Waspy: Direct Ahead-of-Time Compilation of Typed Python to Compact WebAssembly

Kumar Anirudha · Farhaan Bukhsh

Last updated July 8, 2026

Abstract

WebAssembly (Wasm) has become a universal, sandboxed compilation target that runs in every major browser and in a growing family of server-side runtimes. Python, meanwhile, remains one of the world's most widely used languages, yet it reaches Wasm today almost exclusively by shipping an interpreter: Pyodide, CPython's Emscripten builds, MicroPython's Wasm port, and Nuitka-based tools all bundle a Python runtime into the deployed artifact, producing payloads from hundreds of kilobytes to tens of megabytes and paying interpreter startup and dispatch costs on every run. This paper presents *Waspy*, an open-source ahead-of-time compiler, written in Rust, that translates a statically typed subset of Python directly into freestanding WebAssembly modules with no embedded interpreter and no mandatory JavaScript glue. Waspy parses CPython-compatible syntax, lowers it to a typed intermediate representation, emits Wasm bytecode directly, and post-optimizes the binary with Binaryen. The supported subset covers functions, control flow, classes with single inheritance and heap-allocated instances, collections, exceptions, lambdas and comprehensions, and a compiled subset of the standard library. On representative programs, Waspy produces optimized modules of 0.2 to 5.5 KB with end-to-end compile times under 20 ms, three to four orders of magnitude smaller than interpreter-based deployment. We describe the design trade-offs of mapping a dynamic language onto Wasm's static type system and linear memory, the current limitations of the approach, and the roadmap toward garbage collection, growable collections, and generators.

I. Introduction

WebAssembly [1] is a portable, verifiable, sandboxed binary instruction format supported natively by all major browsers and, through standalone runtimes such as Wasmtime and Wasmer, increasingly used for serverless functions, plugin systems, and edge computing. Its promise is simple: compile once, run anywhere, with near-native performance and strong isolation.

Python sits awkwardly against this promise. It is consistently ranked among the most popular programming languages, yet it is an interpreted, dynamically typed language whose reference implementation (CPython) is a large C program. The dominant strategy for running Python on Wasm is therefore to compile *the interpreter* to Wasm and ship it alongside the user's source code. Pyodide [2] compiles CPython and much of the scientific stack with Emscripten; the CPython project itself now ships official Emscripten and WASI build targets [3]; MicroPython provides a smaller Wasm port [4]; and tools such as py2wasm [5] use Nuitka [6] to transpile Python to C before linking a CPython runtime into the output.

All of these inherit the same structural costs. First, **payload size**: the deployed artifact contains a complete language runtime, from hundreds of kilobytes for MicroPython to several megabytes for CPython-based approaches, regardless of how small the user program is. Second, **startup latency**: the interpreter must be instantiated and bootstrapped before the first line of user code runs, which is hostile to cold-start-sensitive environments such as edge functions and browser interactions. Third, **opacity of the interface**: an embedded interpreter exposes an *eval*-style entry point rather than typed, directly callable Wasm exports, complicating composition with other Wasm modules and host tooling.

This paper explores the opposite point in the design space: *compile the Python program itself, not the interpreter*. Waspy is an ahead-of-time (AOT) compiler that translates a statically typed subset of Python di-

rectly into a self-contained Wasm module. Each Python function becomes an exported Wasm function with a typed signature; there is no interpreter, no bytecode, and no required JavaScript shim. The price is coverage: Waspy compiles a subset of Python, anchored on type annotations and augmented with local type inference, rather than the full dynamic language. We argue that this trade is worthwhile for a large class of workloads, such as numeric kernels, business logic, validation rules, embedded scripting, and teaching, where developers want Python's syntax and ergonomics but Wasm's size, speed, and composability.

The contributions of this paper are:

- The design and implementation of Waspy, an open-source (MIT-licensed) Python-to-Wasm AOT compiler written in Rust, with a four-stage pipeline: CPython-compatible parsing, lowering to a typed intermediate representation (IR), direct Wasm emission, and Binaryen post-optimization (Section III).
- A pragmatic mapping of Python semantics (objects, collections, strings, exceptions, and a standard-library subset) onto Wasm's four value types and linear memory, without a garbage collector or runtime library (Section IV).
- An evaluation on representative programs showing optimized module sizes of 233 bytes to 5.5 KB and compile times under 20 ms, compared against interpreter-shipping alternatives (Section V).
- An honest account of the semantic gaps that remain (memory reclamation, growable collections, generators, full closures) and a roadmap for closing them (Section VI).

