
Pyodide

Release 0.23.1

unknown

Apr 14, 2023

CONTENTS

1	What is Pyodide?	3
2	Try Pyodide	5
3	Table of contents	7
4	Communication	189
	Python Module Index	191
	Index	193

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

WHAT IS PYODIDE?

Pyodide is a port of CPython to WebAssembly/[Emscripten](#).

Pyodide 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.

TRY PYODIDE

Try Pyodide in a [REPL](#) directly in your browser (no installation needed).

TABLE OF CONTENTS

3.1 Using Pyodide

3.1.1 Getting started

Try it online

Try Pyodide in a [REPL](#) directly in your browser (no installation needed).

Setup

To include Pyodide in your project you can use the following CDN URL:

```
https://cdn.jsdelivr.net/pyodide/v0.23.1/full/pyodide.js
```

You can also download a release from [GitHub releases](#) or build Pyodide yourself. See *Downloading and deploying Pyodide* for more details.

The `pyodide.js` file defines a single async function called `loadPyodide()` which sets up the Python environment and returns *the Pyodide top level namespace*.

```
async function main() {
  let pyodide = await loadPyodide();
  // Pyodide is now ready to use...
  console.log(pyodide.runPython(`
    import sys
    sys.version
  `));
};
main();
```

Running Python code

Python code is run using the `pyodide.runPython()` function. It takes as input a string of Python code. If the code ends in an expression, it returns the result of the expression, translated to JavaScript objects (see *Type translations*). For example the following code will return the version string as a JavaScript string:

```
pyodide.runPython(`
    import sys
    sys.version
`);
```

After importing Pyodide, only packages from the standard library are available. See *Loading packages* for information about loading additional packages.

Complete example

Create and save a test `index.html` page with the following contents:

```
<!doctype html>
<html>
  <head>
    <script src="https://cdn.jsdelivr.net/pyodide/v0.23.1/full/pyodide.js"></script>
  </head>
  <body>
    Pyodide test page <br>
    Open your browser console to see Pyodide output
    <script type="text/javascript">
      async function main(){
        let pyodide = await loadPyodide();
        console.log(pyodide.runPython(`
          import sys
          sys.version
        `));
        pyodide.runPython("print(1 + 2)");
      }
      main();
    </script>
  </body>
</html>
```

Alternative Example

```
<!doctype html>
<html>
  <head>
    <script src="https://cdn.jsdelivr.net/pyodide/v0.23.1/full/pyodide.js"></script>
  </head>

  <body>
    <p>
      You can execute any Python code. Just enter something in the box below and
```

(continues on next page)

(continued from previous page)

```

    click the button.
  </p>
  <input id="code" value="sum([1, 2, 3, 4, 5])" />
  <button onclick="evaluatePython()">Run</button>
  <br />
  <br />
  <div>Output:</div>
  <textarea id="output" style="width: 100%;" rows="6" disabled></textarea>

  <script>
    const output = document.getElementById("output");
    const code = document.getElementById("code");

    function addToOutput(s) {
      output.value += ">>>" + code.value + "\n" + s + "\n";
    }

    output.value = "Initializing...\n";
    // init Pyodide
    async function main() {
      let pyodide = await loadPyodide();
      output.value += "Ready!\n";
      return pyodide;
    }
    let pyodideReadyPromise = main();

    async function evaluatePython() {
      let pyodide = await pyodideReadyPromise;
      try {
        let output = pyodide.runPython(code.value);
        addToOutput(output);
      } catch (err) {
        addToOutput(err);
      }
    }
  </script>
</body>
</html>

```

Accessing Python scope from JavaScript

All functions and variables defined in the Python global scope are accessible via the `pyodide.globals` object.

For example, if you run the code `x = numpy.ones([3,3])` in Python global scope, you can access the global variable `x` from JavaScript in your browser's developer console with `pyodide.globals.get("x")`. The same goes for functions and imports. See [Type translations](#) for more details.

You can try it yourself in the browser console. Go to the [Pyodide REPL URL](#) and type the following into the browser console:

```

await pyodide.loadPackage("numpy");
pyodide.runPython(`

```

(continues on next page)

(continued from previous page)

```
import numpy
x=numpy.ones((3, 4))
`);
pyodide.globals.get('x').toJs();
// >>> [ Float64Array(4), Float64Array(4), Float64Array(4) ]
```

You can assign new values to Python global variables or create new ones from Javascript.

```
// re-assign a new value to an existing variable
pyodide.globals.set("x", 'x will be now string');

// add the js "alert" function to the Python global scope
// this will show a browser alert if called from Python
pyodide.globals.set("alert", alert);

// add a "square" function to Python global scope
pyodide.globals.set("square", x => x*x);

// Test the new "square" Python function
pyodide.runPython("square(3)");
```

Accessing JavaScript scope from Python

The JavaScript scope can be accessed from Python using the `js` module (see *Importing JavaScript objects into Python*). We can use it to access global variables and functions from Python. For instance, we can directly manipulate the DOM:

```
import js

div = js.document.createElement("div")
div.innerHTML = "<h1>This element was created from Python</h1>"
js.document.body.prepend(div)
```

3.1.2 Downloading and deploying Pyodide

Downloading Pyodide

CDN

Pyodide is available from the JsDelivr CDN

channel	indexURL	Comments	REPL
Latest release	https://cdn.jsdelivr.net/pyodide/v0.23.1/full/	Recommended, cached by the browser	link
Dev (main branch)	https://cdn.jsdelivr.net/pyodide/dev/full/	Re-deployed for each commit on main, no browser caching, should only be used for testing	link

For a given version, several build variants are also available,

- `<version>/full/`: the default full build
- `<version>/debug/`: build with unminified `pyodide.asm.js` useful for debugging

GitHub releases

You can also download Pyodide packages from [GitHub releases](#). The full distribution including all vendored packages is available as `pyodide-0.23.1.tar.bz2`. The full distribution is quite large (200+ megabytes). The minimal set of files needed to start Pyodide is included as `pyodide-core-0.23.1.tar.bz2`. It is intended for use with node which will automatically install missing packages from the cdn – it is the same set of files that are installed if you use `npm install pyodide`. It may also be convenient for other purposes.

You will need to serve these files yourself.

Serving Pyodide packages

Serving locally

With Python 3.7.5+ you can serve Pyodide files locally with `http.server`:

```
python -m http.server
```

from the Pyodide distribution folder. Navigate to <http://localhost:8000/console.html> and the Pyodide repl should load.

Remote deployments

Any service that hosts static files and that correctly sets the WASM MIME type and CORS headers will work. For instance, you can use GitHub Pages or similar services.

For additional suggestions for optimizing the size and load time for Pyodide, see the [Emscripten documentation](#) about deployments.

3.1.3 Using Pyodide

Pyodide may be used in a web browser or a backend JavaScript environment.

Web browsers

To use Pyodide in a web page you need to load `pyodide.js` and initialize Pyodide with `loadPyodide()`.

```
<!doctype html>
<html>
  <head>
    <script src="https://cdn.jsdelivr.net/pyodide/v0.23.1/full/pyodide.js"></script>
  </head>
  <body>
    <script type="text/javascript">
      async function main(){
        let pyodide = await loadPyodide();
        console.log(pyodide.runPython("1 + 2"));
      }
      main();
    </script>
  </body>
</html>
```

See the [Getting started](#) for a walk-through tutorial as well as [Loading packages](#) and [Type translations](#) for a more in depth discussion about existing capabilities.

You can also use the [Pyodide NPM package](#) to integrate Pyodide into your application.

Note: To avoid confusion, note that:

- `cdn.jsdelivr.net/pyodide/` distributes Python packages built with Pyodide as well as `pyodide.js`
 - `cdn.jsdelivr.net/npm/pyodide@0.19.0/` is a mirror of the Pyodide NPM package, which includes none of the WASM files
-

Supported browsers

Pyodide works in any modern web browser with WebAssembly support.

Tier 1 browsers are tested as part of the test suite with continuous integration,

Browser	Minimal supported version	Release date
Firefox	70.0	22 October 2019
Chrome	71.0	4 December 2018

Chrome 89 and 90 have bugs in the webassembly compiler which makes using Pyodide with them unstable. Known problems occur in numpy and have been observed occasionally in other packages. See [#1384](#).

Note: Latest browser versions generally provide more reliable WebAssembly support and will run Pyodide faster, so their use is recommended.

Tier 2 browsers are known to work, but they are not systematically tested in Pyodide,

Browser	Minimal supported version	Release date
Safari	14.0	15 September 2020
Edge	80	26 February 2020

Other browsers with WebAssembly support might also work however they are not officially supported.

Web Workers

By default, WebAssembly runs in the main browser thread, and it can make UI non-responsive for long-running computations.

To avoid this situation, one solution is to run *Pyodide in a WebWorker*.

It's also possible to run *Pyodide in a Service Worker*.

If you're not sure whether you need web workers or service workers, here's an [overview and comparison of the two](#).

Node.js

Note: The following instructions have been tested with Node.js 18.5.0. To use Pyodide with older versions of Node, you might need to use additional command line arguments, see below.

It is now possible to install the [Pyodide npm package](#) in Node.js. To follow these instructions you need at least Pyodide 0.21.0. You can explicitly ask npm to use the alpha version:

```
$ npm install "pyodide@>=0.21.0-alpha.2"
```

Once installed, you can run the following simple script:

```
// hello_python.js
const { loadPyodide } = require("pyodide");

async function hello_python() {
  let pyodide = await loadPyodide();
  return pyodide.runPythonAsync("1+1");
}

hello_python().then((result) => {
  console.log("Python says that 1+1 =", result);
});
```

```
$ node hello_python.js
Loading distutils
Loaded distutils
Python says that 1+1= 2
```

Or you can use the REPL. To start the Node.js REPL with support for top level await, use `node --experimental-repl-await`:

```
$ node --experimental-repl-await
Welcome to Node.js v18.5.0.
Type ".help" for more information.
> const { loadPyodide } = require("pyodide");
undefined
> let pyodide = await loadPyodide();
Loading distutils
Loaded distutils
undefined
> await pyodide.runPythonAsync("1+1");
2
```

Node.js versions <0.17

- Node.js versions 14.x and 16.x: to use certain features of Pyodide you need to manually install `node-fetch`, e.g. by doing `npm install node-fetch`.
- Node.js v14.x: you need to pass the option `--experimental-wasm-bigint` when starting Node. Note that this flag is not documented by `node --help` and moreover, if you pass `--experimental-wasm-bigint` to node >14 it is an error:

```
$ node -v
v14.20.0

$ node --experimental-wasm-bigint hello_python.js
warning: no blob constructor, cannot create blobs with mimetypes
warning: no BlobBuilder
Loading distutils
Loaded distutils
Python says that 1+1= 2
```

If you wish to suppress the blob constructor warnings which appear in node <18, the following code works in node >= v14.18 (do this before calling `loadPyodide`):

```
globalThis.Blob = require("node:buffer").Blob;
```

However, in versions of node < v16.17, this will print an `ExperimentalWarning`. If you also don't want the `ExperimentalWarning`, you can use `cross-blob` to polyfill `Blob`.

Using Pyodide in a web worker

This document describes how to use Pyodide to execute Python scripts asynchronously in a web worker.

Setup

Setup your project to serve `webworker.js`. You should also serve `pyodide.js`, and all its associated `.asm.js`, `.json`, and `.wasm` files as well, though this is not strictly required if `pyodide.js` is pointing to a site serving current versions of these files. The simplest way to serve the required files is to use a CDN, such as `https://cdn.jsdelivr.net/pyodide`. This is the solution presented here.

Update the `webworker.js` sample so that it has as valid URL for `pyodide.js`, and sets `indexURL` to the location of the supporting files.

In your application code create a web worker `new Worker(...)`, and attach listeners to it using its `.onerror` and `.onmessage` methods (listeners).

Communication from the worker to the main thread is done via the `Worker.postMessage()` method (and vice versa).

Detailed example

In this example process we will have three parties involved:

- The **web worker** is responsible for running scripts in its own separate thread.
- The **worker API** exposes a consumer-to-provider communication interface.
- The **consumers** want to run some scripts outside the main thread, so they don't block the main thread.

Consumers

Our goal is to run some Python code in another thread, this other thread will not have access to the main thread objects. Therefore, we will need an API that takes as input not only the Python script we want to run, but also the context on which it relies (some JavaScript variables that we would normally get access to if we were running the Python script in the main thread). Let's first describe what API we would like to have.

Here is an example of consumer that will exchange with the web worker, via the worker interface/API `py-worker.js`. It runs the following Python script using the provided context and a function called `asyncRun()`.

```
import { asyncRun } from './py-worker';

const script = `
import statistics
from js import A_rank
statistics.stdev(A_rank)
`;

const context = {
  A_rank: [0.8, 0.4, 1.2, 3.7, 2.6, 5.8],
};

async function main() {
  try {
    const { results, error } = await asyncRun(script, context);
    if (results) {
      console.log("pyodideWorker return results: ", results);
    } else if (error) {
      console.log("pyodideWorker error: ", error);
    }
  } catch (e) {
    console.log(
      `Error in pyodideWorker at ${e.filename}, Line: ${e.lineno}, ${e.message}`,
    );
  }
}

main();
```

Before writing the API, let's first have a look at how the worker operates. How does our web worker run the script using a given context.

Web worker

Let's start with the definition. A [worker](#) is:

A worker is an object created using a constructor (e.g. `Worker()`) that runs a named JavaScript file — this file contains the code that will run in the worker thread; workers run in another global context that is different from the current window. This context is represented by either a `DedicatedWorkerGlobalScope` object (in the case of dedicated workers - workers that are utilized by a single script), or a `SharedWorkerGlobalScope` (in the case of shared workers - workers that are shared between multiple scripts).

In our case we will use a single worker to execute Python code without interfering with client side rendering (which is done by the main JavaScript thread). The worker does two things:

1. Listen on new messages from the main thread
2. Respond back once it finished executing the Python script

These are the required tasks it should fulfill, but it can do other things. For example, to always load packages `numpy` and `pytz`, you would insert the line `await pyodide.loadPackage(['numpy', 'pytz'])`; as shown below:

```
// webworker.js

// Setup your project to serve `py-worker.js`. You should also serve
// `pyodide.js`, and all its associated `.asm.js`, `.json`,
// and `.wasm` files as well:
importScripts("https://cdn.jsdelivr.net/pyodide/v0.23.1/full/pyodide.js");

async function loadPyodideAndPackages() {
  self.pyodide = await loadPyodide();
  await self.pyodide.loadPackage(["numpy", "pytz"]);
}
let pyodideReadyPromise = loadPyodideAndPackages();

self.onmessage = async (event) => {
  // make sure loading is done
  await pyodideReadyPromise;
  // Don't bother yet with this line, suppose our API is built in such a way:
  const { id, python, ...context } = event.data;
  // The worker copies the context in its own "memory" (an object mapping name to values)
  for (const key of Object.keys(context)) {
    self[key] = context[key];
  }
  // Now is the easy part, the one that is similar to working in the main thread:
  try {
    await self.pyodide.loadPackagesFromImports(python);
    let results = await self.pyodide.runPythonAsync(python);
    self.postMessage({ results, id });
  } catch (error) {
    self.postMessage({ error: error.message, id });
  }
};
```

The worker API

Now that we established what the two sides need and how they operate, let's connect them using this simple API (`py-worker.js`). This part is optional and only a design choice, you could achieve similar results by exchanging message directly between your main thread and the webworker. You would just need to call `.postMessages()` with the right arguments as this API does.

```
const pyodideWorker = new Worker("./dist/webworker.js");

const callbacks = {};

pyodideWorker.onmessage = (event) => {
  const { id, ...data } = event.data;
  const onSuccess = callbacks[id];
  delete callbacks[id];
  onSuccess(data);
};

const asyncRun = (() => {
  let id = 0; // identify a Promise
  return (script, context) => {
    // the id could be generated more carefully
    id = (id + 1) % Number.MAX_SAFE_INTEGER;
    return new Promise((onSuccess) => {
      callbacks[id] = onSuccess;
      pyodideWorker.postMessage({
        ...context,
        python: script,
        id,
      });
    });
  };
})();

export { asyncRun };
```

Caveats

Using a web worker is advantageous because the Python code is run in a separate thread from your main UI, and hence does not impact your application's responsiveness. There are some limitations, however. At present, Pyodide does not support sharing the Python interpreter and packages between multiple web workers or with your main thread. Since web workers are each in their own virtual machine, you also cannot share globals between a web worker and your main thread. Finally, although the web worker is separate from your main thread, the web worker is itself single threaded, so only one Python script will execute at a time.

Loading custom Python code

Pyodide provides a simple API `pyodide.runPython()` to run Python code. However, when your Python code grows bigger, putting hundreds of lines inside `runPython` is not scalable.

For larger projects, the best way to run Python code with Pyodide is:

1. create a Python package
2. load your Python package into the Pyodide (Emscripten) virtual file system
3. import the package with `let mypkg = pyodide.pyimport("mypkgname")`
4. call into your package with `mypkg.some_api(some_args)`.

Using wheels

The best way of serving custom Python code is making it a package in the wheel (.whl) format. If the package is built as a wheel file, you can use `micropip.install()` to install the package. See [Loading packages](#) for more information.

Packages with C extensions

If your Python code contains C extensions, it needs to be built in a specialized way (See [Creating a Pyodide package](#)).

Loading then importing Python code

It is also possible to download and import Python code from an external source. We recommend that you serve all files in an archive, instead of individually downloading each Python script.

From Python

```
// Downloading an archive
await pyodide.runPythonAsync(`
    from pyodide.http import pyfetch
    response = await pyfetch("https://.../your_package.tar.gz") # .zip, .whl, ...
    await response.unpack_archive() # by default, unpacks to the current dir
`)
pkg = pyodide.pyimport("your_package");
pkg.do_something();
```

```
// Downloading a single file
await pyodide.runPythonAsync(`
    from pyodide.http import pyfetch
    response = await pyfetch("https://.../script.py")
    with open("script.py", "wb") as f:
        f.write(await response.bytes())
`)
pkg = pyodide.pyimport("script");
pkg.do_something();
```

What is pyfetch?

Pyodide provides `pyfetch()`, which is a convenient wrapper of JavaScript `fetch`. See *How can I load external files in Pyodide?* for more information.

From JavaScript

```
let response = await fetch("https://.../your_package.tar.gz"); // .zip, .whl, ...
let buffer = await response.arraybuffer();
await pyodide.unpackArchive(buffer); // by default, unpacks to the current dir
pyodide.pyimport("your_package");
```

Warning on unpacking a wheel package

Since a wheel package is actually a zip archive, you can use `pyodide.unpackArchive()` to unpack a wheel package, instead of using `micropip.install()`.

However, `micropip` does dependency resolution when installing packages, while `pyodide.unpackArchive()` simply unpacks the archive. So you must be aware of that each dependencies of a package need to be installed manually before unpacking a wheel.

Future plans: we are planning to support a method for a static dependency resolution (See: [pyodide#2045](#)).

Running external code directly

If you want to run a single Python script from an external source in a simplest way, you can:

```
pyodide.runPython(await (await fetch("https://some_url/.../code.py")).text());
```

Dealing with the file system

Pyodide includes a file system provided by Emscripten. In JavaScript, the Pyodide file system can be accessed through `pyodide.FS` which re-exports the [Emscripten File System API](#)

Example: Reading from the file system

```
pyodide.runPython(`
  with open("/hello.txt", "w") as fh:
    fh.write("hello world!")
`);

let file = pyodide.FS.readFile("/hello.txt", { encoding: "utf8" });
console.log(file); // ==> "hello world!"
```

Example: Writing to the file system

```
let data = "hello world!";
pyodide.FS.writeFile("/hello.txt", data, { encoding: "utf8" });
```

(continues on next page)

(continued from previous page)

```
pyodide.runPython(`
  with open("/hello.txt", "r") as fh:
    data = fh.read()
  print(data)
`);
```

Mounting a file system

The default file system used in Pyodide is [MEMFS](#), which is a virtual in-memory file system. The data stored in MEMFS will be lost when the page is reloaded.

If you wish for files to persist, you can mount other file systems. Other file systems provided by Emscripten are IDBFS, NODEFS, PROXYFS, WORKERFS. Note that some filesystems can only be used in specific runtime environments. See [Emscripten File System API](#) for more details. For instance, to store data persistently between page reloads, one could mount a folder with the [IDBFS file system](#)

```
let mountDir = "/mnt";
pyodide.FS.mkdir(mountDir);
pyodide.FS.mount(pyodide.FS.filesystems.IDBFS, { root: "." }, mountDir);
```

If you are using Node.js you can access the native file system by mounting NODEFS.

```
let mountDir = "/mnt";
pyodide.FS.mkdir(mountDir);
pyodide.FS.mount(pyodide.FS.filesystems.NODEFS, { root: "." }, mountDir);
pyodide.runPython("import os; print(os.listdir('/mnt'))");
// ==> The list of files in the Node working directory
```

(Experimental) Using the native file system in the browser

You can access the native file system from the browser using the [File System Access API](#).

This is experimental

The File System Access API is only supported in Chromium based browsers: Chrome and Edge (as of 2022/08/18).

Mounting a directory

Pyodide provides an API `pyodide.mountNativeFS()` which mounts a `FileSystemDirectoryHandle` into the Pyodide Python file system.

```
const dirHandle = await showDirectoryPicker();

if ((await dirHandle.queryPermission({ mode: "readwrite" })) !== "granted") {
  if (
    (await dirHandle.requestPermission({ mode: "readwrite" })) !== "granted"
  ) {
    throw Error("Unable to read and write directory");
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

const nativefs = await pyodide.mountNativeFS("/mount_dir", dirHandle);

pyodide.runPython(`
    import os
    print(os.listdir('/mount_dir'))
`);

```

Synchronizing changes to native file system

Due to browser limitations, the changes in the mounted file system is not synchronized by default. In order to persist any operations to an native file system, you must call

```

// nativefs is the returned from: await pyodide.mountNativeFS('/mount_dir', dirHandle)
pyodide.runPython(`
    with open('/mount_dir/new_file.txt', 'w') as f:
        f.write("hello");
`);

// new_file.txt does not exist in native file system

await nativefs.syncfs();

// new_file.txt will now exist in native file system

```

or

```
pyodide.FS.syncfs(false, callback_func);
```

Using Pyodide in a service worker

This document describes how to use Pyodide to execute Python scripts in a service worker. Compared to typical web workers, service workers are more related to acting as a network proxy, handling background tasks, and things like caching and offline. See [this article](#) for more info.

Detailed example

For our example, we'll be talking about how we can use a service worker to intercept a fetch call for some data and modify the data. We will have two parties involved:

- The **service worker** which will be intercepting fetch calls for JSON, and modifying the data before returning it
- The **consumer** which will be fetching some JSON data

To keep things simple, all we'll do is add a field to a fetched JSON object, but an example of a more interesting use case is transforming fetched tabular data using numpy, and caching the result before returning it.

Please note that service workers will only work on https and localhost, so you will require a server to be running for this example.

Setup

Setup your project to serve the service worker script `sw.js`, and a `XMLHttpRequest` polyfill - one such polyfill that works in service workers is [xhr-shim](#). You should also serve `pyodide.js`, and all its associated `.asm.js`, `.json`, and `.wasm` files as well, though this is not strictly required if `pyodide.js` is pointing to a site serving current versions of these files. The simplest way to serve the required files is to use a CDN, such as <https://cdn.jsdelivr.net/pyodide>.

Update the `sw.js` sample so that it has a valid URL for `pyodide.js`, and sets `indexURL` to the location of the supporting files.

You'll also need to serve `data.json`, a JSON file containing a simple object - a sample is provided below:

```
{
  "name": "Jem"
}
```

Consumer

In our consumer, we want to register our service worker - in the html below, we're registering a classic-type service worker. For convenience, we also provide a button that fetches data and logs it.

```
<!doctype html>
<html>
  <head>
    <script>
      /* UPDATE PATHS TO POINT TO YOUR ASSETS */
      const SERVICE_WORKER_PATH = "/sw.js";
      const JSON_FILE_PATH = "./data.json";
      /* IF USING MODULE-TYPE SERVICE WORKER, REPLACE THESE OPTIONS */
      const REGISTRATION_OPTIONS = {
        scope: "/",
      };

      // modified snippet from
      // https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_
      ↪Service_Workers
      async function registerServiceWorker() {
        if ("serviceWorker" in navigator) {
          try {
            const registration = await navigator.serviceWorker.register(
              SERVICE_WORKER_PATH,
              REGISTRATION_OPTIONS,
            );
            if (registration.installing) {
              console.log("Service worker installing");
            } else if (registration.waiting) {
              console.log("Service worker installed");
            } else if (registration.active) {
              console.log("Service worker active");
            }
          } catch (error) {
            console.error(`Registration failed with ${error}`);
          }
        }
      }
    </script>
  </head>
  <body>
    <button id="register">Register Service Worker</button>
    <button id="fetch">Fetch Data</button>
  </body>
</html>
```

(continues on next page)

(continued from previous page)

```

    }
  }
}

async function fetchAndLogData() {
  try {
    console.log(await (await fetch(JSON_FILE_PATH)).json());
  } catch (e) {
    console.error("Failed to fetch", e);
  }
}

registerServiceWorker();
</script>
</head>

<body>
  <button onclick="fetchAndLogData()">Fetch and log data</button>
</body>
</html>

```

Service worker

To set up Pyodide in a service worker, you'll need to do the following:

1. Polyfill XMLHttpRequest because it isn't available in service workers' global scopes.
2. Import Pyodide
3. We don't need it for this example, but if you're planning on calling `loadPyodide` after [installation](#) of the service worker, import `pyodide.asm.js` too.

After all the required scripts are imported, we call `loadPyodide` to set up Pyodide, then create a Python function called `modify_data`. This function add a `count` property to an object, where `count` is equal to the number of times `modify_data` is called. We will access this function via a handle assigned to the Javascript variable `modifyData`. We also set up a fetch event handler that intercepts requests for json files so that any JSON object that is fetched is modified using `modifyData`.

```

/* sw.js */
/* MODIFY IMPORT PATHS TO POINT TO YOUR SCRIPTS, REPLACE IF USING MODULE-TYPE WORKER */
// We're using the npm package xhr-shim, which assigns self.XMLHttpRequestShim
importScripts("./node_modules/xhr-shim/src/index.js");
self.XMLHttpRequest = self.XMLHttpRequestShim;
importScripts("./pyodide.js");
// importScripts("./pyodide.asm.js"); // if loading Pyodide after installation phase, you
→ 'll need to import this too

let modifyData;
let pyodide;
loadPyodide({}).then((_pyodide) => {
  pyodide = _pyodide;
  let namespace = pyodide.globals.get("dict")();

```

(continues on next page)

(continued from previous page)

```

pyodide.runPython(
  \
  import json

  counter = 0
  def modify_data(data):
    global counter
    counter += 1
    dict = data.to_py()
    dict['count'] = counter
    return dict
  \
  { globals: namespace },
);

// assign the modify_data function from the Python context to a Javascript variable
modifyData = namespace.get("modify_data");
namespace.destroy();
});

self.addEventListener("fetch", (event) => {
  if (event.request.url.endsWith("json")) {
    if (!modifyData) {
      // For this example, throw so it's clear that the worker isn't ready to modify
      ↪responses
      // This is because we don't want to return a response that isn't modified yet
      // If your service worker would return the same response as a server (eg. it's just
      ↪performing calculations closer to home)
      // then you may want to let the event through without doing anything
      event.respondWith(
        Promise.reject("Python code isn't set up yet, try again in a bit"),
      );
    } else {
      event.respondWith(
        // We aren't using the async await syntax because event.respondWith needs to
        ↪respond synchronously
        // it can't be executing after an awaited promise within the fetch event handler,
        ↪otherwise you'll get this
        // Uncaught (in promise) DOMException: Failed to execute 'respondWith' on
        ↪'FetchEvent': The event has already been responded to
        fetch(event.request)
          .then((v) => v.json())
          .then((originalData) => {
            let proxy = modifyData(originalData);
            let pyproxies = [];

            // Because toJs gives us a Map, we transform it to a plain Javascript object
            ↪before changing it to JSON
            let result = JSON.stringify(
              Object.fromEntries(
                proxy.toJs({
                  pyproxies,

```

(continues on next page)

(continued from previous page)

```

        }),
    ),
};
// Craft the new JSON response
return new Response(result, {
    headers: { "Content-Type": "application/json" },
});
}),
);
}
});

// Code below is for easy iteration during development, you may want to remove or modify
↳ in a prod environment:

// Immediately become the active service worker once installed, so we don't have a stale
↳ service worker intercepting requests
// You can remove this code and achieve a similar thing by enabling "Update on Reload"
↳ in devtools, if supported:
// https://web.dev/service-worker-lifecycle/#update-on-reload
self.addEventListener("install", function () {
    self.skipWaiting();
});

// With this, we won't need to reload the page before the service worker can intercept
↳ fetch requests
// https://developer.mozilla.org/en-US/docs/Web/API/Clients/claim#examples
self.addEventListener("activate", function (event) {
    event.waitUntil(self.clients.claim());
});

```

Using module-type service workers

While classic-type service workers have better cross-browser compatibility at the moment, module-type service workers make it easier to include external libraries in your service workers via ES module imports. There are environments where we can safely assume ES module support in service workers, such as Chromium-based browser extensions' background scripts. With the adjustments outlined below, you should be able to use our example with a module-type service worker.

Setup

Serve `pyodide.mjs` instead of `pyodide.js`, the rest of the setup remains the same.

Consumers

We need to use different registration options on the consumer side. Replace this section of the script:

```
/* IF USING MODULE-TYPE SERVICE WORKER, REPLACE THESE OPTIONS */
const REGISTRATION_OPTIONS = {
  scope: "/",
};
```

With the following:

```
const REGISTRATION_OPTIONS = {
  scope: "/",
  // Note that specifying the type option can cause errors if the browser doesn't support
  ↪ module-type service workers
  type: "module",
};
```

Service worker

On the service worker side, we need to change the way we import scripts. Replace the `importScripts` calls shown below:

```
/* sw.js */
/* MODIFY IMPORT PATHS TO POINT TO YOUR SCRIPTS, REPLACE IF USING MODULE-TYPE WORKER */
// We're using the npm package xhr-shim, which assigns self.XMLHttpRequestShim
importScripts("./node_modules/xhr-shim/src/index.js");
self.XMLHttpRequest = self.XMLHttpRequestShim;
importScripts("./pyodide.js");
// importScripts("./pyodide.asm.js"); // if loading Pyodide after installation phase, you
↪ 'll need to import this too
```

With the following imports:

```
/* sw.js */
/* MODIFY IMPORT PATHS TO POINT TO YOUR SCRIPTS */
// We're using the npm package xhr-shim, which assigns self.XMLHttpRequestShim
import "./node_modules/xhr-shim/src/index.js";
self.XMLHttpRequest = self.XMLHttpRequestShim;
import "./pyodide.asm.js"; // IMPORTANT: This is compulsory in a module-type service
↪ worker, which cannot use importScripts
import { loadPyodide } from "./pyodide.mjs"; // Note the .mjs extension
```

3.1.4 Loading packages

Only the Python standard library is available after importing Pyodide. To use other packages, you'll need to load them using either:

- `micropip.install()` (Python) for pure Python packages with wheels as well as Pyodide packages (including Emscripten/wasm32 binary wheels). It can install packages from PyPI, the JsDelivr CDN or from other URLs.
- `pyodide.loadPackage()` (Javascript) for packages built with Pyodide. This is a function with less overhead but also more limited functionality. `micropip` uses this function to load Pyodide packages. In most cases you should be using `micropip`.

In some cases, and in particular in the REPL, packages are installed implicitly from imports. The Pyodide REPL uses `pyodide.loadPackagesFromImports()` to automatically download all packages that the code snippet imports. This is useful since users might import unexpected packages in REPL. At present, `loadPackagesFromImports()` will not download packages from PyPI, it will only download packages included in the Pyodide distribution. See [Packages built in Pyodide](#) to check the full list of packages included in Pyodide.

How to chose between `micropip.install` and `pyodide.loadPackage`?

While `micropip.install()` is written in Python and `pyodide.loadPackage()` in Javascript this has no incidence on when to use each of these functions. Indeed, you can easily switch languages using the [Type translations](#) with,

- from Javascript,

```
let micropip = pyodide.pyimport(package_name);
```

- from Python,

```
import pyodide_js
await pyodide_js.loadPackage('package_name')
```

Instead, the general advice is to use `micropip.install()` for everything except in the following cases where `pyodide.loadPackage()` might be more appropriate,

- to load `micropip` itself,
- when you are optimizing for size, do not want to install the `micropip` package, and do not need to install packages from PyPI with dependency resolution.

Micropip

Installing packages

Pyodide supports installing following types of packages with `micropip`,

- pure Python wheels from PyPI with `micropip`.
- pure Python and binary wasm32/emscripten wheels (also informally known as “Pyodide packages” or “packages built by Pyodide”) from the JsDelivr CDN and custom URLs. `micropip.install()` is an async Python function which returns a coroutine, so it need to be called with an `await` clause to run.

```
await pyodide.loadPackage("micropip");
const micropip = pyodide.pyimport("micropip");
await micropip.install('snowballstemmer');
pyodide.runPython(`
```

(continues on next page)

(continued from previous page)

```
import snowballstemmer
stemmer = snowballstemmer.stemmer('english')
print(stemmer.stemWords('go goes going gone'.split()))
`);
```

Micropip implements file integrity validation by checking the hash of the downloaded wheel against pre-recorded hash digests from the PyPI JSON API.

Installing wheels from arbitrary URLs

Pure Python wheels can also be installed from any URL with `micropip`,

```
import micropip
micropip.install(
    'https://example.com/files/snowballstemmer-2.0.0-py2.py3-none-any.whl'
)
```

Micropip decides whether a file is a URL based on whether it ends in “.whl” or not. The wheel name in the URL must follow [PEP 427 naming convention](#), which will be the case if the wheels is made using standard Python tools (`pip wheel`, `setup.py bdist_wheel`). Micropip will also install the dependencies of the wheel. If dependency resolution is not desired, you may pass `deps=False`.

Cross-Origin Resource Sharing (CORS)

If the file is on a remote server, the server must set [Cross-Origin Resource Sharing \(CORS\) headers](#) to allow access. If the server doesn’t set CORS headers, you can use a CORS proxy. Note that using third-party CORS proxies has security implications, particularly since we are not able to check the file integrity, unlike with installs from PyPI. See [this stack overflow answer](#) for more information about CORS.

Example

```
<html>
<head>
  <meta charset="utf-8" />
</head>
<body>
  <script type="text/javascript" src="https://cdn.jsdelivr.net/pyodide/v0.23.1/full/
  ↪pyodide.js"></script>
  <script type="text/javascript">
    async function main() {
      let pyodide = await loadPyodide();
      await pyodide.loadPackage("micropip");
      const micropip = pyodide.pyimport("micropip");
      await micropip.install("snowballstemmer");
      await pyodide.runPython(`
import snowballstemmer
stemmer = snowballstemmer.stemmer('english')
print(stemmer.stemWords('go goes going gone'.split()))
`);
```

(continues on next page)

(continued from previous page)

```

    }
    main();
  </script>
</body>
</html>

```

Loading packages with `pyodide.loadPackage()`

Packages included in the official Pyodide repository can be loaded using `loadPackage()`:

```
await pyodide.loadPackage("numpy");
```

It is also possible to load packages from custom URLs:

```
await pyodide.loadPackage(
  "https://foo/bar/numpy-1.22.3-cp310-cp310-emscripten_3_1_13_wasm32.whl",
);
```

The file name must be a valid wheel name.

When you request a package from the official repository, all of the package's dependencies are also loaded. There is no dependency resolution when loading packages from custom URLs. If you want dependency resolution for custom URLs, use `micropip`.

In general, loading a package twice is not permitted. However, one can override a dependency by loading a custom URL with the same package name before loading the dependent package.

Multiple packages can also be loaded at the same time by passing a list to `loadPackage()`.

```
await pyodide.loadPackage(["cyclor", "pytz"]);
```

`loadPackage()` returns a `Promise` which resolves when all the packages are finished loading:

```
let pyodide;
async function main() {
  pyodide = await loadPyodide();
  await pyodide.loadPackage("matplotlib");
  // matplotlib is now available
}
main();
```

Packages built in Pyodide

This is the list of Python packages included with the current version of Pyodide. These packages can be loaded with `pyodide.loadPackage()` or `micropip.install()`. See [Loading packages](#) for information about loading packages. Pure Python packages with wheels on PyPI can be loaded directly from PyPI with `micropip.install()`.

Name	Version
asciitree	0.3.3
astropy	5.2.2
atomicwrites	1.4.1

continues on next page

Table 1 – continued from previous page

Name	Version
attrs	22.2.0
autograd	1.5
bcrypt	4.0.1
beautifulsoup4	4.12.0
biopython	1.81
bitarray	2.7.3
bleach	6.0.0
bokeh	3.1.0
boost-histogram	1.3.2
brotli	1.0.9
cbor-diag	1.0.0
certifi	2022.12.7
cffib	1.15.1
cffib_example	0.1
cftime	1.6.2
click	8.1.3
cligj	0.7.2
cloudpickle	2.2.1
cmyt	1.1.3
colorspacious	1.1.2
coverage	7.2.2
cramjam	2.6.2
cryptography	39.0.2
cssselect	1.2.0
cycler	0.11.0
cytoolz	0.12.1
decorator	5.1.1
demes	0.2.2
distlib	0.3.6
docutils	0.19
exceptiongroup	1.1.1
fastparquet	2023.2.0
fiona	1.8.22
fonttools	4.39.3
freesasa	2.2.0.post3
fsspec	2023.3.0
future	0.18.3
galpy	1.8.3
gensim	4.3.1
geopandas	0.12.2
gmpy2	2.1.5
gsw	3.6.16
h5py	3.7.0
html5lib	1.1
idna	3.4
imageio	2.27.0
iniconfig	2.0.0
jedi	0.18.2
Jinja2	3.1.2

continues on next page

Table 1 – continued from previous page

Name	Version
joblib	1.2.0
jsonschema	4.17.3
kiwisolver	1.4.4
lazy-object-proxy	1.9.0
lightgbm	3.3.5
logbook	1.5.3
lxml	4.9.2
MarkupSafe	2.1.2
matplotlib	3.5.2
matplotlib-pyodide	0.1.1
micropip	0.3.0
mne	1.3.1
more-itertools	9.1.0
mpmath	1.3.0
msgpack	1.0.5
msprime	1.2.0
multidict	6.0.4
munch	2.5.0
mypy	1.1.1
networkx	3.0
newick	1.9.0
nlopt	2.7.0
nlTK	3.8.1
nose	1.3.7
numcodecs	0.11.0
numpy	1.24.2
opencv-python	4.7.0.72
optlang	1.6.1
packaging	23.0
pandas	1.5.3
parso	0.8.3
patsy	0.5.3
Pillow	9.1.1
pillow_heif	0.8.0
pkgconfig	1.5.5
pluggy	1.0.0
py	1.11.0
pyb2d	0.7.2
pyclipper	1.3.0.post4
pycparser	2.21
pycryptodome	3.17
pydantic	1.10.7
pyerfa	2.0.0.3
Pygments	2.14.0
pyheif	0.7.1
pyinstrument	4.4.0
pynacl	1.5.0
pyodide-http	0.2.0
pyparsing	3.0.9

continues on next page

Table 1 – continued from previous page

Name	Version
pyproj	3.4.1
pyrsistent	0.19.3
pytest	7.2.2
pytest-benchmark	4.0.0
python-dateutil	2.8.2
python-magic	0.4.27
python-sat	0.1.7.dev26
python_solvespace	3.0.8
pytz	2023.3
pywavelets	1.4.1
pyxel	1.9.10
pyyaml	6.0
rebound	3.19.8
reboundx	3.7.1
regex	2023.3.23
retrying	1.3.4
RobotRaconteur	0.15.1
ruamel.yaml	0.17.21
scikit-image	0.19.3
scikit-learn	1.2.2
scipy	1.9.3
setuptools	67.6.1
shapely	1.8.2
six	1.16.0
smart_open	6.3.0
soupsieve	2.4
sparseqr	1.2
sqlalchemy	2.0.7
statsmodels	0.13.5
svgwrite	1.4.3
swiglpk	5.0.8
sympy	1.11.1
tblib	1.7.0
termcolor	2.2.0
threadpoolctl	3.1.0
tomli	2.0.1
tomli-w	1.0.0
toolz	0.12.0
tqdm	4.65.0
traits	6.4.1
tskit	0.5.4
typing-extensions	4.5.0
uncertainties	3.1.7
unyt	2.9.5
webencodings	0.5.1
wordcloud	1.8.2.2
wrapt	1.15.0
xarray	2023.3.0
xgboost	1.6.1

continues on next page

Table 1 – continued from previous page

Name	Version
xlrd	2.0.1
yaml	1.8.2
yt	4.1.4
zarr	2.13.3

Using SDL-based packages in Pyodide

This is experimental

SDL support in Pyodide is experimental. Pyodide relies on undocumented behavior of Emscripten and SDL, so it may break or change in the future.

In addition, this feature requires to enable an opt-in flag, `pyodide._api._skip_unwind_fatal_error = true`; which can lead to stack unwinding issues (see [Known issues](#)).

Pyodide provides a way to use SDL-based packages in the browser, This document explains how to use SDL-based packages in Pyodide.

Setting canvas

Before using SDL-based packages, you need to set the canvas to draw on.

The canvas object must be a `HTMLCanvasElement` object, with the `id` attribute set to "canvas". For example, you can set a canvas like this:

```
let sdl2Canvas = document.createElement("canvas");
sdl2Canvas.id = "canvas";
pyodide.canvas.setCanvas2D(sdl2Canvas);
```

See also: [pyodide.canvas](#)

Known issues

There is a known issue that with,

```
pyodide._api._skip_unwind_fatal_error = true;
```

Python call stacks are not being unwound after calling `emscripten_set_main_loop()`.

see: [pyodide#3697](#)

3.1.5 Type translations

In order to communicate between Python and JavaScript, we “translate” objects between the two languages. Depending on the type of the object we either translate the object by implicitly converting it or by proxying it. By “converting” an object we mean producing a new object in the target language which is the equivalent of the object from the source language, for example converting a Python string to the equivalent a JavaScript string. By “proxying” an object we mean producing a special object in the target language that forwards requests to the source language. When we proxy a JavaScript object into Python, the result is a *JsProxy* object. When we proxy a Python object into JavaScript, the result is a *PyProxy* object. A proxied object can be explicitly converted using the explicit conversion methods *JsProxy.to_py()* and *PyProxy.toJs()*.

Python to JavaScript translations occur:

- when returning the final expression from a *pyodide.runPython()* call,
- when *importing Python objects into JavaScript*
- when passing arguments to a JavaScript function called from Python,
- when returning the results of a Python function called from JavaScript,
- when accessing an attribute of a *PyProxy*

JavaScript to Python translations occur:

- when *importing from the js module*
- when passing arguments to a Python function called from JavaScript
- when returning the result of a JavaScript function called from Python
- when accessing an attribute of a *JsProxy*

Memory Leaks and Python to JavaScript translations

Any time a Python to JavaScript translation occurs, it may create a *PyProxy*. To avoid memory leaks, you must store the *PyProxy* and *destroy()* it when you are done with it. See *Calling Python objects from JavaScript* for more info.

Round trip conversions

Translating an object from Python to JavaScript and then back to Python is guaranteed to give an object that is equal to the original object. Furthermore, if the object is proxied into JavaScript, then translation back unwraps the proxy, and the result of the round trip conversion is the original object (in the sense that they live at the same memory address).

Translating an object from JavaScript to Python and then back to JavaScript gives an object that is `===` to the original object. Furthermore, if the object is proxied into Python, then translation back unwraps the proxy, and the result of the round trip conversion is the original object (in the sense that they live at the same memory address). There are a few exceptions:

1. NaN is converted to NaN after a round trip but `NaN !== NaN`,
2. `null` is converted to `undefined` after a round trip, and
3. a `BigInt` will be converted to a `Number` after a round trip unless its absolute value is greater than `Number.MAX_SAFE_INTEGER` (i.e., 2^{53}).

