Python <3 WebAssembly - From Browser to Serverless

Shivay Lamba [1] Gaurav Pandey [2]

[1]shivaylamba@gmail.com [2]hi@pandeygaurav.com

What is WebAssembly?

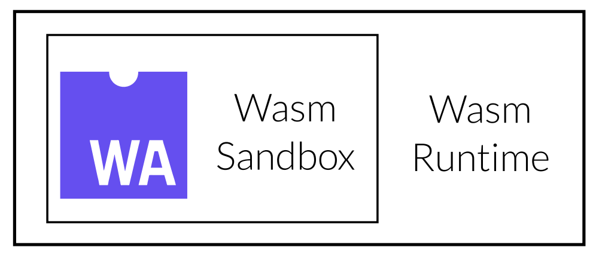
WebAssembly, or WASM, is a binary code format close to assembly and independent of the language and the platform since WebAssembly can be compiled from other languages and can be executed on a browser (Web APIs) or a virtual machine. WebAssembly is an open standard whose main objective is to offer a closely native performance on the web while maintaining compatibility with the current ecosystems and standards.

WASM lets developers bring the performance of languages like C, C++, and Python to the web development area. Webassembly, commonly used to perform demanding operations in the browser, is now being expanded for cloud and serverless workloads.

Features of Webassembly

Security

Each Wasm module executes within a sandboxed environment separated from the host runtime. The model executes independently, and can't escape the sandbox without going through appropriate APIs.



Full Portability

WA
(no host specific calls)
Wasm Runtime
Host OS

A Wasm module is just a binary format with no host-specific OS calls and it runs with limited, local, nondeterminism which sometimes might lead to unexpected behavior. Runtimes invent their own APIs.

Small Artifacts

A Wasm module is also size-efficient when contrasted with containers since each module only has the runtime code and shares the runtime with other Wasm modules. The execution environments are still sandboxed from each other and can depend on different backends.

WA
(only runtime code w/o defs)
artifacts
Wasm Runtime

Python support for WebAssembly

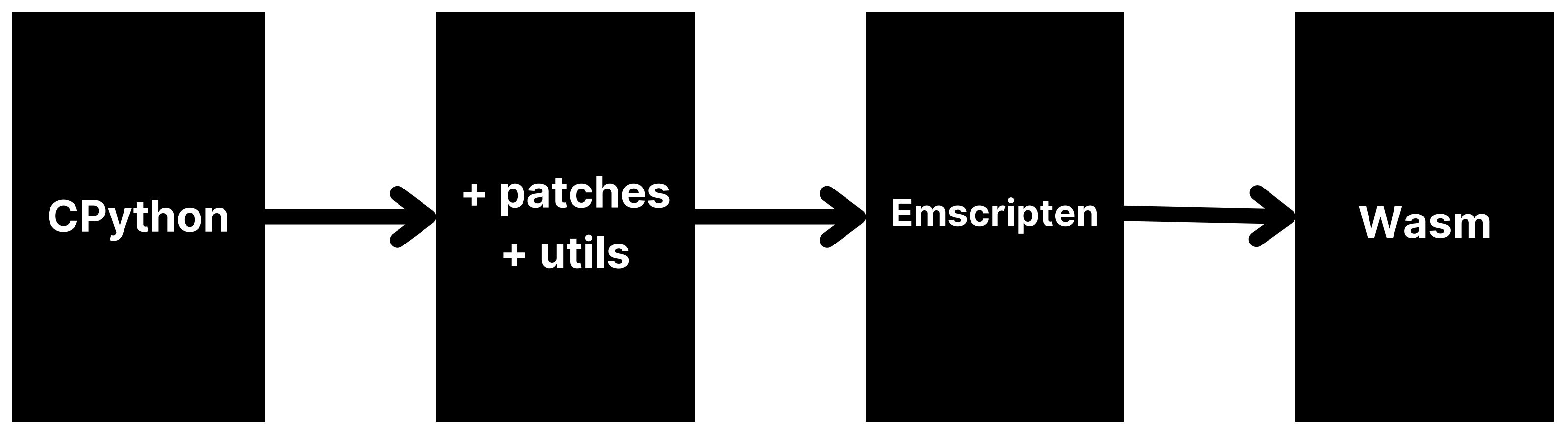
Compiling CPython to WASM : The process of compiling Python to WebAssembly (Wasm) using Emscripten involves several key steps.