II. Background and Related Work

A. WebAssembly as a compilation target

Wasm [1] defines a stack machine with four value types (`i32`, `i64`, `f32`, `f64`), structured control flow, and a single linear memory per module (in the widely deployed core specification). There is no built-in garbage collector in core Wasm, no exceptions (in the baseline widely available everywhere), and no direct access to the host: everything crosses an explicit import/export boundary. Languages with static types and manual memory management (C, C++, Rust, Zig) map onto this model naturally; dynamic languages do not, which is precisely why the interpreter-shipping strategy became the default for Python, Ruby, and PHP.

B. Interpreter-in-Wasm approaches

Pyodide [2] is the most complete Python-on-Wasm environment: full CPython plus NumPy, Pandas, and SciPy compiled with Emscripten, with rich JavaScript interoperability. Its completeness is also its weight: the core runtime download is measured in megabytes before any scientific packages, and instantiation takes seconds on typical connections. PyScript builds a browser framework on top of it. CPython 3.13+ ships official WebAssembly build targets [3] with similar characteristics. MicroPython's Wasm port [4] shrinks the runtime dramatically (hundreds of kilobytes) by re-implementing a Python subset, but still ships an interpreter and its dispatch overhead. RustPython [7], a Python interpreter in Rust, can also run in the browser via Wasm, again as an interpreter.

C. AOT and hybrid compilers for Python

Nuitka [6] transpiles Python to C, but links against libpython and thus retains the full runtime. py2wasm [5] applies Nuitka with a Wasm-targeting toolchain, reporting roughly 3× speedups over interpreted CPython-on-Wasm, but its output still embeds a Python runtime measured in megabytes. Cython [8] and mypyc compile annotated Python to C for use as CPython extension modules; they accelerate Python but do not produce freestanding artifacts. Codon [9] is the closest relative in spirit: an LLVM-based AOT compiler for a statically typed Python dialect achieving C-like performance. Codon, however, targets native code first, is not open source under an OSI license, and does not aim at minimal self-contained Wasm modules. Waspy differs from all of the above in optimizing for exactly that: the smallest possible, dependency-free Wasm module with typed exports, produced by a permissively licensed, embeddable compiler library.

D. Compiling dynamic languages statically

Compiling a dynamic language ahead of time requires either whole-program type inference, gradual typing with annotations, or run-time fallback mechanisms. PyPy [10] exemplifies the run-time route with its meta-tracing JIT, but that approach is unavailable inside a Wasm module, which cannot generate and execute new code at run time. Waspy instead takes the gradual-typing route pioneered in practice by mypy and adopted by Codon and mypyc: function signatures must be annotated (PEP 484 [11]), and local variable types are inferred from initialization and use. This restricts the accepted language but keeps both the compiler and the generated code simple, predictable, and small, which is an explicit design goal.

III. Design and Architecture

Waspy is implemented in Rust (~0.11.0 at the time of writing) and distributed as a library crate, so it can be embedded in build tools, web services, and plugin hosts; a companion runner integration (`wasmrun`) provides a CLI workflow. Fig. 1 shows the pipeline.

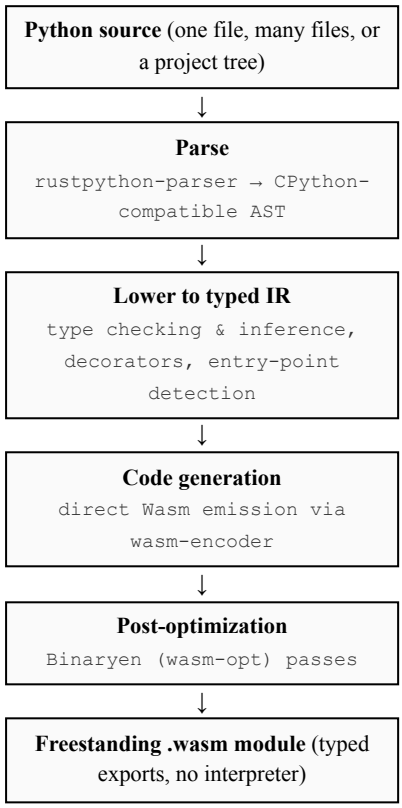

Fig. 1. The Waspy compilation pipeline.

