Compiling Python to WASM

WELCOME TO THE WORLD OF WEBASSEMBLY

HI 👋

Farhaan Bukhsh

Senior Software Engineer OpenCraft
Open edX Core Contributor

@farhaanbukhsh

Kumar Anirudha

Solutions Architect, Product Consultant

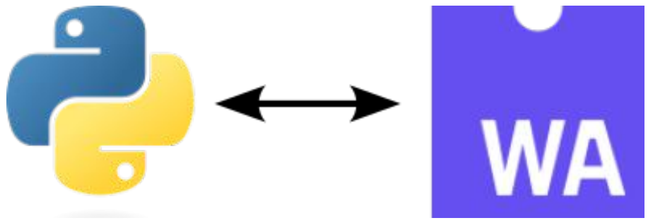
@anistark

The rise of WebAssembly

- WebAssembly (WASM) = Portable bytecode for the web
- Designed for speed, safety, and sandboxing
- Supported by all modern browsers
- Native-like performance on the client
- Opens doors for running non-JS languages on the web

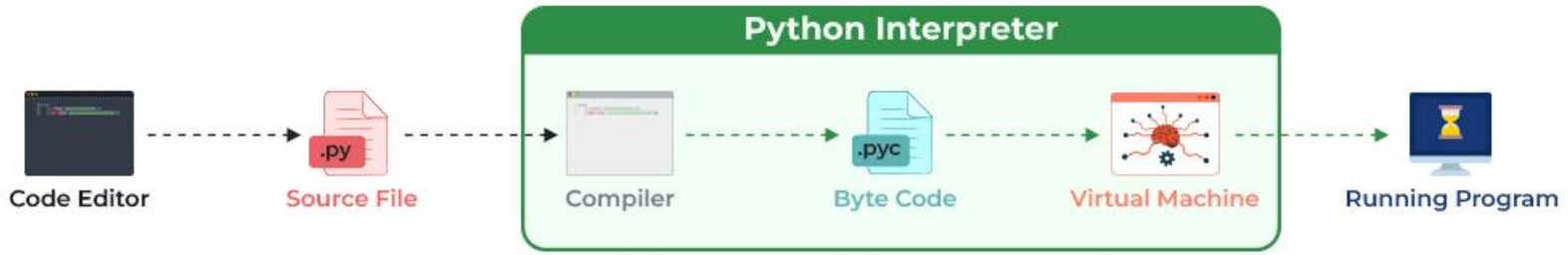
CHALLENGES OF RUNNING PYTHON IN WASM

python’s dynamic nature



Unlike Rust and Go, Python's dynamic typing and runtime features pose challenges for efficient WASM execution.

- Performance overhead.
- Limited support for certain Python features and libraries.



Pyodide: Python for Browser

Pyodide brings the Python runtime to the browser
by compiling CPython to WebAssembly


```
pip install pyodide
```

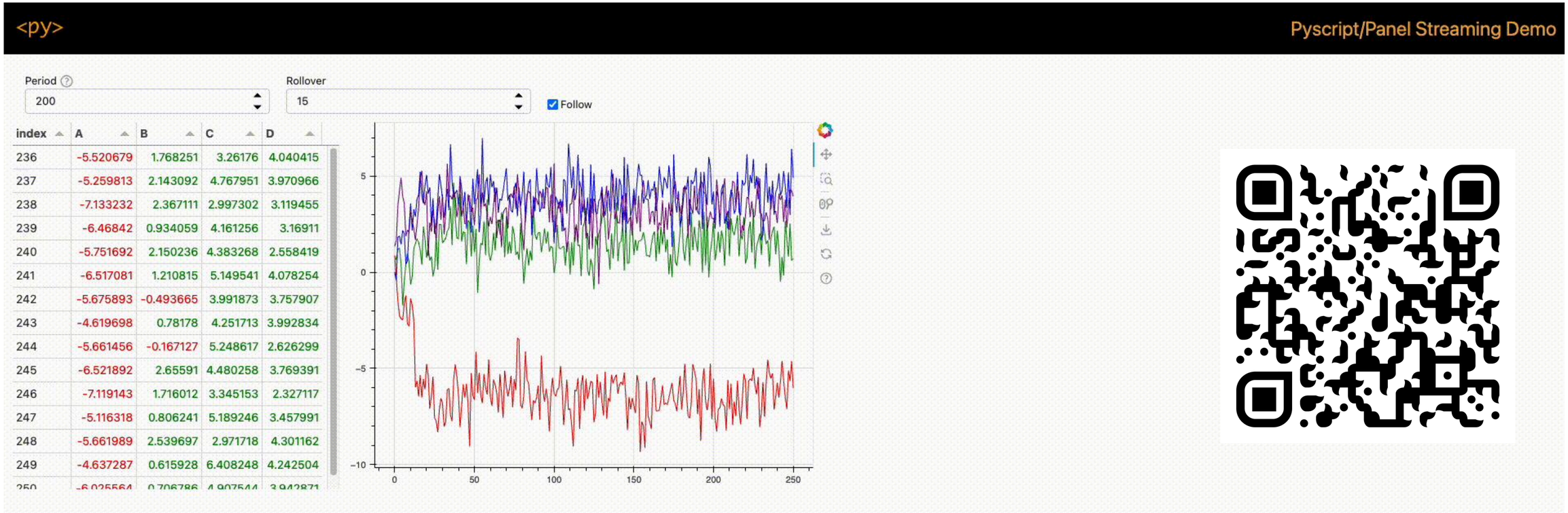
```
import pyodide

async def run_python_code():
    py = await pyodide.loadPyodide()
    result = py.runPython("print('Hello from Pyodide!')")
    print(result)

await run_python_code()
```