Implicit conversions

We implicitly convert immutable types but not mutable types. This ensures that mutable Python objects can be modified from JavaScript and vice-versa. Python has immutable types such as `tuple` and `bytes` that have no equivalent in JavaScript. In order to ensure that round trip translations yield an object of the same type as the original object, we proxy `tuple` and `bytes` objects.

Python to JavaScript

The following immutable types are implicitly converted from Python to JavaScript:

Python	JavaScript
<code>int</code>	<code>Number</code> or <code>BigInt</code> *
<code>float</code>	<code>Number</code>
<code>str</code>	<code>String</code>
<code>bool</code>	<code>Boolean</code>
<code>None</code>	<code>undefined</code>

* An `int` is converted to a `Number` if the absolute value is less than or equal to `Number.MAX_SAFE_INTEGER` otherwise it is converted to a `BigInt`. (If the browser does not support `BigInt` then a `Number` will be used instead. In this case, conversion of large integers from Python to JavaScript is lossy.)

JavaScript to Python

The following immutable types are implicitly converted from JavaScript to Python:

JavaScript	Python
<code>Number</code>	<code>int</code> or <code>float</code> as appropriate*
<code>BigInt</code>	<code>int</code>
<code>String</code>	<code>str</code>
<code>Boolean</code>	<code>bool</code>
<code>undefined</code>	<code>None</code>
<code>null</code>	<code>None</code>

* A `Number` is converted to an `int` if the absolute value is less than or equal to `Number.MAX_SAFE_INTEGER` and its fractional part is zero. Otherwise, it is converted to a `float`.

Proxying

Any of the types not listed above are shared between languages using proxies that allow methods and some operations to be called on the object from the other language.

Proxying from JavaScript into Python

When most JavaScript objects are translated into Python a *JsProxy* is returned. The following operations are currently supported on a *JsProxy*:

Python	JavaScript
<code>str(proxy)</code>	<code>x.toString()</code>
<code>proxy.foo</code>	<code>x.foo</code>
<code>proxy.foo = bar</code>	<code>x.foo = bar</code>
<code>del proxy.foo</code>	<code>delete x.foo</code>
<code>hasattr(proxy, "foo")</code>	<code>"foo" in x</code>
<code>proxy(...)</code>	<code>x(...)</code>
<code>proxy.foo(...)</code>	<code>x.foo(...)</code>
<code>proxy.new(...)</code>	<code>new X(...)</code>
<code>len(proxy)</code>	<code>x.length</code> or <code>x.size</code>
<code>foo in proxy</code>	<code>x.has(foo)</code> or <code>x.includes(foo)</code>
<code>proxy[foo]</code>	<code>x.get(foo)</code>
<code>proxy[foo] = bar</code>	<code>x.set(foo, bar)</code>
<code>del proxy[foo]</code>	<code>x.delete(foo)</code>
<code>proxy1 == proxy2</code>	<code>x === y</code>
<code>proxy.typeof</code>	<code>typeof x</code>
<code>iter(proxy)</code>	<code>x[Symbol.iterator]()</code>
<code>next(proxy)</code>	<code>x.next()</code>
<code>await proxy</code>	<code>await x</code>

Note that each of these operations is only supported if the proxied JavaScript object supports the corresponding operation. See [the JsProxy API docs](#) for the rest of the methods supported on *JsProxy*. Some other code snippets:

```
for v in proxy:
    # do something
```

is equivalent to:

```
for (let v of x) {
    // do something
}
```

The `dir()` method has been overloaded to return all keys on the prototype chain of `x`, so `dir(x)` roughly translates to:

```
function dir(x) {
    let result = [];
    do {
        result.push(...Object.getOwnPropertyNames(x));
    } while ((x = Object.getPrototypeOf(x)));
    return result;
}
```

As a special case, JavaScript `Array`, `HTMLCollection`, and `NodeList` are container types, but instead of using `array.get(7)` to get the 7th element, JavaScript uses `array[7]`. For these cases, we translate:

Python	JavaScript
<code>proxy[idx]</code>	<code>array[idx]</code>
<code>proxy[idx] = val</code>	<code>array[idx] = val</code>
<code>idx in proxy</code>	<code>idx in array</code>
<code>del proxy[idx]</code>	<code>array.splice(idx)</code>

If you need to access the fields in a JavaScript object, you must use `obj.field_name` or if the name of the field is not a valid Python identifier, `getattr(obj, "field name")`. If you want to access the fields of the object like `obj["field name"]` you can use `as_object_map()`:

```
from pyodide.code import run_js

obj = run_js(
    """
    ({
      a: 7,
      b: 9,
      $c: 11
    })
    """
)
obj_map = obj.as_object_map()
assert obj_map["$c"] == 11
```

Another special case comes from the fact that Python reserved words cannot be used as attributes. For instance, `Array.from()` and `Promise.finally()` cannot be directly accessed because they are Python `SyntaxErrors`. Instead we access these attributes with `Array.from_` and `Promise.finally_`. Similarly, to access from Python, `o.from_` you have to use `o.from__` with two underscores (since a single underscore is used for `o.from`). This is reflected in the `dir` of a `JsProxy`:

```
from pyodide.code import run_js
o = run_js("({finally: 1, return: 2, from: 3, from_: 4})")
assert set(dir(o)) == {"finally_", "return_", "from_", "from__"}
```

Proxying from Python into JavaScript

When most Python objects are translated to JavaScript a *PyProxy* is produced.

Fewer operations can be overloaded in JavaScript than in Python, so some operations are more cumbersome on a *PyProxy* than on a *JsProxy*. The following operations are supported:

JavaScript	Python
<code>foo in proxy</code>	<code>hasattr(x, 'foo')</code>
<code>proxy.foo</code>	<code>x.foo</code>
<code>proxy.foo = bar</code>	<code>x.foo = bar</code>
<code>delete proxy.foo</code>	<code>del x.foo</code>
<code>Object.getOwnPropertyNames(proxy)</code>	<code>dir(x)</code>
<code>proxy(...)</code>	<code>x(...)</code>
<code>proxy.foo(...)</code>	<code>x.foo(...)</code>
<code>proxy.length</code>	<code>len(x)</code>
<code>proxy.has(foo)</code>	<code>foo in x</code>
<code>proxy.get(foo)</code>	<code>x[foo]</code>
<code>proxy.set(foo, bar)</code>	<code>x[foo] = bar</code>
<code>proxy.delete(foo)</code>	<code>del x[foo]</code>
<code>proxy.type</code>	<code>type(x)</code>
<code>proxy[Symbol.iterator]()</code>	<code>iter(x)</code>
<code>proxy.next()</code>	<code>next(x)</code>
<code>await proxy</code>	<code>await x</code>

Memory Leaks and PyProxy

Make sure to destroy PyProxies when you are done with them to avoid memory leaks.

```
let foo = pyodide.globals.get('foo');
foo();
foo.destroy();
foo(); // throws Error: Object has already been destroyed
```

Explicit Conversion of Proxies

Python to JavaScript

Explicit conversion of a *PyProxy* into a native JavaScript object is done with the `toJs()` method. You can also perform such a conversion in Python using `to_js()` which behaves in much the same way. By default, the `toJs()` method does a recursive “deep” conversion, to do a shallow conversion use `proxy.toJs({depth : 1})`. In addition to *the normal type conversion*, the `toJs()` method performs the following explicit conversions:

Python	JavaScript
<code>list, tuple</code>	<code>Array</code>
<code>dict</code>	<code>Map</code>
<code>set</code>	<code>Set</code>
<code>a buffer*</code>	<code>TypedArray</code>

* Examples of buffers include `bytes` objects and numpy `Array` objects.

If you need to convert `dict` instead to `Object`, you can pass `Object.fromEntries()` as the `dict_converter` argument: `proxy.toJs({dict_converter : Object.fromEntries})`.

In JavaScript, `Map` and `Set` keys are compared using object identity unless the key is an immutable type (meaning a `String`, a `Number`, a `BigInt`, a `Boolean`, `undefined`, or `null`). On the other hand, in Python, `dict` and `set` keys

are compared using deep equality. If a key is encountered in a `dict` or `set` that would have different semantics in JavaScript than in Python, then a `ConversionError` will be thrown.

See *Using Python Buffer objects from JavaScript* for the behavior of `toJs()` on buffers.

Memory Leaks and `toJs`

The `toJs()` method can create many proxies at arbitrary depth. It is your responsibility to manually `destroy()` these proxies if you wish to avoid memory leaks. The `pyproxies` argument to `toJs()` is designed to help with this:

```
let pyproxies = [];
proxy.toJs({pyproxies});
// Do stuff
// pyproxies contains the list of proxies created by `toJs`. We can destroy them
// when we are done with them
for(let px of pyproxies){
    px.destroy();
}
proxy.destroy();
```

As an alternative, if you wish to assert that the object should be fully converted and no proxies should be created, you can use `proxy.toJs({create_proxies : false})`. If a proxy would be created, a `ConversionError` is raised instead.

JavaScript to Python

Explicit conversion of a `JsProxy` into a native Python object is done with the `JsProxy.to_py()` method. By default, the `to_py()` method does a recursive “deep” conversion, to do a shallow conversion use `proxy.to_py(depth=1)`. The `to_py()` method performs the following explicit conversions:

JavaScript	Python
<code>Array</code>	<code>list</code>
<code>Object*</code>	<code>dict</code>
<code>Map</code>	<code>dict</code>
<code>Set</code>	<code>set</code>

* `to_py()` will only convert an `Object` into a dictionary if its constructor is `Object`, otherwise the object will be left alone. Example:

```
class Test {};
window.x = { "a" : 7, "b" : 2};
window.y = { "a" : 7, "b" : 2};
Object.setPrototypeOf(y, Test.prototype);
pyodide.runPython(`
    from js import x, y
    # x is converted to a dictionary
    assert x.to_py() == { "a" : 7, "b" : 2}
    # y is not a "Plain Old JavaScript Object", it's an instance of type Test so it's not
    ↪ converted
    assert y.to_py() == y
`);
```

In JavaScript, `Map` and `Set` keys are compared using object identity unless the key is an immutable type (meaning a `String`, a `Number`, a `BigInt`, a `Boolean`, `undefined`, or `null`). On the other hand, in Python, `dict` and `set` keys are compared using deep equality. If a key is encountered in a `Map` or `Set` that would have different semantics in Python than in JavaScript, then a `ConversionError` will be thrown. Also, in JavaScript, `true !== 1` and `false !== 0`, but in Python, `True == 1` and `False == 0`. This has the result that a JavaScript map can use `true` and `1` as distinct keys but a Python `dict` cannot. If the JavaScript map contains both `true` and `1` a `ConversionError` will be thrown.

Functions

Calling Python objects from JavaScript

If a Python object is callable, the proxy will be callable too. The arguments will be translated from JavaScript to Python as appropriate, and the return value will be translated from JavaScript back to Python. If the return value is a `PyProxy`, you must explicitly destroy it or else it will be leaked.

An example:

```
let test = pyodide.runPython(`
    def test(x):
        return [n*n for n in x]
    test
`);
let result_py = test([1,2,3,4]);
// result_py is a PyProxy of a list.
let result_js = result_py.toJs();
// result_js is the array [1, 4, 9, 16]
result_py.destroy();
```

If a function is intended to be used from JavaScript, you can use `to_js()` on the return value. This prevents the return value from leaking without requiring the JavaScript code to explicitly destroy it. This is particularly important for callbacks.

```
let test = pyodide.runPython(`
    from pyodide.ffi import to_js
    def test(x):
        return to_js([n*n for n in x])
    test
`);
let result = test([1,2,3,4]);
// result is the array [1, 4, 9, 16], nothing needs to be destroyed.
```

If you need to use a key word argument, use `callKwargs()`. The last argument should be a JavaScript object with the key value arguments.

```
let test = pyodide.runPython(`
    from pyodide.ffi import to_js
    def test(x, *, offset):
        return to_js([n*n + offset for n in x])
    to_js(test)
`);
let result = test.callKwargs([1,2,3,4], { offset : 7});
// result is the array [8, 12, 16, 23]
```

Calling JavaScript functions from Python

What happens when calling a JavaScript function from Python is a bit more complicated than calling a Python function from JavaScript. If there are any keyword arguments, they are combined into a JavaScript object and used as the final argument. Thus, if you call:

```
f(a=2, b=3)
```

then the JavaScript function receives one argument which is a JavaScript object `{a : 2, b : 3}`.

When a JavaScript function is called, if the return value not a [Promise](#), a [Generator](#), or an [AsyncGenerator](#), any arguments that are [PyProxies](#) that were created in the process of argument conversion are also destroyed. If the result is a [PyProxy](#) it is also destroyed.

As a result of this, if a [PyProxy](#) is persisted to be used later, then it must either be copied using [copy\(\)](#) in JavaScript, or it must be created with [create_proxy\(\)](#) or [create_once_callable\(\)](#). If it's only going to be called once use [create_once_callable\(\)](#):

```
from pyodide import create_once_callable
from js import setTimeout
def my_callback():
    print("hi")
setTimeout(create_once_callable(my_callback), 1000)
```

If it's going to be called many times use [create_proxy\(\)](#):

```
from pyodide import create_proxy
from js import document
def my_callback():
    print("hi")
proxy = pyodide.create_proxy(my_callback)
document.body.addEventListener("click", proxy)
# ...
# make sure to hold on to proxy
document.body.removeEventListener("click", proxy)
proxy.destroy()
```

When a JavaScript function returns a [Promise](#) (for example, if the function is an [async function](#)), it is assumed that the [Promise](#) is going to do some work that uses the arguments of the function, so it is not safe to destroy them until the [Promise](#) resolves. In this case, the proxied function returns a Python [Future](#) instead of the original [Promise](#). When the [Promise](#) resolves, the result is converted to Python and the converted value is used to resolve the [Future](#). Then if the result is a [PyProxy](#) it is destroyed. Any [PyProxies](#) created in converting the arguments are also destroyed at this point.

Similarly, if the return value is a [Generator](#) or [AsyncGenerator](#), then the arguments (and all values sent to the generator) are kept alive until it is exhausted, or until [close\(\)](#) is called.

Buffers

Using JavaScript Typed Arrays from Python

JavaScript `ArrayBuffer` and `TypedArray` objects are proxied into Python. Python can't directly access arrays if they are outside the WASM heap, so it's impossible to directly use these proxied buffers as Python buffers. You can convert such a proxy to a Python `memoryview` using the `to_py()` api. This makes it easy to correctly convert the array to a Numpy array using `numpy.asarray()`:

```
self.jsarray = new Float32Array([1,2,3, 4, 5, 6]);
pyodide.runPython(`
    from js import jsarray
    array = jsarray.to_py()
    import numpy as np
    numpy_array = np.asarray(array).reshape((2,3))
    print(numpy_array)
`);
```

After manipulating `numpy_array` you can assign the value back to `jsarray` using `assign()`:

```
pyodide.runPython(`
    numpy_array[1,1] = 77
    jsarray.assign(a)
`);
console.log(jsarray); // [1, 2, 3, 4, 77, 6]
```

The `assign()` and `assign_to()` methods can be used to assign a JavaScript buffer from / to a Python buffer which is appropriately sized and contiguous. The assignment methods will only work if the data types match, the total length of the buffers match, and the Python buffer is contiguous.

Using Python Buffer objects from JavaScript

Python objects supporting the `Python Buffer protocol` are proxied into JavaScript. The data inside the buffer can be accessed via the `toJs()` method or the `getBuffer()` method. The `toJs()` API copies the buffer into JavaScript, whereas the `getBuffer()` method allows low level access to the WASM memory backing the buffer. The `getBuffer()` API is more powerful but requires care to use correctly. For simple use cases the `toJs()` API should be preferred.

If the buffer is zero or one-dimensional, then `toJs()` will in most cases convert it to a single `TypedArray`. However, in the case that the format of the buffer is 's', we will convert the buffer to a string and if the format is '?' we will convert it to an `Array` of booleans.

If the dimension is greater than one, we will convert it to a nested JavaScript array, with the innermost dimension handled in the same way we would handle a 1d array.

An example of a case where you would not want to use the `toJs()` method is when the buffer is bitmapped image data. If for instance you have a 3d buffer shaped 1920 x 1080 x 4, then `toJs()` will be extremely slow. In this case you could use `getBuffer()`. On the other hand, if you have a 3d buffer shaped 1920 x 4 x 1080, the performance of `toJs()` will most likely be satisfactory. Typically, the innermost dimension won't matter for performance.

The `getBuffer()` method can be used to retrieve a reference to a JavaScript typed array that points to the data backing the Python object, combined with other metadata about the buffer format. The metadata is suitable for use with a JavaScript ndarray library if one is present. For instance, if you load the JavaScript `ndarray` package, you can do:

```

let proxy = pyodide.globals.get("some_numpy_ndarray");
let buffer = proxy.getBuffer();
proxy.destroy();
try {
  if (buffer.readonly) {
    // We can't stop you from changing a readonly buffer, but it can cause undefined_
    ↪ behavior.
    throw new Error("Uh-oh, we were planning to change the buffer");
  }
  let array = new ndarray(
    buffer.data,
    buffer.shape,
    buffer.strides,
    buffer.offset,
  );
  // manipulate array here
  // changes will be reflected in the Python ndarray!
} finally {
  buffer.release(); // Release the memory when we're done
}

```

Errors

All entrypoints and exit points from Python code are wrapped in JavaScript `try` blocks. At the boundary between Python and JavaScript, errors are caught, converted between languages, and rethrown.

JavaScript errors are wrapped in a *JsException*. Python exceptions are converted to a *PythonError*. At present if an exception crosses between Python and JavaScript several times, the resulting error message won't be as useful as one might hope.

In order to reduce memory leaks, the *PythonError* has a formatted traceback, but no reference to the original Python exception. The original exception has references to the stack frame and leaking it will leak all the local variables from that stack frame. The actual Python exception will be stored in `sys.last_value` so if you need access to it (for instance to produce a traceback with certain functions filtered out), use that.

Be careful Proxying Stack Frames

If you make a *PyProxy* of `sys.last_value`, you should be especially careful to *destroy()* it when you are done with it, or you may leak a large amount of memory if you don't.

The easiest way is to only handle the exception in Python:

```

pyodide.runPython(`
def reformat_exception():
    from traceback import format_exception
    # Format a modified exception here
    # this just prints it normally but you could for instance filter some frames
    return "".join(
        format_exception(sys.last_type, sys.last_value, sys.last_traceback)
    )
`);
let reformat_exception = pyodide.globals.get("reformat_exception");

```

(continues on next page)

(continued from previous page)

```
try {
  pyodide.runPython(some_code);
} catch(e){
  // replace error message
  e.message = reformat_exception();
  throw e;
}
```

Importing Objects

It is possible to access objects in one language from the global scope in the other language. It is also possible to create custom namespaces and access objects on the custom namespaces.

Importing Python objects into JavaScript

A Python global variable in the `__main__` global scope can be imported into JavaScript using the `pyodide.globals.get()` method. Given the name of the Python global variable, it returns the value of the variable translated to JavaScript.

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

As always, if the result is a `PyProxy` and you care about not leaking the Python object, you must destroy it when you are done. It's also possible to set values in the Python global scope with `pyodide.globals.set()` or remove them with `pyodide.globals.delete()`:

```
pyodide.globals.set("x", 2);
pyodide.runPython("print(x)"); // Prints 2
```

If you execute code with a custom globals dictionary, you can use a similar approach:

```
let my_py_namespace = pyodide.globals.get("dict")();
pyodide.runPython("x=2", my_py_namespace);
let x = my_py_namespace.get("x");
```

To access a Python module from JavaScript, use `pyimport()`:

```
let sys = pyodide.pyimport("sys");
```

Importing JavaScript objects into Python

JavaScript objects in the `globalThis` global scope can be imported into Python using the `js` module.

When importing a name from the `js` module, the `js` module looks up JavaScript attributes of the `globalThis` scope and translates the JavaScript objects into Python.

```
import js
js.document.title = 'New window title'
from js.document.location import reload as reload_page
reload_page()
```

You can also assign to JavaScript global variables in this way:


```
pyodide.runPython("js.x = 2");
console.log(window.x); // 2
```

You can create your own custom JavaScript modules using `pyodide.registerJsModule()` and they will behave like the `js` module except with a custom scope:

```
let my_js_namespace = { x : 3 };
pyodide.registerJsModule("my_js_namespace", my_js_namespace);
pyodide.runPython(`
    from my_js_namespace import x
    print(x) # 3
    my_js_namespace.y = 7
`);
console.log(my_js_namespace.y); // 7
```

If the JavaScript object's name is a reserved Python keyword, the `setattr()` function can be used to access the object by name within the `js` module::

```
lambda = (x) => {return x + 1};
//from js import lambda' will cause a Syntax Error, since 'lambda' is a Python reserved_
↳keyword. Instead:
pyodide.runPython(`
    import js
    js_lambda = getattr(js, 'lambda')
    print(js_lambda(1))
`);
```

If a JavaScript object has a property that is a reserved Python keyword, the `setattr()` and `getattr()` function can be used to access that property by name:

```
people = {global: "lots and lots"};
//Trying to access 'people.global' will raise a Syntax Error, since 'global' is a Python_
↳reserved keyword. Instead:
pyodide.runPython(`
    from js import people
    setattr(people, 'global', 'even more')
    print(getattr(people, 'global'))
`);
```

3.1.6 Pyodide Python compatibility

Python Standard library

Most of the Python standard library is functional, except for the modules listed in the sections below. A large part of the CPython test suite passes except for tests skipped in `src/tests/python_tests.yaml` or via `patches`.

Optional modules

The following stdlib modules are unvendored by default, in order to reduce initial download size of Python distribution. You can load all unvendored stdlib modules when initializing Pyodide with, `loadPyodide({ fullStdLib : true })`. However this has a significant impact on the download size. Instead, it is better to load individual modules as needed using `pyodide.loadPackage()` or `micropip.install()`.

- `distutils`
- `ssl`
- `lzma`
- `sqlite3`
- `test`: it is an exception to the above, since it is not loaded even if `fullStdLib` is set to `true`.

Modules with limited functionality

- `hashlib`: Hash algorithms that are depending on OpenSSL are not available by default. See Python [hashlib documentation](#) for list of algorithms that are dependent on OpenSSL. If you need those algorithms, you need to call `pyodide.loadPackage('hashlib')` or `micropip.install('hashlib')` **before** importing `hashlib`.
- `decimal`: The `decimal` module has C (`_decimal`) and Python (`_pydecimal`) implementations with the same functionality. The Python implementation is not available by default. If you need a Python implementation of `decimal`, you need to call `pyodide.loadPackage('pydecimal')` or `micropip.install('pydecimal')`, then explicitly import `_pydecimal`.
- `pydoc`: Help messages for Python builtins are not available by default in order to reduce the initial download size. You need to call `pyodide.loadPackage('pydoc_data')` or `micropip.install('pydoc_data')` to enable them.
- `webbrowser`: The original `webbrowser` module is not available. Instead, Pyodide includes some method stubs based on browser APIs: `webbrowser.open()`, `webbrowser.open_new()`, `webbrowser.open_new_tab()`.

Removed modules

The following modules are removed from the standard library to reduce download size and since they currently wouldn't work in the WebAssembly VM,

- `curses`
- `dbm`
- `ensurepip`
- `idlelib`
- `lib2to3`
- `tkinter`

- turtle.py
- turtledemo
- venv
- pwd

Included but not working modules

The following modules can be imported, but are not functional due to the limitations of the WebAssembly VM:

- multiprocessing
- threading
- sockets

as well as any functionality that requires these.

3.1.7 Interrupting execution

The native Python interrupt system is based on preemptive multitasking but Web Assembly has no support for preemptive multitasking. Because of this, interrupting execution in Pyodide must be achieved via a different mechanism which takes some effort to set up.

Setting up interrupts

In order to use interrupts you must be using Pyodide in a webworker. You also will need to use a [SharedArrayBuffer](#), which means that your server must set appropriate security headers. See [the MDN docs](#) for more information.

To use the interrupt system, you should create a [SharedArrayBuffer](#) on either the main thread or the worker thread and share it with the other thread. You should use `pyodide.setInterruptBuffer()` to set the interrupt buffer on the Pyodide thread. When you want to indicate an interrupt, write a 2 into the interrupt buffer. When the interrupt signal is processed, Pyodide will set the value of the interrupt buffer back to 0.

By default, when the interrupt fires, a [KeyboardInterrupt](#) is raised. Using the [signal](#) module, it is possible to register a custom Python function to handle `SIGINT`. If you register a custom handler function it will be called instead.

Here is a very basic example. Main thread code:

```
let pyodideWorker = new Worker("pyodideWorker.js");
let interruptBuffer = new Uint8Array(new SharedArrayBuffer(1));
pyodideWorker.postMessage({ cmd: "setInterruptBuffer", interruptBuffer });
function interruptExecution() {
  // 2 stands for SIGINT.
  interruptBuffer[0] = 2;
}
// imagine that interruptButton is a button we want to trigger an interrupt.
interruptButton.addEventListener("click", interruptExecution);
async function runCode(code) {
  // Clear interruptBuffer in case it was accidentally left set after previous code_
  ↪ completed.
  interruptBuffer[0] = 0;
  pyodideWorker.postMessage({ cmd: "runCode", code });
}
```

Worker code:

```
self.addEventListener("message", (msg) => {
  if (msg.data.cmd === "setInterruptBuffer") {
    pyodide.setInterruptBuffer(msg.data.interruptBuffer);
    return;
  }
  if (msg.data.cmd === "runCode") {
    pyodide.runPython(msg.data.code);
    return;
  }
});
```

Allowing JavaScript code to be interrupted

The interrupt system above allows interruption of Python code and also of C code that allows itself to be interrupted by periodically calling `PyErr_CheckSignals()`. There is also a function `pyodide.checkInterrupt()` that allows JavaScript functions called from Python to check for an interrupt. As a simple example, we can implement an interruptible sleep function using `Atomics.wait()`:

```
let blockingSleepBuffer = new Int32Array(new SharedArrayBuffer(4));
function blockingSleep(t) {
  for (let i = 0; i < t * 20; i++) {
    // This Atomics.wait call blocks the thread until the buffer changes or a 50ms
    ↪ timeout elapses.
    // Since we won't change the value in the buffer, this blocks for 50ms.
    Atomics.wait(blockingSleepBuffer, 0, 0, 50);
    // Periodically check for an interrupt to allow a KeyboardInterrupt.
    pyodide.checkInterrupt();
  }
}
```

3.1.8 Redirecting standard streams

Pyodide has three functions `pyodide.setStdin()`, `pyodide.setStdout()`, and `pyodide.setStderr()` that change the behavior of reading from `stdin` and writing to `stdout` and `stderr` respectively.

`setStdin({stdin?, isatty?, error?})` takes a function which should take zero arguments and return either a string or an `ArrayBufferView` of information read from `stdin`. The `isatty` argument signals whether `isatty(stdin)` should be true or false. If you pass `error: true` then reading from `stdin` will return an error. If `setStdin` is called with no arguments, the default value is restored. In Node the default behavior is to read from `process.stdin` and in the browser it is to throw an error.

`setStdout({batched?, raw?, isatty?})` sets the standard out handler and similarly `setStderr` (same arguments) sets the standard error handler. If a `raw` handler is provided then the handler is called with a `number` for each byte of the output to `stdout`. The handler is expected to deal with this in whatever way it prefers. `isatty` again controls whether `isatty(stdout)` returns true or false.

On the other hand, a `batched` handler is only called with complete lines of text (or when the output is flushed). A `batched` handler cannot have `isatty` set to true because it is impossible to use such a handler to make something behave like a `tty`.

Passing neither `raw` nor `batched` sets the default behavior. In Node the default behavior is to write directly to `process.stdout` and `process.stderr` (in this case `isatty` depends on whether `process.stdout` and `process.stderr` are

ttys). In browser, the default behavior is achieved with `pyodide.setStdout({batched: console.log})` and `pyodide.setStderr({batched: console.warn})`.

The arguments `stdin`, `stdout`, and `stderr` to `loadPyodide` provide a diminished amount of control compared to `setStdin`, `setStdout`, and `setStderr`. They all set `isatty` to `false` and use batched processing for `setStdout` and `setStderr`. In most cases, nothing is written or read to any of these streams while Pyodide is starting, so if you need the added flexibility you can wait until Pyodide is loaded and then use the `pyodide.setStdxxx` apis.

3.1.9 API Reference

JavaScript API

Backward compatibility of the API is not guaranteed at this point.

Globals

Functions:

<code>async loadPyodide(options)</code>	Load the main Pyodide wasm module and initialize it.
---	--

`async globalThis.loadPyodide(options)`

Load the main Pyodide wasm module and initialize it.

Arguments

- **options.args** (`string[]`) – Command line arguments to pass to Python on startup. See [Python command line interface options](#) for more details. Default: `[]`
- **options.fullStdLib** (`boolean`) – Load the full Python standard library. Setting this to `false` excludes unvendored modules from the standard library. Default: `false`
- **options.homedir** (`string`) – The home directory which Pyodide will use inside virtual file system. Default: `"/home/pyodide"`
- **options.indexURL** (`string`) – The URL from which Pyodide will load the main Pyodide runtime and packages. It is recommended that you leave this unchanged, providing an incorrect value can cause broken behavior. Default: The url that Pyodide is loaded from with the file name (`pyodide.js` or `pyodide.mjs`) removed.
- **options.jsglobals** (`object`) – The object that Pyodide will use for the `js` module. Default: `globalThis`
- **options.lockFileURL** (`string`) – The URL from which Pyodide will load the Pyodide `repodata.json` lock file. You can produce custom lock files with `micropip.freeze()`. Default: ``${indexURL}/repodata.json``
- **options.stdLibURL** (`string`) – The URL from which to load the standard library `python_stdlib.zip` file. This URL includes the most of the Python standard library. Some stdlib modules were unvendored, and can be loaded separately with `fullStdLib=true` option or by their package name. Default: ``${indexURL}/python_stdlib.zip``
- **options.stderr** (`((msg: string) => void)`) – Override the standard error output callback.
- **options.stdin** (`() => string`) – Override the standard input callback. Should ask the user for one line of input.
- **options.stdout** (`((msg: string) => void)`) – Override the standard output callback.

Returns

`Promise<PyodideInterface>` – The *pyodide* module.

pyodide**Attributes:**

<code>ERRNO_CODES</code>	A map from posix error names to error codes.
<code>FS</code>	An alias to the Emscripten File System API .
<code>PATH</code>	An alias to the Emscripten Path API .
<code>canvas</code>	This provides APIs to set a canvas for rendering graphics.
<code>ffi</code>	See pyodide.ffi
<code>globals</code>	An alias to the global Python namespace.
<code>loadedPackages</code>	The list of packages that Pyodide has loaded.
<code>pyodide_py</code>	An alias to the Python <i>pyodide</i> package.
<code>version</code>	The Pyodide version.

Functions:

<code>checkInterrupt()</code>	Throws a <code>KeyboardInterrupt</code> error if a <code>KeyboardInterrupt</code> has been requested via the interrupt buffer.
<code>isPyProxy(jsobj)</code>	Is jsobj a PyProxy ?
<code>async loadPackage(names, options)</code>	Load packages from the Pyodide distribution or Python wheels by URL.
<code>async loadPackagesFromImports(code, options)</code>	Inspect a Python code chunk and use pyodide.loadPackage() to install any known packages that the code chunk imports.
<code>async mountNativeFS(path, fileSystemHandle)</code>	Mounts a <code>FileSystemDirectoryHandle</code> into the target directory.
<code>pyimport(mod_name)</code>	Imports a module and returns it.
<code>registerComlink(Comlink)</code>	Tell Pyodide about Comlink.
<code>registerJsModule(name, module)</code>	Registers the JavaScript object module as a JavaScript module named name.
<code>runPython(code, options)</code>	Runs a string of Python code from JavaScript, using eval_code() to evaluate the code.
<code>async runPythonAsync(code, options)</code>	Run a Python code string with top level await using eval_code_async() to evaluate the code.
<code>setInterruptBuffer(interrupt_buffer)</code>	Sets the interrupt buffer to be <code>interrupt_buffer</code> .
<code>setStderr(options)</code>	Sets the standard error handler.
<code>setStdin(options)</code>	Set a stdin handler.
<code>setStdout(options)</code>	Sets the standard out handler.
<code>toPy(obj, options)</code>	Convert a JavaScript object to a Python object as best as possible.
<code>unpackArchive(buffer, format, options)</code>	Unpack an archive into a target directory.
<code>unregisterJsModule(name)</code>	Unregisters a JavaScript module with given name that has been previously registered with pyodide.registerJsModule() or register_js_module() .

Classes:

<i>PyBuffer</i>	An alias for <i>pyodide.ffi.PyBufferView</i> .
<i>PyProxyBuffer</i>	An alias for <i>pyodide.ffi.PyBuffer</i> .
<i>PythonError</i>	An alias for <i>pyodide.ffi.PyBuffer</i> .

pyodide.ERRNO_CODES**type:** {[code: string]: number}

A map from posix error names to error codes.

pyodide.FS**type:** anyAn alias to the [Emscripten File System API](#).

This provides a wide range of POSIX-like file/device operations, including [mount](#) which can be used to extend the in-memory filesystem with features like [persistence](#).

While all the file systems implementations are enabled, only the default MEMFS is guaranteed to work in all run-time settings. The implementations are available as members of `FS.filesystems`: IDBFS, NODEFS, PROXYFS, WORKERFS.

pyodide.PATH**type:** anyAn alias to the [Emscripten Path API](#).

This provides a variety of operations for working with file system paths, such as `dirname`, `normalize`, and `splitPath`.

pyodide.canvas**type:** CanvasInterface

This provides APIs to set a canvas for rendering graphics.

For example, you need to set a canvas if you want to use the SDL library. See [Using SDL-based packages in Pyodide](#) for more information.

pyodide.ffiSee [*pyodide.ffi*](#)**pyodide.globals****type:** [*PyProxy*](#)

An alias to the global Python namespace.

For example, to access a variable called `foo` in the Python global scope, use `pyodide.globals.get("foo")`

pyodide.loadedPackages**type:** {[key: string]: string}

The list of packages that Pyodide has loaded. Use `Object.keys(pyodide.loadedPackages)` to get the list of names of loaded packages, and `pyodide.loadedPackages[package_name]` to access install location for a particular `package_name`.

pyodide.pyodide_py**type:** [*PyProxy*](#)An alias to the Python [*pyodide*](#) package.

You can use this to call functions defined in the Pyodide Python package from JavaScript.

pyodide.version**type:** `string`

The Pyodide version.

The version here is a Python version, following [PEP 440](#). This is different from the version in `package.json` which follows the node package manager version convention.

static pyodide.checkInterrupt()

Throws a `KeyboardInterrupt` error if a `KeyboardInterrupt` has been requested via the interrupt buffer.

This can be used to enable keyboard interrupts during execution of JavaScript code, just as `PyErr_CheckSignals()` is used to enable keyboard interrupts during execution of C code.

static pyodide.isPyProxy(jsobj)

Note: Deprecated: Use `obj instanceof pyodide.ffi.PyProxy` instead.

Is `jsobj` a `PyProxy`?

Arguments

- **jsobj** (`any`) – Object to test.

Returns

`boolean` (typeguard for `PyProxy`) –

async pyodide.loadPackage(names, options)

Load packages from the Pyodide distribution or Python wheels by URL.

This installs packages in the virtual filesystem. Packages needs to be imported from Python before it can be used.

This function can only install packages included in the Pyodide distribution, or Python wheels by URL, without dependency resolution. It is significantly more limited in terms of functionality as compared to `micropip`, however it has less overhead and can be faster.

When installing binary wheels by URLs it is user's responsibility to check that the installed binary wheel is compatible in terms of Python and Emscripten versions. Compatibility is not checked during installation time (unlike with `micropip`). If a wheel for the wrong Python/Emscripten version is installed it would fail at import time.

Arguments

- **names** (`string` | `string[]` | `PyProxy`) – Either a single package name or URL or a list of them. URLs can be absolute or relative. The URLs must correspond to Python wheels: either pure Python wheels, with a file name ending with `none-any.whl` or Emscripten/WASM 32 wheels, with a file name ending with `cp<pyversion>_emscripten_<em_version>_wasm32.whl`. The argument can be a `PyProxy` of a list, in which case the list will be converted to JavaScript and the `PyProxy` will be destroyed.
- **options.checkIntegrity** (`boolean`) – If true, check the integrity of the downloaded packages (default: true)
- **options.errorCallback** ((`message: string`) => `void`) – A callback, called with error/warning messages (optional)
- **options.messageCallback** ((`message: string`) => `void`) – A callback, called with progress messages (optional)

Returns`Promise<void>` –**static** `pyodide.pyimport(mod_name)`

Imports a module and returns it.

WarningThis function has a completely different behavior than the old removed `pyimport` function!`pyimport` is roughly equivalent to:

```
pyodide.runPython(`import ${pkgname}; ${pkgname}`);
```

except that the global namespace will not change.

Example:

```
let sysmodule = pyodide.pyimport("sys");
let recursionLimit = sysmodule.getrecursionlimit();
```

Arguments

- **mod_name** (`string`) – The name of the module to import

Returns`PyProxy` – A `PyProxy` for the imported module**static** `pyodide.registerComlink(Comlink)`Tell Pyodide about `Comlink`. Necessary to enable importing `Comlink` proxies into Python.**Arguments**

- **Comlink** (`any`) –

static `pyodide.registerJsModule(name, module)`Registers the JavaScript object `module` as a JavaScript module named `name`. This module can then be imported from Python using the standard Python import system. If another module by the same name has already been imported, this won't have much effect unless you also delete the imported module from `sys.modules`. This calls `register_js_module()`.**Arguments**

- **name** (`string`) – Name of the JavaScript module to add
- **module** (`object`) – JavaScript object backing the module

static `pyodide.runPython(code, options)`Runs a string of Python code from JavaScript, using `eval_code()` to evaluate the code. If the last statement in the Python code is an expression (and the code doesn't end with a semicolon), the value of the expression is returned.**Arguments**

- **code** (`string`) – Python code to evaluate
- **options.globals** (`PyProxy`) – An optional Python dictionary to use as the globals. Defaults to `pyodide.globals`.

- **options.locals** (*PyProxy*) – An optional Python dictionary to use as the locals. Defaults to the same as globals.

Returns

any – The result of the Python code translated to JavaScript. See the documentation for `eval_code()` for more info.

static `pyodide.setInterruptBuffer(interrupt_buffer)`

Sets the interrupt buffer to be `interrupt_buffer`. This is only useful when Pyodide is used in a webworker. The buffer should be a `SharedArrayBuffer` shared with the main browser thread (or another worker). In that case, signal `signum` may be sent by writing `signum` into the interrupt buffer. If `signum` does not satisfy `0 < signum < 65` it will be silently ignored.

You can disable interrupts by calling `setInterruptBuffer(undefined)`.

If you wish to trigger a `KeyboardInterrupt`, write `SIGINT` (a 2) into the interrupt buffer.

By default `SIGINT` raises a `KeyboardInterrupt` and all other signals are ignored. You can install custom signal handlers with the `signal` module. Even signals that normally have special meaning and can't be overridden like `SIGKILL` and `SIGSEGV` are ignored by default and can be used for any purpose you like.

Arguments

- **interrupt_buffer** (`TypedArray`) –

`pyodide.setStderr(options)`

Sets the standard error handler. A batched handler or a raw handler can be provided (both not both). If neither is provided, we restore the default handler.

Arguments

- **options.isatty** (`boolean`) – Should `isatty(stderr)` return `true` or `false`. Can only be set to `true` if a raw handler is provided (default `false`).
- **options.batched** ((`a: string`) => `void`) – A batched handler is called with a string whenever a newline character is written or `stderr` is flushed. In the former case, the received line will end with a newline, in the latter case it will not. `isatty(stderr)` is set to `false` (when using a batched handler, `stderr` is buffered so it is impossible to make a `tty` with it).
- **options.raw** ((`a: number`) => `void`) – A raw handler is called with the handler is called with a *number* for each byte of the output to `stderr`.

`pyodide.setStdin(options)`

Set a `stdin` handler.

The `stdin` handler is called with zero arguments whenever `stdin` is read and the current input buffer is exhausted. It should return one of:

- `null` or `undefined`: these are interpreted as end of file.
- a number
- a string
- an `ArrayBuffer` or `TypedArray` with `BYTES_PER_ELEMENT` equal to 1.

If a number is returned, it is interpreted as a single character code. The number should be between 0 and 255.

If a string is returned, a new line is appended if one is not present and the resulting string is turned into a `Uint8Array` using `TextEncoder`.

Returning a buffer is more efficient and allows returning partial lines of text.

Arguments

- **options.autoEOF** (*boolean*) – Insert an EOF automatically after each string or buffer? (default `true`).
- **options.error** (*boolean*) – If this is set to `true`, attempts to read from `stdin` will always set an IO error.
- **options.isatty** (*boolean*) – Should `isatty(stdin)` be `true` or `false` (default `false`).
- **options.stdin** (*() => null | undefined | string | ArrayBuffer | Uint8Array | number*) – The `stdin` handler.

`pyodide.setStdout(options)`

Sets the standard out handler. A batched handler or a raw handler can be provided (both not both). If neither is provided, we restore the default handler.

Arguments

- **options.isatty** (*boolean*) – Should `isatty(stdout)` return `true` or `false`. Can only be set to `true` if a raw handler is provided (default `false`).
- **options.batched** (*((a: string) => void)*) – A batched handler is called with a string whenever a newline character is written or `stdout` is flushed. In the former case, the received line will end with a newline, in the latter case it will not.
- **options.raw** (*((a: number) => void)*) – A raw handler is called with the handler is called with a *number* for each byte of the output to `stdout`.

static `pyodide.toPy(obj, options)`

Convert a JavaScript object to a Python object as best as possible.

This is similar to `to_py()` but for use from JavaScript. If the object is immutable or a *PyProxy*, it will be returned unchanged. If the object cannot be converted into Python, it will be returned unchanged.

See *JavaScript to Python* for more information.

Arguments

- **obj** (*any*) – The object to convert.
- **options.depth** (*number*) – Optional argument to limit the depth of the conversion.
- **options.defaultConverter** (*((value: any, converter: (value: any) => any, cacheConversion: (input: any, output: any) => void) => any)*) – Optional argument to convert objects with no default conversion. See the documentation of `to_py()`.

Returns

any – The object converted to Python.

static `pyodide.unpackArchive(buffer, format, options)`

Unpack an archive into a target directory.

Arguments

- **buffer** (*ArrayBuffer | TypedArray*) – The archive as an *ArrayBuffer* or *TypedArray*.
- **format** (*string*) – The format of the archive. Should be one of the formats recognized by `shutil.unpack_archive()`. By default the options are `'bztar'`, `'gztar'`, `'tar'`, `'zip'`, and `'wheel'`. Several synonyms are accepted for each format, e.g., for `'gztar'` any of `'gztar'`, `'.tar.gz'`, `'.tgz'`, `'tar.gz'` or `'tgz'` are considered to be synonyms.
- **options.extractDir** (*string*) – The directory to unpack the archive into. Defaults to the working directory.

static `pyodide.unregisterJsModule(name)`

Unregisters a JavaScript module with given name that has been previously registered with `pyodide.registerJsModule()` or `register_js_module()`. If a JavaScript module with that name does not already exist, will throw an error. Note that if the module has already been imported, this won't have much effect unless you also delete the imported module from `sys.modules`. This calls `unregister_js_module()`.

Arguments

- **name** (`string`) – Name of the JavaScript module to remove

`pyodide.PyBuffer`

Note: Deprecated.

An alias for `pyodide.ffi.PyBufferView`.

`pyodide.PyProxyBuffer`

Note: Deprecated.

An alias for `pyodide.ffi.PyBuffer`.

`pyodide.PythonError`

Note: Deprecated.