Emscripten is a powerful toolchain designed to compile C and C++ code into WebAssembly, and while it doesn't directly support Python, it can compile the CPython interpreter, enabling Python code to run in web browsers and other Wasm environments.

To begin, you need to install the Emscripten SDK, which provides essential tools like emcc, a C/C++ compiler that outputs Wasm. Next, you prepare your Python code, ensuring it is compatible with the WebAssembly environment by avoiding unsupported features.

Then, you use emcc to compile the CPython source code along with your Python files into a Wasm module. Optimization flags can be applied during this step to enhance performance or reduce file size. After compiling, you write JavaScript to load and interact with the Wasm module containing the Python interpreter.

Finally, you test and deploy the resulting WebAssembly application.

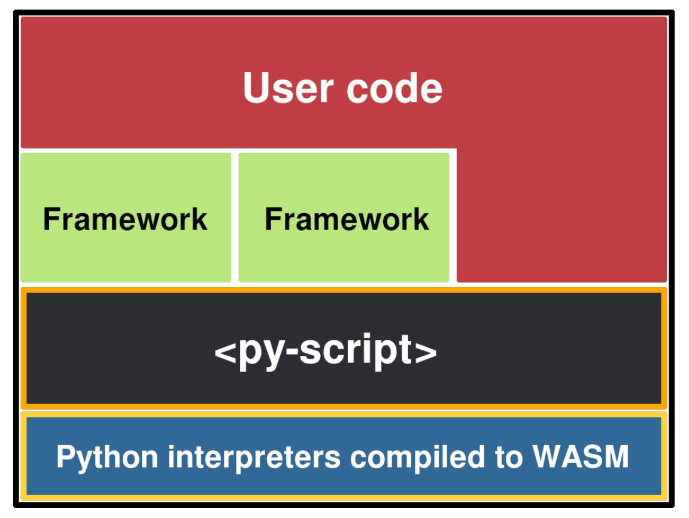


Python in the browser

PyScript - What is PyScript? and Architecture

PyScript is a platform for running Python in modern web browsers. It's architecture has three core concepts:

- A small, efficient and powerful kernel called PolyScript is the foundation upon which PyScript and plugins are built.
- A library called coincident that simplifies and coordinates interactions with web workers.
- The PyScript stack inside the browser is simple and clearly defined.



```html
<html>
    <head>
        <link rel="stylesheet" href="https://pyscript.net/releases/2024.1.1/core.css">
        <script type="module" src="https://pyscript.net/releases/2024.1.1/core.js"></script>
        <title>PyScript Hello World</title>
    </head>
    <body>
        <section class="pyscript">
            This is the current date and time, as computed by Python:
            <script type="py">
                from pyscript import display
                from datetime import datetime
                now = datetime.now()
                display(now.strftime("%m/%d/%Y, %H:%M:%S"))
            </script>
        </section>
    </body>
</html>
```

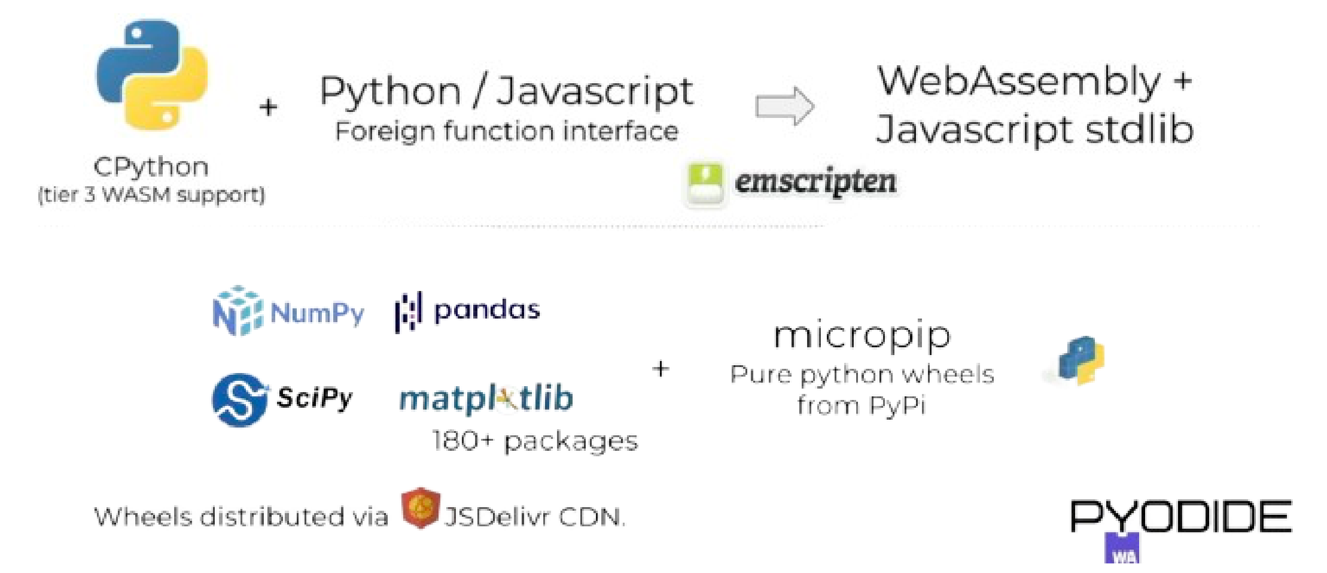
Pyodide - What is Pyodide? and Architecture