LIVE DEMO

Pyodide Demo

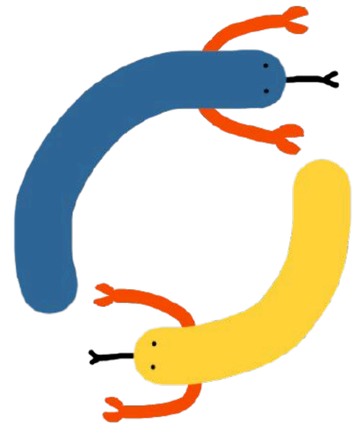
PyScript: Improve Python on Browser

PyScript is an open source platform for Python in the browser.



RustPython: A Rust-based Python Interpreter

RustPython is a Python interpreter written in Rust, optimized for WASM.



```
git clone https://github.com/RustPython/RustPython.git
cd RustPython
cargo build --release --target wasm32-wasi
```

```
use rustpython_vm::Interpreter;

fn main() {
    let interpreter = Interpreter::default();
    interpreter.enter(|vm| {
        vm.run_code("print('Hello from RustPython!')", vm.ctx.new_scope()).unwrap();
    });
}
```

```
wasmtime rustpython.wasm
```

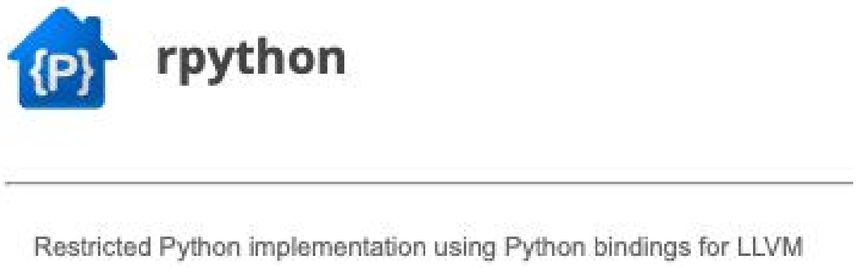
A BETTER WAY?

Python to WASM

Feature	Python *on* WASM	Python *to* WASM
Interpreter needed?	✅ Yes	❌ No
Startup size	❌ Large (MBs)	✅ Tiny (KBs)
Runtime speed	❌ Slower (interpreted)	✅ Near-native
Language support	✅ Full Python	❌ Subset/static
Deployment	❌ Heavy	✅ Lightweight
Use case	Education, prototyping, data science	WASM microservices, games, tooling, embedded scripts

Python Subsets

codon



PYTHON TO WASM

Py2Wasm

py2wasm is a compiler that transforms Python code into WebAssembly
It leverages **Nuitka**, a Python-to-C compiler, to convert Python code into C, which is then compiled into Wasm.


```
Python Source (.py)
    ↓
CPython AST Parser (unchanged)
    ↓
Nuitka Analysis (call graph, dependencies)
    ↓
C++ Code Generation (CPython API calls)
    ↓
Static Analysis for WASM compatibility
    ↓
Emscripten/Clang Compilation
    ↓
WASM Binary + libpython.a bundle
```

Nuitka
the Python Compiler

LIVE DEMO

Py2Wasm Demo

Codon

A high-performance Python-like compiler using LLVM

```
Python Source (.py)
    ↓
Codon Parser (custom Python parser)
    ↓
Type Inference & Checking
    ↓
CIR (Codon Intermediate Representation)
    ↓
Multiple CIR Optimization Passes
    ↓
LLVM IR Generation
    ↓
LLVM Optimization Pipeline
    ↓
Machine Code / WASM (via LLVM backend)
```

SPy

SPy is a subset/variant of Python specifically designed to be statically compilable while retaining a lot of the "useful" dynamic parts of Python.

```
SPy Source (.spy)
    ↓
SPy Parser (Python subset grammar)
    ↓
Meta-Programming Resolution (compile-time)
    ↓
Static Analysis & Type Inference
    ↓
HIR (High-level IR with resolved dynamics)
    ↓
Partial Evaluation & Specialization
    ↓
Target Backend (WASM planned, multiple targets)
```