An alias for `pyodide.ffi.PyBuffer`.

pyodide.ffi

To import types from `pyodide.ffi` you can use for example

```
import type { PyProxy } from "pyodide/ffi";
```

Classes:

<code>PyAsyncGenerator</code>	A <code>PyProxy</code> whose proxied Python object is an asynchronous generator (i.e., it is an instance of <code>AsyncGenerator</code>)
<code>PyAsyncIterable</code>	A <code>PyProxy</code> whose proxied Python object is asynchronous iterable (i.e., has an <code>__aiter__()</code> method).
<code>PyAsyncIterator</code>	A <code>PyProxy</code> whose proxied Python object is an asynchronous iterator
<code>PyAwaitable</code>	A <code>PyProxy</code> whose proxied Python object is awaitable (i.e., has an <code>__await__()</code> method).
<code>PyBuffer</code>	A <code>PyProxy</code> whose proxied Python object supports the Python Buffer Protocol.
<code>PyBufferView</code>	A class to allow access to Python data buffers from JavaScript.

continues on next page

Table 6 – continued from previous page

<i>PyCallable</i>	A <i>PyProxy</i> whose proxied Python object is <i>callable</i> (i.e., has an <code>__call__()</code> method).
<i>PyDict</i>	A <i>PyProxy</i> whose proxied Python object is a <i>dict</i> .
<i>PyGenerator</i>	A <i>PyProxy</i> whose proxied Python object is a <i>generator</i> (i.e., it is an instance of <i>Generator</i>).
<i>PyIterable</i>	A <i>PyProxy</i> whose proxied Python object is <i>iterable</i> (i.e., it has an <code>__iter__()</code> method).
<i>PyIterator</i>	A <i>PyProxy</i> whose proxied Python object is an <i>iterator</i> (i.e., has a <code>send()</code> or <code>__next__()</code> method).
<i>PyProxy</i>	A <i>PyProxy</i> is an object that allows idiomatic use of a Python object from JavaScript.
<i>PyProxyWithGet</i>	A <i>PyProxy</i> whose proxied Python object has a <code>__getitem__()</code> method.
<i>PyProxyWithHas</i>	A <i>PyProxy</i> whose proxied Python object has a <code>__contains__()</code> method.
<i>PyProxyWithLength</i>	A <i>PyProxy</i> whose proxied Python object has a <code>__len__()</code> method.
<i>PyProxyWithSet</i>	A <i>PyProxy</i> whose proxied Python object has a <code>__setitem__()</code> or <code>__delitem__()</code> method.
<i>PythonError</i>	A JavaScript error caused by a Python exception.

class `pyodide.ffi.PyAsyncGenerator()`

A *PyProxy* whose proxied Python object is an *asynchronous generator* (i.e., it is an instance of *AsyncGenerator*)

Extends:

- *PyProxy*

`PyAsyncGenerator.return(v)`

Throws a *GeneratorExit* into the generator and if the *GeneratorExit* is not caught returns the argument value `{done: true, value: v}`. If the generator catches the *GeneratorExit* and returns or yields another value the next value of the generator this is returned in the normal way. If it throws some error other than *GeneratorExit* or *StopAsyncIteration*, that error is propagated. See the documentation for `AsyncGenerator.throw()`

Arguments

- `v (any)` –

Returns

`Promise<IteratorResult<any, any>>` – An Object with two properties: `done` and `value`. When the generator yields `some_value`, `return` returns `{done : false, value : some_value}`. When the generator raises a *StopAsyncIteration* exception, `return` returns `{done : true, value : result_value}`.

`PyAsyncGenerator.throw(exc)`

Throws an exception into the Generator.

See the documentation for `AsyncGenerator.throw()`.

Arguments

- `exc (any)` –

Returns

`Promise<IteratorResult<any, any>>` – An Object with two properties: `done` and

value. When the generator yields `some_value`, `return` returns `{done : false, value : some_value}`. When the generator raises a `StopIteration(result_value)` exception, `return` returns `{done : true, value : result_value}`.

class `pyodide.ffi.PyAsyncIterable()`

A *PyProxy* whose proxied Python object is *asynchronous iterable* (i.e., has an `__aiter__()` method).

Extends:

- *PyProxy*

`PyAsyncIterable.[Symbol.asyncIterator]()`

This translates to the Python code `aiter(obj)`. Return an async iterator associated to the proxy. See the documentation for `Symbol.asyncIterator`.

This will be used implicitly by `for(await let x of proxy){}`.

Returns

`AsyncIterator<any, any, any> –`

class `pyodide.ffi.PyAsyncIterator()`

A *PyProxy* whose proxied Python object is an *asynchronous iterator*

Extends:

- *PyProxy*

`PyAsyncIterator.next(arg=undefined)`

This translates to the Python code `anext(obj)`. Returns the next value of the asynchronous iterator. The argument will be sent to the Python iterator (if it's a generator for instance).

This will be used implicitly by `for(let x of proxy){}`.

Arguments

- `arg (any) –`

Returns

`Promise<IteratorResult<any, any> > –` An Object with two properties: `done` and `value`. When the iterator yields `some_value`, `next` returns `{done : false, value : some_value}`. When the iterator is done, `next` returns `{done : true }`.

class `pyodide.ffi.PyAwaitable()`

A *PyProxy* whose proxied Python object is *awaitable* (i.e., has an `__await__()` method).

Extends:

- *PyProxy*

class `pyodide.ffi.PyBuffer()`

A *PyProxy* whose proxied Python object supports the Python *Buffer Protocol*.

Examples of buffers include `{py:class}`bytes`` objects and `numpy {external+numpy:ref}`arrays``.

Extends:

- *PyProxy*

`PyBuffer.getBuffer(type)`

Get a view of the buffer data which is usable from JavaScript. No copy is ever performed.

We do not support suboffsets, if the buffer requires suboffsets we will throw an error. JavaScript nd array libraries can't handle suboffsets anyways. In this case, you should use the `toJs()` api or copy the buffer to one that doesn't use suboffsets (using e.g., `numpy.ascontiguousarray()`).

If the buffer stores big endian data or half floats, this function will fail without an explicit type argument. For big endian data you can use `toJs()`. `DataView` has support for big endian data, so you might want to pass 'dataview' as the type argument in that case.

Arguments

- **type** (`string`) – The type of the `data` field in the output. Should be one of: "i8", "u8", "u8clamped", "i16", "u16", "i32", "u32", "i32", "u32", "i64", "u64", "f32", "f64, or "dataview". This argument is optional, if absent `getBuffer()` will try to determine the appropriate output type based on the buffer format string (see [Format Strings](#)).

Returns

`PyBufferView` –

`class pyodide.ffi.PyBufferView()`

A class to allow access to Python data buffers from JavaScript. These are produced by `getBuffer()` and cannot be constructed directly. When you are done, release it with the `release()` method. See the [Python Buffer Protocol](#) documentation for more information.

To find the element `x[a_1, ..., a_n]`, you could use the following code:

```
function multiIndexToIndex(pybuff, multiIndex){
  if(multiIndex.length !==pybuff.ndim){
    throw new Error("Wrong length index");
  }
  let idx = pybuff.offset;
  for(let i = 0; i < pybuff.ndim; i++){
    if(multiIndex[i] < 0){
      multiIndex[i] = pybuff.shape[i] - multiIndex[i];
    }
    if(multiIndex[i] < 0 || multiIndex[i] >= pybuff.shape[i]){
      throw new Error("Index out of range");
    }
    idx += multiIndex[i] * pybuff.stride[i];
  }
  return idx;
}
console.log("entry is", pybuff.data[multiIndexToIndex(pybuff, [2, 0, -1])]);
```

Converting between TypedArray types

The following naive code to change the type of a typed array does not work:

```
// Incorrectly convert a TypedArray.
// Produces a Uint16Array that points to the entire WASM memory!
let myarray = new Uint16Array(buffer.data.buffer);
```

Instead, if you want to convert the output TypedArray, you need to say:

```
// Correctly convert a TypedArray.
let myarray = new Uint16Array(
  buffer.data.buffer,
  buffer.data.byteOffset,
  buffer.data.byteLength
);
```

`PyBufferView.c_contiguous`

type: `boolean`

Is it C contiguous? See `memoryview.c_contiguous`.

`PyBufferView.data`

type: `TypedArray`

The actual data. A typed array of an appropriate size backed by a segment of the WASM memory.

The `type` argument of `getBuffer()` determines which sort of `TypedArray` or `DataView` to return. By default `getBuffer()` will look at the format string to determine the most appropriate option. Most often the result is a `Uint8Array`.

Contiguity

If the buffer is not contiguous, the `readonly` `TypedArray` will contain data that is not part of the buffer. Modifying this data leads to undefined behavior.

Read only buffers

If `buffer.readonly` is true, you should not modify the buffer. Modifying a read only buffer leads to undefined behavior.

`PyBufferView.f_contiguous`

type: `boolean`

Is it Fortran contiguous? See `memoryview.f_contiguous`.

`PyBufferView.format`

type: `string`

The format string for the buffer. See [Format Strings](#) and `memoryview.format`.

`PyBufferView.itemsize`

type: `number`

How large is each entry in bytes? See `memoryview.itemsize`.

`PyBufferView.nbytes`

type: `number`

The total number of bytes the buffer takes up. This is equal to `buff.data.byteLength`. See `memoryview.nbytes`.

`PyBufferView.ndim`

type: `number`

The number of dimensions of the buffer. If `ndim` is 0, the buffer represents a single scalar or struct. Otherwise, it represents an array. See `memoryview.ndim`.

`PyBufferView.offset`

type: `number`

The offset of the first entry of the array. For instance if our array is 3d, then you will find `array[0,0,0]` at `pybuf.data[pybuf.offset]`

PyBufferView.readonly**type:** `boolean`

If the data is read only, you should not modify it. There is no way for us to enforce this, but it may cause very weird behavior. See `memoryview.readonly`.

PyBufferView.shape**type:** `number[]`

The shape of the buffer, that is how long it is in each dimension. The length will be equal to `ndim`. For instance, a 2x3x4 array would have shape `[2, 3, 4]`. See `memoryview.shape`.

PyBufferView.strides**type:** `number[]`

An array of of length `ndim` giving the number of elements to skip to get to a new element in each dimension. See the example definition of a `multiIndexToIndex` function above. See `memoryview.strides`.

PyBufferView.release()

Release the buffer. This allows the memory to be reclaimed.

class pyodide.ffi.PyCallable()

A `PyProxy` whose proxied Python object is `callable` (i.e., has an `__call__()` method).

Extends:

- `PyProxy`

PyCallable.apply(thisArg, jsargs)

The `apply()` method calls the specified function with a given `this` value, and arguments provided as an array (or an array-like object). Like `Function.apply()`.

Arguments

- **thisArg** (*any*) – The `this` argument. Has no effect unless the `PyCallable` has `captureThis()` set. If `captureThis()` is set, it will be passed as the first argument to the Python function.
- **jsargs** (*any*) – The array of arguments

Returns

any – The result from the function call.

PyCallable.bind(thisArg, ...jsargs)

The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called. See `Function.bind()`.

If the `PyCallable` does not have `captureThis()` set, the `this` parameter will be discarded. If it does have `captureThis()` set, `thisArg` will be set to the first argument of the Python function. The returned proxy and the original proxy have the same lifetime so destroying either destroys both.

Arguments

- **thisArg** (*any*) – The value to be passed as the `this` parameter to the target function `func` when the bound function is called.
- **jsargs** (*any*) – Extra arguments to prepend to arguments provided to the bound function when invoking `func`.

Returns

`PyProxy` –

`PyCallable.call(thisArg, ...jsargs)`

Calls the function with a given this value and arguments provided individually. See `Function.call()`.

Arguments

- **thisArg** (*any*) – The `this` argument. Has no effect unless the `PyCallable` has `captureThis()` set. If `captureThis()` is set, it will be passed as the first argument to the Python function.
- **jsargs** (*any*) – The arguments

Returns

any – The result from the function call.

`PyCallable.callKwargs(...jsargs)`

Call the function with key word arguments. The last argument must be an object with the keyword arguments.

Arguments

- **jsargs** (*any*) –

Returns

any –

`PyCallable.captureThis()`

Returns a `PyProxy` that passes `this` as the first argument to the Python function. The returned `PyProxy` has the internal `captureThis` property set.

It can then be used as a method on a JavaScript object. The returned proxy and the original proxy have the same lifetime so destroying either destroys both.

For example:

```
let obj = { a : 7 };
pyodide.runPython(`
    def f(self):
        return self.a
`);
// Without captureThis, it doesn't work to use f as a method for obj:
obj.f = pyodide.globals.get("f");
obj.f(); // raises "TypeError: f() missing 1 required positional argument: 'self'
→"
// With captureThis, it works fine:
obj.f = pyodide.globals.get("f").captureThis();
obj.f(); // returns 7
```

Returns

`PyProxy` – The resulting `PyProxy`. It has the same lifetime as the original `PyProxy` but passes `this` to the wrapped function.

`class pyodide.ffi.PyDict()`

A `PyProxy` whose proxied Python object is a `dict`.

Extends:

- `PyProxy`
- `PyProxyWithGet`

- [PyProxyWithSet](#)
- [PyProxyWithHas](#)
- [PyProxyWithLength](#)
- [PyIterable](#)

class `pyodide.ffi.PyGenerator()`

A [PyProxy](#) whose proxied Python object is a [generator](#) (i.e., it is an instance of [Generator](#)).

Extends:

- [PyProxy](#)

`PyGenerator.return(v)`

Throws a [GeneratorExit](#) into the generator and if the [GeneratorExit](#) is not caught returns the argument value {done: true, value: v}. If the generator catches the [GeneratorExit](#) and returns or yields another value the next value of the generator this is returned in the normal way. If it throws some error other than [GeneratorExit](#) or [StopIteration](#), that error is propagated. See the documentation for [Generator.return\(\)](#).

Arguments

- `v` ([any](#)) –

Returns

[IteratorResult](#)<[any](#), [any](#)> – An Object with two properties: done and value. When the generator yields some_value, return returns {done : false, value : some_value}. When the generator raises a [StopIteration\(result_value\)](#) exception, return returns {done : true, value : result_value}.

`PyGenerator.throw(exc)`

Throws an exception into the Generator.

See the documentation for [Generator.throw\(\)](#).

Arguments

- `exc` ([any](#)) –

Returns

[IteratorResult](#)<[any](#), [any](#)> – An Object with two properties: done and value. When the generator yields some_value, return returns {done : false, value : some_value}. When the generator raises a [StopIteration\(result_value\)](#) exception, return returns {done : true, value : result_value}.

class `pyodide.ffi.PyIterable()`

A [PyProxy](#) whose proxied Python object is [iterable](#) (i.e., it has an `__iter__()` method).

Extends:

- [PyProxy](#)

`PyIterable.[Symbol.iterator]()`

This translates to the Python code `iter(obj)`. Return an iterator associated to the proxy. See the documentation for [Symbol.iterator](#).

This will be used implicitly by `for(let x of proxy){}`.

Returns

[Iterator](#)<[any](#), [any](#), [any](#)> –

class `pyodide.ffi.PyIterator()`

A *PyProxy* whose proxied Python object is an *iterator* (i.e., has a `send()` or `__next__()` method).

Extends:

- *PyProxy*

`PyIterator.next(arg=undefined)`

This translates to the Python code `next(obj)`. Returns the next value of the generator. See the documentation for `Generator.next()`. The argument will be sent to the Python generator.

This will be used implicitly by `for(let x of proxy){}`.

Arguments

- `arg (any)` –

Returns

`IteratorResult<any, any>` – An Object with two properties: `done` and `value`. When the generator yields `some_value`, `next` returns `{done : false, value : some_value}`. When the generator raises a `StopIteration` exception, `next` returns `{done : true, value : result_value}`.

class `pyodide.ffi.PyProxy()`

A *PyProxy* is an object that allows idiomatic use of a Python object from JavaScript. See *Proxying from Python into JavaScript*.

`PyProxy.type`

type: `string`

The name of the type of the object.

Usually the value is `"module.name"` but for builtins or interpreter-defined types it is just `"name"`. As pseudocode this is:

```
ty = type(x)
if ty.__module__ == 'builtins' or ty.__module__ == "__main__":
    return ty.__name__
else:
    ty.__module__ + "." + ty.__name__
```

`PyProxy.copy()`

Make a new *PyProxy* pointing to the same Python object. Useful if the *PyProxy* is destroyed somewhere else.

Returns

PyProxy –

`PyProxy.destroy(options)`

Destroy the *PyProxy*. This will release the memory. Any further attempt to use the object will raise an error.

In a browser supporting `FinalizationRegistry`, Pyodide will automatically destroy the *PyProxy* when it is garbage collected, however there is no guarantee that the finalizer will be run in a timely manner so it is better to destroy the proxy explicitly.

Arguments

- `options.destroyRoundtrip (boolean)` –

- **options.message** (*string*) – The error message to print if use is attempted after destroying. Defaults to “Object has already been destroyed”.

`PyProxy.isAwaitable()`

Note: Deprecated: Use `obj isinstance pyodide.ffi.PyAwaitable` instead.

Check whether the *PyProxy* is a *PyAwaitable*

Returns

boolean (typeguard for *PyAwaitable*) –

`PyProxy.isBuffer()`

Note: Deprecated: Use `obj isinstance pyodide.ffi.PyBuffer` instead.

Check whether the *PyProxy* is a *PyBuffer*.

Returns

boolean (typeguard for *PyBuffer*) –

`PyProxy.isCallable()`

Note: Deprecated: `obj isinstance pyodide.ffi.PyCallable` instead.

Check whether the *PyProxy* is a *PyCallable*.

Returns

boolean (typeguard for *PyCallable*) –

`PyProxy.isIterable()`

Note: Deprecated: Use `obj isinstance pyodide.ffi.PyIterable` instead.

Check whether the *PyProxy* is a *PyIterable*.

Returns

boolean (typeguard for *PyIterable*) –

`PyProxy.isIterator()`

Note: Deprecated: Use `obj isinstance pyodide.ffi.PyIterator` instead.

Check whether the *PyProxy* is a *PyIterator*

Returns

boolean (typeguard for *PyIterator*) –

`PyProxy.supportsGet()`

Note: Deprecated: Use `obj isinstance pyodide.ffi.PyProxyWithGet` instead.

Check whether the *PyProxy* is a *PyProxyWithGet*.

Returns

boolean (typeguard for *PyProxyWithGet*) –

`PyProxy.supportsHas()`

Note: Deprecated: Use `obj instanceof pyodide.ffi.PyProxyWithHas` instead.

Check whether the *PyProxy* is a *PyProxyWithHas*.

Returns

boolean (typeguard for *PyProxyWithHas*) –

`PyProxy.supportsLength()`

Note: Deprecated: Use `obj instanceof pyodide.ffi.PyProxyWithLength` instead.

Check whether the *PyProxy* is a *PyProxyWithLength*.

Returns

boolean (typeguard for *PyProxyWithLength*) –

`PyProxy.supportsSet()`

Note: Deprecated: Use `obj instanceof pyodide.ffi.PyProxyWithSet` instead.

Check whether the *PyProxy* is a *PyProxyWithSet*.

Returns

boolean (typeguard for *PyProxyWithSet*) –

`PyProxy.toJs(options)`

Converts the *PyProxy* into a JavaScript object as best as possible. By default does a deep conversion, if a shallow conversion is desired, you can use `proxy.toJs({depth : 1})`. See *Explicit Conversion of PyProxy* for more info.

Arguments

- **options.create_pyproxies** (*boolean*) – If false, `toJs()` will throw a *ConversionError* rather than producing a *PyProxy*.
- **options.depth** (*number*) – How many layers deep to perform the conversion. Defaults to infinite
- **options.pyproxies** (*PyProxy[]*) – If provided, `toJs()` will store all PyProxies created in this list. This allows you to easily destroy all the PyProxies by iterating the list without having to recurse over the generated structure. The most common use case is to create a new empty list, pass the list as `pyproxies`, and then later iterate over `pyproxies` to destroy all of created proxies.
- **options.default_converter** ((*obj: PyProxy*, *convert: (obj: PyProxy) => any*, *cacheConversion: (obj: PyProxy, result: any) => void*) => *any*) – Optional argument to convert objects with no default conversion. See the documentation of `to_js()`.
- **options.dict_converter** ((*array: Iterable<[key: string, value: any]>*) => *any*) – A function to be called on an iterable of pairs [key, value]. Convert this iterable of

pairs to the desired output. For instance, `Object.fromEntries()` would convert the dict to an object, `Array.from()` converts it to an `Array` of pairs, and `(it) => new Map(it)` converts it to a `Map` (which is the default behavior).

Returns

any – The JavaScript object resulting from the conversion.

`PyProxy.toString()`

Returns

string –

class `pyodide.ffi.PyProxyWithGet()`

A *PyProxy* whose proxied Python object has a `__getitem__()` method.

Extends:

- *PyProxy*

`PyProxyWithGet.get(key)`

This translates to the Python code `obj[key]`.

Arguments

- **key** (*any*) – The key to look up.

Returns

any – The corresponding value.

class `pyodide.ffi.PyProxyWithHas()`

A *PyProxy* whose proxied Python object has a `__contains__()` method.

Extends:

- *PyProxy*

`PyProxyWithHas.has(key)`

This translates to the Python code `key in obj`.

Arguments

- **key** (*any*) – The key to check for.

Returns

boolean – Is key present?

class `pyodide.ffi.PyProxyWithLength()`

A *PyProxy* whose proxied Python object has a `__len__()` method.

Extends:

- *PyProxy*

`PyProxyWithLength.length`

type: *number*

The length of the object.

class `pyodide.ffi.PyProxyWithSet()`

A *PyProxy* whose proxied Python object has a `__setitem__()` or `__delitem__()` method.

Extends:

- *PyProxy*

`PyProxyWithSet.delete(key)`

This translates to the Python code `del obj[key]`.

Arguments

- **key** (*any*) – The key to delete.

`PyProxyWithSet.set(key, value)`

This translates to the Python code `obj[key] = value`.

Arguments

- **key** (*any*) – The key to set.
- **value** (*any*) – The value to set it to.

class `pyodide.ffi.PythonError()`

A JavaScript error caused by a Python exception.

In order to reduce the risk of large memory leaks, the `PythonError` contains no reference to the Python exception that caused it. You can find the actual Python exception that caused this error as `sys.last_value`.

See *type translations of errors* for more information.

Avoid leaking stack Frames

If you make a *PyProxy* of `sys.last_value`, you should be especially careful to *destroy()* it when you are done. You may leak a large amount of memory including the local variables of all the stack frames in the traceback if you don't. The easiest way is to only handle the exception in Python.

`PythonError.type`

type: `string`

The name of the Python error class, e.g, `RuntimeError` or `KeyError`.

pyodide.canvas

This provides APIs to set a canvas for rendering graphics.

Functions:

`getCanvas2D()`

`getCanvas3D()`

`setCanvas2D(canvas)`

`setCanvas3D(canvas)`

`pyodide.canvas.getCanvas2D()`

Returns

`undefined` | `HTMLCanvasElement` – The HTML5 canvas element used for 2D rendering. For now, Emscripten only supports one canvas element, so `getCanvas2D` and `getCanvas3D` are the same.

`pyodide.canvas.getCanvas3D()`

Returns

`undefined` | `HTMLCanvasElement` – The HTML5 canvas element used for 3D rendering. For now, Emscripten only supports one canvas element, so `getCanvas2D` and `getCanvas3D` are the same.

`pyodide.canvas.setCanvas2D(canvas)`

Arguments

- **canvas** (`HTMLCanvasElement`) – The HTML5 canvas element to use for 2D rendering. For now, Emscripten only supports one canvas element, so `setCanvas2D` and `setCanvas3D` are the same.

`pyodide.canvas.setCanvas3D(canvas)`

Arguments

- **canvas** (`HTMLCanvasElement`) – The HTML5 canvas element to use for 3D rendering. For now, Emscripten only supports one canvas element, so `setCanvas2D` and `setCanvas3D` are the same.

Python API

Backward compatibility of the API is not guaranteed at this point.

JavaScript Modules

By default there are two JavaScript modules. More can be added with `pyodide.registerJsModule()`. You can import these modules using the Python `import` statement in the normal way.

<code>js</code>	The global JavaScript scope.
<code>pyodide_js</code>	The JavaScript Pyodide module.

Python Modules

<code>pyodide.code</code>	Utilities for evaluating Python and JavaScript code.
<code>pyodide.console</code>	Similar to the Python builtin <code>code</code> module but handles top level await. Used for implementing the Pyodide console.
<code>pyodide.ffi</code>	The <code>JsProxy</code> class and utilities to help interact with JavaScript code.
<code>pyodide.http</code>	Defines <code>pyfetch()</code> and other functions for making network requests.
<code>pyodide.webloop</code>	The Pyodide event loop implementation. This is automatically configured correctly for most use cases it is unlikely you will need it outside of niche use cases.

pyodide.code

Classes:

<code>CodeRunner</code> (source, *, return_mode, mode, ...)	This class allows fine control over the execution of a code block.
---	--

Functions:

<code>eval_code</code> (source[, globals, locals, ...])	Runs a string as Python source code.
<code>async eval_code_async</code> (source[, globals, locals, ...])	Runs a code string asynchronously.
<code>find_imports</code> (source)	Finds the imports in a Python source code string
<code>run_js</code> (code, /)	A wrapper for the <code>eval()</code> function.
<code>should_quiet</code> (source, /)	Should we suppress output?

```
class pyodide.code.CodeRunner(source, *, return_mode='last_expr', mode='exec',
                               quiet_trailing_semicolon=True, filename='<exec>', flags=0)
```

This class allows fine control over the execution of a code block.

It is primarily intended for REPLs and other sophisticated consumers that may wish to add their own AST transformations, separately signal to the user when parsing is complete, etc. The simpler `eval_code()` and `eval_code_async()` apis should be preferred when their flexibility suffices.

Parameters

- **source** (`str`) – The Python source code to run.
- **return_mode** (`Literal['last_expr', 'last_expr_or_assign', 'none']`) – Specifies what should be returned. The options are:
 - **'last_expr'**
return the last expression
 - **'last_expr_or_assign'**
return the last expression or the last assignment.
 - **'none'**
always return None.
- **quiet_trailing_semicolon** (`bool`) – Specifies whether a trailing semicolon should suppress the result or not. When this is `True` executing `"1+1;"` returns `None`, when it is `False`, executing `"1+1;"` return `2`. `True` by default.
- **filename** (`str`) – The file name to use in error messages and stack traces. `'<exec>'` by default.
- **mode** (`str`) – The “mode” to compile in. One of `"exec"`, `"single"`, or `"eval"`. Defaults to `"exec"`. For most purposes it’s unnecessary to use this argument. See the documentation for the built-in `compile()` function.
- **flags** (`int`) – The flags to compile with. See the documentation for the built-in `compile()` function.

ast: `Module`

The ast from parsing source. If you wish to do an ast transform, modify this variable before calling `CodeRunner.compile()`.

code: `Optional[CodeType]`

Once you call `CodeRunner.compile()` the compiled code will be available in the code field. You can modify this variable before calling `CodeRunner.run()` to do a code transform.

compile()

Compile the current value of `self.ast` and store the result in `self.code`.

Can only be used once. Returns `self` (chainable).

Return type

`CodeRunner`

run(globals=None, locals=None)

Executes `self.code`.

Can only be used after calling `compile`. The code may not use top level await, use `CodeRunner.run_async()` for code that uses top level await.

Parameters

- **globals** (`Optional[dict[str, Any]]`) – The global scope in which to execute code. This is used as the `globals` parameter for `exec()`. If `globals` is absent, a new empty dictionary is used.
- **locals** (`Optional[dict[str, Any]]`) – The local scope in which to execute code. This is used as the `locals` parameter for `exec()`. If `locals` is absent, the value of `globals` is used.

Return type

`Any`

Returns

If the last nonwhitespace character of `source` is a semicolon, return `None`. If the last statement is an expression, return the result of the expression. Use the `return_mode` and `quiet_trailing_semicolon` parameters to modify this default behavior.

async run_async(globals=None, locals=None)

Runs `self.code` which may use top level await.

Can only be used after calling `CodeRunner.compile()`. If `self.code` uses top level await, automatically awaits the resulting coroutine.

Parameters

- **globals** (`Optional[dict[str, Any]]`) – The global scope in which to execute code. This is used as the `globals` parameter for `exec()`. If `globals` is absent, a new empty dictionary is used.
- **locals** (`Optional[dict[str, Any]]`) – The local scope in which to execute code. This is used as the `locals` parameter for `exec()`. If `locals` is absent, the value of `globals` is used.

Return type

`Any`

Returns

If the last nonwhitespace character of `source` is a semicolon, return `None`. If the last statement is an expression, return the result of the expression. Use the `return_mode` and `quiet_trailing_semicolon` parameters to modify this default behavior.

```
pyodide.code.eval_code(source, globals=None, locals=None, *, return_mode='last_expr',
                        quiet_trailing_semicolon=True, filename='<exec>', flags=0)
```

Runs a string as Python source code.

Parameters

- **source** (`str`) – The Python source code to run.
- **globals** (`Optional[dict[str, Any]]`) – The global scope in which to execute code. This is used as the `globals` parameter for `exec()`. If `globals` is absent, a new empty dictionary is used.
- **locals** (`Optional[dict[str, Any]]`) – The local scope in which to execute code. This is used as the `locals` parameter for `exec()`. If `locals` is absent, the value of `globals` is used.
- **return_mode** (`Literal['last_expr', 'last_expr_or_assign', 'none']`) – Specifies what should be returned. The options are:
 - `'last_expr'`
return the last expression
 - `'last_expr_or_assign'`
return the last expression or the last assignment.
 - `'none'`
always return None.
- **quiet_trailing_semicolon** (`bool`) – Specifies whether a trailing semicolon should suppress the result or not. When this is `True` executing `"1+1 ;"` returns `None`, when it is `False`, executing `"1+1 ;"` return `2`. `True` by default.
- **filename** (`str`) – The file name to use in error messages and stack traces. `'<exec>'` by default.
- **flags** (`int`) – The flags to compile with. See the documentation for the built-in `compile()` function.

Return type

`Any`

Returns

If the last nonwhitespace character of `source` is a semicolon, return `None`. If the last statement is an expression, return the result of the expression. Use the `return_mode` and `quiet_trailing_semicolon` parameters to modify this default behavior.

Examples

```
>>> from pyodide.code import eval_code
>>> source = "1 + 1"
>>> eval_code(source)
2
>>> source = "1 + 1;"
>>> eval_code(source, quiet_trailing_semicolon=True)
>>> eval_code(source, quiet_trailing_semicolon=False)
2
>>> my_globals = { "y": "100" }
>>> my_locals = { "y": "200" }
```

(continues on next page)

(continued from previous page)

```

>>> source = "print(locals()['y'], globals()['y'])"
>>> eval_code(source, globals=my_globals, locals=my_locals)
200 100
>>> source = "test = 1 + 1"
>>> eval_code(source, return_mode="last_expr_or_assign")
2
>>> eval_code(source, return_mode="last_expr")
>>> eval_code(source, return_mode="none")
>>> source = "print(pyodide)" # Pretend this is open('example_of_filename.py', 'r').
    ↪read()
>>> eval_code(source, filename="example_of_filename.py")
# Traceback will show where in the file the error happened
# ...File "example_of_filename.py", line 1, in <module>...NameError: name 'pyodide'
    ↪is not defined

```

async `pyodide.code.eval_code_async(source, globals=None, locals=None, *, return_mode='last_expr', quiet_trailing_semicolon=True, filename='<exec>', flags=0)`

Runs a code string asynchronously.

Uses `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` to compile the code.

Parameters

- **source** (`str`) – The Python source code to run.
- **globals** (`Optional[dict[str, Any]]`) – The global scope in which to execute code. This is used as the `globals` parameter for `exec()`. If `globals` is absent, a new empty dictionary is used.
- **locals** (`Optional[dict[str, Any]]`) – The local scope in which to execute code. This is used as the `locals` parameter for `exec()`. If `locals` is absent, the value of `globals` is used.
- **return_mode** (`Literal['last_expr', 'last_expr_or_assign', 'none']`) – Specifies what should be returned. The options are:
 - 'last_expr'**
return the last expression
 - 'last_expr_or_assign'**
return the last expression or the last assignment.
 - 'none'**
always return `None`.
- **quiet_trailing_semicolon** (`bool`) – Specifies whether a trailing semicolon should suppress the result or not. When this is `True` executing `"1+1 ;"` returns `None`, when it is `False`, executing `"1+1 ;"` return `2`. `True` by default.
- **filename** (`str`) – The file name to use in error messages and stack traces. `'<exec>'` by default.
- **flags** (`int`) – The flags to compile with. See the documentation for the built-in `compile()` function.

Return type

`Any`

Returns

If the last nonwhitespace character of `source` is a semicolon, return `None`. If the last statement is an expression, return the result of the expression. Use the `return_mode` and `quiet_trailing_semicolon` parameters to modify this default behavior.

`pyodide.code.find_imports(source)`

Finds the imports in a Python source code string

Parameters

source (`str`) – The Python source code to inspect for imports.

Return type

`list[str]`

Returns

A list of module names that are imported in `source`. If `source` is not syntactically correct Python code (after dedenting), returns an empty list.

Examples

```
>>> from pyodide.code import find_imports
>>> source = "import numpy as np; import scipy.stats"
>>> find_imports(source)
['numpy', 'scipy']
```

`pyodide.code.run_js(code, /)`

A wrapper for the `eval()` function.

Runs code as a Javascript code string and returns the result. Unlike `eval()`, if `code` is not a string we raise a `TypeError`.

Parameters

code (`str`) –

Return type

`Any`

`pyodide.code.should_quiet(source, /)`

Should we suppress output?

Return type

`bool`

Returns

True if the last nonwhitespace character of `source` is a semicolon.

Examples

```
>>> should_quiet('1 + 1')
False
>>> should_quiet('1 + 1 ;')
True
>>> should_quiet('1 + 1 # comment ;')
False
```

Parameters**source** (*str*) –**pyodide.console****Classes:**

<code>Console([globals, stdin_callback, ...])</code>	Interactive Pyodide console
<code>ConsoleFuture(syntax_check)</code>	A future with extra fields used as the return value for <code>Console</code> apis.
<code>PyodideConsole([globals, stdin_callback, ...])</code>	A subclass of <code>Console</code> that uses <code>pyodide.loadPackagesFromImports()</code> before running the code.

Functions:

<code>repr_shorten(value[, limit, split, separator])</code>	Compute the string representation of <code>value</code> and shorten it if necessary.
---	--

```
class pyodide.console.Console(globals=None, *, stdin_callback=None, stdout_callback=None,
                              stderr_callback=None, persistent_stream_redirection=False,
                              filename='<console>')
```

Interactive Pyodide console

An interactive console based on the Python standard library `InteractiveConsole` that manages stream redirections and asynchronous execution of the code.

The stream callbacks can be modified directly by assigning to `stdin_callback` (for example) as long as `persistent_stream_redirection` is `False`.

Parameters

- **globals** (`Optional[dict[str, Any]]`) – The global namespace in which to evaluate the code. Defaults to a new empty dictionary.
- **stdin_callback** (`Optional[Callable[[int], str]]`) – Function to call at each read from `sys.stdin`. Defaults to `None`.
- **stdout_callback** (`Optional[Callable[[str], None]]`) – Function to call at each write to `sys.stdout`. Defaults to `None`.
- **stderr_callback** (`Optional[Callable[[str], None]]`) – Function to call at each write to `sys.stderr`. Defaults to `None`.
- **persistent_stream_redirection** (`bool`) – Should redirection of standard streams be kept between calls to `runcode()`? Defaults to `False`.
- **filename** (*str*) – The file name to report in error messages. Defaults to "<console>".

buffer: `list[str]`

The list of lines of code that have been the argument to `push()`.

This is emptied whenever the code is executed.

complete(*source*)

Use Python's `rlcompleter` to complete the source string using the `Console.globals` namespace.

Finds the last “word” in the source string and completes it with `rlcompleter`. Word breaks are determined by the set of characters in `completer_word_break_characters`.

Parameters

source (`str`) – The source string to complete at the end.

Return type

`tuple[list[str], int]`

Returns

- **completions** (`list[str]`) – A list of completion strings.
- **start** (`int`) – The index where completion starts.

Examples

```
>>> shell = Console()
>>> shell.complete("str.isa")
(['str.isalnum(', 'str.isalpha(', 'str.isascii('], 0)
>>> shell.complete("a = 5 ; str.isa")
(['str.isalnum(', 'str.isalpha(', 'str.isascii('], 8)
```

completer_word_break_characters: `str`

The set of characters considered by `complete()` to be word breaks.

formatsyntaxerror(`e`)

Format the syntax error that just occurred.

This doesn’t include a stack trace because there isn’t one. The actual error object is stored into `sys.last_value`.

Parameters

e (`Exception`) –

Return type

`str`

formattraceback(`e`)

Format the exception that just occurred.

The actual error object is stored into `sys.last_value`.

Parameters

e (`BaseException`) –

Return type

`str`

globals: `dict[str, Any]`

The namespace used as the globals

persistent_redirect_streams()

Redirect `stdin/stdout/stdout` persistently

Return type

`None`

persistent_restore_streams()

Restore `stdin/stdout/stderr` if they have been persistently redirected

Return type

`None`

push(line)

Push a line to the interpreter.

The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended.

The return value is the result of calling `runsource()` on the current buffer contents.

Parameters

line (`str`) –

Return type

`ConsoleFuture`

redirect_streams()

A context manager to redirect standard streams.

This supports nesting.

Return type

`Generator[None, None, None]`

async runcode(source, code)

Execute a code object and return the result.

Parameters

- **source** (`str`) –
- **code** (`CodeRunner`) –

Return type

`Any`

runsource(source, filename='<console>')

Compile and run source code in the interpreter.

Parameters

- **source** (`str`) –
- **filename** (`str`) –

Return type

`ConsoleFuture`

stderr_callback: Optional[Callable[[str], None]]

Function to call at each write to `sys.stderr`.

stdin_callback: Optional[Callable[[int], str]]

The function to call at each read from `sys.stdin`

stdout_callback: Optional[Callable[[str], None]]

Function to call at each write to `sys.stdout`.

class `pyodide.console.ConsoleFuture`(*syntax_check*)

A future with extra fields used as the return value for `Console` apis.

Parameters

syntax_check (`Literal`['incomplete', 'syntax-error', 'complete']) –

formatted_error: `Optional`[`str`]

If the Future is rejected, this will be filled with a formatted version of the code. This is a convenience that simplifies code and helps to avoid large memory leaks when using from JavaScript.

syntax_check: `Literal`['incomplete', 'syntax-error', 'complete']

The status of the future. The values mean the following:

‘incomplete’

Input is incomplete. The future has already been resolved with result `None`.

‘syntax-error’

Input contained a syntax error. The future has been rejected with a `SyntaxError`.

‘complete’

The input complete and syntactically correct and asynchronous execution has begun. When the execution is done, the Future will be resolved with the result or rejected with an exception.

class `pyodide.console.PyodideConsole`(*globals=None*, *, *stdin_callback=None*, *stdout_callback=None*, *stderr_callback=None*, *persistent_stream_redirection=False*, *filename='<console>'*)

A subclass of `Console` that uses `pyodide.loadPackagesFromImports()` before running the code.

Parameters

- **globals** (`Optional`[`dict`[`str`, `Any`]]) –
- **stdin_callback** (`Optional`[`Callable`[[`int`], `str`]]) –
- **stdout_callback** (`Optional`[`Callable`[[`str`], `None`]]) –
- **stderr_callback** (`Optional`[`Callable`[[`str`], `None`]]) –
- **persistent_stream_redirection** (`bool`) –
- **filename** (`str`) –

`pyodide.console.repr_shorten`(*value*, *limit=1000*, *split=None*, *separator='...'*)

Compute the string representation of *value* and shorten it if necessary.

This is equivalent to `shorten(repr(value), limit, split, separator)`, but a value error is raised if *limit* is less than 4.

Examples

```
>>> from pyodide.console import repr_shorten
>>> sep = "_"
>>> repr_shorten("abcdefg", limit=8, separator=sep)
"abc_efg"
>>> repr_shorten("abcdefg", limit=12, separator=sep)
"abcdefg"
>>> for i in range(4, 10):
...     repr_shorten(123456789, limit=i, separator=sep)
'12_89'
```

(continues on next page)

(continued from previous page)

```
'12_89'
'123_789'
'123_789'
'1234_6789'
'123456789'
```

Parameters

- **value** (*Any*) –
- **limit** (*int*) –
- **split** (*Optional[int]*) –
- **separator** (*str*) –

Return type*str***pyodide.ffi****Exceptions:**

<i>ConversionError</i>	An error thrown when conversion between JavaScript and Python fails.
<i>JsException</i> (*args, **kwargs)	A JavaScript Error.

Classes:

<i>JsArray</i> ()	A JsProxy of an <i>Array</i> , <i>NodeList</i> , or <i>TypedArray</i>
<i>JsAsyncGenerator</i> ()	A JavaScript <i>AsyncGenerator</i>
<i>JsAsyncIterable</i> ()	A JavaScript async iterable object
<i>JsAsyncIterator</i> ()	A JsProxy of a JavaScript async iterator.
<i>JsBuffer</i> ()	A JsProxy of an array buffer or array buffer view
<i>JsCallable</i> ()	
<i>JsDomElement</i> ()	
<i>JsDoubleProxy</i> ()	A double proxy created with <i>create_proxy()</i> .
<i>JsFetchResponse</i> ()	A <i>JsFetchResponse</i> object represents a <i>Response</i> to a <i>fetch()</i> request.
<i>JsGenerator</i> ()	A JavaScript generator
<i>JsIterable</i> ()	A JavaScript iterable object
<i>JsIterator</i> ()	A JsProxy of a JavaScript iterator.
<i>JsMap</i> ()	A JavaScript Map
<i>JsMutableMap</i> ()	A JavaScript mutable map
<i>JsPromise</i> ()	A <i>JsProxy</i> of a <i>Promise</i> or some other <i>thenable</i> JavaScript object.
<i>JsProxy</i> ()	A proxy to make a JavaScript object behave like a Python object

continues on next page

Table 13 – continued from previous page

*JsTypedArray()***Functions:**

<i>create_once_callable(obj, /)</i>	Wrap a Python Callable in a JavaScript function that can be called once.
<i>create_proxy(obj, /, *[, capture_this, ...])</i>	Create a <i>JsProxy</i> of a <i>PyProxy</i> .
<i>destroy_proxies(pyproxies, /)</i>	Destroy all PyProxies in a JavaScript array.
<i>register_js_module(name, jsproxy)</i>	Registers jsproxy as a JavaScript module named name.
<i>to_js(obj, /, *[, depth, pyproxies, ...])</i>	Convert the object to JavaScript.
<i>unregister_js_module(name)</i>	Unregisters a JavaScript module with given name that has been previously registered with <i>pyodide.registerJsModule()</i> or <i>pyodide.ffi.register_js_module()</i> .

exception `pyodide.ffi.ConversionError`Bases: `Exception`

An error thrown when conversion between JavaScript and Python fails.

class `pyodide.ffi.JsArray`Bases: `JsProxy`, `Generic[T]`A JsProxy of an `Array`, `NodeList`, or `TypedArray`**append**(*object*)

Append object to the end of the list.

Parameters**object** (T) –**Return type**`None`**count**(*x*)

Return the number of times x appears in the list.

Parameters**x** (T) –**Return type**`int`**extend**(*other, /*)

Extend array by appending elements from the iterable.

Parameters**other** (`Iterable[T]`) –**Return type**`None`**index**(*value, start=0, stop=9223372036854775807*)

Return first index at which value appears in the Array.

Raises `ValueError` if the value is not present.

Parameters

- **value** (T) –
- **start** (int) –
- **stop** (int) –

Return type

int

pop(*index=-1*)

Remove and return the item at index (default last).

Raises `IndexError` if list is empty or index is out of range.**Parameters****index** (int) –**Return type**

T

reverse()

Reverse the array in place.

Present only if the wrapped Javascript object is an array.

Return type

None

to_py(*, *depth=-1*, *default_converter=None*)Convert the `JsProxy` to a native Python object as best as possible.See *JavaScript to Python* for more information.**Parameters**

- **depth** (int) – Limit the depth of the conversion. If a shallow conversion is desired, set depth to 1.
- **default_converter** (Optional[Callable[[`JsProxy`, Callable[[`JsProxy`], Any], Callable[[`JsProxy`, Any], None]], Any]]) – If present, this will be invoked whenever Pyodide does not have some built in conversion for the object. If `default_converter` raises an error, the error will be allowed to propagate. Otherwise, the object returned will be used as the conversion. `default_converter` takes three arguments. The first argument is the value to be converted.

