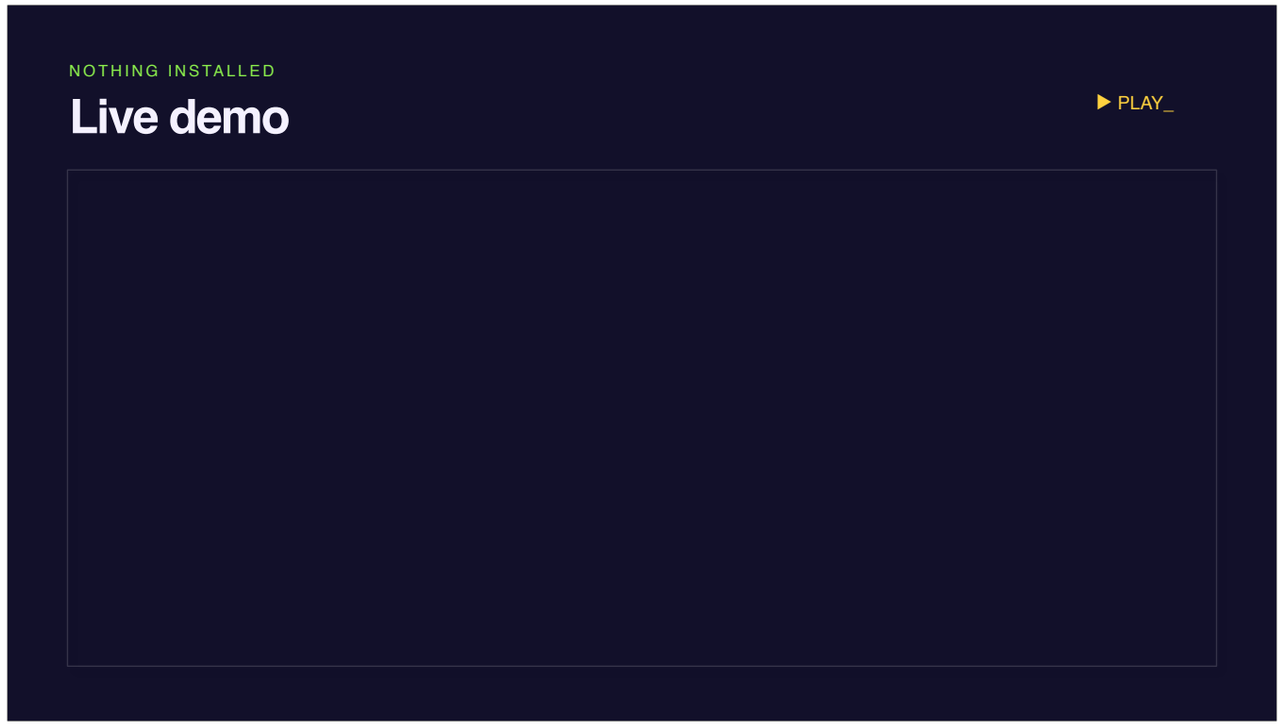
```
from sandlot import  Scene, Sprite

scene = Scene(swatch="sunset" )
cat   =  Sprite("cat", at=(0, 0))

@scene.on_step
def move ():
    cat.right(1 )

if cat.x == 8 :
        scene.win("You reached the flag!")
```

Here's what a learner actually types. Make a scene, add a sprite, and on every step nudge it right until it reaches the flag. Real Python — functions, decorators, conditionals, attributes — but the payoff is a moving cat, not a printed number.



Let's leave the slides. I'll open the editor, write a couple of lines live, and you'll watch the game respond in the same tab — with nothing installed. [Switch to the live tool. Drop a screenshot/GIF into this slide as a fallback.]

Level three. A cool demo is one thing; shipping it to thousands of learners inside a real course platform is another. That's where Open edX comes in.

OPEN EDX

Packaged as an XBlock

Courses are built from XBlocks — modular units. xblock-sandlot drops the whole runtime into a course like any other block.

Authoring
Studio fields to write the exercise, starter code, and goal.

