Running Python packages in the browser with Pyodide

Roman Yurchak

2023/07/20

PYODIDE
WA

About me

Roman Yurchak

ML engineer and founder at Symerio (based in Paris)

Background in computational physics.

Core developer at Pyodide, previously scikit-learn

@RomanYurchak

SYMERIO

Agenda

1. Introducing Pyodide

2. Optimizing Python packages size and load time in the browser

Python in the browser with Pyodide: an overview

What is WebAssembly?

A binary instruction format for a stack-based virtual machine (VM)

- Portable, small code size
- Secure
- No standard APIs or syscalls, only an import mechanism

- Implemented in **browsers**
- Can also be executed in non-web environments including **Node.js** / **Deno**

WASM VM allows to run arbitrary applications, including the CPython interpreter, but with some limitations.

For more details stay for Antonio Cuni's talk at 11:55 in this room

WEBASSEMBLY

Pyodide overview

CPython
(tier 3 WASM support)

+

Python / Javascript
Foreign function interface

WebAssembly +
Javascript stdlib

180+ packages

+

micropip
Pure python wheels
from PyPi

Wheels distributed via  JSDelivr CDN.

PYODIDE
WA

Building Python packages for the web

Pure Python packages can be used directly but require a wheel (`*-any-none.whl`)

Python packages with C (or Rust) extensions need to be cross-compiled for the emscripten/wasm platform (`*-cp311-cp311-emscripten_3_1_32_wasm32.whl`):

- By adding the package to the Pyodide distribution (writing a meta.yaml, similar to conda)
- Or build wheel with pyodide build (uses pypa/build internally)
  - WIP support in cibuildwheel *(thanks to Hood Chatham and Henry Schreiner)*
  - Not yet possible to upload to PyPI, a PEP is necessary

Packages might need workarounds for unsupported functionality in the browser.

There are is also a conda-forge oriented initiative (**emscripten-forge**).

Foreign function interface (JS ↔ Python)

Using Javascript from Python

A Javascript object in global scope can be imported into Python

```
from js import setTimeout

setTimeout(f, 100)
```

Using Python from Javascript

A Python object in global scope can be accessed from Javascript

```
let sum = pyodide.globals.get("sum");

sum([1, 3, 4]); // 8
```

- Automatic conversion of simple native types (float, str, int, ...,)
- Other types are proxied

For more details: pyodide.org/en/stable/usage/type-conversions.html

Ecosystem

A growing ecosystem of projects that use Pyodide:

- **PyScript**: a framework to create rich Python applications in the browser using HTML

- **JupyterLite**: WASM powered Jupyter running in the browser.
    - Experimental support in sphinx-gallery for an interactive gallery of examples (alongside Binder)

- **elilambnz/react-py**: Effortlessly run Python code in your React apps

- WASM versions of dashboard apps:
  stlite (Streamlit), **webdash** (Plotly Dash), **Voici** (Voila / Jupyter)

Optimizing download size and load time

Python apps in the browser are large

Download size was never an optimisation criterion in the Python ecosystem

Lots of code, C extensions produce large .so, particularly with Cython.

Inclusions of extra files in the main package (e.g. `import numpy.tests`)