Return type

list[Any]

Examples

Here are a couple examples of converter functions. In addition to the normal conversions, convert `Date`` to `datetime`:

```
from datetime import datetime
def default_converter(value, _ignored1, _ignored2):
    if value.constructor.name == "Date":
        return datetime.fromtimestamp(d.valueOf()/1000)
    return value
```

Don't create any JsProxies, require a complete conversion or raise an error:

```
def default_converter(_value, _ignored1, _ignored2):
    raise Exception("Failed to completely convert object")
```

The second and third arguments are only needed for converting containers. The second argument is a conversion function which is used to convert the elements of the container with the same settings. The third argument is a “cache” function which is needed to handle self referential containers. Consider the following example. Suppose we have a Javascript Pair class:

```
class Pair {
    constructor(first, second){
        this.first = first;
        this.second = second;
    }
}
```

We can use the following `default_converter` to convert `Pair` to `list`:

```
def default_converter(value, convert, cache):
    if value.constructor.name != "Pair":
        return value
    result = []
    cache(value, result);
    result.append(convert(value.first))
    result.append(convert(value.second))
    return result
```

Note that we have to cache the conversion of `value` before converting `value.first` and `value.second`. To see why, consider a self referential pair:

```
let p = new Pair(0, 0);
p.first = p;
```

Without `cache(value, result);`, converting `p` would lead to an infinite recurse. With it, we can successfully convert `p` to a list such that `l[0]` is `1`.

class pyodide.ffi.JsAsyncGenerator

Bases: [JsAsyncIterable](#)[Tco], [Generic](#)[Tco, Tcontra, Vco]

A JavaScript [AsyncGenerator](#)

A JavaScript object is treated as an async generator if it's `Symbol.toStringTag` is `"AsyncGenerator"`. Most likely this will be because it is a true async generator produced by the JavaScript runtime, but it may be a custom object trying hard to pretend to be an async generator. It should have `next()`, `return()`, and `throw()` methods.

aclose()

Raises a [GeneratorExit](#) at the point where the generator function was paused.

If the generator function then exits gracefully, is already closed, or raises [GeneratorExit](#) (by not catching the exception), `aclose()` returns to its caller. If the generator yields a value, a [RuntimeError](#) is raised. If the generator raises any other exception, it is propagated to the caller. `aclose()` does nothing if the generator has already exited due to an exception or normal exit.

Return type

[Awaitable](#)[None]

asend(*value*, /)

Resumes the execution and “sends” a value into the async generator function.

The *value* argument becomes the result of the current yield expression. The awaitable returned by the `asend()` method will return the next value yielded by the generator or raises `StopAsyncIteration` if the asynchronous generator returns. If the generator returned a value, this value is discarded (because in Python async generators cannot return a value).

When `asend()` is called to start the generator, the argument will be ignored. Unlike in Python, we cannot detect that the generator hasn’t started yet, and no error will be thrown if the argument of a not-started generator is not `None`.

Parameters

value (`Tcontra`) –

Return type

`Awaitable`[`Tco`]

athrow(*error*, /)

Resumes the execution and raises an exception at the point where the generator was paused.

The awaitable returned by `athrow()` method will return the next value yielded by the generator or raises `StopAsyncIteration` if the asynchronous generator returns. If the generator returned a value, this value is discarded (because in Python async generators cannot return a value). If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

Parameters

error (`BaseException`) –

Return type

`Tco`

class `pyodide.ffi.JsAsyncIterable`

Bases: `JsProxy`, `Generic`[`Tco`]

A JavaScript async iterable object

A JavaScript object is async iterable if it has a `Symbol.asyncIterator` method.

class `pyodide.ffi.JsAsyncIterator`

Bases: `JsProxy`, `Generic`[`Tco`]

A `JsProxy` of a JavaScript async iterator.

An object is a `JsAsyncIterator` if it has a `next()` method and either has a `Symbol.asyncIterator` or has no `Symbol.iterator`

class `pyodide.ffi.JsBuffer`

Bases: `JsProxy`

A `JsProxy` of an array buffer or array buffer view

assign(*rhs*, /)

Assign from a Python buffer into the JavaScript buffer.

Parameters

rhs (`Any`) –

Return type

`None`

assign_to(to, /)

Assign to a Python buffer from the JavaScript buffer.

Parameters

to (*Any*) –

Return type

None

from_file(file, /)

Reads from a file into a buffer.

Will try to read a chunk of data the same size as the buffer from the current position of the file.

Example

```
>>> import pytest; pytest.skip()
>>> from js import Uint8Array
>>> # the JsProxy need to be pre-allocated
>>> x = Uint8Array.new(range(10))
>>> with open('file.bin', 'rb') as fh:
...     x.read_file(fh)
which is equivalent to
>>> x = Uint8Array.new(range(10))
>>> with open('file.bin', 'rb') as fh:
...     chunk = fh.read(size=x.byteLength)
...     x.assign(chunk)
but the latter copies the data twice whereas the former only copies the
data once.
```

Parameters

file (*Union*[*IO*[bytes], *IO*[str]]) –

Return type

None

to_bytes()

Convert a buffer to a bytes object.

Copies the data once.

Return type

bytes

to_file(file, /)

Writes a buffer to a file.

Will write the entire contents of the buffer to the current position of the file.

Example

```
>>> import pytest; pytest.skip()
>>> from js import Uint8Array
>>> x = Uint8Array.new(range(10))
>>> with open('file.bin', 'wb') as fh:
...     x.to_file(fh)
which is equivalent to,
>>> with open('file.bin', 'wb') as fh:
...     data = x.to_bytes()
...     fh.write(data)
but the latter copies the data twice whereas the former only copies the
data once.
```

Parameters

file ([Union](#)[[IO](#)[bytes], [IO](#)[str]]) –

Return type

[None](#)

`to_memoryview()`

Convert a buffer to a memoryview.

Copies the data once. This currently has the same effect as [to_py\(\)](#).

Return type

[memoryview](#)

`to_string(encoding=None)`

Convert a buffer to a string object.

Copies the data twice.

The encoding argument will be passed to the [TextDecoder](#) constructor. It should be one of the encodings listed in [the table here](#). The default encoding is utf8.

Parameters

encoding ([Optional](#)[str]) –

Return type

[str](#)

`class pyodide.ffi.JsCallable`

Bases: [JsProxy](#)

`class pyodide.ffi.JsDomElement`

Bases: [JsProxy](#)

`class pyodide.ffi.JsDoubleProxy`

Bases: [JsProxy](#)

A double proxy created with [create_proxy\(\)](#).

`destroy()`

Destroy the proxy.

Return type

[None](#)

unwrap()

Unwrap a double proxy created with [create_proxy\(\)](#) into the wrapped Python object.

Return type

[Any](#)

exception `pyodide.ffi.JsException(*args, **kwargs)`

Bases: [JsProxy](#), [Exception](#)

A JavaScript Error.

These are pickleable unlike other JsProxies.

message: [str](#)

The error message

name: [str](#)

The name of the error type

stack: [str](#)

The JavaScript stack trace

class `pyodide.ffi.JsFetchResponse`

Bases: [JsProxy](#)

A [JsFetchResponse](#) object represents a [Response](#) to a [fetch\(\)](#) request.

class `pyodide.ffi.JsGenerator`

Bases: [JsIterable](#)[Tco], [Generic](#)[Tco, Tcontra, Vco]

A JavaScript generator

A JavaScript object is treated as a generator if its [Symbol.toStringTag](#) is "Generator". Most likely this will be because it is a true [Generator](#) produced by the JavaScript runtime, but it may be a custom object trying hard to pretend to be a generator. It should have [next\(\)](#), [return\(\)](#) and [throw\(\)](#) methods.

close()

Raises a [GeneratorExit](#) at the point where the generator function was paused.

If the generator function then exits gracefully, is already closed, or raises [GeneratorExit](#) (by not catching the exception), [close\(\)](#) returns to its caller. If the generator yields a value, a [RuntimeError](#) is raised. If the generator raises any other exception, it is propagated to the caller. [close\(\)](#) does nothing if the generator has already exited due to an exception or normal exit.

Return type

[None](#)

send(value)

Resumes the execution and “sends” a value into the generator function.

The value argument becomes the result of the current yield expression. The [send\(\)](#) method returns the next value yielded by the generator, or raises [StopIteration](#) if the generator exits without yielding another value. When [send\(\)](#) is called to start the generator, the argument will be ignored. Unlike in Python, we cannot detect that the generator hasn’t started yet, and no error will be thrown if the argument of a not-started generator is not [None](#).

Parameters

value (Tcontra) –

Return type

Tco

throw(*error*, /)

Raises an exception at the point where the generator was paused, and returns the next value yielded by the generator function.

If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

In typical use, this is called with a single exception instance similar to the way the `raise` keyword is used.

For backwards compatibility, however, a second signature is supported, following a convention from older versions of Python. The type argument should be an exception class, and value should be an exception instance. If the value is not provided, the type constructor is called to get an instance. If `traceback` is provided, it is set on the exception, otherwise any existing `__traceback__` attribute stored in value may be cleared.

Parameters

error (`BaseException`) –

Return type

`Tco`

class `pyodide.ffi.JsIterable`

Bases: `JsProxy`, `Generic`[`Tco`]

A JavaScript iterable object

A JavaScript object is iterable if it has a `Symbol.iterator` method.

class `pyodide.ffi.JsIterator`

Bases: `JsProxy`, `Generic`[`Tco`]

A `JsProxy` of a JavaScript iterator.

An object is a `JsAsyncIterator` if it has a `next()` method and either has a `Symbol.iterator` or has no `Symbol.asyncIterator`.

class `pyodide.ffi.JsMap`

Bases: `JsProxy`, `Generic`[`KT`, `VTco`]

A JavaScript Map

To be considered a map, a JavaScript object must have a `get` method, it must have a `size` or a `length` property which is a number (idiomatically it should be called `size`) and it must be iterable.

get(*key*, *default*, /)

If `key` in `self`, returns `self[key]`. Otherwise returns `default`.

Parameters

- **key** (`KT`) –
- **default** (`Optional`[`VTco`]) –

Return type

`VTco`

items()

Return a `ItemsView` for the map.

Return type

`ItemsView`[`KT`, `VTco`]

keys()

Return a `KeysView` for the map.

Return type

`KeysView[KT]`

values()

Return a `ValuesView` for the map.

Return type

`ValuesView[VTco]`

class `pyodide.ffi.JsMutableMap`

Bases: `JsMap[KT, VT]`, `Generic[KT, VT]`

A JavaScript mutable map

To be considered a mutable map, a JavaScript object must have a `get` method, a `has` method, a `size` or a `length` property which is a number (idiomatically it should be called `size`) and it must be iterable.

Instances of the JavaScript builtin `Map` class are `JsMutableMap` s. Also proxies returned by `JsProxy.as_object_map()` are instances of `JsMap` .

clear()

Empty out the map entirely.

Return type

`None`

pop(*key*, *default=None*, /)

If `key` in `self`, return `self[key]` and remove `key` from `self`. Otherwise returns `default`.

Parameters

- **key** (`KT`) –
- **default** (`Optional[VT]`) –

Return type

`VT`

popitem()

Remove some arbitrary `key` , `value` pair from the map and returns the (`key` , `value`) tuple.

Return type

`tuple[KT, VT]`

setdefault(*key*, *default=None*)

If `key` in `self`, return `self[key]`. Otherwise sets `self[key] = default` and returns `default`.

Parameters

- **key** (`KT`) –
- **default** (`Optional[VT]`) –

Return type

`VT`

update(*other=None*, /, ***kwargs*)

Updates `self` from `other` and `kwargs`.

Parameters

- **other** (`Mapping[KT, VT]` | `Iterable[tuple[KT, VT]]`) – Either a mapping or an iterable of pairs. This can be left out.
- **kwargs** (`VT`) – Extra key-values pairs to insert into the map. Only usable for inserting extra strings.

Return type`None`

If `other` is present and is a `Mapping` or has a `keys` method, does

```
for k in other:
    self[k] = other[k]
```

If `other` is present and lacks a `keys` method, does

```
for (k, v) in other:
    self[k] = v
```

In all cases this is followed by:

```
for (k, v) in kwargs.items():
    self[k] = v
```

class `pyodide.ffi.JsPromise`

Bases: `JsProxy`

A `JsProxy` of a `Promise` or some other `thenable` JavaScript object.

A JavaScript object is considered to be a `Promise` if it has a `then` method.

catch(*onrejected*, /)

The `Promise.catch()` API, wrapped to manage the lifetimes of the handler.

Pyodide will automatically release the references to the handler when the promise resolves.

Parameters

onrejected (`Callable[[Any], Any]`) –

Return type`JsPromise`**finally_**(*onfinally*, /)

The `Promise.finally()` API, wrapped to manage the lifetimes of the handler.

Pyodide will automatically release the references to the handler when the promise resolves. Note the trailing underscore in the name; this is needed because `finally` is a reserved keyword in Python.

Parameters

onfinally (`Callable[[], Any]`) –

Return type`JsPromise`**then**(*onfulfilled*, *onrejected*)

The `Promise.then()` API, wrapped to manage the lifetimes of the handlers.

Pyodide will automatically release the references to the handlers when the promise resolves.

Parameters

- **onfulfilled** (`Callable[[Any], Any]`) –

- **onrejected** (Callable[[Any], Any]) –

Return type

JsPromise

class `pyodide.ffi.JsProxy`

Bases: `object`

A proxy to make a JavaScript object behave like a Python object

For more information see the *Type translations* documentation. In particular, see *the list of __dunder__ methods* that are (conditionally) implemented on *JsProxy*.

as_object_map(*, *hereditary=False*)

Returns a new JsProxy that treats the object as a map.

The methods `__getitem__()`, `__setitem__()`, `__contains__()`, `__len__()`, etc will perform lookups via `object[key]` or similar.

Note that `len(x.as_object_map())` evaluates in $O(n)$ time (it iterates over the object and counts how many `ownKeys()` it has). If you need to compute the length in $O(1)$ time, use a real `Map` instead.

Parameters

hereditary (bool) – If True, any “plain old objects” stored as values in the object will be wrapped in *as_object_map* themselves.

Return type

JsMutableMap[str, Any]

Examples

```
from pyodide.code import run_js

o = run_js("({x : {y: 2}})")
# You have to access the properties of o as attributes
assert o.x.y == 2
with pytest.raises(TypeError):
    o["x"] # is not subscriptable

# as_object_map allows us to access the property with getitem
assert o.as_object_map()["x"].y == 2

with pytest.raises(TypeError):
    # The inner object is not subscriptable because hereditary is False.
    o.as_object_map()["x"]["y"]

# When hereditary is True, the inner object is also subscriptable
assert o.as_object_map(hereditary=True)["x"]["y"] == 2
```

js_id: `int`

An id number which can be used as a dictionary/set key if you want to key on JavaScript object identity.

If two JsProxy are made with the same backing JavaScript object, they will have the same `js_id`.

new(**args*, ***kwargs*)

Construct a new instance of the JavaScript object

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
JsProxy

object_entries()

The JavaScript API `Object.entries(object)`

Return type
JsProxy

object_keys()

The JavaScript API `Object.keys(object)`

Return type
JsProxy

object_values()

The JavaScript API `Object.values(object)`

Return type
JsProxy

to_py(*, *depth=-1*, *default_converter=None*)

Convert the *JsProxy* to a native Python object as best as possible.

See *JavaScript to Python* for more information.

Parameters

- **depth** (*int*) – Limit the depth of the conversion. If a shallow conversion is desired, set `depth` to 1.
- **default_converter** (*Optional[Callable[[JsProxy, Callable[[JsProxy], Any], Callable[[JsProxy, Any], None], Any]]*) – If present, this will be invoked whenever Pyodide does not have some built in conversion for the object. If `default_converter` raises an error, the error will be allowed to propagate. Otherwise, the object returned will be used as the conversion. `default_converter` takes three arguments. The first argument is the value to be converted.

Return type
Any

Examples

Here are a couple examples of converter functions. In addition to the normal conversions, convert `Date` to `datetime`:

```
from datetime import datetime
def default_converter(value, _ignored1, _ignored2):
    if value.constructor.name == "Date":
        return datetime.fromtimestamp(d.valueOf()/1000)
    return value
```

Don't create any JsProxies, require a complete conversion or raise an error:

```
def default_converter(_value, _ignored1, _ignored2):
    raise Exception("Failed to completely convert object")
```

The second and third arguments are only needed for converting containers. The second argument is a conversion function which is used to convert the elements of the container with the same settings. The third argument is a “cache” function which is needed to handle self referential containers. Consider the following example. Suppose we have a Javascript Pair class:

```
class Pair {
    constructor(first, second){
        this.first = first;
        this.second = second;
    }
}
```

We can use the following `default_converter` to convert `Pair` to `list`:

```
def default_converter(value, convert, cache):
    if value.constructor.name != "Pair":
        return value
    result = []
    cache(value, result);
    result.append(convert(value.first))
    result.append(convert(value.second))
    return result
```

Note that we have to cache the conversion of `value` before converting `value.first` and `value.second`. To see why, consider a self referential pair:

```
let p = new Pair(0, 0);
p.first = p;
```

Without `cache(value, result);`, converting `p` would lead to an infinite recurse. With it, we can successfully convert `p` to a list such that `l[0]` is `l`.

typeof: `str`

Returns the JavaScript type of the `JsProxy`.

Corresponds to `typeof obj`; in JavaScript. You may also be interested in the `constructor` attribute which returns the type as an object.

class `pyodide.ffi.JsTypedArray`

Bases: `JsBuffer`, `JsArray[int]`

`pyodide.ffi.create_once_callable(obj, /)`

Wrap a Python Callable in a JavaScript function that can be called once.

After being called the proxy will decrement the reference count of the Callable. The JavaScript function also has a `destroy` API that can be used to release the proxy without calling it.

Parameters

`obj` (`Callable[... Any]`) –

Return type

`JsOnceCallable`

`pyodide.ffi.create_proxy(obj, /, *, capture_this=False, roundtrip=True)`

Create a *JsProxy* of a *PyProxy*.

This allows explicit control over the lifetime of the *PyProxy* from Python: call the *destroy()* API when done.

Parameters

- **obj** (*Any*) – The object to wrap.
- **capture_this** (*bool*) – If the object is callable, should *this* be passed as the first argument when calling it from JavaScript.
- **roundtrip** (*bool*) – When the proxy is converted back from JavaScript to Python, if this is *True* it is converted into a double proxy. If *False*, it is unwrapped into a Python object. In the case that *roundtrip* is *True* it is possible to unwrap a double proxy with the *JsDoubleProxy.unwrap()* method. This is useful to allow easier control of lifetimes from Python:

```
from js import o
d = {}
o.d = create_proxy(d, roundtrip=True)
o.d.destroy() # Destroys the proxy created with create_proxy
```

With *roundtrip=False* this would be an error.

Return type

JsDoubleProxy

`pyodide.ffi.destroy_proxies(pyproxies, /)`

Destroy all *PyProxies* in a JavaScript array.

pyproxies must be a JavaScript Array of *PyProxies*. Intended for use with the arrays created from the “*pyproxies*” argument of *toJs()* and *to_js()*. This method is necessary because indexing the Array from Python automatically unwraps the *PyProxy* into the wrapped Python object.

Parameters

pyproxies (*JsArray[Any]*) –

Return type

None

`pyodide.ffi.register_js_module(name, jsproxy)`

Registers *jsproxy* as a JavaScript module named *name*. The module can then be imported from Python using the standard Python import system. If another module by the same name has already been imported, this won’t have much effect unless you also delete the imported module from `sys.modules`. This is called by the JavaScript API *pyodide.registerJsModule()*.

Parameters

- **name** (*str*) – Name of js module
- **jsproxy** (*Any*) – JavaScript object backing the module

Return type

None

`pyodide.ffi.to_js(obj, /, *, depth=-1, pyproxies=None, create_pyproxies=True, dict_converter=None, default_converter=None)`

Convert the object to JavaScript.

This is similar to `toJs()`, but for use from Python. If the object can be implicitly translated to JavaScript, it will be returned unchanged. If the object cannot be converted into JavaScript, this method will return a *JsProxy* of a *PyProxy*, as if you had used `create_proxy()`.

See *Python to JavaScript* for more information.

Parameters

- **obj** (*Any*) – The Python object to convert
- **depth** (*int*) – The maximum depth to do the conversion. Negative numbers are treated as infinite. Set this to 1 to do a shallow conversion.
- **pyproxies** (*Optional[JsProxy]*) – Should be a JavaScript *Array*. If provided, any PyProxies generated will be stored here. You can later use `destroy_proxies()` if you want to destroy the proxies from Python (or from JavaScript you can just iterate over the *Array* and destroy the proxies).
- **create_pyproxies** (*bool*) – If you set this to *False*, `to_js()` will raise an error rather than creating any pyproxies.
- **dict_converter** (*Optional[Callable[[Iterable[JsArray[Any]], JsProxy]]*) – This converter if provided receives a (JavaScript) iterable of (JavaScript) pairs [key, value]. It is expected to return the desired result of the dict conversion. Some suggested values for this argument:
 - `js.Map.new` – similar to the default behavior
 - `js.Array.from` – convert to an array of entries
 - `js.Object.fromEntries` – convert to a JavaScript object
- **default_converter** (*Optional[Callable[[Any, Callable[[Any], JsProxy], Callable[[Any, JsProxy], None]], JsProxy]]*) – If present will be invoked whenever Pyodide does not have some built in conversion for the object. If `default_converter` raises an error, the error will be allowed to propagate. Otherwise, the object returned will be used as the conversion. `default_converter` takes three arguments. The first argument is the value to be converted.

Return type

Any

Examples

Here are some examples demonstrating the usage of the `default_converter` argument.

In addition to the normal conversions, convert JavaScript *Date* objects to *datetime* objects:

```
from datetime import datetime
from js import Date
def default_converter(value, _ignored1, _ignored2):
    if isinstance(value, datetime):
        return Date.new(value.timestamp() * 1000)
    return value
```

Don't create any PyProxies, require a complete conversion or raise an error:

```
def default_converter(_value, _ignored1, _ignored2):
    raise Exception("Failed to completely convert object")
```

The second and third arguments are only needed for converting containers. The second argument is a conversion function which is used to convert the elements of the container with the same settings. The third argument is a “cache” function which is needed to handle self referential containers. Consider the following example. Suppose we have a Python Pair class:

```
class Pair:
    def __init__(self, first, second):
        self.first = first self.second = second
```

We can use the following `default_converter` to convert `Pair` to `Array`:

```
from js import Array

def default_converter(value, convert, cache):
    if not isinstance(value, Pair):
        return value
    result = Array.new() cache(value, result);
    result.push(convert(value.first)) result.push(convert(value.second))
    return result
```

Note that we have to cache the conversion of `value` before converting `value.first` and `value.second`. To see why, consider a self referential pair:

```
p = Pair(0, 0); p.first = p;
```

Without `cache(value, result);`, converting `p` would lead to an infinite recurse. With it, we can successfully convert `p` to an `Array` such that `l[0] === 1`.

`pyodide.ffi.unregister_js_module(name)`

Unregisters a JavaScript module with given name that has been previously registered with `pyodide.registerJsModule()` or `pyodide.ffi.register_js_module()`. If a JavaScript module with that name does not already exist, will raise an error. If the module has already been imported, this won't have much effect unless you also delete the imported module from `sys.modules`. This is called by the JavaScript API `pyodide.unregisterJsModule()`.

Parameters

name (`str`) – Name of the module to unregister

Return type

`None`

Functions:

<code>add_event_listener(elt, event, listener)</code>	Wrapper for JavaScript's <code>addEventListener()</code> which automatically manages the lifetime of a JsProxy corresponding to the <code>listener</code> parameter.
<code>clear_interval(interval_retval)</code>	Wrapper for JavaScript's <code>clearInterval()</code> which automatically manages the lifetime of a JsProxy corresponding to the <code>callback</code> parameter.
<code>clear_timeout(timeout_retval)</code>	Wrapper for JavaScript's <code>clearTimeout()</code> which automatically manages the lifetime of a JsProxy corresponding to the <code>callback</code> parameter.
<code>remove_event_listener(elt, event, listener)</code>	Wrapper for JavaScript's <code>removeEventListener()</code> which automatically manages the lifetime of a JsProxy corresponding to the <code>listener</code> parameter.

continues on next page

Table 15 – continued from previous page

<code>set_interval(callback, interval)</code>	Wrapper for JavaScript's <code>setInterval()</code> which automatically manages the lifetime of a JsProxy corresponding to the <code>callback</code> parameter.
<code>set_timeout(callback, timeout)</code>	Wrapper for JavaScript's <code>setTimeout()</code> which automatically manages the lifetime of a JsProxy corresponding to the <code>callback</code> param.

`pyodide.ffi.wrappers.add_event_listener(elt, event, listener)`

Wrapper for JavaScript's `addEventListener()` which automatically manages the lifetime of a JsProxy corresponding to the `listener` parameter.

Parameters

- `elt` (*JsDomElement*) –
- `event` (*str*) –
- `listener` (*Callable*[[*Any*], *None*]) –

Return type

None

`pyodide.ffi.wrappers.clear_interval(interval_retval)`

Wrapper for JavaScript's `clearInterval()` which automatically manages the lifetime of a JsProxy corresponding to the `callback` parameter.

Parameters

`interval_retval` (*int* | *JsProxy*) –

Return type

None

`pyodide.ffi.wrappers.clear_timeout(timeout_retval)`

Wrapper for JavaScript's `clearTimeout()` which automatically manages the lifetime of a JsProxy corresponding to the `callback` parameter.

Parameters

`timeout_retval` (*int* | *JsProxy*) –

Return type

None

`pyodide.ffi.wrappers.remove_event_listener(elt, event, listener)`

Wrapper for JavaScript's `removeEventListener()` which automatically manages the lifetime of a JsProxy corresponding to the `listener` parameter.

Parameters

- `elt` (*JsDomElement*) –
- `event` (*str*) –
- `listener` (*Callable*[[*Any*], *None*]) –

Return type

None

`pyodide.ffi.wrappers.set_interval(callback, interval)`

Wrapper for JavaScript's `setInterval()` which automatically manages the lifetime of a JsProxy corresponding to the `callback` parameter.

Parameters

- **callback** (`Callable[[], None]`) –
- **interval** (`int`) –

Return type`int | JsProxy``pyodide.ffi.wrappers.set_timeout(callback, timeout)`

Wrapper for JavaScript's `setTimeout()` which automatically manages the lifetime of a `JsProxy` corresponding to the callback param.

Parameters

- **callback** (`Callable[[], None]`) –
- **timeout** (`int`) –

Return type`int | JsProxy`**pyodide.http****Classes:**

<code>FetchResponse(url, js_response)</code>	A wrapper for a Javascript fetch <code>Response</code> .
--	--

Functions:

<code>open_url(url)</code>	Fetches a given URL synchronously.
<code>async pyfetch(url, **kwargs)</code>	Fetch the url and return the response.

class `pyodide.http.FetchResponse(url, js_response)`

A wrapper for a Javascript fetch `Response`.

Parameters

- **url** (`str`) – URL to fetch
- **js_response** (`JsFetchResponse`) – A `JsProxy` of the fetch response

body_used: `bool`

Has the response been used yet?

If so, attempting to retrieve the body again will raise an `OSError`. Use `clone()` first to avoid this. See `Response.bodyUsed`.

async `buffer()`

Return the response body as a Javascript `ArrayBuffer`.

See `Response.arrayBuffer()`.

Return type`JsBuffer`**async** `bytes()`

Return the response body as a bytes object

Return type*bytes***clone()**

Return an identical copy of the *FetchResponse*.

This method exists to allow multiple uses of *FetchResponse* objects. See `Response.clone()`.

Return type*FetchResponse***async json(**kwargs)**

Treat the response body as a JSON string and use `json.loads()` to parse it into a Python object.

Any keyword arguments are passed to `json.loads()`.

Parameters**kwargs** (*Any*) –**Return type***Any***async memoryview()**

Return the response body as a *memoryview* object

Return type*memoryview***ok: bool**

Was the request successful?

See `Response.ok`.

redirected: bool

Was the request redirected?

See `Response.redirected`.

status: int

Response status code

See `Response.status`.

status_text: str

Response status text

See `Response.statusText`.

async string()

Return the response body as a string

Return type*str***type: str**

The type of the response.

See `Response.type`.

async unpack_archive(**, extract_dir=None, format=None*)

Treat the data as an archive and unpack it into target directory.

Assumes that the file is an archive in a format that `shutil` has an unpacker for. The arguments `extract_dir` and `format` are passed directly on to `shutil.unpack_archive()`.

Parameters

- **extract_dir** (`Optional[str]`) – Directory to extract the archive into. If not provided, the current working directory is used.
- **format** (`Optional[str]`) – The archive format: one of "zip", "tar", "gztar", "bztar". Or any other format registered with `shutil.register_unpack_format()`. If not provided, `unpack_archive()` will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a `ValueError` is raised.

Return type

`None`

url: `str`

The url of the response.

The value may be different than the url passed to fetch. See `Response.url`.

`pyodide.http.open_url(url)`

Fetches a given URL synchronously.

The download of binary files is not supported. To download binary files use `pyodide.http.pyfetch()` which is asynchronous.

Parameters

url (`str`) – URL to fetch

Return type

`StringIO`

Returns

The contents of the URL.

async pyodide.http.pyfetch(*url, **kwargs*)

Fetch the url and return the response.

This functions provides a similar API to `fetch()` however it is designed to be convenient to use from Python. The `FetchResponse` has methods with the output types already converted to Python objects.

Parameters

- **url** (`str`) – URL to fetch.
- ****kwargs** (`Any`) – keyword arguments are passed along as optional parameters to the fetch API.

Return type

`FetchResponse`

pyodide.webloop

Classes:

<code>PyodideFuture(*[, loop])</code>	A future with extra <code>then</code> , <code>catch</code> , and <code>finally_</code> methods based on the Javascript promise API.
<code>WebLoop()</code>	A custom event loop for use in Pyodide.
<code>WebLoopPolicy()</code>	A simple event loop policy for managing <code>WebLoop</code> -based event loops.

class `pyodide.webloop.PyodideFuture(*, loop=None)`

A future with extra `then`, `catch`, and `finally_` methods based on the Javascript promise API. `Webloop.create_future` returns these so in practice all futures encountered in Pyodide should be an instance of `PyodideFuture`.

catch(*onrejected*)

Equivalent to `then(None, onrejected)`

Parameters

onrejected (`Callable[[BaseException], object]`) –

Return type

`PyodideFuture[Any]`

finally_(*onfinally*)

When the future is either resolved or rejected, call `onfinally` with no arguments.

Parameters

onfinally (`Callable[[], None]`) –

Return type

`PyodideFuture[T]`

then(*onfulfilled*, *onrejected=None*)

When the Future is done, either execute `onfulfilled` with the result or execute `onrejected` with the exception.

Returns a new Future which will be marked done when either the `onfulfilled` or `onrejected` callback is completed. If the return value of the executed callback is awaitable it will be awaited repeatedly until a nonawaitable value is received. The returned Future will be resolved with that value. If an error is raised, the returned Future will be rejected with the error.

Parameters

- **onfulfilled** (`Optional[Callable[[T], Union[S, Awaitable[S]]]]) – A function called if the Future is fulfilled. This function receives one argument, the fulfillment value.`
- **onrejected** (`Optional[Callable[[BaseException], Union[S, Awaitable[S]]]]) – A function called if the Future is rejected. This function receives one argument, the rejection value.`

Return type

`PyodideFuture[S]`

Returns

A new future to be resolved when the original future is done and the appropriate callback is also done.

class `pyodide.webloop.WebLoop`

A custom event loop for use in Pyodide.

Schedules tasks on the browser event loop. Does no lifecycle management and runs forever.

`run_forever()` and `run_until_complete()` cannot block like a normal event loop would because we only have one thread so blocking would stall the browser event loop and prevent anything from ever happening.

We defer all work to the browser event loop using the `setTimeout()` function. To ensure that this event loop doesn't stall out UI and other browser handling, we want to make sure that each task is scheduled on the browser event loop as a task not as a microtask. `setTimeout(callback, 0)` enqueues the callback as a task so it works well for our purposes.

See the Python [Event Loop](#) documentation.

class `pyodide.webloop.WebLoopPolicy`

A simple event loop policy for managing *WebLoop*-based event loops.

pyodide CLI

This page documents the Pyodide Command Line Interface (CLI) interface. In addition to the commands defined by `pyodide-build`, documented below, other subcommands are defined in external packages (which can be installed with `pip`):

- `pyodide pack`, defined in `pyodide-pack` is a package bundler for Pyodide

pyodide

A command line interface for Pyodide.

Other CLI subcommands are registered via the plugin system by installing Pyodide compatible packages (e.g. `pyodide-build`).

```
pyodide [OPTIONS] COMMAND [ARGS]...
```

Options

--version

auditwheel

Auditwheel-like tool for emscripten wheels and shared libraries.

Registered by: `auditwheel_emscripten`

```
pyodide auditwheel [OPTIONS] COMMAND [ARGS]...
```

copy

Copy shared libraries to the wheel directory. Similar to repair but does not modify the needed section of WASM module.

```
pyodide auditwheel copy [OPTIONS] WHEEL_FILE
```

Options

--libdir <libdir>

Path to the directory containing the shared libraries.

Default

lib

--output-dir <output_dir>

Directory to output repaired wheel or shared library. (default: overwrite the input file)

Arguments

WHEEL_FILE

Required argument

exports

Show exports of a wheel or a shared library file.

```
pyodide auditwheel exports [OPTIONS] WHEEL_OR_SO_FILE
```

Options

--show-type, --no-show-type

Show function type.

Default

False

Arguments

WHEEL_OR_SO_FILE

Required argument

imports

Show imports of a wheel or a shared library file.

```
pyodide auditwheel imports [OPTIONS] WHEEL_OR_SO_FILE
```

Options

--show-type, --no-show-type

Show function type.

Default

False

Arguments

WHEEL_OR_SO_FILE

Required argument

repair

[Experimental] Repair a wheel file: copy shared libraries to the wheel directory and modify the path in the wheel file.

```
pyodide auditwheel repair [OPTIONS] WHEEL_FILE
```

Options

--libdir <libdir>

Path to the directory containing the shared libraries.

Default

lib

--output-dir <output_dir>

Directory to output repaired wheel or shared library. (default: overwrite the input file)

Arguments

WHEEL_FILE

Required argument

show

Show shared library dependencies of a wheel or a shared library file.

```
pyodide auditwheel show [OPTIONS] WHEEL_OR_SO_FILE
```

Arguments

WHEEL_OR_SO_FILE

Required argument

build

Use pypa/build to build a Python package from source, pypi or url.

Registered by: pyodide-build

```
pyodide build [OPTIONS] [SOURCE_LOCATION]
```

Options

--output-directory <output_directory>

which directory should the output be placed into?

Default

./dist

-r, --requirements <requirements_txt>

Build a list of package requirements from a requirements.txt file

Default

--exports <exports>

Which symbols should be exported when linking .so files?

Default

requested

--build-dependencies, --no-build-dependencies

Fetch non-pyodide dependencies from pypi and build them too.

Default

False

--output-lockfile <output_lockfile>

Output list of resolved dependencies to a file in requirements.txt format

Default

--skip-dependency <skip_dependency>

Skip building or resolving a single dependency. Use multiple times or provide a comma separated list to skip multiple dependencies.

Default

--compression-level <compression_level>
 Compression level to use for the created zip file
Default
 6

Arguments

SOURCE_LOCATION
 Optional argument

build-recipes

Build packages using yaml recipes and create repodata.json
 Registered by: pyodide-build

```
pyodide build-recipes [OPTIONS] PACKAGES...
```

Options

--recipe-dir <recipe_dir>
 The directory containing the recipe of packages. If not specified, the default is *./packages*

--no-deps, --no-no-deps
 If true, do not build dependencies of the specified packages.
Default
 False

--install, --no-install
 If true, install the built packages into the *install_dir*. If false, build packages without installing.
Default
 False

--install-dir <install_dir>
 Path to install built packages and repodata.json. If not specified, the default is *./dist*.

--cflags <cflags>
 Extra compiling flags. Default: *SIDE_MODULE_CFLAGS*

--cxxflags <cxxflags>
 Extra compiling flags. Default: *SIDE_MODULE_CXXFLAGS*

--ldflags <ldflags>
 Extra linking flags. Default: *SIDE_MODULE_LDFLAGS*

--target-install-dir <target_install_dir>
 The path to the target Python installation. Default: *TARGETINSTALLDIR*

--host-install-dir <host_install_dir>
 Directory for installing built host packages. Default: *HOSTINSTALLDIR*

--log-dir <log_dir>

Directory to place log files

--force-rebuild, --no-force-rebuild

Force rebuild of all packages regardless of whether they appear to have been updated

Default

False

--continue

Continue a build from the middle. For debugging. Implies ‘--force-rebuild’

Default

False

--n-jobs <n_jobs>

Number of packages to build in parallel (default: # of cores in the system)

--compression-level <compression_level>

Level of zip compression to apply when installing. 0 means no compression.

Default

6

Arguments

PACKAGES

Required argument(s)

config

Manage config variables used in pyodide

Registered by: pyodide-build

```
pyodide config [OPTIONS] COMMAND [ARGS]...
```

get

Get a value of a single config variable used in pyodide

```
pyodide config get [OPTIONS] CONFIG_VAR
```

Arguments

CONFIG_VAR

Required argument

list

List config variables used in pyodide

```
pyodide config list [OPTIONS]
```

py-compile

Compile .py files to .pyc in a wheel, a zip file, or a folder with wheels or zip files.

If the provided folder contains the *repodata.json* file, it will be rewritten with the updated wheel / zip file paths and sha256 checksums.

Registered by: pyodide-build

```
pyodide py-compile [OPTIONS] PATH
```

Options

--silent, --no-silent

Silent mode, do not print anything.

Default

False

--keep, --no-keep

Keep the original wheel / zip file.

Default

False

--compression-level <compression_level>

Compression level to use for the created zip file

Default

6

Arguments

PATH

Required argument

skeleton

Add a new package build recipe or update an existing recipe

Registered by: pyodide-build

```
pyodide skeleton [OPTIONS] COMMAND [ARGS]...
```

pypi

Create a new package from PyPI.

```
pyodide skeleton pypi [OPTIONS] NAME
```

Options

-u, --update

Update an existing recipe instead of creating a new one

Default

False

--update-patched

Force update the package even if it contains patches.

Default

False

--version <version>

The version of the package, if not specified, latest version will be used.

--source-format <source_format>

Which source format is preferred. Options are wheel or sdist. If not specified, then either a wheel or an sdist will be used.

--recipe-dir <recipe_dir>

The directory containing the recipe of packages. If not specified, the default is <cwd>/packages.

Arguments

NAME

Required argument

venv

Create a Pyodide virtual environment

Registered by: pyodide-build

```
pyodide venv [OPTIONS] DEST
```


Arguments

DEST

Required argument

3.1.10 Frequently Asked Questions

How can I load external files in Pyodide?

If you are using Pyodide in the browser, you should download external files and save them to the virtual file system. The recommended way to do this is to zip the files and unpack them into the file system with `pyodide.unpackArchive()`:

```
let zipResponse = await fetch("myfiles.zip");
let zipBinary = await zipResponse.arrayBuffer();
pyodide.unpackArchive(zipBinary, "zip");
```

You can also download the files from Python using `pyfetch()`, which is a convenient wrapper of JavaScript `fetch()`:

```
await pyodide.runPythonAsync(`
    from pyodide.http import pyfetch
    response = await pyfetch("https://some_url/myfiles.zip")
    await response.unpack_archive()
`)
```

If you are working in Node.js, you can mount a native folder into the file system as follows:

```
FS.mkdir("/local_directory");
FS.mount(NODEFS, { root: "some/local/filepath" }, "/local_directory");
```

Then you can access the mounted folder from Python via the `/local_directory` mount.

Why can't I just use urllib or requests?

We currently can't use such packages since sockets are not available in Pyodide. See [Write http.client in terms of Web APIs](#) for more information.

Why can't I load files from the local file system?

For security reasons JavaScript in the browser is not allowed to load local data files (for example, `file:///path/to/local/file.data`). You will run into Network Errors, due to the [Same Origin Policy](#). There is a [File System API](#) supported in Chrome but not in Firefox or Safari. See [\(Experimental\) Using the native file system in the browser](#) for experimental local file system support.

For development purposes, you can serve your files with a [web server](#).

How can I execute code in a custom namespace?

The second argument to `pyodide.runPython()` is an options object which may include a `globals` element which is a namespace for code to read from and write to. The provided namespace must be a Python dictionary.

```
let my_namespace = pyodide.globals.get("dict")();
pyodide.runPython(`x = 1 + 1`, { globals: my_namespace });
pyodide.runPython(`y = x ** x`, { globals: my_namespace });
my_namespace.get("y"); // ==> 4
```

You can also use this approach to inject variables from JavaScript into the Python namespace, for example:

```
let my_namespace = pyodide.toPy({ x: 2, y: [1, 2, 3] });
pyodide.runPython(
  `
    assert x == y[1]
    z = x ** x
  `,
  { globals: my_namespace }
);
my_namespace.get("z"); // ==> 4
```

How to detect that code is run with Pyodide?

At run time, you can check if Python is built with Emscripten (which is the case for Pyodide) with,

```
import sys

if sys.platform == 'emscripten':
    # running in Pyodide or other Emscripten based build
```

To detect that a code is running with Pyodide specifically, you can check for the loaded `pyodide` module,

```
import sys

if "pyodide" in sys.modules:
    # running in Pyodide
```

This however will not work at build time (i.e. in a `setup.py`) due to the way the Pyodide build system works. It first compiles packages with the host compiler (e.g. `gcc`) and then re-runs the compilation commands with `emsdk`. So the `setup.py` is never run inside the Pyodide environment.

To detect Pyodide, at build time use,

```
import os

if "PYODIDE" in os.environ:
    # building for Pyodide
```

How do I create custom Python packages from JavaScript?

Put a collection of functions into a JavaScript object and use `pyodide.registerJsModule()`: JavaScript:

```
let my_module = {
  f: function (x) {
    return x * x + 1;
  },
  g: function (x) {
    console.log(`Calling g on argument ${x}`);
    return x;
  },
  submodule: {
    h: function (x) {
      return x * x - 1;
    },
    c: 2,
  },
};
pyodide.registerJsModule("my_js_module", my_module);
```

You can import your package like a normal Python package:

```
import my_js_module
from my_js_module.submodule import h, c
assert my_js_module.f(7) == 50
assert h(9) == 80
assert c == 2
```

How can I send a Python object from my server to Pyodide?

The best way to do this is with pickle. If the version of Python used in the server exactly matches the version of Python used in the client, then objects that can be successfully pickled can be sent to the client and unpickled in Pyodide. If the versions of Python are different then for instance sending AST is unlikely to work since there are breaking changes to Python AST in most Python minor versions.

Similarly when pickling Python objects defined in a Python package, the package version needs to match exactly between the server and pyodide.

Generally, pickles are portable between architectures (here amd64 and wasm32). The rare cases when they are not portable, for instance currently tree based models in scikit-learn, can be considered as a bug in the upstream library.

Security Issues with pickle

Unpickling data is similar to `eval`. On any public-facing server it is a really bad idea to unpickle any data sent from the client. For sending data from client to server, try some other serialization format like JSON.

How can I use a Python function as an event handler?

Note that the most straight forward way of doing this will not work:

```
from js import document
def f(*args):
    document.querySelector("h1").innerHTML += "<.<"

document.body.addEventListener('click', f)
```

Now every time you click, an error will be raised (see *Calling JavaScript functions from Python*).

To do this correctly use `create_proxy()` as follows:

```
from js import document
from pyodide.ffi import create_proxy
def f(*args):
    document.querySelector("h1").innerHTML += "<.<"

proxy_f = create_proxy(f)
document.body.addEventListener('click', proxy_f)
# Store proxy_f in Python then later:
document.body.removeEventListener('click', proxy_f)
proxy_f.destroy()
```

How can I use fetch with optional arguments from Python?