A. Parsing

Waspy reuses `rustpython-parser`, the parser of the RustPython project, and therefore accepts standard CPython grammar. This is a deliberate separation of concerns: Waspy imposes no syntactic dialect. Any accepted program is valid Python that also runs under CPython, which preserves the developer's existing tooling (editors, linters, type checkers, tests) and makes the semantics auditable by differential testing against CPython.

B. Typed intermediate representation

The AST is lowered into a compact typed IR of modules, functions, statements, and expressions. Types enter the IR from three sources: (1) mandatory PEP 484 annotations on function parameters and return values, (2) literal and operator typing, and (3) forward local inference, in which a variable's type is fixed by its first assignment and checked at subsequent uses, with automatic coercion between compatible numeric types.

The IR stage also resolves decorators through a registry, detects `if __name__ == "__main__"` entry points and synthesizes a corresponding exported entry function, and merges multiple source files (or a whole project directory, discovered via lightweight import analysis) into a single IR module so that cross-file calls compile to direct Wasm calls.

C. Code generation

Code generation emits Wasm bytecode directly through the `wasm-encoder` crate; there is no C, LLVM, or text-format intermediary. Each Python function becomes one Wasm function, exported under its original name. Python locals map to Wasm locals; control flow (`if`/`elif`/`else`, `while`, `for` over ranges and collections, `break`/`continue` including inside nested `if`/`try` blocks) maps to Wasm structured `block`/`loop`/`br` instructions, which fit Python's structured control flow with no relooping needed. Short-circuit `and`/`or` compile to conditional branches. Because the module is closed at compile time, all calls are direct and the whole program is visible to the optimizer.

D. Post-optimization

The raw module is passed through Binaryen [12], applying its standard optimization pipeline (dead code elimination, inlining, local coalescing, instruction simplification, and size-directed passes). Optimization is on by default and can be disabled for debugging or for downstream toolchains that prefer to optimize themselves. The compiler can additionally emit function metadata and a self-contained HTML test harness for immediate execution in a browser.

IV. Mapping Python onto Wasm

A. Scalar types

Table I shows the value mapping. Integers map to `i32`, floats to IEEE-754 `f64`, and booleans to `i32` 0/1. The full arithmetic operator set is supported, including floor division, modulo with Python sign semantics, and exponentiation.

TABLE I. Python-to-WebAssembly type mapping

Python type	Wasm representation
`int`	`i32` value
`float`	`f64` value
`bool`	`i32` (0 / 1)
`str`	pointer (`i32`) into linear memory; length-prefixed
`list`, `tuple`, `range`	pointer to fixed-capacity region in linear memory
`dict`	pointer to key/value region in linear memory
`set`	pointer to open-addressing hash table in linear memory
class instance	pointer to heap-allocated field block (`__alloc`)

B. Strings and collections

Strings live in linear memory and support slicing, concatenation, formatting, and more than twenty methods, with compile-time evaluation where operands are constant. Lists, tuples, dicts, sets, and ranges are laid out as regions in linear memory with capacities fixed at compile time from their literals; loops that build collections allocate fresh per-iteration regions to preserve value semantics. Sets use an open-addressing hash table, giving expected-constant membership tests; `in`/`not in`, indexing, and the common method surface are supported across collection types, including float-element collections using full-width `f64` slots.

C. Objects and classes

Class instances are heap-allocated through a compiler-generated bump allocator (`__alloc`) over linear memory. Multiple live instances per class are supported; instances can be passed to and returned from functions. Waspy supports single inheritance with `super()` delegation and `isinstance` checks, resolved statically where possible. Because the class hierarchy is closed at compile time, method calls compile to direct calls rather than dictionary lookups. This is one of the places where the subset restriction converts Python's most expensive dynamic feature into zero-overhead code.