github.com/spylang/spy

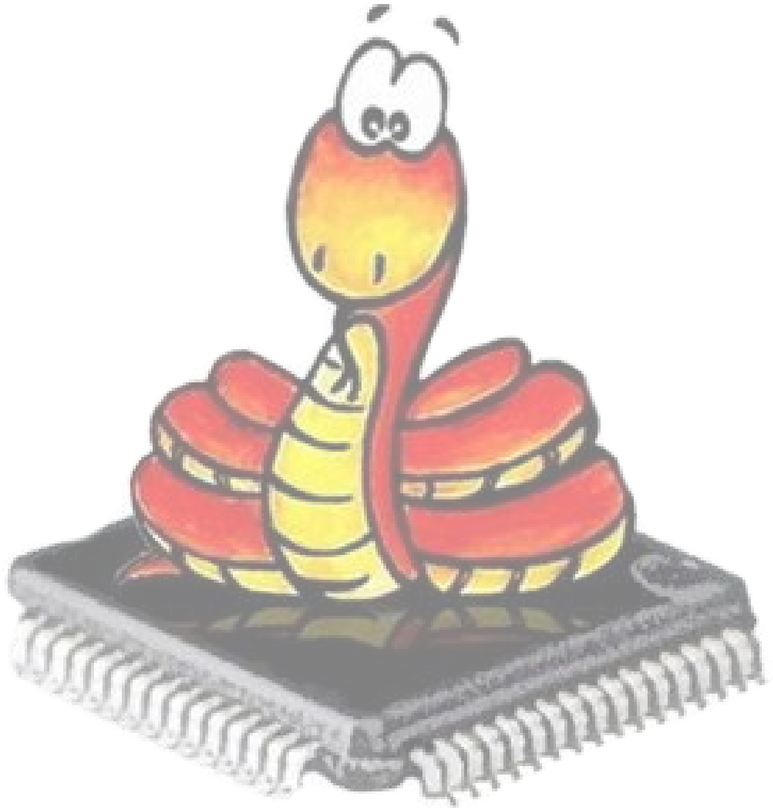
NEW

EXPLORING CURRENT SOLUTIONS

MicroPython

MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments.

```
Python Source (.py)
    ↓
MicroPython Lexer/Parser
    ↓
MicroPython AST (simplified vs CPython)
    ↓
MicroPython Bytecode Compiler
    ↓
MicroPython Bytecode (.mpy format)
    ↓
MicroPython VM Execution Engine
    ↓
Emscripten Compilation (VM → WASM)
    ↓
WASM Module (VM + Runtime + Optional Bytecode)
```



WHERE IT ALL BREAKS

Current Limitations

Current Limitations Across All Projects

1. Dynamic Features: **eval()**, **exec()**, runtime code generation
2. Import System: Dynamic imports, **importlib** manipulation
3. Introspection: Deep runtime introspection capabilities
4. Standard Library: Many modules require system calls unavailable in WASM
5. C Extensions: Native extension modules don't work

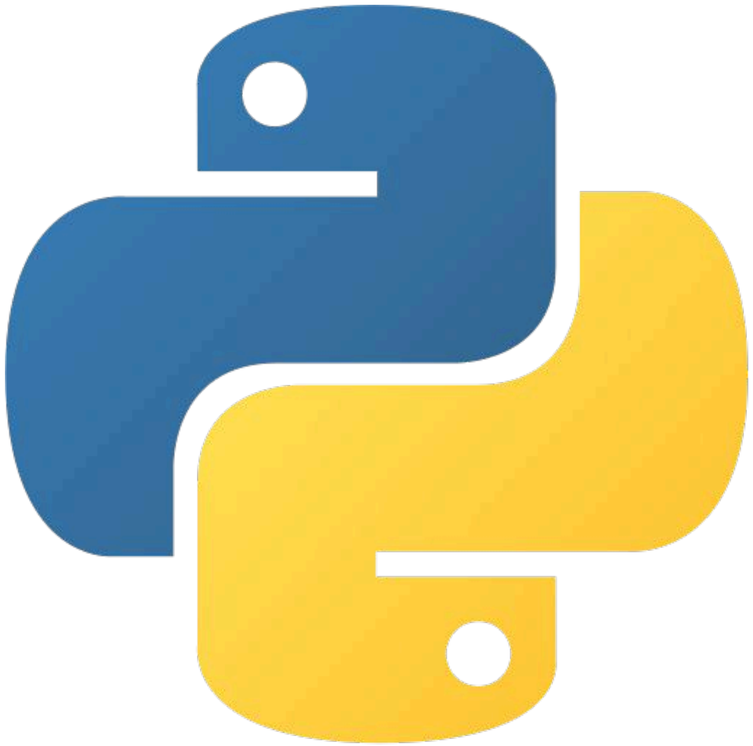
WASM-Specific Challenges

1. Memory Management: Reference counting vs GC integration
2. Threading: WASM threading model limitations
3. I/O: WASI interface constraints
4. Binary Size: Balancing features vs deployment size
5. Startup Time: Cold start performance for edge deployment

INTRODUCING

waspy

A Python to WebAssembly compiler written in Rust.