The most obvious translation of the JavaScript code won't work:

```
import json
resp = await js.fetch('/someurl', {
    "method": "POST",
    "body": json.dumps({ "some" : "json" }),
    "credentials": "same-origin",
    "headers": { "Content-Type": "application/json" }
})
```

The `fetch()` API ignores the options that we attempted to provide. You can do this correctly in one of two ways:

```
import json
from pyodide.ffi import to_js
from js import Object
resp = await js.fetch('example.com/some_api',
    method= "POST",
    body= json.dumps({ "some" : "json" }),
    credentials= "same-origin",
    headers= Object.fromEntries(to_js({ "Content-Type": "application/json" })),
)
```

or:

```
import json
from pyodide.ffi import to_js
```

(continues on next page)

(continued from previous page)

```

from js import Object
resp = await js.fetch('example.com/some_api', to_js({
    "method": "POST",
    "body": json.dumps({ "some" : "json" }),
    "credentials": "same-origin",
    "headers": { "Content-Type": "application/json" }
}), dict_converter=Object.fromEntries)

```

How can I control the behavior of stdin / stdout / stderr?

If you wish to override stdin, stdout or stderr for the entire Pyodide runtime, you can pass options to `loadPyodide()`: If you say

```

loadPyodide({
    stdin: stdin_func,
    stdout: stdout_func,
    stderr: stderr_func,
});

```

then every time a line is written to stdout (resp. stderr), `stdout_func` (resp `stderr_func`) will be called on the line. Every time stdin is read, `stdin_func` will be called with zero arguments. It is expected to return a string which is interpreted as a line of text.

You can also use the functions `pyodide.setStdin()`, `pyodide.setStdout()`, and `pyodide.setStderr()`.

Temporary redirection works much the same as it does in native Python: you can overwrite `sys.stdin`, `sys.stdout`, and `sys.stderr` respectively. If you want to do it temporarily, it's recommended to use `contextlib.redirect_stdout()` and `contextlib.redirect_stderr()` There is no `contextlib.redirect_stdin()` but it is easy to make your own as follows:

```

from contextlib import _RedirectStream
class redirect_stdin(_RedirectStream):
    _stream = "stdin"

```

For example, if you do:

```

from io import StringIO
with redirect_stdin(StringIO("\n".join(["eval", "asyncio.ensure_future", "functools.
↪reduce", "quit"])))):
    help()

```

it will print:

```

Welcome to Python 3.10's help utility!
<...OMITTED LINES>
Help on built-in function eval in module builtins:
eval(source, globals=None, locals=None, /)
    Evaluate the given source in the context of globals and locals.
<...OMITTED LINES>
Help on function ensure_future in asyncio:
asyncio.ensure_future = ensure_future(coro_or_future, *, loop=None)
    Wrap a coroutine or an awaitable in a future.
<...OMITTED LINES>

```

(continues on next page)

(continued from previous page)

Help on built-in function reduce in functools:

```
functools.reduce = reduce(...)
```

```
    reduce(function, sequence[, initial]) -> value
```

Apply a function of two arguments cumulatively to the items of a sequence,

```
<...OMITTED LINES>
```

You are now leaving help and returning to the Python interpreter.

Why can't Micropip find a "pure Python wheel" for a package?

When installing a Python package from PyPI, micropip will produce an error if it cannot find a pure Python wheel. To determine if a package has a pure Python wheel manually, you can open its PyPi page (for instance <https://pypi.org/project/snowballstemmer/>) and go to the "Download files" tab. If this tab doesn't contain a file `*py3-none-any.whl` then the pure Python wheel is missing.

This can happen for two reasons,

1. either the package is pure Python (you can check language composition for a package on Github), and its maintainers didn't upload a wheel. In this case, you can report this issue to the package issue tracker. As a temporary solution, you can also [build the wheel](#) yourself, upload it to some temporary location and install it with micropip from the corresponding URL.
2. or the package has binary extensions (e.g. C, Fortran or Rust), in which case it needs to be packaged in Pyodide. Please open [an issue](#) after checking that an issue for this package doesn't exist already. Then follow [Creating a Pyodide package](#).

How can I change the behavior of `runPython()` and `runPythonAsync()`?

You can directly call Python functions from JavaScript. For most purposes it makes sense to make your own Python function as an entrypoint and call that instead of redefining `runPython`. The definitions of `runPython()` and `runPythonAsync()` are very simple:

```
function runPython(code) {
  pyodide.pyodide_py.code.eval_code(code, pyodide.globals);
}
```

```
async function runPythonAsync(code) {
  return await pyodide.pyodide_py.code.eval_code_async(code, pyodide.globals);
}
```

To make your own version of `runPython()` you could do:

```
const my_eval_code = pyodide.runPython(`
  from pyodide.code import eval_code
  def my_eval_code(code, globals=None, locals=None):
    extra_info = None
    result = eval_code(code, globals, locals)
    return globals["extra_info"], result
  my_eval_code
`)

function myRunPython(code){
```

(continues on next page)

(continued from previous page)

```

    return my_eval_code(code, pyodide.globals);
}

```

Then `myRunPython("2+7")` returns `[None, 9]` and `myRunPython("extra_info='hello' ; 2 + 2")` returns `['hello', 4]`. If you want to change which packages `pyodide.loadPackagesFromImports()` loads, you can monkey patch `pyodide.code.find_imports()` which takes code as an argument and returns a list of packages imported.

Why can't I import a file I just wrote to the file system?

For example:

```

from pathlib import Path
Path("mymodule.py").write_text("""\
def hello():
    print("hello world!")
""")
)
from mymodule import hello # may raise "ModuleNotFoundError: No module named 'mymodule'"
hello()

```

If you see this error, call `importlib.invalidate_caches()` before importing the module:

```

import importlib
from pathlib import Path
Path("mymodule.py").write_text("""\
def hello():
    print("hello world!")
""")
)
importlib.invalidate_caches() # Make sure Python notices the new .py file
from mymodule import hello
hello()

```

Why changes made to IndexedDB don't persist?

Unlike other filesystems, IndexedDB (`pyodide.FS.filesystem.IDBFS`) is an asynchronous filesystem. This is because browsers offer only asynchronous interfaces for IndexedDB. So in order to persist changes, you have to call `pyodide.FS.syncfs()`. See [Emscripten File System API](#) for more details.

How can I access JavaScript objects/attributes in Python if their names are Python keywords?

Some JavaScript objects may have names or attributes which are also [Python Keywords](#), making them difficult to interact with when importing them into Python. For example, all three of the following uses of `runPython` will throw a `SyntaxError`:

```

//The built-in method Array.from() overlaps with Python's "from"
pyodide.runPython(`from js import Array; print(Array.from([1,2,3]))`);

//"global" is a valid attribute name in JS, but a reserved keyword in Python

```

(continues on next page)

(continued from previous page)

```
people = {global: "lots and lots"};
pyodide.runPython(`from js import people; print(people.global)`);

// "lambda" is a valid object name in JS, but a reserved keyword in Python
lambda = (x) => {return x + 1};
pyodide.runPython(`from js import lambda; print(lambda(1))`);
```

For JS objects with attributes that are Python reserved keywords, `getattr()` and `setattr()` can be used to access the attribute by name:

```
pyodide.runPython(`
    from js import Array
    fromFunc = getattr(Array, 'from')
    print(fromFunc([1,2,3]))
`);

people = {global: "lots and lots"};
pyodide.runPython(`
    from js import people
    setattr(people, 'global', 'even more')
    print(getattr(people, 'global'))
`);
```

For objects whose names are keywords, one can similarly use `getattr()` on the `js` module itself:

```
lambda = (x) => {return x + 1};
pyodide.runPython(`
    import js
    js_lambda = getattr(js, 'lambda')
    print(js_lambda(1))
`);
```

3.2 Development

The Development section helps Pyodide contributors to find information about the development process including making packages to support third party libraries.

3.2.1 Building from sources

Warning: If you are building the latest development version of Pyodide from the `main` branch, please make sure to follow the build instructions from the dev version of the documentation at pyodide.org/en/latest/

Pyodide can be built from sources on different platforms,

- on **Linux** it is easiest using the Pyodide Docker image. This approach works with any native operating system as long as Docker is installed. You can also build on your native Linux OS if the correct build prerequisites are installed.

- on **MacOS** it is recommended to install dependencies via conda-forge or using Homebrew, particularly with the M1 ARM CPU. Building with Docker is possible but very slow.
- It is not possible to build on **Windows**, but you can use [Windows Subsystem for Linux](#) to create a Linux build environment.

Build instructions

Using Docker

We provide a Debian-based x86_64 Docker image ([pyodide/pyodide-env](#)) on Docker Hub with the dependencies already installed to make it easier to build Pyodide.

Note: These Docker images are also available from the Github packages at github.com/orgs/pyodide/packages.

1. Install Docker
2. From a git checkout of Pyodide, run `./run_docker`
3. Run `make` to build.

Note: You can control the resources allocated to the build by setting the env vars `EMSDK_NUM_CORE`, `EMCC_CORES` and `PYODIDE_JOBS` (the default for each is 4).

If running `make` deterministically stops at some point, increasing the maximum RAM usage available to the docker container might help. (The RAM available to the container is different from the physical RAM capacity of the machine.) Ideally, at least 3 GB of RAM should be available to the docker container to build Pyodide smoothly. These settings can be changed via Docker preferences (see [here](#)).

You can edit the files in the shared `pyodide` source folder on your host machine (outside of Docker), and then repeatedly run `make` inside the Docker environment to test your changes.

Using make

Make sure the prerequisites for [emsdk](#) are installed. Pyodide will build a custom, patched version of `emsdk`, so there is no need to build it yourself prior.

You need Python 3.11.2 to run the build scripts. To make sure that the correct Python is used during the build it is recommended to use a [virtual environment](#) or a conda environment.

Linux

To build on Linux, you need:

- A working native compiler toolchain, enough to build [CPython](#).
- CMake (required to install Emscripten)

Linux with conda

You would need a working native compiler toolchain, enough to build CPython, for example,

- `apt install build-essential` on Debian based systems.
- Conda which can be installed from [MiniForge](#)

Then install the required Python version and other build dependencies in a separate conda environment,

```
conda env create -f environment.yml
conda activate conda-forge
```

MacOS with conda

You would need,

- System libraries in the root directory: `xcode-select --install`
- Conda which can be installed using [Miniforge](#) (both for Intel and M1 CPU)

Then install the required Python version and other build dependencies in a separate conda environment,

```
conda env create -f environment.yml
conda activate conda-forge
```

MacOS with Homebrew

To build on MacOS with Homebrew, you need:

- System command line tools `xcode-select --install`
- [Homebrew](#) for installing dependencies
- `brew install coreutils cmake autoconf automaker libtool`
- It is also recommended installing the GNU patch and GNU sed (`brew install gpatch gnu-sed`) and re-defining them temporarily as `patch` and `sed`.

Note: If you encounter issues with the requirements, it is useful to check the exact list in the [Dockerfile](#) which is tested in the CI.

You can install the Python dependencies from the requirement file at the root of Pyodide folder: `pip install -r requirements.txt`

After installing the build prerequisites, run from the command line:

```
make
```

Partial builds

To build a subset of available packages in Pyodide, set the environment variable `PYODIDE_PACKAGES` to a comma separated list of packages. For instance,

```
PYODIDE_PACKAGES="toolz,attrs" make
```

Dependencies of the listed packages will be built automatically as well. The package names must match the folder names in `packages/` exactly; in particular they are case-sensitive.

If `PYODIDE_PACKAGES` is not set, a minimal set of packages necessary to run the core test suite is installed, including “micropip”, “pyparsing”, “pytz”, “packaging”, “Jinja2”, “regex”. This is equivalent to setting `PYODIDE_PACKAGES='tag:core'` meta-package. Other supported meta-packages are,

- “tag:min-scipy-stack”: includes the “core” meta-package as well as some core packages from the scientific python stack and their dependencies: “numpy”, “scipy”, “pandas”, “matplotlib”, “scikit-learn”, “joblib”, “pytest”. This option is non exhaustive and is mainly intended to make build faster while testing a diverse set of scientific packages.
- “*” builds all packages
- You can exclude a package by prefixing it with “!”.

micropip and distutils are always automatically included.

Environment variables

The following environment variables additionally impact the build:

- `PYODIDE_JOBS`: the `-j` option passed to the `emmake make` command when applicable for parallel compilation. Default: 3.
- `PYODIDE_BASE_URL`: Base URL where Pyodide packages are deployed. It must end with a trailing `/`. Default: `./` to load Pyodide packages from the same base URL path as where `pyodide.js` is located. Example: `https://cdn.jsdelivr.net/pyodide/v0.23.1/full/`
- `EXTRA_CFLAGS` : Add extra compilation flags.
- `EXTRA_LDFLAGS` : Add extra linker flags.

Setting `EXTRA_CFLAGS="-D DEBUG_F"` provides detailed diagnostic information whenever error branches are taken inside the Pyodide core code. These error messages are frequently helpful even when the problem is a fatal configuration problem and Pyodide cannot even be initialized. These error branches occur also in correctly working code, but they are relatively uncommon so in practice the amount of noise generated isn’t too large. The shorthand `make debug` automatically sets this flag.

In certain cases, setting `EXTRA_LDFLAGS="-s ASSERTIONS=1` or `ASSERTIONS=2` can also be helpful, but this slows down the linking and the runtime speed of Pyodide a lot and generates a large amount of noise in the console.

3.2.2 Creating a Pyodide package

It is recommended to look into how other similar packages are built in Pyodide. If you encounter difficulties in building your package after trying the steps listed here, open a [new Pyodide issue](#).

Determining if creating a Pyodide package is necessary

If you wish to use a package in Pyodide that is not already included in the [packages folder](#), first you need to determine whether it is necessary to package it for Pyodide. Ideally, you should start this process with package dependencies.

Most pure Python packages can be installed directly from PyPI with `micropip.install()` if they have a pure Python wheel. Check if this is the case by trying `micropip.install("package-name")`.

If there is no wheel on PyPI, but you believe there is nothing preventing it (it is a Python package without C extensions):

- you can create the wheel yourself by running

```
python -m pip install build
python -m build
```

from within the package folder where the `setup.py` are located. See the [Python packaging guide](#) for more details. Then upload the wheel file somewhere (not to PyPI) and install it with micropip via its URL.

- please open an issue in the package repository asking the authors to upload the wheel.

If however the package has C extensions or its code requires patching, then continue to the next steps.

Note: To determine if a package has C extensions, check if its `setup.py` contains any compilation commands.

Building Python wheels (out of tree)

Starting with Pyodide 0.22.0, it is now possible to build Python wheels for Pyodide for many packages separately from the Pyodide package tree. See *Building and testing Python packages out of tree* for more details.

Building a Python package (in tree)

This section documents how to add a new package to the Pyodide distribution.

As a starting point, you may want to look at the `meta.yaml` files for some other Pyodide packages in the [packages/ folder](#).

Prerequisites

First clone the Pyodide git repository:

```
git clone https://github.com/pyodide/pyodide
cd pyodide
```

If you have trouble with missing dependencies (or are not running linux) you can use the `pyodide-env` docker container with:

```
./run_docker
```

This will mount the current working directory as `/src` within the container so if you build the package within the container the files created will persist in the directory after you exit the container.

You should install `pyodide-build`:

```
pip install -e pyodide-build
```

If you want to build the package, you will need to build Python which you can do as follows:

```
make -C emsdk
make -C cpython
```

This also builds the appropriate version of Emscripten.

Creating the `meta.yaml` file

To build a Python package in tree, you need to create a `meta.yaml` file that defines a “recipe” which may include build commands and “patches” (source code edits), amongst other things.

If your package is on PyPI, the easiest place to start is with the `pyodide skeleton pypi` command. Run

```
pyodide skeleton pypi <package-name>
```

This will generate a `meta.yaml` file under `packages/<package-name>/` (see [The `meta.yaml` specification](#)). The `pyodide cli` tool will populate the latest version, the download link and the sha256 hash by querying PyPI.

It doesn’t currently handle package dependencies, so you will need to specify those yourself in the `requirements` section of the `meta.yaml` file.

```
requirements:
  host:
    # Dependencies that are needed to build the package
    - cffi
  run:
    # Dependencies that are needed to run the package
    - cffi
    - numpy
```

Note: To determine build and runtime dependencies, including for non Python libraries, it is often useful to check if the package was already built on [conda-forge](#) look at the corresponding `meta.yaml` file. This can be done either by checking if the URL <https://github.com/conda-forge/<package-name>-feedstock/blob/master/recipe/meta.yaml> exists, or by searching the [conda-forge GitHub org](#) for the package name.

The Pyodide `meta.yaml` file format was inspired by the one in conda, however it is not strictly compatible.

Building the package

Once the `meta.yaml` file is ready, build the package with the following command

```
pyodide build-recipes <package-name> --install
```

and see if there are any errors.

Loading the package

If the build succeeds you can try to load the package:

1. Serve the dist directory with `python -m http.server --directory ./dist`
2. Open `localhost:<port>/console.html` and try to import the package
3. You can test the package in the repl

Fixing build issues

If there are errors you might need to add a build script to set You can add extra build commands to the `meta.yaml` like this:

```
build:
  script: |
    wget https://example.com/file.tar.gz
    export MY_ENV_VARIABLE=FOO
```

You can also inject extra compile and link flags with the `cflags` and `ldflags` keys. You can modify the wheel after it is built with the `post:` key.

If you need to patch the package's source to fix build issues, see the section on Generating patches below.

Writing tests for your package

The tests should go in one or more files like `packages/<package-name>/test_xxx.py`. Most packages have one test file named `test_<package-name>.py`. The tests should look like:

```
from pytest_pyodide import run_in_pyodide

@run_in_pyodide(packages=["<package-name>"])
def test_mytestname(selenium):
    import <package-name>
    assert package.do_something() == 5
    # ...
```

If you want to run your package's full pytest test suite and your package vendors tests you can do it like:

```
from pytest_pyodide import run_in_pyodide

@run_in_pyodide(packages=["<package-name>-tests", "pytest"])
def test_mytestname(selenium):
```

(continues on next page)

(continued from previous page)

```
import pytest
pytest.main(["--pyargs", "<package-name>", "-k", "some_filter", ...])
```

you can put whatever command line arguments you would pass to `pytest` as separate entries in the list. For more info on `run_in_pyodide` see [pytest-pyodide](#).

Generating patches

If the package has a git repository, the easiest way to make a patch is usually:

1. Clone the git repository of the package. You might want to use the options `git clone --depth 1 --branch <version>`. Find the appropriate tag given the version of the package you are trying to modify.
2. Make a new branch with `git checkout -b pyodide-version` (e.g., `pyodide-1.21.4`).
3. Make whatever changes you want. Commit them. Please split your changes up into focused commits. Write detailed commit messages! People will read them in the future, particularly when migrating patches or trying to decide if they are no longer needed. The first line of each commit message will also be used in the patch file name.
4. Use `git format-patch <version> -o <pyodide-root>/packages/<package-name>/patches/` to generate a patch file for your changes and store it directly into the patches folder.
5. You also need to add the patches to the `meta.yaml` file:

```
source:
  url: https://files.pythonhosted.org/packages/somehash/some-pkg-1.2.3.tar.gz
  sha256: somehash
patches:
  - 0001-patch-some-thing.patch
  - 0002-patch-some-other-thing.patch
```

The following command will write out the properly formatted file list to use in the patches key:

```
find patches/ -type f | sed 's/^/    - /g'
```

Migrating Patches

When you want to upgrade the version of a package, you will need to migrate the patches. To do this:

1. Clone the git repository of the package. You might want to use the options `git clone --depth 1 --branch <version-tag>`.
2. Make a new branch with `git checkout -b pyodide-old-version` (e.g., `pyodide-1.21.4`).
3. Apply the current patches with `git am <pyodide-root>/packages/<package-name>/patches/*`.
4. Make a new branch `git checkout -b pyodide-new-version` (e.g., `pyodide-1.22.0`).
5. Rebasing the patches with `git rebase old-version --onto new-version` (e.g., `git rebase pyodide-1.21.4 --onto pyodide-1.22.0`). Resolve any rebase conflicts. If a patch has been upstreamed, you can drop it with `git rebase --skip`.
6. Remove old patches with `rm <pyodide-root>/packages/<package-name>/patches/*`.
7. Use `git format-patch <version-tag> -o <pyodide-root>/packages/<package-name>/patches/` to generate new patch files.

Upstream your patches!

Please create PRs or issues to discuss with the package maintainers to try to find ways to include your patches into the package. Many package maintainers are very receptive to including Pyodide-related patches and they reduce future maintenance work for us.

The package build pipeline

Pyodide includes a toolchain to add new third-party Python libraries to the build. We automate the following steps:

- If source is a url (not in-tree):
 - Download a source archive or a pure python wheel (usually from PyPI)
 - Confirm integrity of the package by comparing it to a checksum
 - If building from source (not from a wheel):
 - * Apply patches, if any, to the source distribution
 - * Add extra files, if any, to the source distribution
- If the source is not a wheel (building from a source archive or an in-tree source):
 - Run `build/script` if present
 - Modify the `PATH` to point to wrappers for `gfortran`, `gcc`, `g++`, `ar`, and `ld` that preempt compiler calls, rewrite the arguments, and pass them to the appropriate emscripten compiler tools.
 - Using `pypa/build`:
 - * Create an isolated build environment. Install symbolic links from this isolated environment to “host” copies of certain unisolated packages.
 - * Install the build dependencies requested in the package `build-requires`. (We ignore all version constraints on the unisolated packages, but version constraints on other packages are respected.
 - * Run the [PEP 517](#) build backend associated to the project to generate a wheel.
- Unpack the wheel with `python -m wheel unpack`.
- Run the `build/post` script in the unpacked wheel directory if it's present.
- Unvendor unit tests included in the installation folder to a separate zip file `<package name>-tests.zip`
- Repack the wheel with `python -m wheel pack`

Lastly, a `repodata.json` file is created containing the dependency tree of all packages, so `pyodide.loadPackage()` can load a package's dependencies automatically.

Partial Rebuilds

By default, each time you run `pyodide build-recipes`, it will delete the entire source directory and replace it with a fresh copy from the download url. This is to ensure build repeatability. For debugging purposes, this is likely to be undesirable. If you want to try out a modified source tree, you can pass the flag `--continue` and `build-recipes` will try to build from the existing source tree. This can cause various issues, but if it works it is much more convenient.

Using the `--continue` flag, you can modify the sources in tree to fix the build, then when it works, copy the modified sources into your checked out copy of the package source repository and use `git format-patch` to generate the patch.

C library dependencies

Some Python packages depend on certain C libraries, e.g. `lxml` depends on `libxml`.

To package a C library, create a directory in `packages/` for the C library. In the directory, you should write `meta.yaml` that specifies metadata about the library. See [The meta.yaml specification](#) for more details.

The minimal example of `meta.yaml` for a C library is:

```
package:
  name: <name>
  version: <version>

source:
  url: <url>
  sha256: <sha256>

requirements:
  run:
    - <requirement>

build:
  type: static_library
  script: |
    emconfigure ./configure
    emmake make -j ${PYODIDE_JOBS:-3}
```

You can use the `meta.yaml` of other C libraries such as `libxml` as a starting point.

After packaging a C library, it can be added as a dependency of a Python package like a normal dependency. See `lxml` and `libxml` for an example (and also `scipy` and `CLAPACK`).

Remark: Certain C libraries come as emscripten ports, and do not have to be built manually. They can be used by adding e.g. `-s USE_ZLIB` in the `cflags` of the Python package. See e.g. `matplotlib` for an example. [The full list of libraries with Emscripten ports is here.](#)

Structure of a Pyodide package

Pyodide is obtained by compiling CPython into WebAssembly. As such, it loads packages the same way as CPython — it looks for relevant files `.py` and `.so` files in the directories in `sys.path`. When installing a package, our job is to install our `.py` and `.so` files in the right location in emscripten’s virtual filesystem.

Wheels are just zip archives, and to install them we unzip them into the `site-packages` directory. If there are any `.so` files, we also need to load them at install time: WebAssembly must be loaded asynchronously, but Python imports are synchronous so it is impossible to load `.so` files lazily.

The meta.yaml specification

Packages are defined by writing a `meta.yaml` file. The format of these files is based on the `meta.yaml` files used to build [Conda packages](#), though it is much more limited. The most important limitation is that Pyodide assumes there will only be one version of a given library available, whereas Conda allows the user to specify the versions of each package that they want to install. Despite the limitations, it is recommended to use existing conda package definitions as a starting point to create Pyodide packages. In general, however, one should not expect Conda packages to “just work” with Pyodide, see [#795](#)

This is unstable

The Pyodide build system is under fairly active development (as of 2022/03/13). The next couple of releases are likely to include breaking changes.

The supported keys in the `meta.yaml` file are described below.

`package`

`package/name`

The name of the package. It must match the name of the package used when expanding the tarball, which is sometimes different from the name of the package in the Python namespace when installed. It must also match the name of the directory in which the `meta.yaml` file is placed. It can only contain alphanumeric characters, `-`, and `_`.

`package/version`

The version of the package.

`package/top-level`

The list of top-level import name for the package. This key is used in `pyodide.loadPackagesFromImports()`. For example, the top-level import name for the `scikit-learn` is `sklearn`. Some packages may have multiple top-level import names. For instance, `setuptools` exposes `setuptools` and `pkg_resources` as a top-level import names.

`package/tag`

The list of tags of the package. This is meta information used to group packages by functionality. Normally this is not needed. The following tags are currently used in Pyodide:

- `always`: This package is always built.
- `core`: This package is used in the Pyodide core test suite.
- `min-scipy-stack`: This package is part of the minimal scipy stack.

source**source/url**

The URL of the source tarball.

The tarball may be in any of the formats supported by Python's `shutil.unpack_archive()`: `tar`, `gztar`, `bztar`, `xztar`, and `zip`.

source/extract_dir

The top level directory name of the contents of the source tarball (i.e. once you extract the tarball, all the contents are in the directory named `source/extract_dir`). This defaults to the tarball name (sans extension).

source/path

Alternatively to `source/url`, a relative or absolute path can be specified as package source. This is useful for local testing or building packages which are not available online in the required format.

If a path is specified, any provided checksums are ignored.

source/sha256

The SHA256 checksum of the tarball. It is recommended to use SHA256 instead of MD5. At most one checksum entry should be provided per package.

source/patches

A list of patch files to apply after expanding the tarball. These are applied using `patch -p1` from the root of the source tree.

source/extras

Extra files to add to the source tree. This should be a list where each entry is a pair of the form `(src, dst)`. The `src` path is relative to the directory in which the `meta.yaml` file resides. The `dst` path is relative to the root of source tree (the expanded tarball).

build**build/cflags**

Extra arguments to pass to the compiler when building for WebAssembly.

(This key is not in the Conda spec).

build/cxxflags

Extra arguments to pass to the compiler when building C++ files for WebAssembly. Note that both `cflags` and `cxxflags` will be used when compiling C++ files. A common example would be to use `-std=c++11` for code that makes use of C++11 features.

(This key is not in the Conda spec).

build/ldflags

Extra arguments to pass to the linker when building for WebAssembly.

(This key is not in the Conda spec).

build/exports

Which symbols should be exported from the shared object files. Possible values are:

- `pyinit`: The default. Only export Python module initialization symbols of the form `PyInit_some_module`.
- `requested`: Export the functions that are marked as exported in the object files. Switch to this if `pyinit` doesn't work. Useful for packages that use `ctypes` or `dlsym` to access symbols.
- `whole_archive`: Uses `-Wl,--whole-archive` to force inclusion of all symbols. Use this when neither `pyinit` nor `explicit` work.

build/backend-flags

Extra flags to pass to the build backend (e.g., `setuptools`, `flit`, etc).

build/type

Type of the package. Possible values are:

- `package` (default): A normal Python package, built to a wheel file.
- `static_library`: A static library.
- `shared_library`: A shared library.
- `cpython_module`: A CPython stdlib extension module. This is used for unvendoring CPython modules, and should not be used for other purposes.

If you are building ordinary Python package, you don't need to set this key. But if you are building a static or shared library, you need to set this to `static_library` or `shared_library` respectively.

Static and shared libraries are not Python packages themselves, but are needed for other python packages. For libraries, the script specified in the `build/script` section is run to compile the library.

The difference between `static_library` and `shared_library` is that `static_library` is statically linked into the other packages, so it is required only in the build time, while `shared_library` is dynamically linked, so it is required in the runtime. When building a shared library, you should copy the built libraries into the `$DISTDIR`. Files or folders in this folder will be packaged to make the Pyodide package.

See the [zlib meta.yaml](#) for an example of a static library specification, and the [CLAPACK meta.yaml](#) for an example of a shared library specification.

build/script

The script section is required for a library package (build/library set to true). For a Python package this section is optional. If it is specified for a Python package, the script section will be run before the build system runs `setup.py`. This script is run by `bash` in the directory where the tarball was extracted.

There are special environment variables defined:

- `$PKGDIR`: The directory in which the `meta.yaml` file resides.
- `$PKG_VERSION`: The version of the package
- `$PKG_BUILD_DIR`: The directory where the tarball was extracted.
- `$DISTDIR`: The directory where the built wheel or library should be placed. If you are building a shared library, you should copy the built libraries into this directory.

(These keys are not in the Conda spec).

build/cross-script

This script will run *after* `build/script`. The difference is that it runs with the target environment variables and `sysconfigdata` and with the `pywasmcross` compiler symlinks. Any changes to the environment will persist to the main build step but will not be seen in the `build/post` step (or anything else done outside of the cross build environment). The working directory for this script is the source directory.

build/post

Shell commands to run after building the library. This script is run by `bash` in the directory where `meta.yaml` file resides. The `${PKG_BUILD_DIR}/dist` will contain the built wheel unpacked with `python -m wheel unpack` so it's possible to manually add, delete, change, move files etc. See the [setuptools meta.yaml](https://github.com/pyodide/pyodide/blob/main/packages/setuptools/meta.yaml) for an example of the usage of this key.

build/unvendor-tests

Whether to unvendor tests found in the installation folder to a separate package `<package-name>-tests`. If this option is true and no tests are found, the test package will not be created. Default: true.

build/vendor-sharedlib

If set to true, shared libraries that are required by the package will be vendored into the package after the build. This is similar to what `auditwheel repair` does, but it is done in a way that is compatible with Pyodide and Emscripten dynamic linking. Default: false.

requirements

requirements/run

A list of required packages at runtime.

(Unlike conda, this only supports package names, not versions).

requirements/host

A list of Pyodide packages that are required when building a package. It represents packages that need to be specific to the target platform.

For instance, when building `libxml`, `zlib` needs to be built for WASM first, and so it's a host dependency. This is unrelated to the fact that the build system might already have `zlib` present.

requirements/executable

A list of executables that are required when building a package.

Note that unlike conda, specifying executables in this key doesn't actually install any of them. This key exists to halt build earlier if required executables are not available.

test

test/imports

List of imports to test after the package is built.

Supported Environment Variables

The following environment variables can be used in the scripts in the `meta.yaml` files:

- `PYODIDE_ROOT`: The path to the base Pyodide directory
- `PYMAJOR`: Current major Python version
- `PYMINOR`: Current minor Python version
- `PYMICRO`: Current micro Python version
- `SIDE_MODULE_CFLAGS`: The standard CFLAGS for a side module. Use when compiling libraries or shared libraries.
- `SIDE_MODULE_LDFLAGS`: The standard LDFLAGS for a side module. Use when linking a shared library.
- `NUMPY_LIB`: Use `-L$NUMPY_LIB` as a ldflag when linking `-lnpymath` or `-lnpyrandom`.

Rust/PyO3 Packages

We currently build cryptography which is a Rust extension built with PyO3 and `setuptools-rust`. It should be reasonably easy to build other Rust extensions. If you want to build a package with Rust extension, you will need Rust ≥ 1.41 , and you need to set the rustup toolchain to `nightly`, and the target to `wasm32-unknown-emscrip` in the build script as shown [here](#), but other than that there may be no other issues if you are lucky.

As mentioned [here](#), by default certain wasm-related RUSTFLAGS are set during `build.script` and can be removed with `export RUSTFLAGS=""`.

If your project builds using maturin, you need to use maturin 0.14.14 or later. It is pretty easy to patch an existing project (see `projects/fastparquet/meta.yaml` for an example)

3.2.3 Building and testing Python packages out of tree

This is some information about how to build and test Python packages against Pyodide out of tree (for instance in your package's CI or for use with private packages).

Pyodide currently only supports Linux for out of tree builds, though there is a good change it will work in MacOS too. If you are using Windows, try Windows Subsystem for Linux.

Building binary packages for Pyodide

If your package is a pure Python package (i.e., if the wheel ends in `py3-none-any.whl`) then follow the official PyPA documentation on building [wheels](#) Otherwise, the procedure is as follows.

Install pyodide-build

```
pip install pyodide-build
```

Set up Emscripten

You need to download the Emscripten developer toolkit:

```
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
```

then you can install the appropriate version of Emscripten:

```
PYODIDE_EMSCRIPTEN_VERSION=$(pyodide config get emscripten_version)
./emsdk install ${PYODIDE_EMSCRIPTEN_VERSION}
./emsdk activate ${PYODIDE_EMSCRIPTEN_VERSION}
source emsdk_env.sh
```

If you restart your shell, you will need to run `source emsdk_env.sh` again.

Build the WASM/Emscripten wheel

Change directory into the package folder where the `setup.py` or `pyproject.toml` file is located. You should be in a shell session where you ran `source emsdk_env.sh`. Then run

```
pyodide build
```

in the package folder. This command produces a wheel in the `dist/` folder, similarly to the [PyPA build](#) command.

If you need to add custom compiler / linker flags to the compiler invocations, you can set the `CFLAGS`, `CXXFLAGS` and `LDFLAGS` environment variables. For instance, to make a debug build, you can use: `CFLAGS=-g2 LDFLAGS=g2 pyodide build`.

`pyodide build` invokes a slightly modified version of the `pypa/build` build frontend so the behavior should be similar to what happens if you do:

```
pip install build
python -m build
```

If you run into problems, make sure that building a native wheel with `pypa/build` works. If it does, then please open an issue about it.

Serve the wheel

Serve the wheel via a file server e.g., `python3.10 -m http.server --directory dist`. Then you can install it with `pyodide.loadPackage` or `micropip.install` by URL.

Notes

- the resulting package wheels have a file name of the form `*-cp310-cp310-emscripten_3_1_27_wasm32.whl` and are compatible only for a given Python and Emscripten versions. In the Pyodide distribution, Python and Emscripten are updated simultaneously.
- for now, PyPi does not support emscripten/wasm32 wheels so you will not be able to upload them there.

Testing packages against Pyodide

Pyodide provides an experimental command line runner for testing packages against Pyodide. Using it requires nodejs version 14 or newer.

The way it works is simple: you can create a virtual environment with:

```
pyodide venv .venv-pyodide
```

Activate it just like a normal virtual environment:

```
source .venv-pyodide/bin/activate
```

As a warning, things are pretty weird inside of the Pyodide virtual environment because `python` points to the Pyodide Python runtime. Any program that uses Python and is sensitive to the current virtual environment will probably break.

You can install whatever dependencies you need with `pip`. For a pure Python package, the following will work:


```
pip install -e .
```

For a binary package, you will need to build a wheel with `pyodide build` and then point `pip` directly to the built wheel. For now, editable installs won't work with binary packages.

```
# Build the binary package
pyodide build
# Install it
pip install dist/the_wheel-cp310-cp310-emscripten_3_1_20_wasm32.whl[tests]
```

To test, you can generally run the same script as you would usually do. For many packages this will be:

```
python -m pytest
```

but for instance `numpy` uses a file called `runtests.py`; the following works:

```
python runtests.py
```

and you can pass options to it just like normal. Currently `subprocess` doesn't work, so if you have a test runner that uses `subprocess` then it cannot be used.

Build Github actions example

Here is a complete example of a Github Actions workflow for building a Python wheel out of tree:

```
runs-on: ubuntu-22.04
steps:
- uses: actions/checkout@v3
- uses: actions/setup-python@v4
  with:
    python-version: 3.11.2
- run: |
    pip install pyodide-build>=0.23.0
    echo EMSCRIPTEN_VERSION=$(pyodide config get emscripten_version) >> $GITHUB_ENV
- uses: mymindstorm/setup-emsdk@v12
  with:
    version: ${{ env.EMSCRIPTEN_VERSION }}
- run: pyodide build
```

For an example “in the wild” of a github action to build and test a wheel against Pyodide, see [the numpy CI](#)

3.2.4 How to Contribute

Thank you for your interest in contributing to Pyodide! There are many ways to contribute, and we appreciate all of them. Here are some guidelines & pointers for diving into it.

Development Workflow

To contribute code, see the following steps,

1. Fork the Pyodide repository <https://github.com/pyodide/pyodide> on Github.
2. If you are on Linux, you can skip this step. On Windows and MacOS you have a choice. The first option is to manually install Docker:
 - on MacOS follow [these instructions](#)
 - on Windows, [install WSL 2](#), then Docker. Note that Windows filesystem access from WSL2 is very slow and should be avoided when building Pyodide.

The second option is to use a service that provides a Linux development environment, such as

- [Github Codespaces](#)
- [gitpod.io](#)
- or a remote Linux VM with SSH connection.

3. Clone your fork of Pyodide

```
git clone https://github.com/<your-username>/pyodide.git
```

and add the upstream remote,

```
git remote add upstream https://github.com/pyodide/pyodide.git
```

4. While the build will happen inside Docker you still need a development environment with Python 3.10 and ideally Node.js. These can be installed for instance with,

```
conda create -c conda-forge -n pyodide-env python=3.10.2 nodejs
conda activate pyodide-env
```

or via your system package manager.

5. Install requirements (it's recommended to use a virtualenv or a conda env),

```
pip install -r requirements.txt
```

6. Enable [pre-commit](#) for code style,

```
pre-commit install
```

This will run a set of linters for each commit.

7. Follow [Building from sources](#) instructions.
8. See [Testing and benchmarking](#) documentation.

Code of Conduct

Pyodide has adopted a *Code of Conduct* that we expect all contributors and core members to adhere to.

Development

Work on Pyodide happens on GitHub. Core members and contributors can make Pull Requests to fix issues and add features, which all go through the same review process. We'll detail how you can start making PRs below.

We'll do our best to keep `main` in a non-breaking state, ideally with tests always passing. The unfortunate reality of software development is sometimes things break. As such, `main` cannot be expected to remain reliable at all times. We recommend using the latest stable version of Pyodide.

Pyodide follows [semantic versioning](#) - major versions for breaking changes (x.0.0), minor versions for new features (0.x.0), and patches for bug fixes (0.0.x).

We keep a file, [docs/changelog.md](#), outlining changes to Pyodide in each release. We like to think of the audience for changelogs as non-developers who primarily run the latest stable. So the change log will primarily outline user-visible changes such as new features and deprecations, and will exclude things that might otherwise be inconsequential to the end user experience, such as infrastructure or refactoring.

Bugs & Issues

We use [Github Issues](#) for announcing and discussing bugs and features. Use [this link](#) to report a bug or issue. We provide a template to give you a guide for how to file optimally. If you have the chance, please search the existing issues before reporting a bug. It's possible that someone else has already reported your error. This doesn't always work, and sometimes it's hard to know what to search for, so consider this extra credit. We won't mind if you accidentally file a duplicate report.

Core contributors are monitoring new issues & comments all the time, and will label & organize issues to align with development priorities.

How to Contribute

Pull requests are the primary mechanism we use to change Pyodide. GitHub itself has some [great documentation](#) on using the Pull Request feature. We use the “fork and pull” model [described here](#), where contributors push changes to their personal fork and create pull requests to bring those changes into the source repository.

Please make pull requests against the `main` branch.

If you're looking for a way to jump in and contribute, our list of [good first issues](#) is a great place to start.

If you'd like to fix a currently-filed issue, please take a look at the comment thread on the issue to ensure no one is already working on it. If no one has claimed the issue, make a comment stating you'd like to tackle it in a PR. If someone has claimed the issue but has not worked on it in a few weeks, make a comment asking if you can take over, and we'll figure it out from there.

We use [pytest](#), driving [Selenium](#) as our testing framework. Every PR will automatically run through our tests, and our test framework will alert you on GitHub if your PR doesn't pass all of them. If your PR fails a test, try to figure out whether or not you can update your code to make the test pass again, or ask for help. As a policy we will not accept a PR that fails any of our tests, and will likely ask you to add tests if your PR adds new functionality. Writing tests can be scary, but they make open-source contributions easier for everyone to assess. Take a moment and look through how we've written our tests, and try to make your tests match. If you are having trouble, we can help you get started on our test-writing journey.

All code submissions should pass `make lint`. Python is checked with `flake8`, `black` and `mypy`. JavaScript is checked with `prettier`. C is checked against the Mozilla style in `clang-format`.

Contributing to the “core” C Code

See *Contributing to the “core” C Code*.

Documentation

Documentation is a critical part of any open source project, and we are very welcome to any documentation improvements. Pyodide has a documentation written in Markdown in the `docs/` folder. We use the [MyST](#) for parsing Markdown in `sphinx`. You may want to have a look at the [MyST syntax guide](#) when contributing, in particular regarding [cross-referencing sections](#).

Building the docs

From the directory `docs`, first install the Python dependencies with `pip install -r requirements-doc.txt`. You also need to install `JsDoc`, which is a node dependency. Install it with `sudo npm install -g jsdoc`. Then to build the docs run `make html`. The built documentation will be in the subdirectory `docs/_build/html`. To view them, `cd` into `_build/html` and start a file server, for instance `http-server`.

Migrating patches

It often happens that patches need to be migrated between different versions of upstream packages.

If patches fail to apply automatically, one solution can be to

1. Checkout the initial version of the upstream package in a separate repo, and create a branch from it.
2. Add existing patches with `git apply <path.path>`
3. Checkout the new version of the upstream package and create a branch from it.
4. Cherry-pick patches to the new version,

```
git cherry-pick <commit-hash>
```

and resolve conflicts.

5. Re-export last `N` commits as patches e.g.

```
git format-patch -<N> -N --no-stat HEAD -o <out_dir>
```

Maintainer information

For information about making releases see [Maintainer information](#).

License

All contributions to Pyodide will be licensed under the [Mozilla Public License 2.0 \(MPL 2.0\)](#). This is considered a “weak copyleft” license. Check out the [tldrLegal](#) entry for more information, as well as Mozilla’s [MPL 2.0 FAQ](#) if you need further clarification on what is and isn’t permitted.

Get in Touch

- **Gitter:** [#pyodide](#) channel at gitter.im

Contributing to the “core” C Code

This file is intended as guidelines to help contributors trying to modify the C source files in `src/core`.

What the files do

The primary purpose of `core` is to implement *type translations* between Python and JavaScript. Here is a breakdown of the purposes of the files.

- `main` – responsible for configuring and initializing the Python interpreter, initializing the other source files, and creating the `_pyodide_core` module which is used to expose Python objects to `pyodide_py`. `main.c` also tries to generate fatal initialization error messages to help with debugging when there is a mistake in the initialization code.
- `keyboard_interrupt` – This sets up the keyboard interrupts system for using Pyodide with a webworker.

Backend utilities

- `hiwire` – A helper framework. It is impossible for `wasn` to directly hold owning references to JavaScript objects. The primary purpose of `hiwire` is to act as a surrogate owner for JavaScript references by holding the references in a JavaScript Map. `hiwire` also defines a wide variety of `EM_JS` helper functions to do JavaScript operations on the held objects. The primary type that `hiwire` exports is `JsRef`. References are created with `Hiwire.new_value` (only can be done from JavaScript) and must be destroyed from C with `hiwire_decref` or `hiwire_CLEAR`, or from JavaScript with `Hiwire.decref`.
- `error_handling` – defines macros useful for error propagation and for adapting JavaScript functions to the CPython calling convention. See more in the *[Error Handling Macros](#)* section.

Type conversion from JavaScript to Python

- `js2python` – Translates basic types from JavaScript to Python, leaves more complicated stuff to `jsproxy`.
- `jsproxy` – Defines Python classes to proxy complex JavaScript types into Python. A complex file responsible for many of the core behaviors of Pyodide.