Progress
Persistence and completion, tracked like any graded unit.

Runtime
JS worker host + snapshot renderer bundled in the block.

In Open edX, courses are built from XBlocks — modular, reusable units. We package the whole thing as xblock-sandlot: Studio fields to author an exercise, persistence and completion tracking, and the JS worker host plus snapshot renderer bundled in. Authors drop it into a course like any other block.

WASM vs CodeJail

CodeJail

server-side execution

- Runs Python on *your* servers (AppArmor)
- Hardened hosts, scaled per learner
- A network round-trip on every run
- Ops burden + real attack surface

WASM / Pyodide

in-browser execution

- Runs in the *learner's* browser tab
- Sandboxed by the browser itself
- Instant feedback, no round-trip
- Zero server-side execution to scale

Compare that to how Open edX has traditionally run untrusted code: CodeJail, an AppArmor sandbox on your servers. It works, but you own hardened hosts, you scale them per learner, and every run is a network round-trip. Moving execution into the browser deletes that whole tier.

What you get

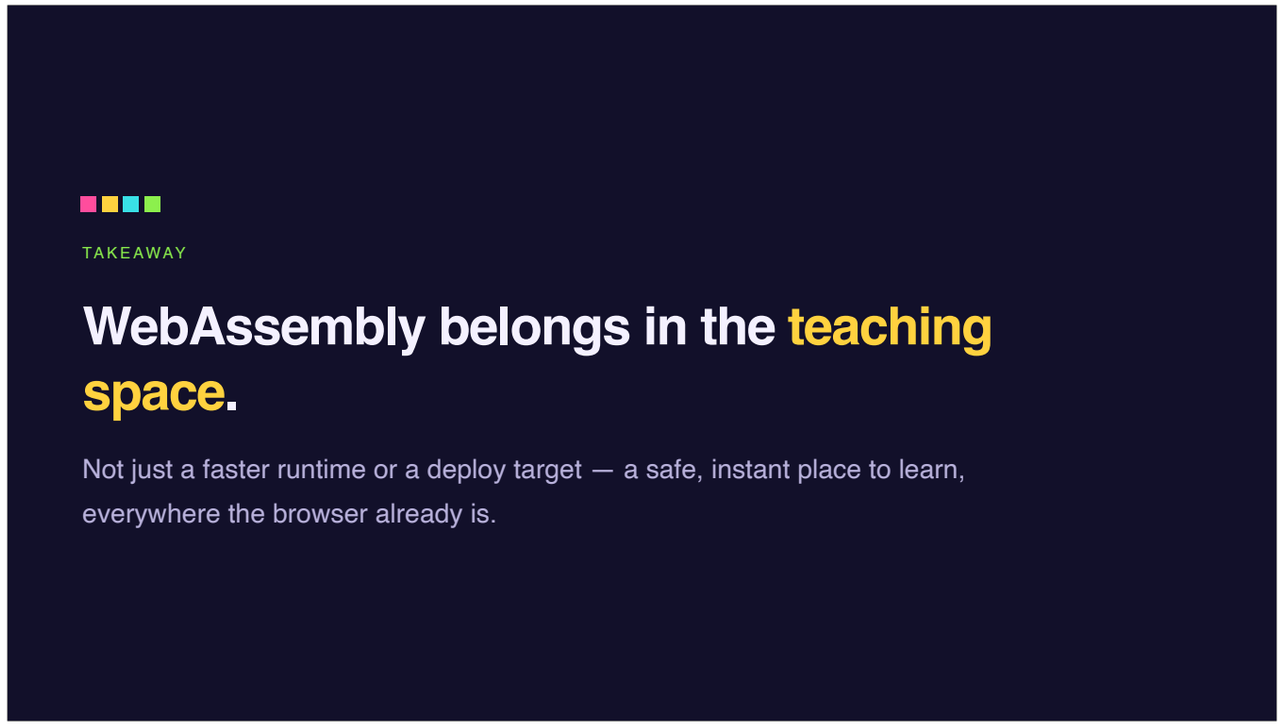
A **Zero-install**
Open a link, start coding.