Pyodide is a Python distribution for the browser and Node.js based on WebAssembly/Emscripten.

It makes it possible to install and run Python packages in the browser with micropip. Any pure Python package with a wheel available on PyPI is supported. Many packages with C extensions have also been ported for use with Pyodide. These include many general-purpose packages such as regex, PyYAML, lxml and scientific Python packages including NumPy, pandas, SciPy, Matplotlib, and scikit-learn.

Pyodide comes with a robust Javascript ⍰ Python foreign function interface so that you can freely mix these two languages in your code with minimal friction. This includes full support for error handling (throw an error in one language, catch it in the other), async/await, and much more.

When used inside a browser, Python has full access to the Web APIs.



Python in the serverless and cloud

WebAssembly in Serverless with Spin

Spin is an open Source Developer Tool for building WebAssembly based serverless applications in just a few commands.

The Spin Python SDK is an experimental toolkit designed to help developers build serverless applications using Python, leveraging WebAssembly (Wasm) for deployment. This SDK is part of Fermyon's Spin framework,

- The older version was created using Py2Wasm: A Python to WebAssembly compiler.
- The newer version is implemented with Compontentize.py: a tool to convert a Python application to a WebAssembly component

```python
# Import necessary modules from the Spin SDK
from spin_sdk import http
from spin_sdk.http import Request, Response
from spin_sdk import llm


# Define a handler class for incoming HTTP requests
class IncomingHandler(http.IncomingHandler):
    # Define the method to handle incoming requests
    def handle_request(self, request: Request) -> Response:
        # Use the LLM to generate code for array implementation in multiple languages
        res = llm.infer_with_options(
            "llama2-chat",  # Specify the LLM model to use
            "write code to implement an Array in Java, C++, JavaScript and Rust",  # prompt
            llm.InferencingParams(temperature=0.5, max_tokens=1024)  # inference parameters
        )

        # Return the generated code as an HTTP response
        return Response(
            200,  # HTTP status code
            {"content-type": "text/plain"},  # Response headers
            bytes(res.text, "utf-8")  # Response body
        )
```

Pyodide Support in Cloudflare Workers

Cloudflare Workers leverage Pyodide by integrating it directly into the Workers runtime, allowing developers to write serverless functions in Python. This integration utilizes Pyodide's WebAssembly-compiled Python interpreter and its foreign function interface to provide access to JavaScript APIs and dynamic linking of Python packages.

```python
from js import Response
async def on_fetch(request, env):
    await env.FOO.put("bar", "baz")
    bar = await env.FOO.get("bar")
    return Response.new(bar) # returns "baz"
```

This example shows how bindings work in Python Workers: Put a key into Workers KV, and then read it.

Python support in WASI

WASI is a system interface standard for WebAssembly. Through a combination of C compilers that can target WebAssembly and wasi-libc providing POSIX-compatible shims for WASI, it's possible for CPython to run on a WASI host/runtime as a guest.

To build for WASI, you will need to cross-compile CPython. This requires a C compiler just like building for Unix as well as:

- C compiler that can target WebAssembly (for example, WASI SDK)
- WASI host/runtime (for example, Wasmtime)