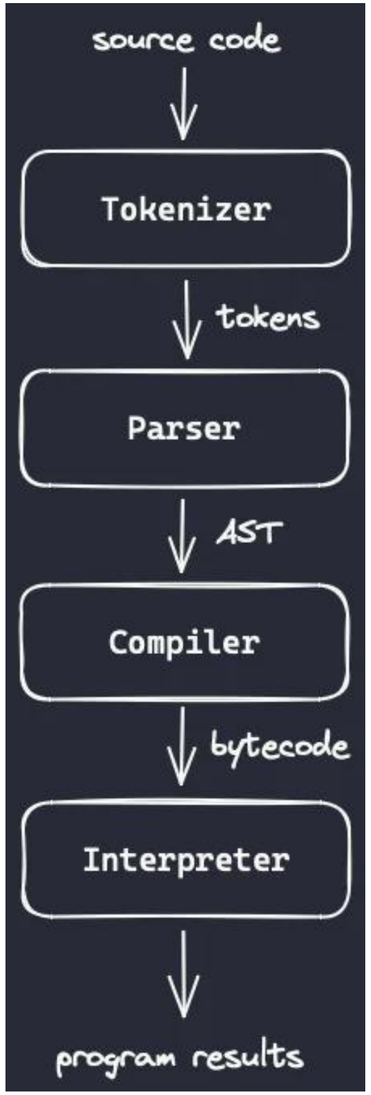


open source

github.com/anistark/waspy

HOW WASPY WORKS

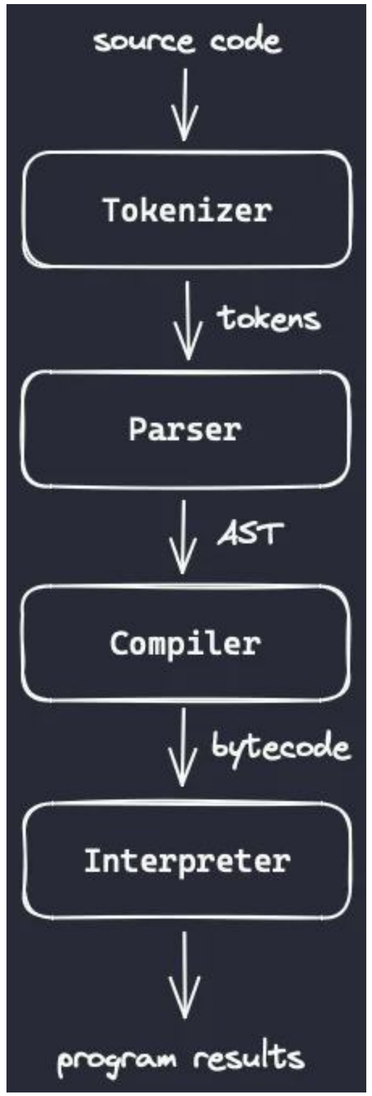
waspy

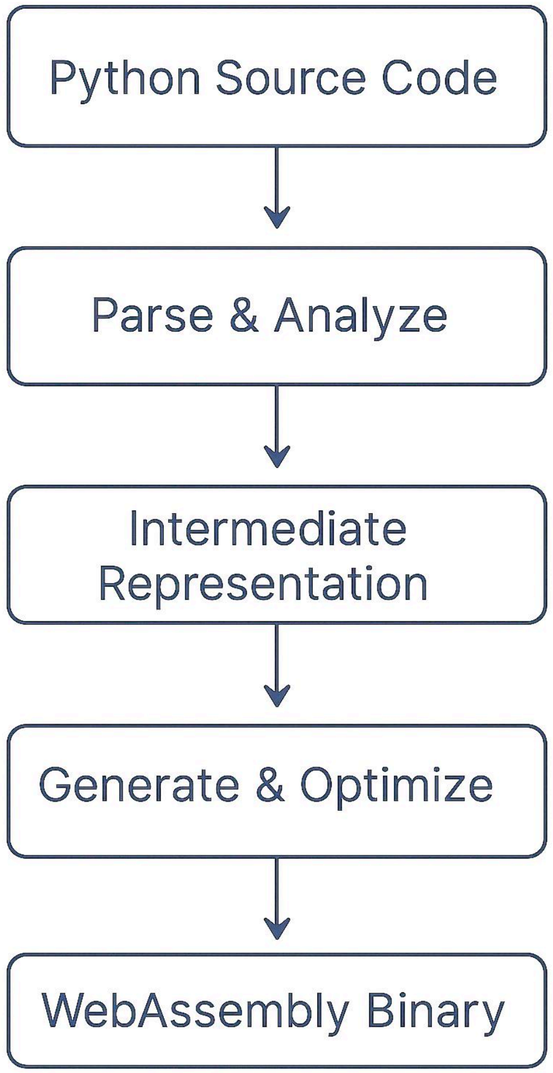


github.com/anistark/waspy

HOW WASPY WORKS

waspy

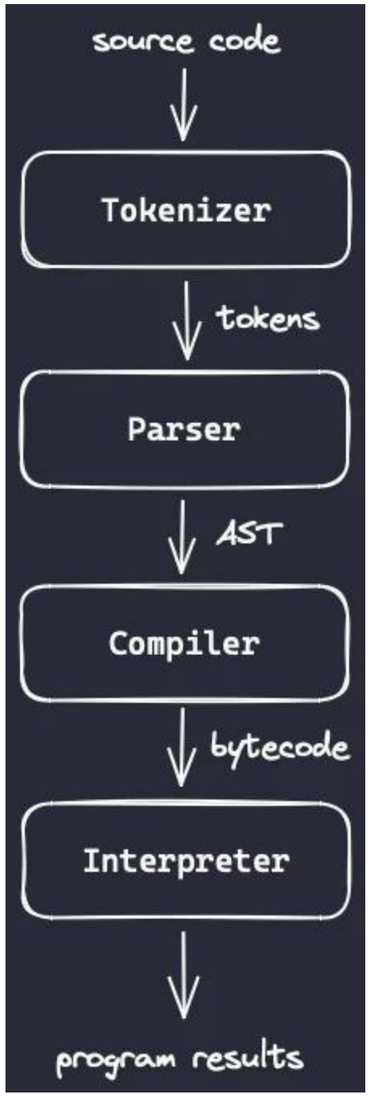


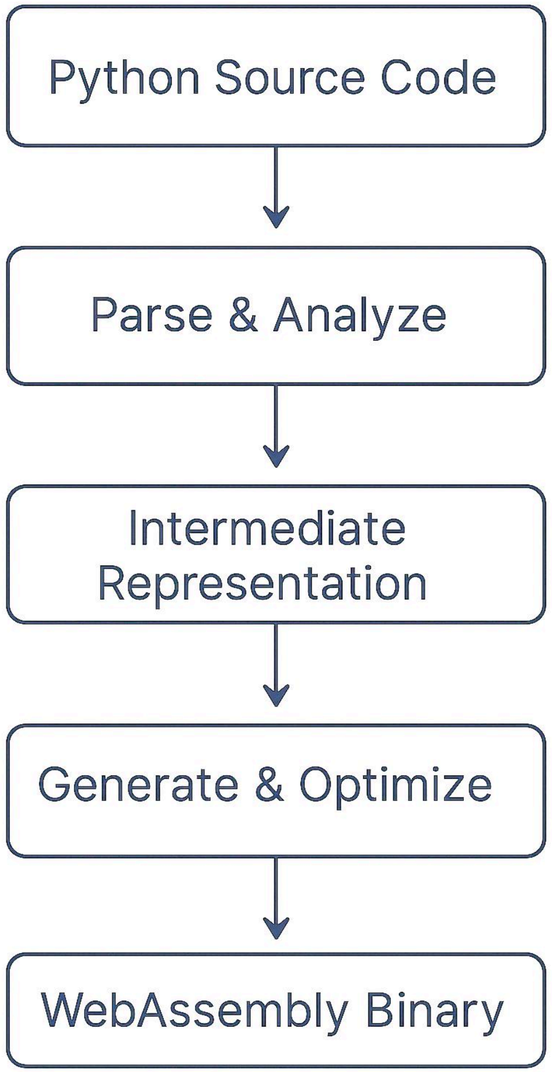


github.com/anistark/waspy

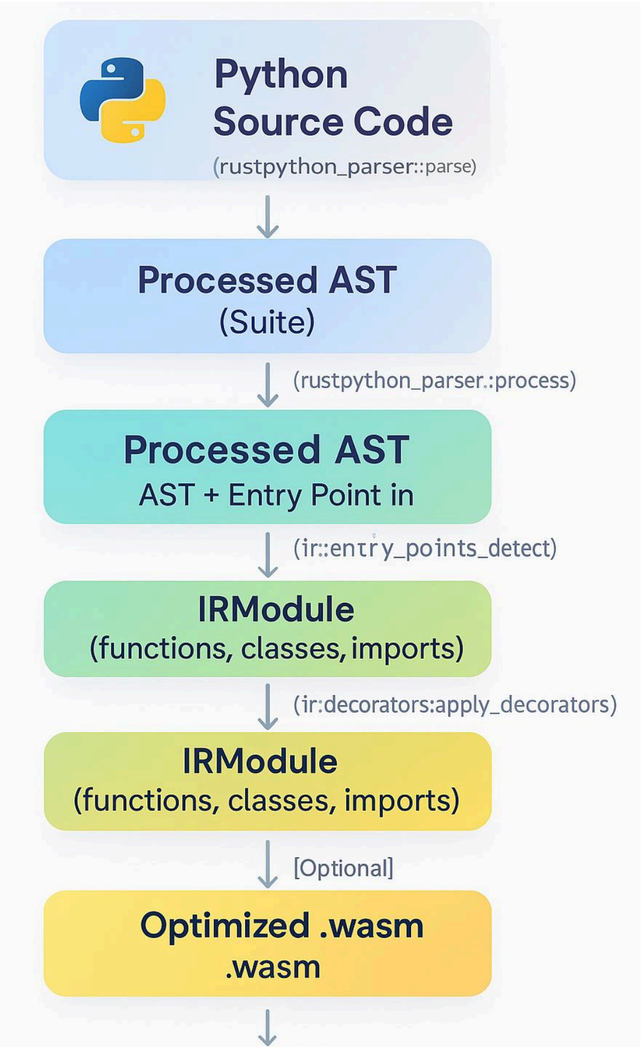
HOW WASPY WORKS

waspy





github.com/anistark/waspy



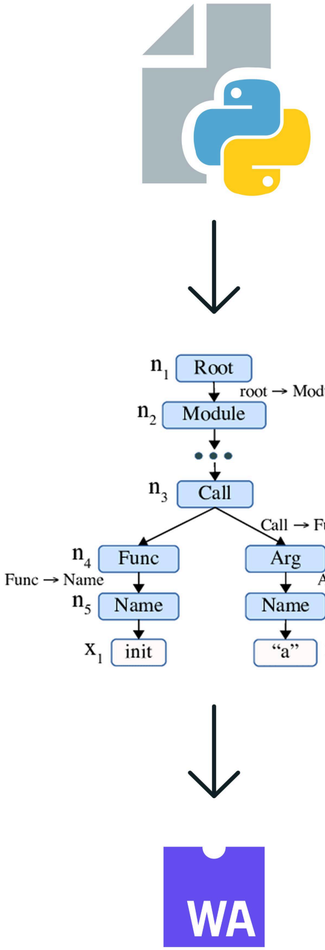
waspy

github.com/anistark/waspy

```
cargo add waspy
```

```rust
use waspy::compile_python_to_wasm;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let python_code = r#"
    def add(a: int, b: int) -> int:
        return a + b

    def fibonacci(n: int) -> int:
        if n <= 1:
            return n
        a = 0
        b = 1
        i = 2
        while i <= n:
            temp = a + b
            a = b
            b = temp
            i = i + 1
        return b
    "#;

    let wasm = compile_python_to_wasm(python_code)?;
    // Write to file or use the WebAssembly binary
    std::fs::write("output.wasm", &wasm)?;

    Ok(())
}
```

```javascript
// Browser or Node.js