Type conversion from Python to JavaScript

- `python2js` – Translates types from Python to JavaScript, implicitly converting basic types and creating proxies for others. It also implements explicit conversion from Python to JavaScript (the `toJs` method).
- `python2js_buffer` – Attempts to convert Python objects that implement the Python [Buffer Protocol](#). This includes bytes objects, `memoryviews`, `array.array` and a wide variety of types exposed by extension modules like `numpy`. If the data is a 1d array in a contiguous block it can be sliced directly out of the wasm heap to produce a JavaScript `TypedArray`, but JavaScript does not have native support for pointers, so higher dimensional arrays are more complicated.
- `pyproxy` – Defines a JavaScript Proxy object that passes calls through to a Python object. Another important core file, `PyProxy.apply` is the primary entrypoint into Python code. `pyproxy.c` is much simpler than `jsproxy.c` though.

CPython APIs

Conventions for indicating errors

The two main ways to indicate errors:

1. If the function returns a pointer, (most often `PyObject*`, `char*`, or `const char*`) then to indicate an error set an exception and return `NULL`.
2. If the function returns `int` or `float` and a correct output must be nonnegative, to indicate an error set an exception and return `-1`.

Certain functions have “successful errors” like `PyIter_Next` (successful error is `StopIteration`) and `PyDict_GetItemWithError` (successful error is `KeyError`). These functions will return `NULL` without setting an exception to indicate the “successful error” occurred. Check what happened with `PyErr_Occurred`. Also, functions that return `int` for which `-1` is a valid return value will return `-1` with no error set to indicate that the result is `-1` and `-1` with an error set if an error did occur. The simplest way to handle this is to always check `PyErr_Occurred`.

Lastly, the argument parsing functions `PyArg_ParseTuple`, `PyArg_Parse`, etc are edge cases. These return `true` on success and return `false` and set an error on failure.

Python APIs to avoid:

- `PyDict_GetItem`, `PyDict_GetItemString`, and `_PyDict_GetItemId` These APIs do not do correct error reporting and there is talk in the Python community of deprecating them going forward. Instead, use `PyDict_GetItemWithError` and `_PyDict_GetItemIdWithError` (there is no `PyDict_GetItemStringWithError` API because use of `GetXString` APIs is also discouraged).
- `PyObject_HasAttrString`, `PyObject_GetAttrString`, `PyDict_GetItemString`, `PyDict_SetItemString`, `PyMapping_HasKeyString` etc, etc. These APIs cause wasteful repeated string conversion. If the string you are using is a constant, e.g., `PyDict_GetItemString(dict, "identifier")`, then make an id with `Py_Identifier(identifier)` and then use `_PyDict_GetItemId(&PyId_identifier)`. If the string is not constant, convert it to a Python object with `PyUnicode_FromString()` and then use e.g., `PyDict_GetItem`.
- `PyModule_AddObject`. This steals a reference on success but not on failure and requires unique cleanup code. Instead, use `PyObject_SetAttr`.

Error Handling Macros

The file `error_handling.h` defines several macros to help make error handling as simple and uniform as possible.

Error Propagation Macros

In a language with exception handling as a feature, error propagation requires no explicit code, it is only if you want to prevent an error from propagating that you use a `try/catch` block. On the other hand, in C all error propagation must be done explicitly.

We define macros to help make error propagation look as simple and uniform as possible. They can only be used in a function with a `finally:` label which should handle resource cleanup for both the success branch and all the failing branches (see structure of functions section below). When compiled with `DEBUG_F`, these commands will write a message to `console.error` reporting the line, function, and file where the error occurred.

- `FAIL()` – unconditionally `goto finally;`
- `FAIL_IF_NULL(ptr)` – `goto finally;` if `ptr == NULL`. This should be used with any function that returns a pointer and follows the standard Python calling convention.
- `FAIL_IF_MINUS_ONE(num)` – `goto finally;` if `num == -1`. This should be used with any function that returns a number and follows the standard Python calling convention.
- `FAIL_IF_NONZERO(num)` – `goto finally;` if `num != 0`. Can be used with functions that return any nonzero error code on failure.
- `FAIL_IF_ERR_OCCURRED()` – `goto finally;` if the Python error indicator is set (in other words if `PyErr_Occurred()`).
- `FAIL_IF_ERR_MATCHES(python_err_type)` – `goto finally;` if `PyErr_ExceptionMatches(python_err_type)`, for example `FAIL_IF_ERR_MATCHES(PyExc_AttributeError);`

JavaScript to CPython calling convention adaptors

If we call a JavaScript function from C and that JavaScript function throws an error, it is impossible to catch it in C. We define two `EM_JS` adaptors to convert from the JavaScript calling convention to the CPython calling convention. The point of this is to ensure that errors that occur in `EM_JS` functions can be handled in C code using the `FAIL_*` macros. When compiled with `DEBUG_F`, when a JavaScript error is thrown a message will also be written to `console.error`. The wrappers do roughly the following:

```
try {
  // body of function here
} catch (e) {
  // wrap e in a Python exception and set the Python error indicator
  // return error code
}
```

There are two variants: `EM_JS_NUM` returns `-1` as the error code, `EM_JS_REF` returns `NULL == 0` as the error code. A couple of simple examples: Use `EM_JS_REF` when return value is a `JsRef`:

```
EM_JS_REF(JsRef, hiwire_call, (JsRef idfunc, JsRef idargs), {
  let jsfunc = Hiwire.get_value(idfunc);
  let jsargs = Hiwire.get_value(idargs);
  return Hiwire.new_value(jsfunc(... jsargs));
});
```

Use `EM_JS_REF` when return value is a `PyObject`:

```
EM_JS_REF(PyObject*, __js2python, (JsRef id), {  
    // body here  
});
```

If the function returns `void`, use `EM_JS_NUM` with return type `errcode`. `errcode` is a typedef for `int`. `EM_JS_NUM` will automatically return `-1` if an error occurs and `0` if not:

```
EM_JS_NUM(errcode, hiwire_set_member_int, (JsRef idobj, int idx, JsRef idval), {  
    Hiwire.get_value(idobj)[idx] = Hiwire.get_value(idval);  
});
```

If the function returns `int` or `bool` use `EM_JS_NUM`:

```
EM_JS_NUM(int, hiwire_get_length, (JsRef idobj), {  
    return Hiwire.get_value(idobj).length;  
});
```

These wrappers enable the following sort of code:

```
try:  
    jsfunc()  
except JsException:  
    print("Caught an exception thrown in JavaScript!")
```

Structure of functions

In C it takes special care to correctly and cleanly handle both reference counting and exception propagation. In Python (or other higher level languages), all references are released in an implicit finally block at the end of the function. Implicitly, it is as if you wrote:

```
def f():  
    try: # implicit  
        a = do_something()  
        b = do_something_else()  
        c = a + b  
        return some_func(c)  
    finally:  
        # implicit, free references both on successful exit and on exception  
        decref(a)  
        decref(b)  
        decref(c)
```

Freeing all references at the end of the function allows us to separate reference counting boilerplate from the “actual logic” of the function definition. When a function does correct error propagation, there will be many different execution paths, roughly linearly many in the length of the function. For example, the above pseudocode could exit in five different ways: `do_something` could raise an exception, `do_something_else` could raise an exception, `a + b` could raise an exception, `some_func` could raise an exception, or the function could return successfully. (Even a Python function like `def f(a,b,c,d): return (a + b) * c - d` has four execution paths.) The point of the `try/finally` block is that we know the resources are freed correctly without checking once for each execution path.

To do this, we divide any function that produces more than a couple of owned `PyObject`*s or `JsRefs` into several “segments”. The more owned references there are in a function and the longer it is, the more important it becomes

to follow this style carefully. By being as consistent as possible, we reduce the burden on people reading the code to double-check that you are not leaking memory or errors. In short functions it is fine to do something ad hoc.

1. The guard block. The first block of a function does sanity checks on the inputs and argument parsing, but only to the extent possible without creating any owned references. If you check more complicated invariants on the inputs in a way that requires creating owned references, this logic belongs in the body block.

Here's an example of a METH_VARARGS function:

```
PyObject*
JsImport_CreateModule(PyObject* self, PyObject* args)
{
    // Guard
    PyObject* name;
    PyObject* jsproxy;
    // PyArg_UnpackTuple uses an unusual calling convention:
    // It returns `false` on failure...
    if (!PyArg_UnpackTuple(args, "create_module", 2, 2, &spec, &jsproxy)) {
        return NULL;
    }
    if (!JsProxy_Check(jsproxy)) {
        PyErr_SetString(PyExc_TypeError, "package is not an instance of jsproxy");
        return NULL;
    }
}
```

2. Forward declaration of owned references. This starts by declaring a success flag `bool success = false`. This will be used in the finally block to decide whether the finally block was entered after a successful execution or after an error. Then declare every reference counted variable that we will create during execution of the function. Finally, declare the variable that we are planning to return. Typically, this will be called `result`, but in this case the function is named `CreateModule` so we name the return variable `module`.

```
bool success = false;
// Note: these are all the objects that we will own. If a function returns
// a borrow, we XINCREf the result so that we can CLEAR it in the finally block.
// Reference counting is hard, so it's good to be as explicit and consistent
// as possible!
PyObject* sys_modules = NULL;
PyObject* importlib_machinery = NULL;
PyObject* ModuleSpec = NULL;
PyObject* spec = NULL;
PyObject* __dir__ = NULL;
PyObject* module_dict = NULL;
// result
PyObject* module = NULL;
```

3. The body of the function. The vast majority of API calls can return error codes. You MUST check every fallible API for an error. Also, as you are writing the code, you should look up every Python API you use that returns a reference to determine whether it returns a borrowed reference or a new one. If it returns a borrowed reference, immediately `Py_XINCREf()` the result to convert it into an owned reference (before `FAIL_IF_NULL`, to be consistent with the case where you use custom error handling).

```
name = PyUnicode_FromString(name_utf8);
FAIL_IF_NULL(name);
sys_modules = PyImport_GetModuleDict(); // returns borrow
Py_XINCREf(sys_modules);
```

(continues on next page)

(continued from previous page)

```

FAIL_IF_NULL(sys_modules);
module = PyDict_GetItemWithError(sys_modules, name); // returns borrow
Py_XINCREF(module);
FAIL_IF_NULL(module);
if(module && !JsImport_Check(module)){
    PyErr_Format(PyExc_KeyError,
        "Cannot mount with name '%s': there is an existing module by this name that was
↳not mounted with 'pyodide.mountPackage'."
        , name
    );
    FAIL();
}
// ... [SNIP]

```

4. The finally block. Here we will clear all the variables we declared at the top in exactly the same order. Do not clear the arguments! They are borrowed. According to the standard Python function calling convention, they are the responsibility of the calling code.

```

success = true;
finally:
    Py_CLEAR(sys_modules);
    Py_CLEAR(importlib_machinery);
    Py_CLEAR(ModuleSpec);
    Py_CLEAR(spec);
    Py_CLEAR(__dir__);
    Py_CLEAR(module_dict);
    if(!success){
        Py_CLEAR(result);
    }
    return result;
}

```

One case where you do need to `Py_CLEAR` a variable in the body of a function is if that variable is allocated in a loop:

```

// refcounted variable declarations
PyObject* pyentry = NULL;
// ... other stuff
Py_ssize_t n = PySequence_Length(pylist);
for (Py_ssize_t i = 0; i < n; i++) {
    pyentry = PySequence_GetItem(pydir, i);
    FAIL_IF_MINUS_ONE(do_something(pyentry));
    Py_CLEAR(pyentry); // important to use Py_CLEAR and not Py_decref.
}

success = true
finally:
    // have to clear pyentry at end too in case do_something failed in the loop body
    Py_CLEAR(pyentry);

```

Testing

Any nonstatic C function called `some_name` defined not using `EM_JS` will be exposed as `pyodide._module._some_name`, and this can be used in tests to good effect. If the arguments / return value are not just numbers and booleans, it may take some effort to set up the function call.

If you want to test an `EM_JS` function, consider moving the body of the function to an API defined on `Module`. You should still wrap the function with `EM_JS_REF` or `EM_JS_NUM` in order to get a function with the CPython calling convention.

Maintainer information

Making a release

For branch organization we use a variation of the [GitHub Flow](#) with the latest release branch named `stable` (due to ReadTheDocs constraints).

Release Instructions

1. From the root directory of the repository run

```
./tools/bump_version.py --new-version <new_version>
# ./tools/bump_version.py --new_version <new_version> --dry-run
```

and check that the diff is correct with `git diff`. Try using `ripgrep` to make sure there are no extra old versions lying around e.g., `rg -F "0.18"`, `rg -F dev0`, `rg -F dev.0`.

2. Make sure the change log is up-to-date. (Skip for alpha releases.)

- Indicate the release date in the change log.
- Generate the list of contributors for the release at the end of the changelog entry with,

```
git shortlog -s LAST_TAG.. | cut -f2- | grep -v '\[bot\]' | sort --ignore-case
↪ | tr '\n' ';' | sed 's/;/, /g;s/, $//' | fold -s
```

where `LAST_TAG` is the tag for the last release.

3. Make a PR with the updates from steps 1 and 2. Merge the PR.
4. (Major release only.) Assuming the upstream `stable` branch exists, rename it to a release branch for the previous major version. For instance if last release was, `0.20.0`, the corresponding release branch would be `0.20.X`,

```
git fetch upstream
git checkout stable
git checkout -b 0.20.X
git push upstream 0.20.X
git branch -D stable    # delete locally
```

5. Create a tag `X.Y.Z` (without leading `v`) and push it to upstream,

```
git tag X.Y.Z
git push upstream X.Y.Z
```

Wait for the CI to pass and create the release on GitHub.

6. (Major release only). Create a new `stable` branch from this tag,

```
git checkout -b stable
git push upstream stable --force
```

7. Revert the release commit. If making a major release, increment the version to the next development version specified by Semantic Versioning.

```
# If you just released 0.22.0, then set the next version to 0.23.0
./tools/bump_version.py --new-version 0.23.0.dev0
```

8. Update these instructions with any relevant changes.

Making a minor release

For a minor release, commits need to be added to the `stable` branch, ideally via a PR. This can be done with either,

- `git cherry picking` individual commits,

```
git checkout stable
git pull
git checkout -b backport-branch
git cherry-pick <commit-hash>
```

- or with interactive rebase,

```
git fetch upstream
git checkout stable
git pull
git checkout -b backport-branch
git rebase -i upstream/main
```

and indicate which commits to take from `main` in the UI.

Then follow the relevant steps from [Release Instructions](#).

Making an alpha release

Name the first alpha release `x.x.xa1` and in subsequent alphas increment the final number. Follow the relevant steps from [Release Instructions](#).

Fixing documentation for a released version

Cherry pick the corresponding documentation commits to the `stable` branch. Use `[skip ci]` in the commit message.

Upgrading pyodide to a new version of CPython

For example: v3.11.1 -> v3.11.2

1. Download the **Gzipped source tarball** at <https://www.python.org/downloads/release/python-3112> into `downloads/`
2. `shasum -a 256 downloads/Python-3.11.2.tgz > cpython/checksums`
3. `git grep --name-only "3.11.1" # All these files will need to be updated.`
4. After updating the Python version in `Dockerfile`, create a new Docker image.
 - A maintainer must click `Run workflow` on https://github.com/pyodide/pyodide/actions/workflows/docker_image.yml
5. That workflow will build and upload a new Docker image to <https://hub.docker.com/r/pyodide/pyodide-env/tags>
6. Modify the image name in `.circleci/config.yml` to match the image tag on Docker Hub.
 - `image: pyodide/pyodide-env:20230301-chrome109-firefox109-py311`
7. Modify the `PYODIDE_IMAGE_TAG` in `run_docker` to match the image tag on Docker Hub.
 - `PYODIDE_IMAGE_TAG="20230301-chrome109-firefox109-py311"`
8. Rebase any patches which do not apply cleanly.
9. Create a pull request and fix any failing tests. This may be complicated for major releases of CPython.

3.2.5 Testing and benchmarking

Testing

Running the Python test suite

1. Install the following dependencies into the default Python installation:

```
pip install pytest-pyodide pytest-httpserver
```

`pytest-pyodide` is a `pytest` plugin for testing Pyodide and third-party applications that use Pyodide.

See: [pytest-pyodide](#) for more information.

2. Install [geckodriver](#) or [chromedriver](#) and check that they are in your `PATH`.
3. To run the test suite, run `pytest` from the root directory of Pyodide:

```
pytest
```

There are 3 test locations that are collected by `pytest`,

- `src/tests/`: general Pyodide tests and tests running the CPython test suite
- `pyodide-build/pyodide_build/tests/`: tests related to Pyodide build system (do not require selenium or playwright to run)
- `packages/*/test_*`: package specific tests.

You can run the tests from a specific file with:

```
pytest path/to/test/file.py
```

Some browsers sometimes produce informative errors than others so if you are getting confusing errors it is worth rerunning the test on each browser. You can use `--runtime` commandline option to specify the browser runtime.

```
pytest --runtime firefox
pytest --runtime chrome
pytest --runtime node
```

Custom test marks

We support custom test marks:

`@pytest.mark.skip_refcount_check` and `pytest.mark.skip_pyproxy_check` disable respectively the check for JavaScript references and the check for PyProxies. If a test creates JavaScript references or PyProxies and does not clean them up, by default the tests will fail. If a test is known to leak objects, it is possible to disable these checks with these markers.

Running the JavaScript test suite

To run tests on the JavaScript Pyodide package using Mocha, run the following commands,

```
cd src/js
npm test
```

To check TypeScript type definitions run,

```
npx tsd
```

Manual interactive testing

To run tests manually:

1. Build Pyodide, perhaps in the docker image
2. From outside of the docker image, cd into the dist directory and run `python -m http.server`.
3. Once the webserver is running, simple interactive testing can be run by visiting the URL: `http://localhost:<PORT>/console.html`. It's recommended to use `pyodide.runPython` in the browser console rather than using the repl.

Benchmarking

To run common benchmarks to understand Pyodide's performance, begin by installing the same prerequisites as for testing. Then run:

```
PYODIDE_PACKAGES="numpy,matplotlib" make benchmark
```

Linting

We lint with `pre-commit`.

Python is linted with `ruff`, `black` and `mypy`. JavaScript, markdown, yaml, and html are linted with `prettier`. C is linted with `clang-format`.

To lint the code, run:

```
pre-commit run -a
```

You can have the linter automatically run whenever you commit by running

```
pip install pre-commit
pre-commit install
```

and this can later be disabled with

```
pre-commit uninstall
```

If you don't lint your code, certain lint errors will be fixed automatically by `pre-commit.ci` which will push fixes to your branch. If you want to push more commits, you will either have to pull in the remote changes or force push.

3.2.6 Debugging tips

See [Emscripten's page about debugging](#) which has extensive info about the various debugging options available. The [Wasm Binary Toolkit](#) is super helpful for analyzing `.wasm`, `.so`, `.a`, and `.o` files.

Also whenever you can reproduce a bug in chromium make sure to use a chromium-based browser (e.g., chrome) for debugging. They are better at it.

Run prettier on pyodide.asm.js

Before doing any debugger I strongly recommend running `npx prettier -w pyodide.asm.js`. This makes everything much easier.

Linker error: function signature mismatch

You may get linker errors as follows:

```
wasm-ld: error: function signature mismatch: some_func
>>> defined as (i32, i32) -> i32 in some_static_lib.a(a.o)
>>> defined as (i32) -> i32 in b.o
```

This is especially common in Scipy. Oftentimes it isn't too hard to figure out what is going wrong because it told you the both the symbol name (`some_func`) and the object files involved (this is much easier than the runtime version of this error!). If you can't tell what is going on from looking at the source files, it's time to pull out `wasm-objdump`. In this case `a.o` is part of `some_static_lib.a` so you first need to get it out with `ar -x some_static_lib.a a.o`. Now we can check if `a.o` imports or defines `some_func`. To check for imports, use `wasm-objdump a.o -j Import -x | grep some_func`. If `a.o` is importing `some_func` you should see a line like: `- func[0] sig=1 <env.some_func> <- env.some_func` in the output.

If not, you will see nothing or things like `some_func2`. To check if `a.o` defines `some_func` (this is a bit redundant because you can conclude whether or not does from whether it imports it) we can use: `wasm-objdump a.o -j`

Function `-x | grep some_func`, if `a.o` defines `some_func` you will see something like: `- func[0] sig=0 <some_func>`.

Now the question is what these signatures mean (though we already know this from the linker error). To find out what signature 0 is, you can use `wasm-objdump a.o -j Type -x | grep "type\[0\]"`.

Using this, we can verify that `a.o` imports `some_func` with signature `(i32, i32) -> i32` but `b.o` exports it with signature `(i32) -> i32`, hence the linker error.

This process works in basically the same way for already-linked `.so` and `.wasm` files, which can help if you get the load-time version of this linker error.

Misencoded Wasm

On a very rare occasion you may run into a misencoded object file. This can cause different tools to crash, `wasm-ld` may panic, etc. `wasm-objdump` will just generate a useless error message. In this case, I recommend `wasm-objdump -s --debug 2>&1 | grep -i error -C 20` (or pipe to `less`), which will result in more diagnostic information. Sometimes the crash happens quite a lot later than the actual error, look for suspiciously large constants, these are often the first sign of something gone haywire.

After this, you can get out a hex editor and consult the [WebAssembly binary specification](#) Cross reference against the hex addresses appearing in `wasm-objdump --debug`. With enough diligence you can locate the problem.

Debugging RuntimeError: function signature mismatch

First recompile with `-g2`. `-g2` keeps symbols but won't try to use C source maps which mostly make our life harder (though it may be helpful to link one copy with `-g2` and one with `-g3` and run them at the same time cf [Using C source maps](#)).

The browser console will show something like the following. Click on the innermost stack trace:

```

Stack (most recent call first):
  File "<console>", line 1 in <module>
  File "/lib/python3.11/site-packages/_pyodide/_base.py", line 351 in run_async
  File "/lib/python3.11/site-packages/pyodide/console.py", line 362 in runcode
  File "/lib/python3.11/site-packages/pyodide/console.py", line 474 in runcode
  File "/lib/python3.11/asyncio/events.py", line 80 in _run
  File "/lib/python3.11/site-packages/pyodide/webloop.py", line 151 in run_handle
  ✖ ▶ Uncaught RuntimeError: null function or function signature mismatch
    at cfunction_call (pyodide.asm.wasm:0x1e325c)
    at _PyObject_MakeTpCall (pyodide.asm.wasm:0x1a16b7)
    at PyObject_Vectorcall (pyodide.asm.wasm:0x1a1ca6)
    at _PyEval_EvalFrameDefault (pyodide.asm.wasm:0x271855)
    at PyEval_EvalCode (pyodide.asm.wasm:0x269262)
    at builtin_eval (pyodide.asm.wasm:0x266613)
    at cfunction_vectorcall_FASTCALL (pyodide.asm.wasm:0x1e2e61)
    at PyObject_Vectorcall (pyodide.asm.wasm:0x1a1cb8)
    at _PyEval_EvalFrameDefault (pyodide.asm.wasm:0x271855)
    at gen_send_ex2 (pyodide.asm.wasm:0x1b548a)
  
```

click here

Clicking the offset will (hopefully) take you to the corresponding `wasm` instruction, which should be a `call_indirect`. If the offset is too large (somewhere between `0x02000000` and `0x03000000`) you will instead see `;;` text is truncated due to size, see [Dealing with ;; text is truncated due to size](#). In this example we see the following:


```

0x01e3255 end $label2
0x01e3256 local.get $var3
0x01e3258 local.get $var1
0x01e325a local.get $var8
0x01e325c call_indirect (param i32 i32) (result i32)
0x01e325f end $label1
0x01e3260 i32.const 0
0x01e3262 call $_Py_CheckFunctionResult
0x01e3265 end $label0
0x01e3266 local.set $var0
0x01e3268 local.get $var4

```

click here ←

So we think we are calling a function pointer with signature (param i32 i32) (result i32) meaning that it takes two i32 inputs and returns one i32 output. Set a breakpoint by clicking on the address, then refresh the page and run the reproduction again. Sometimes these are on really hot code paths (as in the present example) so you probably only want to set the breakpoint once Pyodide is finished loading. If your reproduction passes through the breakpoint multiple times before crashing you can do the usual chore of counting how many times you have to press “Resume” before the crash. Suppose you’ve done all this, and we’ve got the vm stopped at the bad instruction just before crashing:

```

0x05cf i32.add
0x05d0 call $PyModule_AddType
0x05d2 drop
0x05d3 local.get $var1
0x05d5 )
0x05d6 (func $zero (;5;) (result i32)
0x05d6 (local $var0 i32)
0x05da global.get $_Py_NoneStruct
0x05dc local.tee $var0
0x05de local.get $var0
0x05e0 i32.load
0x05e3 i32.const 1
0x05e5 i32.add
0x05e6 i32.store
0x05e9 local.get $var0
0x05eb )
0x05ec (func $one (;6;) (param $var0 i32) (result i32)
0x05ec (local $var1 i32)
0x05f0 global.get $_Py_NoneStruct
0x05f2 local.tee $var1
0x05f4 local.get $var1
0x05f6 i32.load
0x05f9 i32.const 1
0x05fb i32.add
0x05fc i32.store
0x05ff local.get $var1
0x0601 )
0x0602 (func $two (;7;) (param $var0 i32) (param $var1 i32)
0x0602 (local $var2 i32)
0x0606 global.get $_Py_NoneStruct
0x0608 local.tee $var2
0x060a local.get $var2
0x060c i32.load
0x060f i32.const 1
0x0611 )

```

Paused on breakpoint

Watch

Breakpoints

☒ pyodide.asm.wasm:0x1e325c

Scope

Expression

stack: Stack

- 0: i32 {value: 2852320}
- 1: i32 {value: 15116816}
- 2: i32 {value: 13749792}
- 3: i32 {value: 2798452}
- 4: i32 {value: 13109}

Local

- \$var0: i32 {value: 15116816}
- \$var1: i32 {value: 2798452}
- \$var2: i32 {value: 0}
- \$var3: i32 {value: 13749792}
- \$var4: i32 {value: 9139648}
- \$var5: i32 {value: 2852320}
- \$var6: i32 {value: 18200624}
- \$var7: i32 {value: 1}
- \$var8: i32 {value: 13109}

Module

Call Stack

- \$function_call

The function pointer ←

The bottom value on the stack is the function pointer. In this case it’s the fourth item on the stack, so you can type the following into the console:

```

> pyodide._module.wasmTable.get(stack[4].value) // stack[4].value === 13109
< f $one() { [native code] }

```

So the bad function pointer’s symbol is one! Now clicking on \$one brings you to the source for it:

```
(func $one (;6;) (param $var0 i32) (result i32)
  (local $var1 i32)
  global.get $_Py_NoneStruct
  local.tee $var1
  local.get $var1
```

and we see the function pointer has signature (param \$var0 i32) (result i32), meaning it takes one i32 input and returns one i32 output. Note that if the function had void return type it might look like (param \$var0 i32 \$var1 i32) (with no result). Confusion between i32 and void return type is the single most common cause of this error.

Now we basically know the cause of the trouble. You can look up `cfunction_call` in the CPython source code with the help of `ripgrep` and locate the line that generates this call, and look up one in the appropriate source and find the signature. Another approach to locate the call site would be to recompile with `-g3` and use source maps *Using C source maps* to locate the problematic source code. With the same process of reproduce crash ==> click innermost stack frame ==> see source file and line where the error occurs. In this case we see that the crash is on the line:

```
result = _PyCFunction_trampolineCall(meth, self, args);
```

in the file `/src/cpython/build/Python-3.11.0dev0/Objects/methodobject.c`. Unfortunately, source maps are useless for the harder problem of finding the callee because compiling with `-g3` increases the number of function pointers so the function pointer we are calling is in a different spot. I know of no way to determine the bad function pointer when compiling with `-g3`.

Sometimes (particularly with Scipy/CLAPACK) the issue will be a mismatch between (param i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32) (result i32) and (param i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32) (result i32)

(14 vs 15 parameters) which might be a little hard to discern. I copy the signature into the Javascript console and run `"i32 ... i32".split(" ").length` in this case.

Dealing with ;; text is truncated due to size

If you are debugging and run into the dreaded `;; text is truncated due to size` error message, the solution is to compile a modified version of Chrome devtools with a larger `wasm` size cap. Surprisingly, this is not actually all that hard.

These instructions are adapted from here: <https://www.diverto.hr/en/blog/2020-08-15-WebAssembly-limit/>

In short,

```
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
./fetch devtools-frontend
cd devtools-frontend
```

Apply the following change:

```
--- a/front_end/entrypoints/wasmparser_worker/WasmParserWorker.ts
+++ b/front_end/entrypoints/wasmparser_worker/WasmParserWorker.ts
@@ -55,7 +55,7 @@ export function disassembleWASM(
  const lines = [];
  const offsets = [];
  const functionBodyOffsets = [];
-  const MAX_LINES = 1000 * 1000;
```

(continues on next page)

(continued from previous page)

```
+  const MAX_LINES = 12 * 1000 * 1000;
    let chunkSize: number = 128 * 1024;
    let buffer: Uint8Array = new Uint8Array(chunkSize);
    let pendingSize = 0;
```

Then build with:

```
gn gen out/Default
autoninja -C out/Default
```

then

```
cd out/Default/resources/inspector_overlay/
python http.server <some_port>
```

and then you can start a version of chrome using the modified devtools:

```
chrome --custom-devtools-frontend=http://localhost:<some_port>/
```

Using C source maps

Chromium has support for [DWARF info](#) which can be very helpful for debugging in certain circumstances.

I haven't used this very much because it is often not very beneficial. The biggest issue is that I have found no way to toggle between viewing the C source and the WebAssembly. In particular, if source maps are available, the debugger gives no way to view the current line in the wasm. What's worse is that even if it fails to find the source map, it won't fall back to displaying the source map. To *prevent* this, relink the code with `-g2`.

Typically once I have isolated the interesting line of C code, I need to see what is going on at an instruction-level. This limitation means that it is generally easier to work directly with instructions. One work around is to load a copy of Pyodide with the source maps next to one without the source maps. This situation is rapidly improving both on Emscripten's side and on the browser side. To build Pyodide with DWARF, you should set `DBGFLAGS="-g3 -gseparate-dwarf"`.

If you are building in the docker image, you will get error 404s when the browser tries to look up the source maps because the path `/src/cpython/...` doesn't exist. One dumb solution is `sudo ln -s $(pwd) /src`. It might not be the best idea to link some random directory into root, if you manage to destroy your computer with this please don't blame me. In particular, if you later want to remove this link make sure not to remove `/srv` instead! The correct solution is to use `--source-map-base`, but I can't seem to get it to work.

3.3 Project

The Project section gives additional information about the project's organization and latest releases.

3.3.1 What is Pyodide?

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

Pyodide 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.

History

Pyodide was created in 2018 by [Michael Droettboom](#) at Mozilla as part of the [Iodide](#) project. Iodide is an experimental web-based notebook environment for literate scientific computing and communication.

Contributing

See the [contributing guide](#) for tips on filing issues, making changes, and submitting pull requests. Pyodide is an independent and community-driven open-source project. The decision-making process is outlined in [Governance and Decision-making](#).

Citing

If you use Pyodide for a scientific publication, we would appreciate citations. Please find us [on Zenodo](#) and use the citation for the version you are using. You can replace the full author list from there with “The Pyodide development team” like in the example below:

```
@software{pyodide_2021,
  author      = {The Pyodide development team},
  title       = {pyodide/pyodide},
  month       = aug,
  year        = 2021,
  publisher    = {Zenodo},
  version     = {0.23.1},
  doi         = {10.5281/zenodo.5156931},
  url         = {https://doi.org/10.5281/zenodo.5156931}
}
```

Communication

- Blog: blog.pyodide.org
- Mailing list: mail.python.org/mailman3/lists/pyodide.python.org/
- Gitter: gitter.im/pyodide/community
- Twitter: twitter.com/pyodide
- Stack Overflow: stackoverflow.com/questions/tagged/pyodide

Donations

We accept donations to the Pyodide project at opencollective.com/pyodide. All donations are processed by the [Open Source Collective](#) – a nonprofit organization that acts as our fiscal host.

Funds will be mostly spent to organize in-person code sprints and to cover infrastructure costs for distributing packages built with Pyodide.

License

Pyodide uses the [Mozilla Public License Version 2.0](#).

Infrastructure support

We would like to thank,

- [Mozilla](#) and [CircleCI](#) for Continuous Integration resources
- [JsDelivr](#) for providing a CDN for Pyodide packages
- [ReadTheDocs](#) for hosting the documentation.

3.3.2 Roadmap

This document lists general directions that core developers are interested to see developed in Pyodide. The fact that an item is listed here is in no way a promise that it will happen, as resources are limited. Rather, it is an indication that help is welcomed on this topic.

Improve documentation

Our API documentation is fairly detailed, but they need more introductory information like tutorials. We also want to add more information to the FAQ and improve the organization. It would also be good to find some way to include interactive code pens in the documentation.

Reducing download sizes and initialization times

At present a first load of Pyodide requires a 6.4 MB download, and the environment initialization takes 4 to 5 seconds. Subsequent page loads are faster since assets are cached in the browser. Both of these indicators can likely be improved, by optimizing compilation parameters, minifying the Python standard library and packages, reducing the number of exported symbols. To figure out where to devote the effort, we need a better profiling system for the load process.

See issue [#646](#).

Improve performance of Python code in Pyodide

Across [benchmarks](#) Pyodide is currently around 3x to 5x slower than native Python.

At the same time, C code compiled to WebAssembly typically runs between near native speed and 2x to 2.5x times slower (Jangda et al. 2019 [PDF](#)). It is therefore very likely that the performance of Python code in Pyodide can be improved with some focused effort.

In addition, scientific Python code would benefit from packaging a high performance BLAS library such as BLIS.

See issue [#1120](#).

Find a better way to compile Fortran

Currently, we use f2c to cross compile Fortran to C. This does not work very well because f2c only fully supports Fortran 77 code. LAPACK has used more modern Fortran features since 2008 and Scipy has adopted more recent Fortran as well. f2c still successfully generates code for all but 6 functions in Scipy + LAPACK, but much of the generated code is slightly wrong and requires extensive patching. There are still a large number of fatal errors due to call signature incompatibilities.

If we could use an LLVM-based Fortran compiler as a part of the Emscripten toolchain, most of these problems would be solved. There are several promising projects heading in that direction including flang and lfortran.

See [scipy/scipy#15290](#).

Better project sustainability

Some of the challenges that Pyodide faces, such as maintaining a collection of build recipes, dependency resolution from PyPI, etc are already solved in either Python or JavaScript ecosystems. We should therefore strive to better re-use existing tooling, and seeking synergies with existing initiatives in this space, such as conda-forge.

See issue [#795](#).

Improve support for WebWorkers

WebWorkers are necessary in order to run computational tasks in the browser without hanging the user interface. Currently, Pyodide can run in a WebWorker, however the user experience and reliability can be improved.

See issue [#1504](#).

Synchronous IO

The majority of existing I/O APIs are synchronous. Unless we can support synchronous IO, much of the existing Python ecosystem cannot be ported. There are several different approaches to this, we would like to support at least one method.

See issue [#1503](#).

Write `http.client` in terms of Web APIs

Python packages make an extensive use of packages such as `requests` to synchronously fetch data. We currently can't use such packages since sockets are not available in Pyodide. We could however try to re-implement some stdlib libraries with Web APIs, potentially making this possible.

Because `http.client` is a synchronous API, we first need support for synchronous IO.

See issue [#140](#).

3.3.3 Code of Conduct

Conduct

We are committed to providing a friendly, safe and welcoming environment for all, regardless of level of experience, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, nationality, or other similar characteristic.

- Please be kind and courteous. There's no need to be mean or rude.
- Please avoid using usernames that are overtly sexual or otherwise might detract from a friendly, safe, and welcoming environment for all.
- Respect that people have differences of opinion and that every design or implementation choice carries trade-offs. There is seldom a single right answer.
- We borrow the Recurse Center's "[social rules](#)": no feigning surprise, no well-actually's, no backseat driving, and no subtle -isms.
- Please keep unstructured critique to a minimum. If you have solid ideas you want to experiment with, make a fork and see how it works. All feedback should be constructive in nature. If you need more detailed guidance around giving feedback, consult [Digital Ocean's Code of Conduct](#)
- It is unacceptable to insult, demean, or harass anyone. We interpret the term "harassment" as defined in the [Citizen Code of Conduct](#); if you are not sure about what harassment entails, please read their definition. In particular, we don't tolerate behavior that excludes people in socially marginalized groups.
- Private harassment is also unacceptable. No matter who you are, please contact any of the Pyodide core team members immediately if you are being harassed or made uncomfortable by a community member. Whether you are a regular contributor or a newcomer, we care about making this community a safe place for you and we've got your back.
- Likewise spamming, trolling, flaming, baiting or other attention-stealing behavior is not welcome.

Moderation

These are the policies for upholding our community’s standards of conduct. If you feel that a thread needs moderation, please contact the Pyodide core team.

1. Remarks that violate the Pyodide standards of conduct are not allowed. This includes hateful, hurtful, oppressive, or exclusionary remarks. (Cursing is allowed, but never targeting another community member, and never in a hateful manner.)
2. Remarks that moderators find inappropriate are not allowed, even if they do not break a rule explicitly listed in the code of conduct.
3. Moderators will first respond to such remarks with a warning.
4. If the warning is unheeded, the offending community member will be temporarily banned.
5. If the community member comes back and continues to make trouble, they will be permanently banned.
6. Moderators may choose at their discretion to un-ban the community member if they offer the offended party a genuine apology.
7. If a moderator bans someone and you think it was unjustified, please take it up with that moderator, or with a different moderator, in private. Complaints about bans in-channel are not allowed.
8. Moderators are held to a higher standard than other community members. If a moderator creates an inappropriate situation, they should expect less leeway than others.
9. In the Pyodide community we strive to go the extra mile to look out for each other. Don’t just aim to be technically unimpeachable, try to be your best self. In particular, avoid flirting with offensive or sensitive issues, particularly if they’re off-topic; this all too often leads to unnecessary fights, hurt feelings, and damaged trust; worse, it can drive people away from the community entirely.
10. If someone takes issue with something you said or did, resist the urge to be defensive. Just stop doing what it was they complained about and apologize. Even if you feel you were misinterpreted or unfairly accused, chances are good there was something you could have communicated better — remember that it’s your responsibility to make your fellow Pyodide community members comfortable. Everyone wants to get along and we are all here first and foremost because we want to talk about science and cool technology. You will find that people will be eager to assume good intent and forgive as long as you earn their trust.
11. The enforcement policies listed above apply to all official Pyodide venues. If you wish to use this code of conduct for your own project, consider making a copy with your own moderation policy so as to avoid confusion.

Adapted from the [the Rust Code of Conduct](#), with further reference from [Digital Ocean Code of Conduct](#), the [Recurse Center](#), the [Citizen Code of Conduct](#), and the [Contributor Covenant](#).

3.3.4 Governance and Decision-making

The purpose of this document is to formalize the governance process used by the Pyodide project, to clarify how decisions are made and how the various members of our community interact. This document establishes a decision-making structure that takes into account feedback from all members of the community and strives to find consensus, while avoiding deadlocks.

Anyone with an interest in the project can join the community, contribute to the project design and participate in the decision making process. This document describes how to participate and earn merit in the Pyodide community.

Roles And Responsibilities

Contributors

Contributors are community members who contribute in concrete ways to the project. Anyone can become a contributor, and contributions can take many forms, for instance, answering user questions – not only code – as detailed in [How to Contribute](#).

Community members team

The community members team is composed of community members who have permission on Github to label and close issues. Their work is crucial to improve the communication in the project.

After participating in Pyodide development with pull requests and reviews for a period of time, any contributor may become a member of the team. The process for adding team members is modeled on the [CPython project](#). Any core developer is welcome to propose a Pyodide contributor to join the community members team. Other core developers are then consulted: while it is expected that most acceptances will be unanimous, a two-thirds majority is enough.

Core developers

Core developers are community members who have shown that they are dedicated to the continued development of the project through ongoing engagement with the community. They have shown they can be trusted to maintain Pyodide with care. Being a core developer allows contributors to more easily carry on with their project related activities by giving them direct access to the project's repository and is represented as being a member of the core team on the Pyodide [GitHub organization](#). Core developers are expected to review code contributions, can merge approved pull requests, can cast votes for and against merging a pull-request, and can make decisions about major changes to the API (all contributors are welcome to participate in the discussion).

New core developers can be nominated by any existing core developers. Once they have been nominated, there will be a vote by the current core developers. Voting on new core developers is one of the few activities that takes place on the project's private communication channels. While it is expected that most votes will be unanimous, a two-thirds majority of the cast votes is enough. The vote needs to be open for at least one week.

Core developers that have not contributed to the project (commits or GitHub comments) in the past two years will be asked if they want to become emeritus core developers and recant their commit and voting rights until they become active again.

Decision Making Process

Decisions about the future of the project are made through discussion with all members of the community. All non-sensitive project management discussion takes place on the project contributors' [issue tracker](#) and on [Github discussion](#). Occasionally, sensitive discussion occurs on a private communication channels.

Pyodide uses a “consensus seeking” process for making decisions. The group tries to find a resolution that has no open objections among core developers. At any point during the discussion, any core-developer can call for a vote, which will conclude two weeks from the call for the vote. This is what we hereafter may refer to as “the decision making process”.

Decisions (in addition to adding core developers as above) are made according to the following rules:

- **Maintenance changes**, include for instance improving the wording in the documentation, updating CI or dependencies. Core developers are expected to give “reasonable time” to others to give their opinion on the Pull Request in case they're not confident that others would agree. If no further review on the Pull Request is received

within this time, it can be merged. If a review is received, then the consensus rules from the following section apply.

- **Code changes in general, and especially those impacting user facing APIs**, as well as more significant documentation changes, require review and approval by a core developer and no objections raised by any core developer (lazy consensus). This process happens on the pull-request page.
- **Changes to the governance model** use the same decision process outlined above.