B **Safe by default**
Code never leaves the device.

C **Less infrastructure**
Static hosting, no exec servers.

D **Better learner UX**
Instant, visual, responsive.

Net effect, four wins. Zero-install for the learner. Safe by default because code never leaves their device. Far less infrastructure to run. And a tighter, more responsive experience. That's the case for WASM in education.



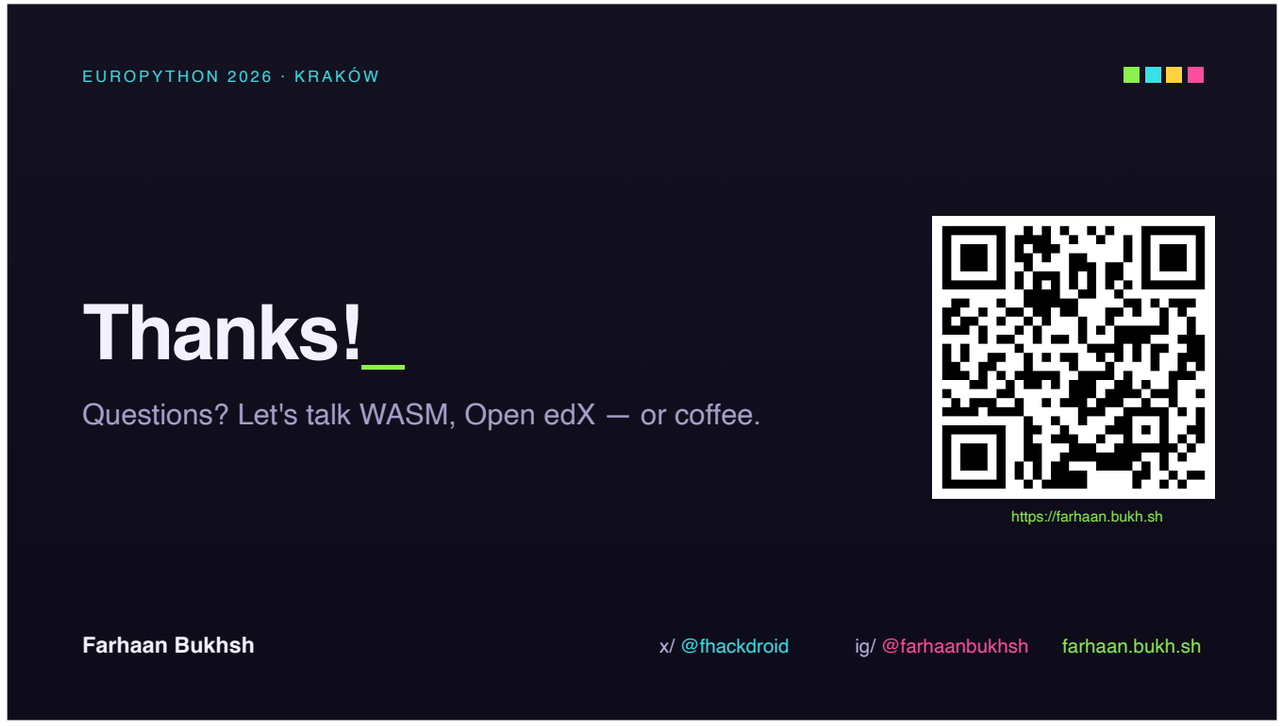
If you take one thing away: WebAssembly isn't just a performance trick or a deployment target. It's a genuinely new place to teach — safe, instant, and everywhere the browser already is.

Try it

playsandlot	the project — editor, runtime, harness
sandlot	the dependency-free teaching library
xblock-sandlot	the Open edX integration

→ github.com/farhaanbukhsh/playsandlot

Everything's open. playsandlot is the project; sandlot is the teaching library; xblock-sandlot is the Open edX integration. Fork it, write an exercise, tell me what breaks.



Thank you. I'm Farhaan Bukhsh — find me on the socials below, and I'm around all conference. Happy to talk WASM, Open edX, or pour-over coffee. Questions?