WebAssembly.instantiate(wasmBuffer).then(result => {
  const instance = result.instance;
  console.log(instance.exports.factorial(5)); // 120
  console.log(instance.exports.max_num(42, 17)); //
42
});
```



LIVE DEMO

waspy Demo

CURRENT STATE

waspy

github.com/anistark/waspy

ARITHMETIC — COMPLETED

- ✅ All operators (+, -, *, /, %, //, **)
- ✅ Type coercion (int/float)
- ✅ Bitwise operations (&, |, ^, <<, >>, ~)
- ✅ Unary operations (-, +, not)

CONTROL FLOW — COMPLETED

- ✅ if/elif/else statements
- ✅ while loops
- ✅ Comparison operations (==, !=, <, <=, >, >=)
- ✅ Boolean logic (and, or, not) with short-circuit

FUNCTIONS — COMPLETED

- ✅ Function definitions with parameters
- ✅ Type annotations (params & return types)
- ✅ Function calls between compiled functions
- ✅ Multiple functions per module

TYPE SYSTEM — COMPLETED

- ✅ Basic types (int, float, bool)
- ✅ Generic types (List[T], Dict[K,V], Tuple[T,...])
- ✅ Union/Optional types
- ✅ Custom class type annotations

VARIABLES & ASSIGNMENT — COMPLETED

- ✅ Variable declarations and assignments
- ✅ Augmented assignment (+=, -=, *=, etc.)
- ✅ Attribute assignment (obj.attr = value)
- ✅ Type inference from usage