3.3.5 Change Log

Version 0.23.1

Deployment

- Fix Export `python_stdlib.zip` in `package.json`. [#3723](#)

CLI

- Enhancement `pyodide build` now accepts an `--output-directory` argument. [#3746](#)
- Fix Fix `pyodide py-compile` not to ignore the `--compression-level` option when applied on a single file. [#3727](#)
- Fix Fix an issue where the `pyodide venv` command did not work correctly in `pyodide-build` version 0.23.0 because of missing `python_stdlib.zip`. [#3760](#)
- Fix `python -m pip` works correctly in the Pyodide venv now. [#3761](#)
- Fix Executables installed in a Pyodide virtual environment now run in Pyodide not in the host Python. [#3752](#)

Build System

- Fix Fix `PYODIDE_ROOT` to point the correct directory when running out-of-tree build. [#3751](#)

Version 0.23.0

March 30, 2023

General

- Update Pyodide now runs Python 3.11.2 which officially supports WebAssembly as a [PEP11 Tier 3](#) platform. [#3252](#), [#3614](#)
- Update We now build `libpyodide.a` so the Pyodide foreign function interface can be experimentally linked into other Emscripten builds of Python. [#3335](#)
- Enhancement Updated Emscripten to version 3.1.32 [#3471](#), [#3517](#), [#3599](#)

JavaScript API

- **BREAKING CHANGE** Type exports of `PyProxy` subtypes have been moved from `pyodide` to `pyodide/ffi` and many of them have changed names. The original exports are still available but they are deprecated. [#3523](#)
- **BREAKING CHANGE** The methods for checking `PyProxy` capabilities (e.g., `supportsHas`, `isCallable`) are now deprecated. Use e.g., `instanceof pyodide.ffi.PyCallable` instead. [#3523](#)
- **Enhancement** Added subclasses of `PyProxy` for each mixin. These can be used to check whether a `PyProxy` supports a given set of methods with `instanceof` e.g., `x instanceof pyodide.ffi.PyDict`. [#3523](#)
- **Enhancement** Added `stdLibURL` parameter to `loadPyodide` allowing to customize the URL from which the Python standard library is loaded. [#3670](#)
- **Enhancement** Checking whether an object is an instance of a `PyProxy` now only recognizes a `PyProxy` generated from the same Python interpreter. This means that creating multiple interpreters and importing a `PyProxy` from one into another no longer causes a fatal error. [#3545](#)
- **Enhancement** `as_object_map` now accepts a keyword argument `hereditary`. If set to `True` and indexing the object returns a plain-old-object, then the return value will be automatically mapped in `as_object_map` as well. [#3638](#)
- **Enhancement** A `JsProxy` of a JavaScript error object can be directly thrown as Python exceptions. Previously Pyodide automatically wrapped them in a `JsException` but that is no longer needed – now `JsException` inherits from both `JsProxy` and `Exception`. [#3455](#)
- **Enhancement** `runPython` and `runPythonAsync` now accept a `locals` argument. [#3618](#)
- **Fix** Calling `loadPyodide` repeatedly in Node no longer results in `MaxListenersExceededWarning`. Also, calling `loadPyodide` in Node v14 no longer changes unhandled rejections in promises. [#3542](#)
- **Fix** If the `locals` argument to `eval_code` or `eval_code_async` is `None` it now uses `locals=globals` as the documentation says. [#3580](#)

Python standard library

- **BREAKING CHANGE** Unvendored `_pydecimal` and `pydoc_data` from the standard library. Now these modules need to be loaded with `pyodide.loadPackage` or `micropip.install`, or auto-loaded via `imports` in `pyodide.runPythonAsync` [#3525](#)
- **BREAKING CHANGE** Test files of `stdlib` `ctypes` and `unittest` are now moved to `test/ctypes` and `test/unittest` respectively. This change is adapted from [CPython 3.12](#). [#3507](#)

Deployment

- **BREAKING CHANGE** Pyodide no longer uses Emscripten preload plugin, hence `pyodide.asm.data` is removed, in favor of `python_stdlib.zip`. This change normally shouldn't affect users, but if you were using this file in a bundler, you will need to remove it. [#3584](#)
- **BREAKING CHANGE** `pyodide.py.tar` file is removed. This change normally shouldn't affect users, but if you were using this file in a bundler, you will need to remove it. [#3621](#)
- **BREAKING CHANGE** Python standard libraries are now vendored in a zipfile: `/lib/python{version}.zip` in the in-browser MEMFS file system. If you need to access the standard library source code, you need to unpack the zip file. For example: `import shutil; shutil.unpack_archive('/lib/python311.zip', '/lib/python3.11', 'zip')` [#3584](#)

- Fix Improves the compression of wheel files with the JsDelivr CDN. For browsers that support the Brotli compression (most modern ones) this should result in a size reduction of 20-30%. Also most many pyodide CLI sub-commands now support `--compression-level` as an optional parameter. [#3655](#)
- BREAKING CHANGE Following libraries are now not linked to the Pyodide main module: `libgl`, `libal`, `libhtml5`. This normally shouldn't affect users, but if you are using these libraries in a package that are built out-of-tree, you will need to link them to the package manually. [#3505](#)

Python / JavaScript Foreign Function Interface

- Fix PyProxies of Async iterators are now async iterable JavaScript objects. The code:

```
for await (let x of async_iterator_pyproxy) {  
  // ...  
}
```

would previously fail with `TypeError: async_iterator_pyproxy is not async iterable`. (Python *async iterables* that were not also iterators were already *async iterable*, the problem was only with Python objects that are both *async iterable* and an *async iterator*.) [#3708](#)

- Enhancement A py-compiled build which has smaller and faster-to-load packages is now deployed under <https://cdn.jsdelivr.net/pyodide/v0.23.0/pyc/> (also for future versions). The exceptions obtained with this builds will not include code snippets however. [#3701](#)
- BREAKING CHANGE Removed support for calling functions from the root of `pyodide` package directly. This has been deprecated since v0.21.0. Now all functions are only available under submodules. [#3677](#)
- BREAKING CHANGE Removed support for passing the “message” argument to `PyProxy.destroy` in a positional argument. This has been deprecated since v0.22.0. [#3677](#)
- Enhancement Python does not allow reserved words to be used as attributes. For instance, `Array.from` is a `SyntaxError`. (JavaScript has a more robust parser which can handle this.) To handle this, if an attribute to a `JsProxy` consists of a Python reserved word followed by one or more underscores, we remove a single underscore from the end of the attribute. For instance, `Array.from_` would access `from` on the underlying JavaScript object, whereas `o.from__` accesses the `from_` attribute. [#3617](#)

Build System

- BREAKING CHANGE When building meta-packages (`core` and `min-scipy-stack`), you must prefix `tag:` to the meta-package name. For example, to build the `core` meta-package, you must run `pyodide build-recipes tag:core`, or `PYODIDE_PACKAGES="tag:core" make`. [#3444](#)
- Enhancement Add `--build-dependencies` to `pyodide build` command to fetch and build dependencies of a package being built. Also adds `--skip-dependency` to ignore selected dependencies. [#3310](#)
- Enhancement Added `pyodide build` support for building a list of packages from a `requirements.txt` file with `pyodide build -r <requirements.txt>`. Also can output a list of chosen dependencies in the same format when building a package and dependencies using the `--output-lockfile <lockfile.txt>` argument. This enables repeatable builds of packages. [#3469](#)
- Enhancement Added `package/tag` key to the `meta.yaml` spec to group packages. [#3444](#)
- Enhancement `pyodide build-recipes` now autodetects the number of CPU cores in the system and uses them for parallel builds. [#3559](#) [#3598](#)
- Fix Fixed pip install error when installing cross build environment. [#3562](#)
- Enhancement Response files are now correctly handled when calculating exported symbols. [#3645](#)

- Fix Fix occasional build failure when building rust packages. [#3607](#)
- Enhancement Improved logging in `pyodide-build` with rich. [#3442](#)
- Enhancement `pyodide build-recipes` now accepts `--no-deps` parameter, which skips building dependencies of the package. This replaces `pyodide-build buildpkg`. [#3520](#)
- Enhancement `pyodide build-recipes` now works out-of-tree.

Pyodide CLI

- BREAKING CHANGE Removed deprecated CLI entrypoints `pyodide-build buildall` which is replaced by `pyodide build-recipes`, and `pyodide-build mkpkg` which is replaced by `pyodide skeleton pypi`. [#3668](#).
- Feature Added `pyodide py-compile` CLI command that `py` compiles a wheel or a zip file, converting `.py` files to `.pyc` files. It can also be applied to a folder with wheels / zip files. If the input folder contains the `repodata.json` the paths and checksums it contains will also be updated [#3253](#) [#3700](#)
- Feature Added `pyodide create-zipfile` CLI command that creates a zip file of a directory. This command is hidden by default since it is not intended for use by end users. [#3411](#) [#3463](#)

REPL

- Fix Non-breaking space characters are now automatically converted to regular spaces in `pyodide REPL`. [#3558](#)
- Enhancement Allow changing the build type used in the REPL by passing the `build` argument to the REPL URL. For instance, <https://pyodide.org/en/latest/console.html?build=debug> will load debug dev build. [#3671](#)

Packages

- New packages: `fastparquet` [#3590](#), `cramjam` [#3590](#), `pynacl` [#3500](#), `pyxel` [#3508](#). `mypy` [#3504](#), `multidict` [#3581](#), `yaml` [#3702](#), `idna` [#3702](#), `cbor-diag` [#3581](#).
- Upgraded to micropip 0.3.0 (see [changelog](#) [#3709](#))
- Added experimental *support for SDL based packages* [#3508](#)
- Upgraded packages: see the list of packages versions in this release in *Packages built in Pyodide*.

List of Contributors

Alexey Ignatiev, Andrea Giammarchi, Arpit, Christian Clauss, Deepak Cherian, Eli Lamb, Feodor Fitsner, Gyeongjae Choi, Hood Chatham, Jeff Glass, Jo Bovy, Joe Marshall, josephrocca, Loïc Estève, martinRenou, messense, Nicholas Bollweg, Roman Yurchak, TheOnlyWayUp, Victor Blomqvist, Ye Joo Park

Version 0.22.1

January 25, 2023

- BREAKING CHANGE `setStdin` now accepts an extra `autoEOF` parameter. If `true`, it will insert an EOF automatically after each string or buffer. Defaults to `true`. This also affects the behavior of the `stdin` argument to `loadPyodide`. #3488
- Fix `from pyodide.ffi import *` doesn't raise an `ImportError` anymore. #3484
- Enhancement Pyodide displays a better message when someone calls `posix exit` or `os._exit`. #3496

Package Loading

- Fix Fix incorrect error message when loading a package include in Pyodide fails. #3435

Build system

- Fix Emscripten is no longer required to create a Pyodide virtual environment. #3485
- Fix Fixed a bug where `pyodide build` would fail on package that use CMake, when run multiple times. #3445
- Fix `pyodide build`: Don't pass the directory to the build backend args, only pass the arguments. #3490
- Fix `pyodide config` won't print extra messages anymore. #3483
- Fix Pass the same environment variables for out of tree builds as for in tree builds. #3495

Version 0.22.0

January 3, 2023

See the [release notes](#) for a summary.

Deployment and testing

- BREAKING CHANGE `pyodide-cdn2.iodide.io` is not available anymore. Please use `https://cdn.jsdelivr.net/pyodide` instead. #3150.
- BREAKING CHANGE We don't publish pre-built Pyodide docker images anymore. Note that `./run_docker --pre-built` was not working for a while and it was actually equivalent to `./run_docker`. If you need to build a single Python wheel out of tree, you can use the `pyodide build` command instead. See [our blog post](#) for more information. #3342.
- Enhancement The releases are now called `pyodide-{version}.tar.gz` rather than `pyodide-build-{version}.tar.gz` #2996
- Enhancement Added a new release file called `pyodide-core-{version}.tar.gz` intended for use in Node. It contains the files needed to start Pyodide and no additional packages. #2999
- Enhancement The full test suite is now run in Safari #2578, #3095.
- Enhancement Added Gitpod configuration to the repository. #3201

Foreign function interface

JsProxy / JavaScript from Python

- Enhancement Implemented `reverse`, `__reversed__`, `count`, `index`, `append`, and `pop` for JsProxy of Javascript arrays so that they implement the `collections.abc.MutableSequence` API. [#2970](#)
- Enhancement Implemented methods `keys`, `items`, `values`, `get`, `pop`, `setdefault`, `popitem`, `update`, and `clear` for JsProxy of map-like objects so that they implement the `collections.abc.MutableMapping` API. [#3275](#)
- Enhancement It's now possible to destructure a JavaScript array, map, or object returned by `as_object_map` with a `match` statement. [#2906](#)
- Enhancement Added `then`, `catch`, and `finally_` methods to the Futures used by Pyodide's event loop so they can be used like Promises. [#2997](#)
- Enhancement `create_proxy` now takes an optional `roundtrip` parameter. If this is set to `True`, then when the proxy is converted back to Python, it is converted back to the same double proxy. This allows the proxy to be destroyed from Python even if no reference is retained. [#3163](#), [#3369](#)
- Enhancement A JsProxy of a function now has a `__get__` descriptor method, so it's possible to use a JavaScript function as a Python method. When the method is called, this will be a PyProxy pointing to the Python object the method is called on. [#3130](#)
- Enhancement A JsProxy now has an `as_object_map` method. This will treat the object as a mapping over its `ownKeys` so for instance: `run_js("{a:2, b:3}").as_object_map()["a"]` will return 2. These implement `collections.abc.MutableMapping`. [#3273](#), [#3295](#), [#3297](#)
- Enhancement Split up the JsProxy documentation class into several classes, e.g., `JsBuffer`, `JsPromise`, etc. Implemented `issubclass` and `isinstance` on the various synthetic and real JsProxy classes so that they behave the way one might naively expect them to (or at least closer to that than it was before). [#3277](#)
- Enhancement Added type parameters to many of the JsProxy subtypes. [#3387](#)
- Enhancement Added `JsGenerator` and `JsIterator` types to `pyodide.ffi`. Added `send` method to `JsIterators` and `throw`, and `close` methods to `JsGenerators`. [#3294](#)
- Enhancement It is now possible to use asynchronous JavaScript iterables, iterators and generators from Python. This includes support for `aiter` for async iterables, `anext` and `asend` for async iterators, and `athrow` and `aclose` for async generators. [#3285](#), [#3299](#), [#3339](#)
- Enhancement JavaScript generators and async generators that are created from Python now are wrapped so that Python objects sent to them as arguments or from `.send` / `.asend` are kept alive until the generator is exhausted or `.closed`. This makes generators significantly more ergonomic to use, at the cost of making memory leaks more likely if the generator is never finalized. [#3317](#)
- Enhancement Added a mypy typeshed for some common functionality for the `js` module. [#3298](#)
- Enhancement mypy understands the types of more things now. [#3385](#)
- Fix Fixed bug in `split` argument of `pyodide.console.repr_shorten`. Added `shorten` function. [#3178](#)

PyProxy / Using Python from JavaScript

- Enhancement Added a type field to `PythonError` (e.g., a `StopIteration` error would have `e.type === "StopIteration"`) [#3289](#)
- Enhancement It is now possible to use asynchronous Python generators from JavaScript. [#3290](#)
- Enhancement PyProxies of synchronous and asynchronous Python generators now support `return` and `throw` APIs that behave like the ones on JavaScript generators. [#3346](#)
- Enhancement It is possible to make a `PyProxy` that takes `this` as the first argument using the `PyProxy.captureThis` method. The `create_proxy` method also has a `capture_this` argument which causes the `PyProxy` to receive `this` as the first argument if set to `True` [#3103](#), [#3145](#)

JavaScript API

- Enhancement Users can do a static import of `pyodide/pyodide.asm.js` to avoid issues with dynamic imports. This allows the use of Pyodide with module-type service workers. [#3070](#)
- Enhancement Added a new API `pyodide.mountNativeFS` which mounts a `FileSystemDirectoryHandle` into the Pyodide file system. [#2987](#)
- Enhancement `loadPyodide` has a new option called `args`. This list will be passed as command line arguments to the Python interpreter at start up. [#3021](#), [#3282](#)
- Removed “Python initialization complete” message printed when loading is finished. `{pr}`3247`
- BREAKING CHANGE The `messageCallback` and `errorCallback` argument to `loadPackage` and `loadPackagesFromImports` is now passed as named arguments. The old usage still works with a deprecation warning. [#3149](#)
- Enhancement `loadPackage` and `loadPackagesFromImports` now accepts a new option `checkIntegrity`. If set to `False`, integrity check for Python Packages will be disabled.
- Enhancement Added APIs `pyodide.setStdin`, `pyodide.setStdout`, `pyodide.setStderr` for changing the stream handlers after loading Pyodide. Also added more careful control over whether `isatty` returns `true` or `false` on `stdin`, `stdout`, and `stderr`. [#3268](#)

Package Loading

- Enhancement Pyodide now shows more helpful error messages when importing packages that are included in Pyodide fails. [#3137](#), [#3263](#)
- Fix Shared libraries with version suffixes are now handled correctly. [#3154](#)
- BREAKING CHANGE Unvendored the `sqlite3` module from the standard library. Before `sqlite3` was included by default. Now it needs to be loaded with `pyodide.loadPackage` or `micropip.install`. [#2946](#)
- BREAKING CHANGE The Pyodide Python package is installed into `/lib/python3.10` rather than `/lib/python3.10/site-packages`. [#3022](#)
- BREAKING CHANGE The `matplotlib` HTML5 backends are now available as part of the `matplotlib-pyodide` package. If you use the default backend from Pyodide, no changes are necessary. However, if you previously specified the backend with `matplotlib.use`, the URL is now different. See [package readme](#) for more details. [#3061](#)
- BREAKING CHANGE The `micropip` package was moved to a separate repository `pyodide/micropip`. In addition to installing the version shipped with a given Pyodide release, you can also install a different `micropip` version from `PyPi` with,


```
await pyodide.loadPackage('packaging')
await pyodide.loadPackage('<URL of the micropip wheel on PyPI>')
```

from Javascript. From Python you can import the Javascript Pyodide package,

```
import pyodide_js
```

and call the same functions as above. #3122

- Enhancement The parsing and validation of `meta.yaml` according to the specification is now done more rigorously with Pydantic. #3079
- BREAKING CHANGE The `source/md5` checksum field is not longer supported in `meta.yaml` files, use `source/sha256` instead #3079
- BREAKING CHANGE `pyodide_build.io.parse_package_config` function is removed in favor of `pyodide_build.MetaConfig.from_yaml` #3079
- Fix `ctypes.util.find_library` will now search WASM modules from `LD_LIBRARY_PATH`. #3353

Build System

- Enhancement Updated Emscripten to version 3.1.27 #2958, #2950, #3027, #3107, #3148, #3236, #3239, #3280, #3314
- Enhancement Added `requirements/host` key to the `meta.yaml` spec to allow host dependencies that are required for building packages. #2132
- Enhancement Added `package/top-level` key to the `meta.yaml` spec to calculate top-level import names for the package. Previously `test/imports` key was used for this purpose. #3006
- Enhancement Added `build/vendor-sharedlib` key to the `meta.yaml` spec which vendors shared libraries into the wheel after building. #3234 #3264
- Enhancement Added `build/type` key to the `meta.yaml` spec which specifies the type of the package. #3238
- Enhancement Added `requirements/executable` key to the `meta.yaml` spec which specifies the list of executables required for building a package. #3300
- BREAKING CHANGE `build/library` and `build/sharedlibrary` key in the `meta.yaml` spec are removed. Use `build/type` instead. #3238
- Fix Fixed a bug that `backend-flags` propagated to dependencies. #3153
- Fix Fixed a bug that shared libraries are not copied into distribution directory when it is already built. #3212
- Enhancement Added a system for making Pyodide virtual environments. This is for testing out of tree builds. For more information, see [the documentation](#). #2976, #3039, #3040, #3044, #3096, #3098, #3108, #3109, #3241
- Added a new CLI command `pyodide skeleton` which creates a package build recipe. `pyodide-build mkpkg` will be replaced by `pyodide skeleton pypi`. #3175
- Added a new CLI command `pyodide build-recipes` which build packages from recipe folder. It replaces `pyodide-build buildall`. #3196 #3279
- Added a new CLI command `pyodide config` which shows config variables used in Pyodide. #3376
- Added subcommands for `pyodide build` which builds packages from various sources.

command	result
<code>pyodide build pypi</code>	build or fetch a single package from pypi
<code>pyodide build source</code>	build the current source folder (same as <code>pyodide build</code>)
<code>pyodide build url</code>	build or fetch a package from a url either <code>tgz</code> , <code>tar.gz</code> <code>zip</code> or <code>wheel</code>
<code>#3196</code>	

Packages

- New packages: `pycryptodome` #2965, `coverage-py` #3053, `bcrypt` #3125, `lightgbm` #3138, `pyheif`, `pillow_heif`, `libheif`, `libde265` #3161, `wordcloud` #3173, `gdal`, `fiona`, `geopandas` #3213, the standard library `_hashlib` module #3206, `pyinstrument` #3258, `gensim` #3326, `smart_open` #3326, `pyodide-http` #3355.
- Fix Scipy CSR data is now handled correctly in XGBoost. #3194
- Update Upgraded packages: SciPy 1.9.1 #3043, pandas 1.5.0 #3134, numpy 1.23.3 #3284, scikit-learn 1.1.3 #3324 as well as most of the other packages #3348 #3365. See *Packages built in Pyodide* for more details.
- Fix Fix scipy handling of exceptions that are raised from C++ code. #3384.

List of Contributors

Aierie, dataxerik, David Lechner, Deepak Cherian, Filipe, Gyeongjae Choi, Hood Chatham, H.Yamada, Jacques Boscq, Jeremy Tuloup, Joe Marshall, John Wason, Loïc Estève, partev, Patrick Arminio, Péter Ferenc Gyarmati, Prete, Qijia Liu, Roman Yurchak, ryanking13, skelsec, Starz0r, Will Lachance, YeonWoo, Yizhi Liu

Version 0.21.3

September 15, 2022

- Fix When loading `sqlite3`, `loadPackage` no longer also loads `nlTK` and `regex`. #3001
- Fix Packages are now loaded in a topologically sorted order regarding their dependencies. #3020
- **BREAKING CHANGE** Loading the `soupsieve` package will not automatically load `beautifulsoup4` together. #3020
- Fix Fix the incorrect package name `ruamel` to `ruamel.yaml`. #3036
- Fix `loadPyodide` will now raise error when the version of JavaScript and Python Pyodide package does not match. #3074
- Enhancement Pyodide now works with a content security policy that doesn't include `unsafe-eval`. It is still necessary to include `wasm-unsafe-eval` (and probably always will be). Since current Safari versions have no support for `wasm-unsafe-eval`, it is necessary to include `unsafe-eval` in order to work in Safari. This will likely be fixed in the next Safari release: https://bugs.webkit.org/show_bug.cgi?id=235408 #3075
- Fix It works again to use `loadPyodide` with a relative URL as `indexURL` (this was a regression in v0.21.2). #3077
- Fix Add `url` to list of pollyfilled packages for webpack compatibility. #3080
- Fix Fixed warnings like `Critical dependency: the request of a dependency is an expression.` when using Pyodide with webpack. #3080
- Enhancement Add binary files to exports in JavaScript package #3085.

- Fix Source maps are included in the distribution again (reverting [#3015](#) included in 0.21.2) and if there is a variable in top level scope called `__dirname` we use that for the `indexURL`. [#3088](#)
- Fix `PyProxy.apply` now correctly handles the case when something unexpected is passed as the second argument. [#3101](#)

Version 0.21.2

August 29, 2022

- Fix The standard library packages `ssl` and `lzma` can now be installed with `pyodide.loadPackage("ssl")` or `micropip.install("ssl")` (previously they had a leading underscore and it was only possible to load them with `pyodide.loadPackage`). [#3003](#)
- Fix If a wheel path is passed to `pyodide.loadPackage`, it will now be resolved relative to `document.location` (in browser) or relative to the current working directory (in Node) rather than relative to `indexURL`. [#3013](#), [#3011](#)
- Fix Fixed a bug in Emscripten that caused Pyodide to fail in Jest. [#3014](#)
- Fix It now works to pass a relative url to `indexURL`. Also, the calculated index URL now works even if `node` is run with `--enable-source-maps`. [#3015](#)

Version 0.21.1

August 22, 2022

- New packages: the standard library `lzma` module [#2939](#)
- Enhancement Pyodide now shows more helpful error messages when importing unvendored or removed stdlib modules fails. [#2973](#)
- **BREAKING CHANGE** The default value of `fullStdLib` in `loadPyodide` has been changed to `false`. This means Pyodide now will not load some stdlib modules like `distutils`, `ssl`, and `sqlite3` by default. See [Pyodide Python compatibility](#) for detail. If `fullStdLib` is set to `true`, it will load all unvendored stdlib modules. However, setting `fullStdLib` to `true` will increase the initial Pyodide load time. So it is preferable to explicitly load the required module. [#2998](#)
- Enhancement `pyodide build` now checks that the correct version of the Emscripten compiler is used. [#2975](#), [#2990](#)
- Fix Pyodide works in Safari v14 again. It was broken in v0.21.0 [#2994](#)

Version 0.21.0

August 9, 2022

[See the release notes for a summary.](#)

Build system

- Enhancement Emscripten was updated to Version 3.1.14 [#2775](#), [#2679](#), [#2672](#)
- Fix Fix building on macOS [#2360](#) [#2554](#)
- Enhancement Update Typescript target to ES2017 to generate more modern Javascript code. [#2471](#)
- Enhancement We now put our built files into the `dist` directory rather than the `build` directory. [#2387](#)
- Fix The build will error out earlier if `cmake` or `libtool` are not installed. [#2423](#)
- Enhancement The platform tags of wheels now include the Emscripten version in them. This should help ensure ABI compatibility if Emscripten wheels are distributed outside of the main Pyodide distribution. [#2610](#)
- Enhancement The build system now uses the `sysconfigdata` from the target Python rather than the host Python. [#2516](#)
- Enhancement Pyodide now builds with `-sWASM_BIGINT`. [#2643](#)
- Enhancement Added `cross-script` key to the `meta.yaml` spec to allow executing custom logic in the cross build environment. [#2734](#)

Pyodide Module and type conversions

- API Change All functions were moved out of the root `pyodide` package into various submodules. For backwards compatibility, they will be available from the root package (raising a `FutureWarning`) until v0.23.0. [#2787](#), [#2790](#)
- Enhancement `loadPyodide` no longer uses any global state, so it can be used more than once in the same thread. This is recommended if a network request causes a loading failure, if there is a fatal error, if you damage the state of the runtime so badly that it is no longer usable, or for certain testing purposes. It is not recommended for creating multiple execution environments, for which you should use `pyodide.runPython(code, { globals : some_dict })`; [#2391](#)
- Enhancement `pyodide.unpackArchive` now accepts any `ArrayBufferView` or `ArrayBuffer` as first argument, rather than only a `Uint8Array`. [#2451](#)
- Feature Added `pyodide.code.run_js` API. [#2426](#)
- Fix `BigInt`'s between $2^{\{32*n - 1\}}$ and $2^{\{32*n\}}$ no longer get translated to negative Python ints. [#2484](#)
- Fix Pyodide now correctly handles JavaScript objects with `null` constructor. [#2520](#)
- Fix Fix garbage collection of `once_callable` [#2401](#)
- Enhancement Added the `js_id` attribute to `JsProxy` to allow using JavaScript object identity as a dictionary key. [#2515](#)
- Fix Fixed a bug with `toJs` when used with recursive structures and the `dictConverter` argument. [#2533](#)
- Enhancement Added Python wrappers `set_timeout`, `clear_timeout`, `set_interval`, `clear_interval`, `add_event_listener` and `remove_event_listener` for the corresponding JavaScript functions. [#2456](#)
- Fix If a request fails due to CORS, `pyfetch` now raises an `OSError` not a `JSException`. [#2598](#)
- Enhancement Pyodide now directly exposes the Emscripten `PATH` and `ERRNO_CODES` APIs. [#2582](#)
- Fix The `bool` operator on a `JsProxy` now behaves more consistently: it returns `False` if JavaScript would say that `!!x` is `false`, or if `x` is an empty container. Otherwise it returns `True`. [#2803](#)
- Fix Fix `loadPyodide` errors for the Windows Node environment. [#2888](#)
- Enhancement Implemented slice subscripting, `+=`, and `extend` for `JsProxy` of Javascript arrays. [#2907](#)

REPL

- Enhancement Add a spinner while the REPL is loading [#2635](#)
- Enhancement Cursor blinking in the REPL can be disabled by setting `noblink` in URL search params. [#2666](#)
- Fix Fix a REPL error in printing high-dimensional lists. [#2517](#) [#2919](#)
- Fix Fix output bug with using `input()` on online console [#2509](#)

micropip and package loading

- API Change `packages.json` which contains the dependency graph for packages was renamed to `repodata.json` to avoid confusion with `package.json` used in JavaScript packages.
- Enhancement Added SHA-256 hash of package to entries in `repodata.json` [#2455](#)
- Enhancement Integrity of Pyodide packages is now verified before loading them. This is for now limited to browser environments. [#2513](#)
- Enhancement `micropip` supports loading wheels from the Emscripten file system using the `emfs:` protocol now. [#2767](#)
- Enhancement It is now possible to use an alternate `repodata.json` lockfile by passing the `lockFileURL` option to `loadPyodide`. This is particularly intended to be used with `micropip.freeze`. [#2645](#)
- Fix `micropip` now correctly handles package names that include dashes [#2414](#)
- Enhancement Allow passing credentials to `micropip.install()` [#2458](#)
- Enhancement `micropip.install()` now accepts a `deps` parameter. If set to `False`, `micropip` will not install dependencies of the package. [#2433](#)
- Fix `micropip` now correctly compares packages with prerelease version [#2532](#)
- Enhancement `micropip.install()` now accepts a `pre` parameter. If set to `True`, `micropip` will include pre-release and development versions. [#2542](#)
- Enhancement `micropip` was refactored to improve readability and ease of maintenance. [#2561](#), [#2563](#), [#2564](#), [#2565](#), [#2568](#)
- Enhancement Various error messages were fine tuned and improved. [#2562](#), [#2558](#)
- Enhancement `micropip` was adjusted to keep its state in the wheel `.dist-info` directories which improves consistency with the Python standard library and other tools used to install packages. [#2572](#)
- Enhancement `micropip` can now be used to install Emscripten binary wheels. [#2591](#)
- Enhancement Added `micropip.freeze` to record the current set of loaded packages into a `repodata.json` file. [#2581](#)
- Fix `micropip.list` now works correctly when there are packages that are installed via `pyodide.loadPackage` from a custom URL. [#2743](#)
- Fix `micropip` now skips package versions which do not follow PEP440. [#2754](#)
- Fix `micropip` supports extra markers in packages correctly now. [#2584](#)

Packages

- Enhancement Update sqlite version to latest stable release [#2477](#) and [#2518](#)
- Enhancement Pillow now supports WEBP image format [#2407](#).
- Enhancement Pillow and opencv-python now support the TIFF image format. [#2762](#)
- Pandas is now compiled with `-Oz`, which significantly speeds up loading the library on Chrome [#2457](#)
- New packages: opencv-python [#2305](#), ffmpeg [#2305](#), libwebp [#2305](#), h5py, pkgconfig and libhdf5 [#2411](#), bitarray [#2459](#), gsw [#2511](#), cftime [#2504](#), svgwrite, jsonschema, tskit [#2506](#), xarray [#2538](#), demes, libgsl, newick, ruamel, msprime [#2548](#), gmpy2 [#2665](#), xgboost [#2537](#), galpy [#2676](#), shapely, geos [#2725](#), suitesparse, sparseqr [#2685](#), libtiff [#2762](#), pytest-benchmark [#2799](#), termcolor [#2809](#), sqlite3, libproj, pyproj, certifi [#2555](#), rebound [#2868](#), reboundx [#2909](#), pyclipper [#2886](#), brotli [#2925](#), python-magic [#2941](#)

Miscellaneous

- Fix We now tell packagers (e.g., Webpack) to ignore npm-specific imports when packing files for the browser. [#2468](#)
- Enhancement `run_in_pyodide` now has support for pytest assertion rewriting and decorators such as `pytest.mark.parametrize` and `hypothesis`. [#2510](#), [#2541](#)
- BREAKING CHANGE `pyodide_build.testing` is removed. `run_in_pyodide` decorator can now be accessed through `pytest-pyodide` package. [#2418](#)

List of contributors

Alexey Ignatiev, Andrey Smelter, andrzej, Antonio Cuni, Ben Jeffery, Brian Benjamin Maranville, David Lechner, drag-oncoder047, echorand (Amit Saha), Filipe, Frank, Gyeongjae Choi, Hanno Rein, haoran1062, Henry Schreiner, Hood Chatham, Jason Grout, jmdyck, Jo Bovy, John Wason, josephrocca, Kyle Cutler, Lester Fan, Liumeo, lukemarsden, Mario Gersbach, Matt Toad, Michael Droettboom, Michael Gilbert, Michael Neil, Mu-Tsun Tsai, Nicholas Bollweg, pysathq, Ricardo Prins, Rob Gries, Roman Yurchak, Ryan May, Ryan Russell, stonebig, Szymswiat, Tobias Megies, Vic Kumar, Victor, Wei Ji, Will Lachance

Version 0.20.0

April 9th, 2022

[See the release notes for a summary.](#)

CPython and stdlib

- Update Pyodide now runs Python 3.10.2. [#2225](#)
- Enhancement All `ctypes` tests pass now except for `test_callback_too_many_args` (and we have a plan to fix `test_callback_too_many_args` upstream). `libffi-emsripten` now also passes all `libffi` tests. [#2350](#)

Packages

- Fix matplotlib now loads multiple fonts correctly [#2271](#)
- New packages: boost-histogram [#2174](#), cryptography v3.3.2 [#2263](#), the standard library ssl module [#2263](#), python-solvespace v3.0.7, lazy-object-proxy [#2320](#).
- Many more scipy linking errors were fixed, mostly related to the Fortran f2c ABI for string arguments. There are still some fatal errors in the Scipy test suite, but none seem to be simple linker errors. [#2289](#)
- Removed pyodide-interrupts. If you were using this for some reason, use `pyodide.setInterruptBuffer` instead. [#2309](#)
- Most included packages were updated to the latest version. See *Packages built in Pyodide* for a full list.

Type translations

- Fix Python tracebacks now include Javascript frames when Python calls a Javascript function. [#2123](#)
- Enhancement Added a `default_converter` argument to `JsProxy.to_py` and `pyodide.toPy` which is used to process any object that doesn't have a built-in conversion to Python. Also added a `default_converter` argument to `PyProxy.toJs` and `pyodide.ffi.to_js` to convert. [#2170](#) and [#2208](#)
- Enhancement Async Python functions called from Javascript now have the resulting coroutine automatically scheduled. For instance, this makes it possible to use an async Python function as a Javascript event handler. [#2319](#)

Javascript package

- Enhancement It is no longer necessary to provide `indexURL` to `loadPyodide`. [#2292](#)
- BREAKING CHANGE The `globals` argument to `pyodide.runPython` and `pyodide.runPythonAsync` is now passed as a named argument. The old usage still works with a deprecation warning. [#2300](#)
- Enhancement The Javascript package was migrated to Typescript. [#2130](#) and [#2133](#)
- Fix Fix importing pyodide with ESM syntax in a module type web worker. [#2220](#)
- Enhancement When Pyodide is loaded as an ES6 module, no global `loadPyodide` variable is created (instead, it should be accessed as an attribute on the module). [#2249](#)
- Fix The type `Py2JsResult` has been replaced with `any` which is more accurate. For backwards compatibility, we still export `Py2JsResult` as an alias for `any`. [#2277](#)
- Fix Pyodide now loads correctly even if `requirejs` is included. [#2283](#)
- Enhancement Added robust handling for non-`Error` objects thrown by Javascript code. This mostly should never happen since well behaved Javascript code ought to throw errors. But it's better not to completely crash if it throws something else. [#2294](#)

pyodide_build

- Enhancement Pyodide now uses Python wheel files to distribute packages rather than the `emscripten file_packager.py` format. [#2027](#)
- Enhancement Pyodide now uses `pypa/build` to build packages. We (mostly) use build isolation, so we can build packages that require conflicting versions of `setuptools` or alternative build backends. [#2272](#)
- Enhancement Most pure Python packages were switched to use the wheels directly from PyPI rather than rebuilding them. [#2126](#)
- Enhancement Added support for C++ exceptions in packages. Now C++ extensions compiled and linked with `-fexceptions` can catch C++ exceptions. Furthermore, uncaught C++ exceptions will be formatted in a human-readable way. [#2178](#)
- BREAKING CHANGE Removed the `skip-host` key from the `meta.yaml` format. If needed, install a host copy of the package with `pip` instead. [#2256](#)

Uncategorized

- Enhancement The interrupt buffer can be used to raise all 64 signals now, not just `SIGINT`. Write a number between `1 <= signal <= 64` into the interrupt buffer to trigger the corresponding signal. By default everything but `SIGINT` will be ignored. Any value written into the interrupt buffer outside of the range from 1 to 64 will be silently discarded. [#2301](#)
- Enhancement Updated to Emscripten 2.0.27. [#2295](#)
- BREAKING CHANGE The `extractDir` argument to `pyodide.unpackArchive` is now passed as a named argument. The old usage still works with a deprecation warning. [#2300](#)
- Enhancement Support ANSI escape codes in the Pyodide console. [#2345](#)
- Fix `pyodide_build` can now be installed in non-editable ways. [#2351](#)

List of contributors

Boris Feld, Christian Staudt, Gabriel Fougerson, Gyeongjae Choi, Henry Schreiner, Hood Chatham, Jo Bovy, Karthikeyan Singaravelan, Leo Psidom, Liumeo, Luka Mamukashvili, Madhur Tandon, Paul Korzhyk, Roman Yurchak, Seungmin Kim, Thorsten Beier, Tom White, and Will Lachance

Version 0.19.1

February 19, 2022

Packages

- New packages: `sqlalchemy` [#2112](#), `pydantic` [#2117](#), `wrapt` [#2165](#)
- Update Upgraded packages: `pyb2d` (0.7.2), [#2117](#)
- Fix A fatal error in `scipy.stats.binom.ppf` has been fixed. [#2109](#)
- Fix Type signature mismatches in some numpy comparators have been fixed. [#2110](#)

Type translations

- Fix The “PyProxy has already been destroyed” error message has been improved with some context information. [#2121](#)

REPL

- Enhancement Pressing TAB in REPL no longer triggers completion when input is whitespace. [#2125](#)

List of contributors

Christian Staudt, Gyeongjae Choi, Hood Chatham, Liumeo, Paul Korzyk, Roman Yurchak, Seungmin Kim, Thorsten Beier

Version 0.19.0

January 10, 2021

[See the release notes for a summary.](#)

Python package

- Enhancement If `find_imports` is used on code that contains a syntax error, it will return an empty list instead of raising a `SyntaxError`. [#1819](#)
- Enhancement Added the `pyodide.http.pyfetch` API which provides a convenience wrapper for the Javascript `fetch` API. The API returns a response object with various methods that convert the data into various types while minimizing the number of times the data is copied. [#1865](#)
- Enhancement Added the `unpack_archive` API to the `pyodide.http.FetchResponse` object which treats the response body as an archive and uses `shutil` to unpack it. [#1935](#)
- Fix The Pyodide event loop now works correctly with cancelled handles. In particular, `asyncio.wait_for` now functions as expected. [#2022](#)

JavaScript package

- Fix `loadPyodide` no longer fails in the presence of a user-defined global named `process`. [#1849](#)
- Fix Various webpack buildtime and runtime compatibility issues were fixed. [#1900](#)
- Enhancement Added the `pyodide.pyimport` API to import a Python module and return it as a `PyProxy`. Warning: this is different from the original `pyimport` API which was removed in this version. [#1944](#)
- Enhancement Added the `pyodide.unpackArchive` API which unpacks an archive represented as an `ArrayBuffer` into the working directory. This is intended as a way to install packages from a local application. [#1944](#)
- API Change `loadPyodide` now accepts a `homedir` parameter which sets home directory of Pyodide virtual file system. [#1936](#)
- BREAKING CHANGE The default working directory(home directory) inside the Pyodide virtual file system has been changed from `/` to `/home/pyodide`. To get the previous behavior, you can
 - call `os.chdir("/")` in Python to change working directory or

- call `loadPyodide` with the `homedir="/"` argument [#1936](#)

Python / JavaScript type conversions

- **BREAKING CHANGE** Updated the calling convention when a JavaScript function is called from Python to improve memory management of PyProxies. PyProxy arguments and return values are automatically destroyed when the function is finished. [#1573](#)
- **Enhancement** Added `JsProxy.to_string`, `JsProxy.to_bytes`, and `JsProxy.to_memoryview` to allow for conversion of `TypedArray` to standard Python types without unneeded copies. [#1864](#)
- **Enhancement** Added `JsProxy.to_file` and `JsProxy.from_file` to allow reading and writing Javascript buffers to files as a byte stream without unneeded copies. [#1864](#)
- **Fix** It is now possible to destroy a borrowed attribute PyProxy of a PyProxy (as introduced by [#1636](#)) before destroying the root PyProxy. [#1854](#)
- **Fix** If `__iter__()` raises an error, it is now handled correctly by the `PyProxy[Symbol.iterator()]` method. [#1871](#)
- **Fix** Borrowed attribute PyProxys are no longer destroyed when the root PyProxy is garbage collected (because it was leaked). Doing so has no benefit to nonleaky code and turns some leaky code into broken code (see [#1855](#) for an example). [#1870](#)
- **Fix** Improved the way that `pyodide.globals.get("builtin_name")` works. Before we used `__main__.__dict__.update(builtins.__dict__)` which led to several undesirable effects such as `__name__` being equal to "builtins". Now we use a proxy wrapper to replace `pyodide.globals.get` with a function that looks up the name on builtins if lookup on globals fails. [#1905](#)
- **Enhancement** Coroutines have their memory managed in a more convenient way. In particular, now it is only necessary to either `await` the coroutine or call one of `.then`, `.except` or `.finally` to prevent a leak. It is no longer necessary to manually destroy the coroutine. Example: before:

```
async function runPythonAsync(code, globals) {
  let coroutine = Module.pyodide_py.eval_code_async(code, globals);
  try {
    return await coroutine;
  } finally {
    coroutine.destroy();
  }
}
```

After:

```
async function runPythonAsync(code, globals) {
  return await Module.pyodide_py.eval_code_async(code, globals);
}
```

[#2030](#)

pyodide-build

- API Change By default only a minimal set of packages is built. To build all packages set `PYODIDE_PACKAGES='*'` In addition, `make minimal` was removed, since it is now equivalent to `make` without extra arguments. #1801
- Enhancement It is now possible to use `pyodide-build buildall` and `pyodide-build buildpkg` directly. #2063
- Enhancement Added a `--force-rebuild` flag to `buildall` and `buildpkg` which rebuilds the package even if it looks like it doesn't need to be rebuilt. Added a `--continue` flag which keeps the same source tree for the package and can continue from the middle of a build. #2069
- Enhancement Changes to environment variables in the build script are now seen in the compile and post build scripts. #1706
- Fix Fix usability issues with `pyodide-build mkpkg` CLI. #1828
- Enhancement Better support for `ccache` when building Pyodide #1805
- Fix Fix compile error `wasm-ld: error: unknown argument: --sort-common` and `wasm-ld: error: unknown argument: --as-needed` in ArchLinux. #1965

micropip

- Fix micropip now raises an error when installing a non-pure python wheel directly from a url. #1859
- Enhancement `micropip.install()` now accepts a `keep_going` parameter. If set to `True`, micropip reports all identifiable dependencies that don't have pure Python wheels, instead of failing after processing the first one. #1976
- Enhancement Added a new API `micropip.list()` which returns the list of installed packages by micropip. #2012

Packages

- Enhancement Unit tests are now unvendored from Python packages and included in a separate package `<package name>-tests`. This results in a 20% size reduction on average for packages that vendor tests (e.g. `numpy`, `pandas`, `scipy`). #1832
- Update Upgraded SciPy to 1.7.3. There are known issues with some SciPy components, the current status of the scipy test suite is [here](#) #2065
- Fix The built-in `pwd` module of Python, which provides a Unix specific feature, is now unvendored. #1883
- Fix `pillow` and `imageio` now correctly encode/decode grayscale and black-and-white JPEG images. #2028
- Fix The `numpy fft` module now works correctly. #2028
- New packages: `logbook` #1920, `pyb2d` #1968, and `threadpoolctl` (a dependency of `scikit-learn`) #2065
- Upgraded packages: `numpy` (1.21.4) #1934, `scikit-learn` (1.0.2) #2065, `scikit-image` (0.19.1) #2005, `msgpack` (1.0.3) #2071, `astropy` (5.0.3) #2086, `statsmodels` (0.13.1) #2073, `pillow` (9.0.0) #2085. This list is not exhaustive, refer to `packages.json` for the full list.