D. Exceptions

Core Wasm (as universally deployed) lacks native exception handling, so Waspy compiles `try`/`except`/`finally`/`raise` into structured control flow: raising sets an error state and branches to the innermost matching handler, and `finally` blocks are duplicated along normal and exceptional edges. This keeps the output compatible with every Wasm runtime, at the cost of some code-size duplication in exception-heavy code.

E. Standard library subset

Rather than shipping a runtime library, Waspy implements a subset of the standard library *inside the compiler*: calls into `math`, `random`, `json`, `re`, `datetime`, `logging`, `collections`, `itertools`, `functools`, `sys`, and `os`/`os.path` are recognized during lowering and compiled to inline Wasm implementations or intrinsic sequences. Only code that is actually used is emitted, so a program that calls `math.sqrt` pays for square root and nothing else, in contrast to interpreter bundles, where the whole library is present regardless.

F. Developer experience

Compilation errors carry source locations and specific error categories (parse errors, type errors, unsupported features), with warnings for constructs that compile but deserve attention. The library exposes four verbosity levels and options for debug info, metadata extraction, and HTML harness generation. The test suite currently comprises 89 unit and integration tests exercising the pipeline end to end.

V. Evaluation

We evaluate the two properties Waspy is designed for: output size and compilation speed. All measurements were taken with Waspy 0.11.0 (optimizations enabled) on an Apple-silicon macOS machine, compiling the example programs distributed with the project. Table II reports optimized module sizes.

TABLE II. Optimized Wasm module sizes for example programs

Program	Exercises	Size (bytes)
exceptions	try/except/finally, raise	233
basic_operations	arithmetic, comparisons	422
control_flow	if/while/for, break/continue	575
calculator	multi-function module	921
typed_demo	type system, coercions	1,196
nested_collections	lists/dicts/sets, nesting	5,528

Every module is self-contained and directly instantiable: `WebAssembly.instantiate(bytes)` in a browser or Node.js, or any standalone runtime, with each Python function available as a typed export. End-to-end compilation (parse through Binaryen) completes in under 20 ms for these programs, making Waspy practical inside interactive tooling and build pipelines.

Table III positions these numbers against the interpreter-shipping alternatives. The figures for other systems are approximate published deployment sizes for a minimal "hello, add two numbers" workload; they

should be read as orders of magnitude, not benchmarks, since the systems differ enormously in language coverage.

TABLE III. Deployment characteristics for a small Python function on Wasm (orders of magnitude; coverage differs greatly)

System	Artifact contains	Payload	Python coverage
Pyodide [2]	CPython + stdlib (+packages)	> 6 MB	effectively full
CPython/Emscripten [3]	CPython + stdlib	MBs	full (per platform tier)
py2wasm [5]	transpiled C + CPython runtime	MBs	near-full
MicroPython Wasm [4]	MicroPython VM	100s of KB	large subset
Waspy	**compiled user code only**	**0.2 to 6 KB**	typed subset

The gap of three to four orders of magnitude is structural, not incidental: it is the difference between shipping a program and shipping a language implementation. For workloads inside Waspy's subset, this translates directly into faster cold starts, lower bandwidth, and modules small enough to inline into HTML pages, smart-contract-style environments, or per-request edge deployments.

VI. Limitations and Roadmap

Waspy is under active development, and its current limitations are documented rather than hidden:

- **No memory reclamation.** The bump allocator has no `free`; instances live until the module is torn down and `__del__` never runs. Reference counting or integration with the Wasm GC proposal is the highest-priority roadmap item.
- **Fixed-capacity collections.** Collection capacity is fixed at compile time from literals; growth past the initial capacity (e.g., unbounded `.append`) is unsafe. Runtime reallocation is planned.
- **Generators and closures.** `yield` does not yet preserve execution state, and closure variable capture analysis is incomplete. Lambdas, basic closures, and list comprehensions do compile.
- **Imports and I/O.** Only the bundled stdlib subset can be imported inside a module; user-written `.py` module imports are handled at the project-compilation level rather than as a general import system, and file I/O awaits a WASI-backed implementation.