DECORATORS — COMPLETED

- ✅ @memoize decorator (caching)
- ✅ @debug decorator (logging)
- ✅ @timer decorator (performance)
- ✅ Custom decorator registration

PROJECT MANAGEMENT — COMPLETED

- ✅ Multi-file compilation to single WASM
- ✅ Dependency analysis & circular detection
- ✅ Entry point detection (__main__.py)
- ✅ Config parsing (setup.py, pyproject.toml)

OPTIMIZATION — COMPLETED

- ✅ WebAssembly optimization (Binaryen)
- ✅ Compiler options & configuration
- ✅ Error handling with detailed messages
- ✅ Memory layout optimization

STRINGS — IN PROGRESS

- ✅ String literals and constants
- 🔄 String indexing (returns bytes, not char)
- ❌ String methods (.upper(), .lower(), .split())
- ❌ String concatenation (+) & slicing

38% Complete

COLLECTIONS — IN PROGRESS

- ✅ List literals ([1, 2, 3]) - parsing only
- ✅ Dict literals ({'key': 'value'}) - parsing only
- ❌ List/Dict indexing & operations
- ❌ Dynamic resizing & memory management

50% Complete

CLASSES & OOP — IN PROGRESS

- ✅ Class definitions & method parsing
- ✅ Basic inheritance (base class tracking)
- ❌ Object instantiation & method calls
- ❌ Special methods (__init__, __str__, etc.)