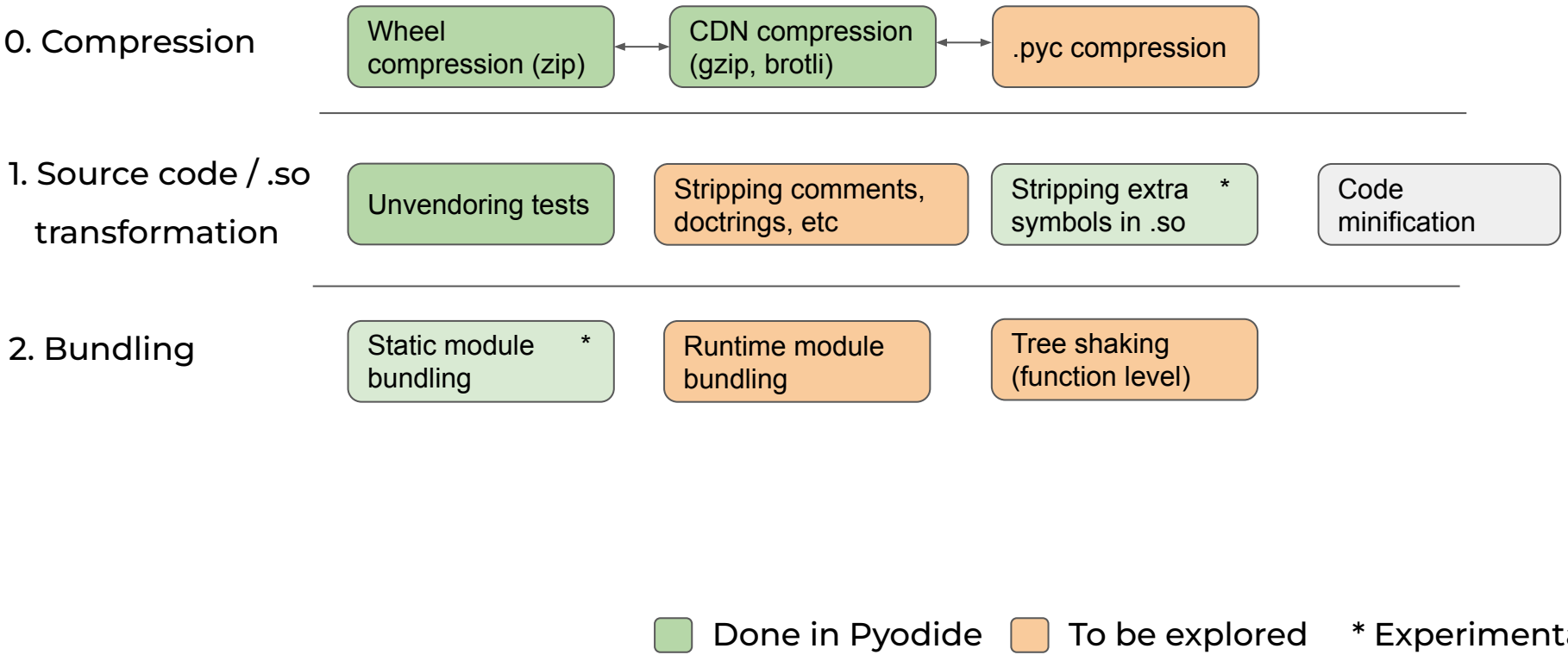
Otherwise see MicroPython but no package ecosystem

Example of loading pandas

200	GET	distutils.tar	x-tar	961.10 KB	960 KB
200	GET	favicon.ico	x-icon	667 B	766 B
200	GET	jquery	js	32.36 KB	87.40 KB
200	GET	jquery.terminal.min.css	css	5.30 KB	22.83 KB
200	GET	jquery.terminal.min.js	js	54.38 KB	162.60 KB
200	GET	numpy-1.22.3-cp310-cp310-emscripten_wasm32.whl	octet-...	3.53 MB	3.53 MB
200	GET	packages.json	json	5.86 KB	27.39 KB
200	GET	pandas-1.4.2-cp310-cp310-emscripten_wasm32.whl	octet-...	4.97 MB	4.97 MB
200	GET	pyodide.asm.data	wasm	3.21 MB	5.14 MB
200	GET	pyodide.asm.js	js	315.25 KB	1.91 MB
200	GET	pyodide.asm.wasm	wasm	3.04 MB	9.05 MB
200	GET	pyodide.js	js	14.75 KB	44.92 KB
200	GET	pyodide_py.tar	x-tar	101.13 KB	100 KB
200	GET	pyparsing-3.0.7-py3-none-any.whl	octet-...	96.89 KB	95.75 KB
200	GET	python_dateutil-2.8.2-py2.py3-none-any.whl	octet-...	243.04 KB	241.90 KB
200	GET	pytz-2022.1-py2.py3-none-any.whl	octet-...	492.86 KB	491.72 KB
200	GET	setuptools-62.0.0-py3-none-any.whl	octet-...	773.12 KB	771.98 KB
200	GET	six-1.16.0-py2.py3-none-any.whl	octet-...	11.93 KB	10.79 KB

20 requests | 27.56 MB / 17.79 MB transferred | Finish: 14.27 s | DOMContentLoaded: 18

Can we optimize Python packages for the web?



Better compression of Python wheels

Wheels (PEP 427) are .zip files

- DEFLATE compression: standard but not state of the art

Content Delivery Network (CDN) compression

- CDN ↔ browser, transparent to end user
- **Brotli** compression
- Great for source code (.py) and .wasm files
- For Python bytecode (.pyc) → **an additional domain specific compressor could help.**

Compression ratio (x times)	stdlib (.py)	stdlib (.pyc)
Gzip	4.3 x	3.5 x
Brotli	6.1 x	5 x

Avoid double zip & CDN compression.

Reducing code size via AST transforms

Package source code in Python can be rewritten to be more compact.

`Example on the stdlib:`

- Removing comments, group imports (`-27%`)
- Normalize parentheses, double/single quotes
- Removing docstrings (`-16%`)
- Renaming locals (or even globals) to be shorter

Loss of usability

Less Pythonic

Relatively robust, aside from the last option.

Package load & import time in the browser

In addition to the download time (*network dependent*),

1. **.py**: are parsed and compiled by the Python interpreter before running them
2. **.pyc**: are run directly, but don't include code snippets in error tracebacks
3. **.wasm**: need to be compiled by the browser to specialize for the target architecture (10-60 MB/s).