The path to 1.0 also includes completing the object model (classmethod/staticmethod, dataclasses), hashed dictionary lookups (sets already use hashing), and module caching. Two properties of the architecture make us optimistic about closing these gaps incrementally: the typed IR isolates each feature's lowering, and CPython compatibility means every new feature can be validated differentially against the reference implementation.

VII. Use Cases and Outlook

The design point Waspy occupies (tiny, typed, freestanding modules from annotated Python) serves several concrete audiences. Web developers can compile hot numeric or logic kernels to Wasm without leaving Python or adopting Rust/C++. Platform engineers get a safe way to accept user-supplied Python logic (rules, pricing, validation) and run it sandboxed at the edge with microsecond instantiation. Educators get a transparent, 20 ms-feedback compiler whose entire pipeline (AST, typed IR, Wasm text) is small enough to read, making it a practical vehicle for teaching compiler construction with a familiar source language. Because Waspy is an embeddable MIT-licensed library on crates.io, it can also serve as compilation infrastructure inside larger systems, from notebook environments to plugin runtimes.

VIII. Conclusion

Interpreter-in-Wasm systems answer the question "how do we run all of Python on WebAssembly?" Waspy answers a different one: "how much of Python can we run on WebAssembly for (almost) free?" The answer is already substantial: typed functions, control flow, classes with inheritance, collections, exceptions, and a compiled stdlib subset, delivered as freestanding modules of a few hundred bytes to a few kilobytes, compiled in milliseconds by a small, embeddable, permissively licensed Rust library. As the supported subset grows toward garbage collection, growable collections, and generators, we believe direct AOT compilation will become the default way to put Python *programs*, rather than Python *interpreters*, on the web and the edge.

Availability

Waspy is open source under the MIT license. Source code, examples, and documentation: `https://github.com/anistark/waspy`; releases are published on crates.io (`cargo add waspy`).

Acknowledgments

Waspy is a community open-source project, and this work reflects the efforts of its wider contributor base. We thank all contributors to the Waspy repository, whose individual contributions are recorded at `https://github.com/anistark/waspy/graphs/contributors`, as well as the maintainers of the upstream projects Waspy builds on: rustpython-parser, wasm-encoder, and Binaryen.

References

1. A. Haas et al., "Bringing the Web up to Speed with WebAssembly," in *Proc. 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185-200.
2. The Pyodide development team, "Pyodide: Python distribution for the browser and Node.js based on WebAssembly," https://pyodide.org, accessed Jul. 2026.
3. Python Software Foundation, "CPython WebAssembly (Emscripten and WASI) build targets," https://docs.python.org, accessed Jul. 2026.
4. MicroPython project, "MicroPython WebAssembly port," https://github.com/micropython/micropython/tree/master/ports/webassembly, accessed Jul. 2026.
5. Wasmer, Inc., "py2wasm: A Python to WebAssembly compiler," https://wasmer.io/posts/py2wasm-a-python-to-wasm-compiler, 2024.
6. K. Hayen, "Nuitka: The Python compiler," https://nuitka.net, accessed Jul. 2026.
7. RustPython contributors, "RustPython: A Python interpreter written in Rust," https://rustpython.github.io, accessed Jul. 2026.
8. S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The Best of Both Worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31-39, 2011.
9. A. Shajii, G. Ramirez, H. Smajlović, J. Ray, B. Berger, S. Amarasinghe, and I. Numanagić, "Codon: A Compiler for High-Performance Pythonic Applications and DSLs," in *Proc. 32nd ACM SIGPLAN Int. Conf. on Compiler Construction (CC)*, 2023, pp. 191-202.
10. C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the Meta-Level: PyPy's Tracing JIT Compiler," in *Proc. 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, 2009, pp. 18-25.
11. G. van Rossum, J. Lehtosalo, and Ł. Langa, "PEP 484: Type Hints," Python Enhancement Proposals, 2014, https://peps.python.org/pep-0484/.
12. A. Zakai and the Binaryen contributors, "Binaryen: Optimizer and compiler/toolchain library for WebAssembly," https://github.com/WebAssembly/binaryen, accessed Jul. 2026.