50% Complete

ADVANCED CONTROL FLOW — IN PROGRESS

- 🔄 for loops (syntax parsing only)
- 🔄 try/except (syntax parsing only)
- 🔄 with statements (syntax parsing only)
- ❌ Actual iteration & exception handling

38% Complete

BUILT-IN FUNCTIONS — IN PROGRESS

- ✅ Type conversions (int(), float(), str(), bool())
- 🔄 len() - recognized but not implemented
- 🔄 print() - recognized but no actual output
- ❌ min(), max(), sum(), abs(), round()

50% Complete

IMPORT SYSTEM — IN PROGRESS

- ✅ Import syntax parsing (all types)
- ✅ Conditional imports (try/except)
- ✅ Dynamic imports (__import__, importlib)
- ❌ Actual module loading & execution

75% Complete

WEB INTEROP — TODO

- ❌ JavaScript bridge & value conversion
- ❌ DOM manipulation APIs
- ❌ Event handling system
- ❌ HTTP requests (fetch API)

0% Complete

STANDARD LIBRARY — TODO

- ❌ sys module (system parameters)
- ❌ os module (operating system interface)
- ❌ math module (mathematical functions)
- ❌ json module (JSON encoding/decoding)

0% Complete

GENERATORS & ITERATORS — TODO

- ❌ yield statement & generator functions
- ❌ Iterator protocol (__iter__, __next__)
- ❌ range() function implementation
- ❌ for loop iteration over collections

0% Complete

ASYNC/AWAIT — TODO

- ❌ Promise object representation
- ❌ async/await syntax support
- ❌ Asynchronous function execution
- ❌ Concurrent task management

0% Complete

COMPREHENSIONS — TODO

- ❌ List comprehensions ([x for x in list])
- ❌ Dict comprehensions ({k: v for k, v in items})
- ❌ Set comprehensions ({x for x in list})
- ❌ Generator expressions (x for x in list)

0% Complete

EXCEPTION HANDLING — TODO

- ❌ Exception throwing (raise statements)
- ❌ Exception catching by type
- ❌ Built-in exception types
- ❌ Exception propagation & finally blocks

0% Complete

ADVANCED DATA TYPES — TODO

- ❌ Set type ({1, 2, 3}) & operations
- ❌ Tuple operations & unpacking
- ❌ String formatting (f-strings, .format())
- ❌ Bytes type & binary data handling

0% Complete

MEMORY MANAGEMENT — TODO

- ❌ Garbage collection system
- ❌ Reference counting & leak prevention
- ❌ Memory pools for efficient allocation
- ❌ WASM memory optimization

0% Complete

ADVANCED OOP — TODO

- ❌ Multiple inheritance & method resolution
- ❌ Metaclasses & class creation
- ❌ Property decorators (@property, @setter)
- ❌ Static methods (@staticmethod, @classmethod)

0% Complete

GRAPHICS & CANVAS — TODO

- ❌ Canvas API (2D drawing)
- ❌ WebGL support (3D graphics)
- ❌ Animation & rendering loops
- ❌ Image manipulation & processing