Pyodide distributes .py and .pyc wheels separately

Balance between these points depend on the package.

- **Example**: for scientific computing packages, 3. can be the bottleneck.

Import time outside of WASM also matters:

- **Example**: `import pandas` takes 0.5 s on Linux → several seconds in the browser

Module inclusion with static detection

Static code analysis can be used to determine the dependency graph e.g. with `modulefinder`, or `modulegraph`:

1. Find imports in the application code
2. Compute the modules dependency graph
3. Only include the needed modules

Challenges

- Modules detected whether they are actually used not not, including optional functionality (e.g. `pandas.DataFrame.to_parquet` → imports `pyarrow`)
- Python is a dynamic language, imports can be done dynamically
- Imports in C Python API. Also loading non Python `.so`

WASM not the only application, also in application bundlers: `pyinstaller`, `py2app`, ...

Module inclusion with runtime detection

Observation: in the browser we are distributing Python, but also the “OS” (Emscripten) which we can modify.

Possible to identify opened files for an application at runtime by intercepting system calls.

```
Detected 8 dependencies with a total size of 10.54 MB  (uncompressed: 40.99 MB)
In total 425 files and 54 dynamic libraries were accessed.

                                    Packing..
```

No	Package	All files	.so libs	Size (MB)	Reduction
1	distutils.tar	101 → 0	0 → 0	0.26 → 0.00	100.0 %
2	numpy-1.22.3-cp310-cp310-emsc…	418 → 94	19 → 13	3.63 → 2.49	31.4 %
3	pandas-1.4.2-cp310-cp310-emsc…	469 → 283	42 → 41	5.11 → 4.50	12.0 %
4	pyparsing-3.0.7-py3-none-any.…	17 → 0	0 → 0	0.10 → 0.00	100.0 %
5	python_dateutil-2.8.2-py2.py3…	25 → 15	0 → 0	0.24 → 0.22	9.4 %
6	pytz-2022.1-py2.py3-none-any.…	612 → 5	0 → 0	0.43 → 0.02	96.1 %
7	setuptools-62.0.0-py3-none-an…	213 → 0	0 → 0	0.76 → 0.00	100.0 %
8	six-1.16.0-py2.py3-none-any.w…	6 → 1	0 → 0	0.01 → 0.01	18.5 %

```
Wrote pyodide-package-bundle.zip with 7.36 MB (30.2% reduction)
```

Experimental implementation in `pyodide-pack`

Lazy imports

Bundling approaches are still limited by top level import code style (PEP 8).

Example for `pathlib.py`

```
import fnmatch
import functools
import io
import ntpath
import os
import posixpath
import re
import sys
import warnings
from _collections_abc import Sequence
from errno import ENOENT, ENOTDIR, EBADF, ELOOP
from operator import attrgetter
from stat import S_ISDIR, S_ISLNK, S_ISREG, S_ISSOCK, S_ISBLK, S_ISCHR, S_ISFIFO
from urllib.parse import quote_from_bytes as urlquote_from_bytes
```

Lazy imports (PEP 690) might be a partial solution for runtime detection, but PEP was rejected.

Lazy imports with a remote file-system?

In the browser packages are installed and imported for each page load.

Less strict boundary: `await pyodide.runPythonAsync('import numpy')` installs and imports

What if we loaded files as needed from a remote file system?

Will be possible do to sync once **JS Promise integration** is supported (thanks Hood Chatham) and enabled in browsers.

Likely not practical due to network latency

Example: `import pandas` *accesses 450 files → 9 sec (with 200ms latency & 10 parallel downloads)*

Hybrid approaches might be with exploring.

Conclusion

- Pyodide was started 5 years ago by Michael Droettboom at Mozilla. Now Python with WASM is a relatively robust technology with a growing ecosystem.
- Allows use case for Python that were not previously possible
- Reducing the size of Python applications for the web is challenging
  - tools are being developed
  - but community effort might be need to make Python ecosystem web friendlier

Many other improvement topics: pyodide.org/en/stable/project/roadmap.html

New contributors are very welcome!

Many low hanging fruit in the Python for WASM ecosystem.

PYODIDE
WA

Acknowledgement

Pyodide project

Hood Chatham

Michael Droettboom

Gyeongjae Choi

Joe Marshal

Henry Schreiner

Pyodide commiters and users who engaged in discussions on the issue tracker.

Pyodide sponsors 

Community

Emscripten, CPython, JSDelivr CDN

Discussions at WASM Summit @ EuroPython

JupyterLite, pyscript, Basthon, Irydium, Iodide maintainers

Python package maintainers for reviewing patches to improve Pyodide compatibility

PYODIDE WA

Thank you!

github.com/pyodide/pyodide

Also join us for the Pyodide sprint @ EuroPython, 22 - 23 July, 2023

@RomanYurchak @pyodide