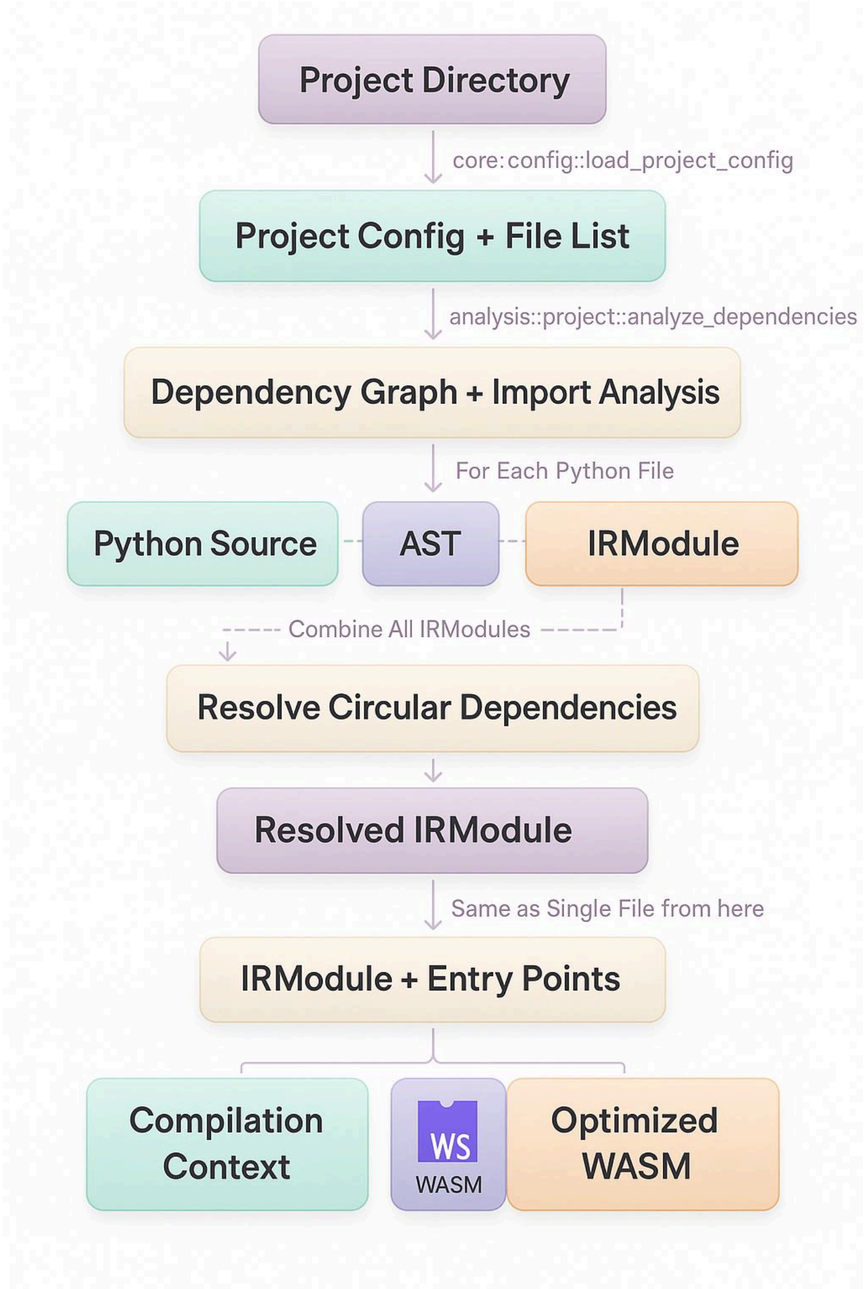
0% Complete

33%

COMPLETE

PYTHON PROJECT SUPPORT

waspy



github.com/anistark/waspy

HIGHLY EXPERIMENTAL

ALTERNATIVE APPROACH

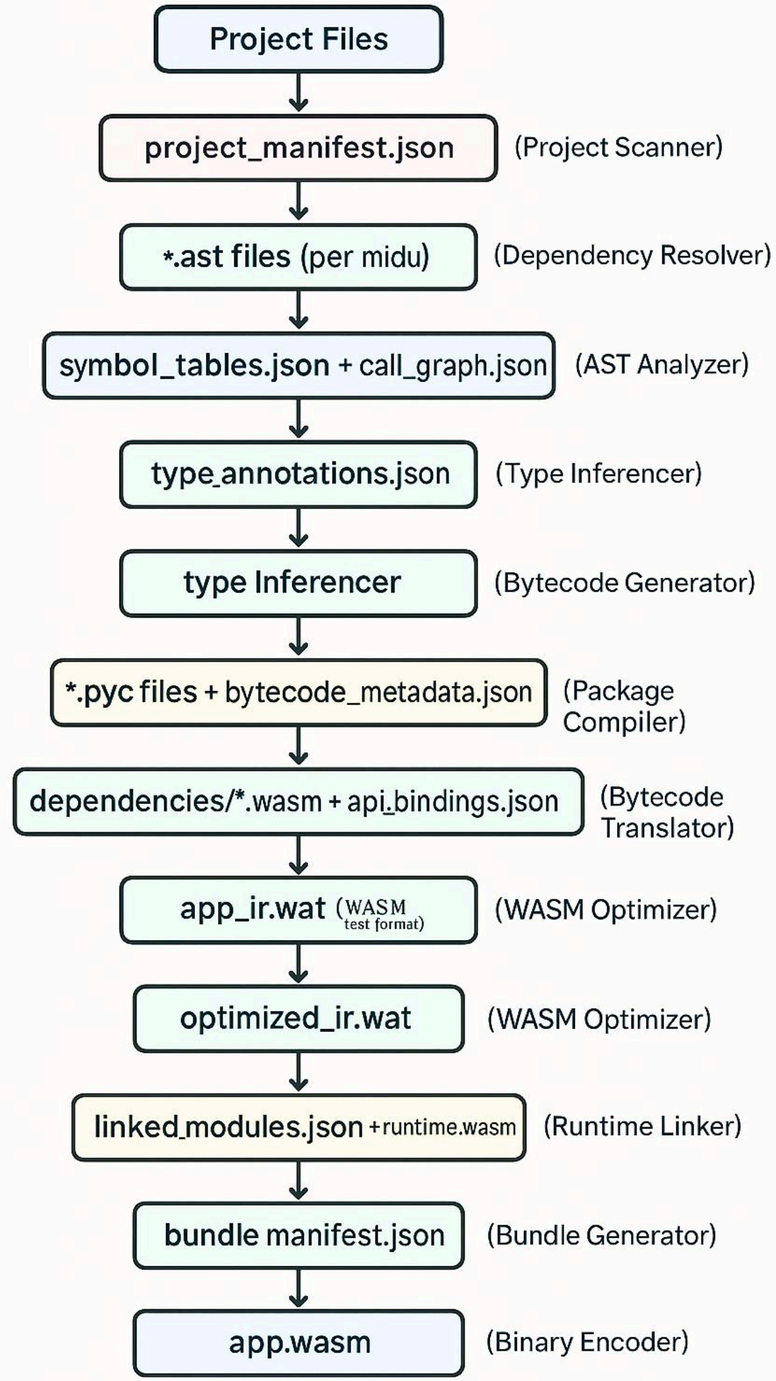
In early discussion

- C-API Bridge
- Extension/Module for CPython
- JIT and AOT support

PEP 489

Multi-phase extension module initialization

python™



Thank you for your patience

Q & A

@kranirudha

Find us on 𝕏

@fhackdroid