Uncategorized

- Enhancement `PyErr_CheckSignals` now works with the keyboard interrupt system so that cooperative C extensions can be interrupted. Also, added the `pyodide.checkInterrupt` function so Javascript code can opt to be interrupted. [#1294](#)
- Fix The `_` variable is now set by the Pyodide repl just like it is set in the native Python repl. [#1904](#)
- Enhancement `pyodide-env` and `pyodide` Docker images are now available from both the [Docker Hub](#) and from the [Github Package registry](#). [#1995](#)
- Fix The console now correctly handles it when an object's `__repr__` function raises an exception. [#2021](#)
- Enhancement Removed the `-s EMULATE_FUNCTION_POINTER_CASTS` flag, yielding large benefits in speed, stack usage, and code size. [#2019](#)

List of contributors

Alexey Ignatiev, Alex Hall, Bart Broere, Cyrille Bogaert, etienne, Grimmer, Grimmer Kang, Gyeongjae Choi, Hao Zhang, Hood Chatham, Ian Clester, Jan Max Meyer, LeoPsidom, Liumeo, Michael Christensen, Owen Ou, Roman Yurchak, Seungmin Kim, Sylvain, Thorsten Beier, Wei Ouyang, Will Lachance

Version 0.18.1

September 16, 2021

Console

- Fix Ctrl+C handling in console now works correctly with multiline input. New behavior more closely approximates the behavior of the native Python console. [#1790](#)
- Fix Fix the repr of Python objects (including lists and dicts) in console [#1780](#)
- Fix The “long output truncated” message now appears on a separate line as intended. [#1814](#)
- Fix The streams that are used to redirect stdin and stdout in the console now define `isatty` to return `True`. This fixes `pytest`. [#1822](#)

Python package

- Fix Avoid circular references when `runsource` raises `SyntaxError` [#1758](#)

JavaScript package

- Fix The `pyodide.setInterruptBuffer` command is now publicly exposed again, as it was in v0.17.0. [#1797](#)

Python / JavaScript type conversions

- Fix Conversion of very large strings from JavaScript to Python works again. [#1806](#)
- Fix Fixed a use after free bug in the error handling code. [#1816](#)

Packages

- Fix pillow now correctly encodes/decodes RGB JPEG image format. [#1818](#)

Micellaneous

- Fix Patched emscripten to make the system calls to duplicate file descriptors closer to posix-compliant. In particular, this fixes the use of dup on pipes and temporary files, as needed by pytest. [#1823](#)

Version 0.18.0

August 3rd, 2021

General

- Update Pyodide now runs Python 3.9.5. [#1637](#)
- Enhancement Pyodide can experimentally be used in Node.js [#1689](#)
- Enhancement Pyodide now directly exposes the [Emscripten filesystem API](#), allowing for direct manipulation of the in-memory filesystem [#1692](#)
- Enhancement Pyodide's support of [emscripten file systems](#) is expanded from the default MEMFS to include IDBFS, NODEFS, PROXYFS, and WORKERFS, allowing for custom persistence strategies depending on execution environment [#1596](#)
- API Change The `packages.json` schema for Pyodide was redesigned for better compatibility with conda. [#1700](#)
- API Change `run_docker` no longer binds any port to the docker image by default. [#1750](#)

Standard library

- API Change The following standard library modules are now available as standalone packages
 - `distlib`

They are loaded by default in `loadPyodide`, however this behavior can be disabled with the `fullStdLib` parameter set to `false`. All optional stdlib modules can then be loaded as needed with `pyodide.loadPackage`. [#1543](#)

- Enhancement The standard library module `audioop` is now included, making the `wave`, `sndhdr`, `aifc`, and `sunau` modules usable. [#1623](#)
- Enhancement Added support for `ctypes`. [#1656](#)

JavaScript package

- Enhancement The Pyodide JavaScript package is released to npm under npmjs.com/package/pyodide #1762
- API Change `loadPyodide` no longer automatically stores the API into a global variable called `pyodide`. To get old behavior, say `globalThis.pyodide = await loadPyodide({...})`. #1597
- Enhancement `loadPyodide` now accepts callback functions for `stdin`, `stdout` and `stderr` #1728
- Enhancement Pyodide now ships with first party typescript types for the entire JavaScript API (though no typings are available for `PyProxy` fields). #1601
- Enhancement It is now possible to import `Comlink` objects into Pyodide after using `pyodide.registerComlink` #1642
- Enhancement If a Python error occurs in a reentrant `runPython` call, the error will be propagated into the outer `runPython` context as the original error type. This is particularly important if the error is a `KeyboardInterrupt`. #1447

Python package

- Enhancement Added a new `pyodide.code.CodeRunner` API for finer control than `eval_code` and `eval_code_async`. Designed with the needs of REPL implementations in mind. #1563
- Enhancement Added `pyodide.console.Console` class closely based on the Python standard library `InteractiveConsole` but with support for top level `await` and stream redirection. Also added the subclass `pyodide.console.PyodideConsole` which automatically uses `pyodide.loadPackagesFromImports` on the code before running it. #1125, #1155, #1635
- Fix `pyodide.code.eval_code_async` no longer automatically awaits a returned coroutine or attempts to await a returned generator object (which triggered an error). #1563

Python / JavaScript type conversions

- API Change `pyodide.runPythonAsync` no longer automatically calls `pyodide.loadPackagesFromImports`. #1538.
- Enhancement Added the `PyProxy.callKwargs` method to allow using Python functions with keyword arguments from JavaScript. #1539
- Enhancement Added the `PyProxy.copy` method. #1549 #1630
- API Change Updated the method resolution order on `PyProxy`. Performing a lookup on a `PyProxy` will prefer to pick a method from the `PyProxy` api, if no such method is found, it will use `getattr` on the proxied object. Prefixing a name with `$` forces `getattr`. For instance, `PyProxy.destroy` now always refers to the method that destroys the proxy, whereas `PyProxy.$destroy` refers to an attribute or method called `destroy` on the proxied object. #1604
- API Change It is now possible to use `Symbol` keys with `PyProxies`. These `Symbol` keys put markers on the `PyProxy` that can be used by external code. They will not currently be copied by `PyProxy.copy`. #1696
- Enhancement Memory management of `PyProxy` fields has been changed so that fields looked up on a `PyProxy` are “borrowed” and have their lifetime attached to the base `PyProxy`. This is intended to allow for more idiomatic usage. (See #1617.) #1636
- API Change The `depth` argument to `toJs` is now passed as an option, so `toJs(n)` in v0.17 changed to `toJs({depth : n})`. Similarly, `pyodide.toPy` now takes `depth` as a named argument. Also `to_js` and `to_py` only take `depth` as a keyword argument. #1721

- API Change `PyProxy.toJs` and `pyodide.ffi.to_js` now take an option `pyproxies`, if a JavaScript Array is passed for this, then any proxies created during conversion will be placed into this array. This allows easy cleanup later. The `create_pyproxies` option can be used to disable creation of pyproxies during conversion (instead a `ConversionError` is raised). [#1726](#)
- API Change `toJs` and `to_js` now take an option `dict_converter` which will be called on a JavaScript iterable of two-element Arrays as the final step of converting dictionaries. For instance, pass `Object.fromEntries` to convert to an object or `Array.from` to convert to an array of pairs. [#1742](#)

pyodide-build

- API Change `pyodide-build` is now an installable Python package, with an identically named CLI entrypoint that replaces `bin/pyodide` which is removed [#1566](#)

micropip

- Fix micropip now correctly handles packages that have mixed case names. (See [#1614](#)). [#1615](#)
- Enhancement micropip now resolves dependencies correctly for old versions of packages (it used to always use the dependencies from the most recent version, see [#1619](#) and [#1745](#)). micropip also will resolve dependencies for wheels loaded from custom urls. [#1753](#)

Packages

- Enhancement `matplotlib` now comes with a new renderer based on the `html5` canvas element. [#1579](#) It is optional and the current default backend is still the `agg` backend compiled to `wasm`.
- Enhancement Updated a number of packages included in Pyodide.

List of contributors

Albertas Gimbutas, Andreas Klostermann, Arfy Slowy, daoXian, Devin Neal, fuyutarow, Grimmer, Guido Zuidhof, Gyeongjae Choi, Hood Chatham, Ian Clester, Itay Dafna, Jeremy Tuloup, jmsmdy, LinasNas, Madhur Tandon, Michael Christensen, Nicholas Bollweg, Ondřej Staněk, Paul m. p. P, Piet Brömmel, Roman Yurchak, stefnotch, Syrus Akbary, Teon L Brooks, Waldir

Version 0.17.0

April 21, 2021

See the 0-17-0-release-notes for more information.

Improvements to package loading and dynamic linking

- Enhancement Uses the emscripten preload plugin system to preload .so files in packages
- Enhancement Support for shared library packages. This is used for CLAPACK which makes scipy a lot smaller. [#1236](#)
- Fix Pyodide and included packages can now be used with Safari v14+. Safari v13 has also been observed to work on some (but not all) devices.

Python / JS type conversions

- Feature A JsProxy of a JavaScript Promise or other awaitable object is now a Python awaitable. [#880](#)
- API Change Instead of automatically converting Python lists and dicts into JavaScript, they are now wrapped in PyProxy. Added a new PyProxy.toJs API to request the conversion behavior that used to be implicit. [#1167](#)
- API Change Added JsProxy.to_py API to convert a JavaScript object to Python. [#1244](#)
- Feature Flexible jsimports: it now possible to add custom Python “packages” backed by JavaScript code, like the js package. The js package is now implemented using this system. [#1146](#)
- Feature A PyProxy of a Python coroutine or awaitable is now an awaitable JavaScript object. Awaiting a coroutine will schedule it to run on the Python event loop using `asyncio.ensure_future`. [#1170](#)
- Enhancement Made PyProxy of an iterable Python object an iterable Js object: defined the `[Symbol.iterator]` method, can be used like `for(let x of proxy)`. Made a PyProxy of a Python iterator an iterator: `proxy.next()` is translated to `next(it)`. Made a PyProxy of a Python generator into a JavaScript generator: `proxy.next(val)` is translated to `gen.send(val)`. [#1180](#)
- API Change Updated PyProxy so that if the wrapped Python object supports `__getitem__` access, then the wrapper has `get`, `set`, `has`, and `delete` methods which do `obj[key]`, `obj[key] = val`, `key in obj` and `del obj[key]` respectively. [#1175](#)
- API Change The `pyodide.pyimport` function is deprecated in favor of using `pyodide.globals.get('key')`. [#1367](#)
- API Change Added `PyProxy.getBuffer` API to allow direct access to Python buffers as JavaScript TypedArrays. [#1215](#)
- API Change The innermost level of a buffer converted to JavaScript used to be a TypedArray if the buffer was contiguous and otherwise an Array. Now the innermost level will be a TypedArray unless the buffer format code is a “?” in which case it will be an Array of booleans, or if the format code is a “s” in which case the innermost level will be converted to a string. [#1376](#)
- Enhancement JavaScript BigInts are converted into Python int and Python ints larger than 2^{53} are converted into BigInt. [#1407](#)
- API Change Added `pyodide.isPyProxy` to test if an object is a PyProxy. [#1456](#)
- Enhancement PyProxy and PyBuffer objects are now garbage collected if the browser supports FinalizationRegistry. [#1306](#)
- Enhancement Automatic conversion of JavaScript functions to CPython calling conventions. [#1051](#), [#1080](#)
- Enhancement Automatic detection of fatal errors. In this case Pyodide will produce both a JavaScript and a Python stack trace with explicit instruction to open a bug report. [pr{1151}](#), [pr{1390}](#), [pr{1478}](#).
- Enhancement Systematic memory leak detection in the test suite and a large number of fixed to memory leaks. [pr{1340}](#)

- Fix `getattr` and `dir` on `JsProxy` now report consistent results and include all names defined on the Python dictionary backing `JsProxy`. [#1017](#)
- Fix `JsProxy.__bool__` now produces more consistent results: both `bool(window)` and `bool(zero-arg-callback)` were `False` but now are `True`. Conversely, `bool(empty_js_set)` and `bool(empty_js_map)` were `True` but now are `False`. [#1061](#)
- Fix When calling a JavaScript function from Python without keyword arguments, Pyodide no longer passes a `PyProxy`-wrapped `NULL` pointer as the last argument. [#1033](#)
- Fix `JsBoundMethod` is now a subclass of `JsProxy`, which fixes nested attribute access and various other strange bugs. [#1124](#)
- Fix JavaScript functions imported like `from js import fetch` no longer trigger “invalid invocation” errors (issue [#461](#)) and `js.fetch("some_url")` also works now (issue [#768](#)). [#1126](#)
- Fix JavaScript bound method calls now work correctly with keyword arguments. [#1138](#)
- Fix JavaScript constructor calls now work correctly with keyword arguments. [#1433](#)

pyodide-py package

- Feature Added a Python event loop to support `asyncio` by scheduling coroutines to run as jobs on the browser event loop. This event loop is available by default and automatically enabled by any relevant `asyncio` API, so for instance `asyncio.ensure_future` works without any configuration. [#1158](#)
- API Change Removed `as_nested_list` API in favor of `JsProxy.to_py`. [#1345](#)

pyodide-js

- API Change Removed iodide-specific code in `pyodide.js`. This breaks compatibility with `iodide`. [#878](#), [#981](#)
- API Change Removed the `pyodide.autocomplete` API, use `Jedi` directly instead. [#1066](#)
- API Change Removed `pyodide.repr` API. [#1067](#)
- Fix If `messageCallback` and `errorCallback` are supplied to `pyodide.loadPackage`, `pyodide.runPythonAsync` and `pyodide.loadPackagesFromImport`, then the messages are no longer automatically logged to the console.
- Feature `runPythonAsync` now runs the code with `eval_code_async`. In particular, it is possible to use top-level `await` inside of `runPythonAsync`.
- `eval_code` now accepts separate `globals` and `locals` parameters. [#1083](#)
- Added the `pyodide.setInterruptBuffer` API. This can be used to set a `SharedArrayBuffer` to be the keyboard interrupt buffer. If Pyodide is running on a webworker, the main thread can signal to the webworker that it should raise a `KeyboardInterrupt` by writing to the interrupt buffer. [#1148](#) and [#1173](#)
- Changed the loading method: added an `async` function `loadPyodide` to load Pyodide to use instead of `languagePluginURL` and `languagePluginLoader`. The change is currently backwards compatible, but the old approach is deprecated. [#1363](#)
- `runPythonAsync` now accepts `globals` parameter. [#1914](#)

micropip

- Feature micropip now supports installing wheels from relative URLs. [#872](#)
- API Change micropip.install now returns a Python Future instead of a JavaScript Promise. [#1324](#)
- Fix micropip.install now interacts correctly with `pyodide.loadPackage()`. [#1457](#)
- Fix micropip.install now handles version constraints correctly even if there is a version of the package available from the Pyodide indexURL.

Build system

- Enhancement Updated to latest emscripten 2.0.13 with the upstream LLVM backend [#1102](#)
- API Change Use upstream `file_packager.py`, and stop checking package abi versions. The `PYODIDE_PACKAGE_ABI` environment variable is no longer used, but is still set as some packages use it to detect whether it is being built for Pyodide. This usage is deprecated, and a new environment variable `PYODIDE` is introduced for this purpose.
As part of the change, `Module.checkABI` is no longer present. [#991](#)
- uglifyjs and lessc no longer need to be installed in the system during build [#878](#).
- Enhancement Reduce the size of the core Pyodide package [#987](#).
- Enhancement Optionally to disable docker port binding [#1423](#).
- Enhancement Run arbitrary command in docker [#1424](#)
- Docker images for Pyodide are now accessible at `pyodide/pyodide-env` and `pyodide/pyodide`.
- Enhancement Option to run docker in non-interactive mode [#1641](#)

REPL

- Fix In `console.html`: sync behavior, full stdout/stderr support, clean namespace, bigger font, correct result representation, clean traceback [#1125](#) and [#1141](#)
- Fix Switched from Jedi to `rlcompleter` for completion in `pyodide.console.InteractiveConsole` and so in `console.html`. This fixes some completion issues (see [#821](#) and [#1160](#))
- Enhancement Support top-level await in the console [#1459](#)

Packages

- `six`, `jedi` and `parso` are no longer vendored in the main Pyodide package, and need to be loaded explicitly [#1010](#), [#987](#).
- Updated packages [#1021](#), [#1338](#), [#1460](#).
- Added Plotly version 4.14.3 and retrying dependency [#1419](#)

List of contributors

(in alphabetic order)

Aditya Shankar, casatir, Dexter Chua, dmondev, Frederik Braun, Hood Chatham, Jan Max Meyer, Jeremy Tuloup, joemarshall, leafjolt, Michael Greminger, Mireille Raad, Ondřej Staněk, Paul m. p. P, rdb, Roman Yurchak, Rudolfs

Version 0.16.1

December 25, 2020

Note: due to a CI deployment issue the 0.16.0 release was skipped and replaced by 0.16.1 with identical contents.

- Pyodide files are distributed by [JsDelivr](https://cdn.jsdelivr.net/pyodide/v0.16.1/full/pyodide.js), <https://cdn.jsdelivr.net/pyodide/v0.16.1/full/pyodide.js> The previous CDN `pyodide-cdn2.iodide.io` still works and there are no plans for deprecating it. However please use JsDelivr as a more sustainable solution, including for earlier Pyodide versions.

Python and the standard library

- Pyodide includes CPython 3.8.2 [#712](#)
- ENH Patches for the threading module were removed in all packages. Importing the module, and a subset of functionality (e.g. locks) works, while starting a new thread will produce an exception, as expected. [#796](#). See [#237](#) for the current status of the threading support.
- ENH The multiprocessing module is now included, and will not fail at import, thus avoiding the necessity to patch included packages. Starting a new process will produce an exception due to the limitation of the WebAssembly VM with the following message: `Resource temporarily unavailable` [#796](#).

Python / JS type conversions

- FIX Only call `Py_INCREF()` once when proxied by `PyProxy` [#708](#)
- JavaScript exceptions can now be raised and caught in Python. They are wrapped in `pyodide.JsException`. [#891](#)

pyodide-py package and micropip

- The `pyodide.py` file was transformed to a `pyodide-py` package. The imports remain the same so this change is transparent to the users [#909](#).
- FIX Get last version from PyPI when installing a module via micropip [#846](#).
- Suppress REPL results returned by `pyodide.eval_code` by adding a semicolon [#876](#).
- Enable monkey patching of `eval_code` and `find_imports` to customize behavior of `runPython` and `runPythonAsync` [#941](#).

Build system

- Updated docker image to Debian buster, resulting in smaller images. [#815](#)
- Pre-built docker images are now available as [iodide-project/pyodide](#) [#787](#)
- Host Python is no longer compiled, reducing compilation time. This also implies that Python 3.8 is now required to build Pyodide. It can for instance be installed with conda. [#830](#)
- FIX Infer package tarball directory from source URL [#687](#)
- Updated to emscripten 1.38.44 and binaryen v86 (see related [commits](#))
- Updated default `--ldflags` argument to `pyodide_build` scripts to equal what Pyodide actually uses. [#817](#)
- Replace C lz4 implementation with the (upstream) JavaScript implementation. [#851](#)
- Pyodide deployment URL can now be specified with the `PYODIDE_BASE_URL` environment variable during build. The `pyodide_dev.js` is no longer distributed. To get an equivalent behavior with `pyodide.js`, set

```
window.languagePluginUrl = "./";
```

before loading it. [#855](#)

- Build runtime C libraries (e.g. libxml) via package build system with correct dependency resolution [#927](#)
- Pyodide can now be built in a conda virtual environment [#835](#)

Other improvements

- Modify MEMFS timestamp handling to support better caching. This in particular allows to import newly created Python modules without invalidating import caches [#893](#)

Packages

- New packages: `freesasa`, `lxml`, `python-sat`, `traits`, `astropy`, `pillow`, `scikit-image`, `imageio`, `numcodecs`, `msgpack`, `asciitree`, `zarr`
Note that due to the large size and the experimental state of the `scipy` package, packages that depend on `scipy` (including `scikit-image`, `scikit-learn`) will take longer to load, use a lot of memory and may experience failures.
- Updated packages: `numpy` 1.15.4, `pandas` 1.0.5, `matplotlib` 3.3.3 among others.
- New package [pyodide-interrupt](#), useful for handling interrupts in Pyodide (see project description for details).

Backward incompatible changes

- Dropped support for loading `.wasm` files with incorrect MIME type, following [#851](#)

List of contributors

abolger, Aditya Shankar, Akshay Philar, Alexey Ignatiev, Aray Karjauv, casatir, chigozienri, Christian glacet, Dexter Chua, Frithjof, Hood Chatham, Jan Max Meyer, Jay Harris, jcaesar, Joseph D. Long, Matthew Turk, Michael Greminger, Michael Panchenko, mojighahar, Nicolas Ollinger, Ram Rachum, Roman Yurchak, Sergio, Seungmin Kim, Shyam Saladi, smkm, Wei Ouyang

Version 0.15.0

May 19, 2020

- Upgrades Pyodide to CPython 3.7.4.
- micropip no longer uses a CORS proxy to install pure Python packages from PyPI. Packages are now installed from PyPI directly.
- micropip can now be used from web workers.
- Adds support for installing pure Python wheels from arbitrary URLs with micropip.
- The CDN URL for Pyodide changed to <https://pyodide-cdn2.iodide.io/v0.15.0/full/pyodide.js> It now supports versioning and should provide faster downloads. The latest release can be accessed via <https://pyodide-cdn2.iodide.io/latest/full/>
- Adds `messageCallback` and `errorCallback` to `pyodide.loadPackage`.
- Reduces the initial memory footprint (`TOTAL_MEMORY`) from 1 GiB to 5 MiB. More memory will be allocated as needed.
- When building from source, only a subset of packages can be built by setting the `PYODIDE_PACKAGES` environment variable. See [partial builds documentation](#) for more details.
- New packages: future, autograd

Version 0.14.3

Dec 11, 2019

- Convert JavaScript numbers containing integers, e.g. `3.0`, to a real Python long (e.g. `3`).
- Adds `__bool__` method to for `JsProxy` objects.
- Adds a JavaScript-side auto completion function for Iodide that uses `jedi`.
- New packages: nltk, jeudi, statsmodels, regex, cytoolz, xlrd, uncertainties

Version 0.14.0

Aug 14, 2019

- The built-in `sqlite` and `bz2` modules of Python are now enabled.
- Adds support for auto-completion based on `jedi` when used in `iodide`

Version 0.13.0

May 31, 2019

- Tagged versions of Pyodide are now deployed to Netlify.

Version 0.12.0

May 3, 2019

User improvements:

- Packages with pure Python wheels can now be loaded directly from PyPI. See `micropip` for more information.
- Thanks to PEP 562, you can now `import js` from Python and use it to access anything in the global JavaScript namespace.
- Passing a Python object to JavaScript always creates the same object in JavaScript. This makes APIs like `removeEventListener` usable.
- Calling `dir()` in Python on a JavaScript proxy now works.
- Passing an `ArrayBuffer` from JavaScript to Python now correctly creates a `memoryview` object.
- Pyodide now works on Safari.

Version 0.11.0

Apr 12, 2019

User improvements:

- Support for built-in modules:
 - `sqlite`, `crypt`
- New packages: `mne`

Developer improvements:

- The `mkpkg` command will now select an appropriate archive to use, rather than just using the first.
- The included version of `emscripten` has been upgraded to 1.38.30 (plus a bugfix).
- New packages: `jinja2`, `MarkupSafe`

Version 0.10.0

Mar 21, 2019

User improvements:

- New packages: `html5lib`, `pygments`, `beautifulsoup4`, `soupsieve`, `docutils`, `bleach`, `mne`

Developer improvements:

- `console.html` provides a simple text-only interactive console to test local changes to Pyodide. The existing notebooks based on legacy versions of Iodide have been removed.
- The `run_docker` script can now be configured with environment variables.

Pyodide Deprecation Timeline

Each Pyodide release may deprecate certain features from previous releases in a backward incompatible way. If a feature is deprecated, it will continue to work until its removal, but raise warnings. We try to ensure deprecations are done over at least two minor(feature) releases, however, as Pyodide is still in beta state, this list is subject to change and some features can be removed without deprecation warnings. More details about each item can often be found in the *Change Log*.

0.25.0

- Typescript type imports for `PyProxy` subtypes from `pyodide` will be removed.
- The methods `PyProxy.supportsHas`, `PyProxy.isCallable`, etc will be removed.

0.24.0

- The `messageCallback` and `errorCallback` argument to `loadPackage` and `loadPackagesFromImports` will be passed as a named argument only.
- `Py2JsResult` will be removed.

0.23.0

- Names that used to be in the root `pyodide` module and were moved to submodules will no longer be available in the root module.
- The “message” argument to `PyProxy.destroy` method will no longer be accepted as a positional argument.

0.21.0

- The `globals` argument to `runPython` and `runPythonAsync` will be passed as a named argument only.
- The `extractDir` argument to `unpackArchive` will be passed as a named argument only.

0.20.0

- The `skip-host` key will be removed from the `meta.yaml` format. If needed, install a host copy of the package with `pip` instead.
- `pyodide-interrupts` module will be removed. If you were using this for some reason, use `setInterruptBuffer()` instead.

0.19.0

- The default working directory (home directory) inside the Pyodide virtual file system has been changed from / to /home/pyodide. To get the previous behavior, you can
 - call `os.chdir("/")` in Python to change working directory or
 - call `loadPyodide()` with the `homedir="/"` argument
- When a JavaScript function is called from Python, PyProxy arguments and return values will be automatically destroyed when the function is finished.

3.3.6 Related Projects

WebAssembly ecosystem

- [emscripten](#) is the compiler toolchain for WebAssembly that made Pyodide possible.

Notebook environments, IDEs, and REPLs

- [Iodide](#) is a notebook-like environment for literate scientific computing and communication for the web. It is no longer actively maintained. Historically, Pyodide started as plugin for iodide.
- [Starboard notebook](#) is an in-browser literal notebook runtime that uses Pyodide for Python.
- [Basthon notebook](#) is a static fork of Jupyter notebook with a Pyodide kernel (currently in French).
- [JupyterLite](#) is a JupyterLab distribution that runs entirely in the browser, based on Pyodide.
- [futurecoder](#) is an interactive Python course running on Pyodide. It includes an [IDE](#) with a REPL, debuggers, and automatic installation of any imported packages supported by Pyodide's `micropip`.

Workarounds for common WASM and browser limitations

- [pyodide-http](#) Provides patches for widely used http libraries to make them work in Pyodide environments like JupyterLite.

Dashboards and visualization

- [WebDash](#) is a Plotly Dash distribution that runs entirely in the browser, using Pyodide.
- [Flet](#) is a UI framework for your Pyodide apps based on Flutter.

Other projects

- [wc-code](#) is a library to run JavaScript, Python, and Theme in the browser with inline code blocks. It uses Pyodide to execute Python code.
- [SymPy Beta](#) is a fork of SymPy Gamma. It's an in-browser answer engine with a Pyodide backend.
- [react-py](#) is a library that allows for easy integration of Pyodide in React applications. It provides convenient hooks for running Python code.

COMMUNICATION

- Blog: blog.pyodide.org
- Mailing list: mail.python.org/mailman3/lists/pyodide.python.org/
- Gitter: gitter.im/pyodide/community
- Twitter: twitter.com/pyodide
- Stack Overflow: stackoverflow.com/questions/tagged/pyodide

PYTHON MODULE INDEX

p

- `pyodide.code`, [70](#)
- `pyodide.console`, [75](#)
- `pyodide.ffi`, [79](#)
- `pyodide.ffi.wrappers`, [95](#)
- `pyodide.http`, [97](#)
- `pyodide.webloop`, [100](#)

Symbols

- build-dependencies
 - pyodide-build command line option, 104
- cflags
 - pyodide-build-recipes command line option, 105
- compression-level
 - pyodide-build command line option, 104
 - pyodide-build-recipes command line option, 106
 - pyodide-py-compile command line option, 107
- continue
 - pyodide-build-recipes command line option, 106
- cxxflags
 - pyodide-build-recipes command line option, 105
- exports
 - pyodide-build command line option, 104
- force-rebuild
 - pyodide-build-recipes command line option, 106
- host-install-dir
 - pyodide-build-recipes command line option, 105
- install
 - pyodide-build-recipes command line option, 105
- install-dir
 - pyodide-build-recipes command line option, 105
- keep
 - pyodide-py-compile command line option, 107
- ldflags
 - pyodide-build-recipes command line option, 105
- libdir
 - pyodide-auditwheel-copy command line option, 102
 - pyodide-auditwheel-repair command line option, 103
- log-dir
 - pyodide-build-recipes command line option, 105
- n-jobs
 - pyodide-build-recipes command line option, 106
- no-build-dependencies
 - pyodide-build command line option, 104
- no-deps
 - pyodide-build-recipes command line option, 105
- no-force-rebuild
 - pyodide-build-recipes command line option, 106
- no-install
 - pyodide-build-recipes command line option, 105
- no-keep
 - pyodide-py-compile command line option, 107
- no-no-deps
 - pyodide-build-recipes command line option, 105
- no-show-type
 - pyodide-auditwheel-exports command line option, 102
 - pyodide-auditwheel-imports command line option, 103
- no-silent
 - pyodide-py-compile command line option, 107
- output-dir
 - pyodide-auditwheel-copy command line option, 102
 - pyodide-auditwheel-repair command line option, 103
- output-directory
 - pyodide-build command line option, 104
- output-lockfile
 - pyodide-build command line option, 104
- recipe-dir
 - option, 103

pyodide-build-recipes command line option, 105
pyodide-skeleton-pypi command line option, 108
--requirements
 pyodide-build command line option, 104
--show-type
 pyodide-auditwheel-exports command line option, 102
 pyodide-auditwheel-imports command line option, 103
--silent
 pyodide-py-compile command line option, 107
--skip-dependency
 pyodide-build command line option, 104
--source-format
 pyodide-skeleton-pypi command line option, 108
--target-install-dir
 pyodide-build-recipes command line option, 105
--update
 pyodide-skeleton-pypi command line option, 108
--update-patched
 pyodide-skeleton-pypi command line option, 108
--version
 pyodide command line option, 101
 pyodide-skeleton-pypi command line option, 108
-r
 pyodide-build command line option, 104
-u
 pyodide-skeleton-pypi command line option, 108

A

aclose() (*pyodide.ffi.JsAsyncGenerator method*), 82
add_event_listener() (*in module pyodide.ffi.wrappers*), 96
append() (*pyodide.ffi.JsArray method*), 80
as_object_map() (*pyodide.ffi.JsProxy method*), 90
asend() (*pyodide.ffi.JsAsyncGenerator method*), 82
assign() (*pyodide.ffi.JsBuffer method*), 83
assign_to() (*pyodide.ffi.JsBuffer method*), 83
ast (*pyodide.code.CodeRunner attribute*), 70
athrow() (*pyodide.ffi.JsAsyncGenerator method*), 83

B

body_used (*pyodide.http.FetchResponse property*), 97
buffer (*pyodide.console.Console attribute*), 75
buffer() (*pyodide.http.FetchResponse method*), 97

bytes() (*pyodide.http.FetchResponse method*), 97

C

canvas (*None attribute*), 51
catch() (*pyodide.ffi.JsPromise method*), 89
catch() (*pyodide.webloop.PyodideFuture method*), 100
checkInterrupt() (*built-in static function*), 52
clear() (*pyodide.ffi.JsMutableMap method*), 88
clear_interval() (*in module pyodide.ffi.wrappers*), 96
clear_timeout() (*in module pyodide.ffi.wrappers*), 96
clone() (*pyodide.http.FetchResponse method*), 98
close() (*pyodide.ffi.JsGenerator method*), 86
code (*pyodide.code.CodeRunner attribute*), 70
CodeRunner (*class in pyodide.code*), 70
compile() (*pyodide.code.CodeRunner method*), 71
complete() (*pyodide.console.Console method*), 75
completer_word_break_characters (*pyodide.console.Console attribute*), 76
CONFIG_VAR
 pyodide-config-get command line option, 106

Console (*class in pyodide.console*), 75
ConsoleFuture (*class in pyodide.console*), 77
ConversionError, 80
count() (*pyodide.ffi.JsArray method*), 80
create_once_callable() (*in module pyodide.ffi*), 92
create_proxy() (*in module pyodide.ffi*), 92

D

DEST
 pyodide-venv command line option, 109
destroy() (*pyodide.ffi.JsDoubleProxy method*), 85
destroy_proxies() (*in module pyodide.ffi*), 93

E

ERRNO_CODES (*None attribute*), 51
eval_code() (*in module pyodide.code*), 71
eval_code_async() (*in module pyodide.code*), 73
extend() (*pyodide.ffi.JsArray method*), 80

F

FetchResponse (*class in pyodide.http*), 97
ffi (*None attribute*), 51
finally_() (*pyodide.ffi.JsPromise method*), 89
finally_() (*pyodide.webloop.PyodideFuture method*), 100
find_imports() (*in module pyodide.code*), 74
formatsyntaxerror() (*pyodide.console.Console method*), 76
formatted_error (*pyodide.console.ConsoleFuture attribute*), 78
formattraceback() (*pyodide.console.Console method*), 76

`from_file()` (*pyodide.ffi.JsBuffer* method), 84
`FS` (*None* attribute), 51

G

`get()` (*pyodide.ffi.JsMap* method), 87
`getCanvas2D()` (*built-in* function), 68
`getCanvas3D()` (*built-in* function), 68
`globals` (*None* attribute), 51
`globals` (*pyodide.console.Console* attribute), 76
`globalThis` (*module*), 49

I

`index()` (*pyodide.ffi.JsArray* method), 80
`isPyProxy()` (*built-in* static function), 52
`items()` (*pyodide.ffi.JsMap* method), 87

J

`js_id` (*pyodide.ffi.JsProxy* property), 90
`JsArray` (*class* in *pyodide.ffi*), 80
`JsAsyncGenerator` (*class* in *pyodide.ffi*), 82
`JsAsyncIterable` (*class* in *pyodide.ffi*), 83
`JsAsyncIterator` (*class* in *pyodide.ffi*), 83
`JsBuffer` (*class* in *pyodide.ffi*), 83
`JsCallable` (*class* in *pyodide.ffi*), 85
`JsDomElement` (*class* in *pyodide.ffi*), 85
`JsDoubleProxy` (*class* in *pyodide.ffi*), 85
`JsException`, 86
`JsFetchResponse` (*class* in *pyodide.ffi*), 86
`JsGenerator` (*class* in *pyodide.ffi*), 86
`JsIterable` (*class* in *pyodide.ffi*), 87
`JsIterator` (*class* in *pyodide.ffi*), 87
`JsMap` (*class* in *pyodide.ffi*), 87
`JsMutableMap` (*class* in *pyodide.ffi*), 88
`json()` (*pyodide.http.FetchResponse* method), 98
`JsPromise` (*class* in *pyodide.ffi*), 89
`JsProxy` (*class* in *pyodide.ffi*), 90
`JsTypedArray` (*class* in *pyodide.ffi*), 92

K

`keys()` (*pyodide.ffi.JsMap* method), 87

L

`loadedPackages` (*None* attribute), 51
`loadPackage()` (*built-in* function), 52
`loadPyodide()` (*built-in* function), 49

M

`memoryview()` (*pyodide.http.FetchResponse* method), 98
`message` (*pyodide.ffi.JsException* attribute), 86
`module`
 pyodide.code, 70
 pyodide.console, 75
 pyodide.ffi, 79

pyodide.ffi.wrappers, 95
pyodide.http, 97
pyodide.webloop, 100

N

NAME

pyodide-skeleton-pypi command line
 option, 108

`name` (*pyodide.ffi.JsException* attribute), 86
`new()` (*pyodide.ffi.JsProxy* method), 90

O

`object_entries()` (*pyodide.ffi.JsProxy* method), 91
`object_keys()` (*pyodide.ffi.JsProxy* method), 91
`object_values()` (*pyodide.ffi.JsProxy* method), 91
`ok` (*pyodide.http.FetchResponse* property), 98
`open_url()` (*in* module *pyodide.http*), 99

P

PACKAGES

pyodide-build-recipes command line
 option, 106

PATH

pyodide-py-compile command line option,
 107

`PATH` (*None* attribute), 51

`persistent_redirect_streams()` (*pyo-*
dide.console.Console method), 76

`persistent_restore_streams()` (*pyo-*
dide.console.Console method), 76

`pop()` (*pyodide.ffi.JsArray* method), 81

`pop()` (*pyodide.ffi.JsMutableMap* method), 88

`popitem()` (*pyodide.ffi.JsMutableMap* method), 88

`push()` (*pyodide.console.Console* method), 77

`PyAsyncGenerator()` (*class*), 57

`PyAsyncGenerator.return()` (*PyAsyncGenerator*
method), 57

`PyAsyncGenerator.throw()` (*PyAsyncGenerator*
method), 57

`PyAsyncIterable()` (*class*), 58

`PyAsyncIterable.[Symbol.asyncIterator]()`
 (*PyAsyncIterable* method), 58

`PyAsyncIterator()` (*class*), 58

`PyAsyncIterator.next()` (*PyAsyncIterator* method),
 58

`PyAwaitable()` (*class*), 58

`PyBuffer` (*None* attribute), 56

`PyBuffer()` (*class*), 58

`PyBuffer.getBuffer()` (*PyBuffer* method), 58

`PyBufferView()` (*class*), 59

`PyBufferView.c_contiguous` (*PyBufferView* at-
 tribute), 59

`PyBufferView.data` (*PyBufferView* attribute), 60

`PyBufferView.f_contiguous` (*PyBufferView attribute*), 60
`PyBufferView.format` (*PyBufferView attribute*), 60
`PyBufferView.itemsize` (*PyBufferView attribute*), 60
`PyBufferView.nbytes` (*PyBufferView attribute*), 60
`PyBufferView.ndim` (*PyBufferView attribute*), 60
`PyBufferView.offset` (*PyBufferView attribute*), 60
`PyBufferView.readonly` (*PyBufferView attribute*), 60
`PyBufferView.release()` (*PyBufferView method*), 61
`PyBufferView.shape` (*PyBufferView attribute*), 61
`PyBufferView.strides` (*PyBufferView attribute*), 61
`PyCallable()` (*class*), 61
`PyCallable.apply()` (*PyCallable method*), 61
`PyCallable.bind()` (*PyCallable method*), 61
`PyCallable.call()` (*PyCallable method*), 61
`PyCallable.callKwargs()` (*PyCallable method*), 62
`PyCallable.captureThis()` (*PyCallable method*), 62
`PyDict()` (*class*), 62
`pyfetch()` (*in module pyodide.http*), 99
`PyGenerator()` (*class*), 63
`PyGenerator.return()` (*PyGenerator method*), 63
`PyGenerator.throw()` (*PyGenerator method*), 63
`pyimport()` (*built-in static function*), 53
`PyIterable()` (*class*), 63
`PyIterable.[Symboliterator]()` (*PyIterable method*), 63
`PyIterator()` (*class*), 63
`PyIterator.next()` (*PyIterator method*), 64
`pyodide` (*module*), 51
`pyodide` command line option
 --version, 101
`pyodide.canvas` (*module*), 68
`pyodide.code`
 module, 70
`pyodide.console`
 module, 75
`pyodide.ffi`
 module, 79
`pyodide.ffi` (*module*), 57
`pyodide.ffi.wrappers`
 module, 95
`pyodide.http`
 module, 97
`pyodide.webloop`
 module, 100
`pyodide_py` (*None attribute*), 51
`pyodide-auditwheel-copy` command line option
 --libdir, 102
 --output-dir, 102
 WHEEL_FILE, 102
`pyodide-auditwheel-exports` command line option
 --no-show-type, 102
 --show-type, 102
 WHEEL_OR_SO_FILE, 102
`pyodide-auditwheel-imports` command line option
 --no-show-type, 103
 --show-type, 103
 WHEEL_OR_SO_FILE, 103
`pyodide-auditwheel-repair` command line option
 --libdir, 103
 --output-dir, 103
 WHEEL_FILE, 103
`pyodide-auditwheel-show` command line option
 WHEEL_OR_SO_FILE, 104
`pyodide-build` command line option
 --build-dependencies, 104
 --compression-level, 104
 --exports, 104
 --no-build-dependencies, 104
 --output-directory, 104
 --output-lockfile, 104
 --requirements, 104
 --skip-dependency, 104
 -r, 104
 SOURCE_LOCATION, 105
`pyodide-build-recipes` command line option
 --cflags, 105
 --compression-level, 106
 --continue, 106
 --cxxflags, 105
 --force-rebuild, 106
 --host-install-dir, 105
 --install, 105
 --install-dir, 105
 --ldflags, 105
 --log-dir, 105
 --n-jobs, 106
 --no-deps, 105
 --no-force-rebuild, 106
 --no-install, 105
 --no-no-deps, 105
 --recipe-dir, 105
 --target-install-dir, 105
 PACKAGES, 106
`pyodide-config-get` command line option
 CONFIG_VAR, 106
`pyodide-py-compile` command line option
 --compression-level, 107
 --keep, 107
 --no-keep, 107
 --no-silent, 107
 --silent, 107
 PATH, 107
`pyodide-skeleton-pypi` command line option
 --recipe-dir, 108

--source-format, 108
 --update, 108
 --update-patched, 108
 --version, 108
 -u, 108
 NAME, 108
 pyodide-venv command line option
 DEST, 109
 PyodideConsole (class in *pyodide.console*), 78
 PyodideFuture (class in *pyodide.webloop*), 100
 PyProxy() (class), 64
 PyProxy.copy() (PyProxy method), 64
 PyProxy.destroy() (PyProxy method), 64
 PyProxy.isAwaitable() (PyProxy method), 65
 PyProxy.isBuffer() (PyProxy method), 65
 PyProxy.isCallable() (PyProxy method), 65
 PyProxy.isIterable() (PyProxy method), 65
 PyProxy.isIterator() (PyProxy method), 65
 PyProxy.supportsGet() (PyProxy method), 65
 PyProxy.supportsHas() (PyProxy method), 66
 PyProxy.supportsLength() (PyProxy method), 66
 PyProxy.supportsSet() (PyProxy method), 66
 PyProxy.toJs() (PyProxy method), 66
 PyProxy.toString() (PyProxy method), 67
 PyProxy.type (PyProxy attribute), 64
 PyProxyBuffer (None attribute), 56
 PyProxyWithGet() (class), 67
 PyProxyWithGet.get() (PyProxyWithGet method), 67
 PyProxyWithHas() (class), 67
 PyProxyWithHas.has() (PyProxyWithHas method), 67
 PyProxyWithLength() (class), 67
 PyProxyWithLength.length (PyProxyWithLength attribute), 67
 PyProxyWithSet() (class), 67
 PyProxyWithSet.delete() (PyProxyWithSet method), 67
 PyProxyWithSet.set() (PyProxyWithSet method), 68
 Python Enhancement Proposals
 PEP 440, 52
 PEP 517, 124
 PythonError (None attribute), 56
 PythonError() (class), 68
 PythonError.type (PythonError attribute), 68

R

redirect_streams() (pyodide.console.Console method), 77
 redirected (pyodide.http.FetchResponse property), 98
 register_js_module() (in module *pyodide.ffi*), 93
 registerComlink() (built-in static function), 53
 registerJsModule() (built-in static function), 53
 remove_event_listener() (in module *pyodide.ffi.wrappers*), 96
 repr_shorten() (in module *pyodide.console*), 78

reverse() (pyodide.ffi.JsArray method), 81
 run() (pyodide.code.CodeRunner method), 71
 run_async() (pyodide.code.CodeRunner method), 71
 run_js() (in module *pyodide.code*), 74
 runcode() (pyodide.console.Console method), 77
 runPython() (built-in static function), 53
 runsource() (pyodide.console.Console method), 77

S

send() (pyodide.ffi.JsGenerator method), 86
 set_interval() (in module *pyodide.ffi.wrappers*), 96
 set_timeout() (in module *pyodide.ffi.wrappers*), 97
 setCanvas2D() (built-in function), 69
 setCanvas3D() (built-in function), 69
 setdefault() (pyodide.ffi.JsMutableMap method), 88
 setInterruptBuffer() (built-in static function), 54
 setStderr() (built-in function), 54
 setStdin() (built-in function), 54
 setStdout() (built-in function), 55
 should_quiet() (in module *pyodide.code*), 74
 SOURCE_LOCATION
 pyodide-build command line option, 105
 stack (pyodide.ffi.JsException attribute), 86
 status (pyodide.http.FetchResponse property), 98
 status_text (pyodide.http.FetchResponse property), 98
 stderr_callback (pyodide.console.Console attribute), 77
 stdin_callback (pyodide.console.Console attribute), 77
 stdout_callback (pyodide.console.Console attribute), 77
 string() (pyodide.http.FetchResponse method), 98
 syntax_check (pyodide.console.ConsoleFuture attribute), 78

T

then() (pyodide.ffi.JsPromise method), 89
 then() (pyodide.webloop.PyodideFuture method), 100
 throw() (pyodide.ffi.JsGenerator method), 86
 to_bytes() (pyodide.ffi.JsBuffer method), 84
 to_file() (pyodide.ffi.JsBuffer method), 84
 to_js() (in module *pyodide.ffi*), 93
 to_memoryview() (pyodide.ffi.JsBuffer method), 85
 to_py() (pyodide.ffi.JsArray method), 81
 to_py() (pyodide.ffi.JsProxy method), 91
 to_string() (pyodide.ffi.JsBuffer method), 85
 toPy() (built-in static function), 55
 type (pyodide.http.FetchResponse property), 98
 typeof (pyodide.ffi.JsProxy property), 92

U

unpack_archive() (pyodide.http.FetchResponse method), 98
 unpackArchive() (built-in static function), 55

`unregister_js_module()` (*in module `pyodide.ffi`*), 95
`unregisterJsModule()` (*built-in static function*), 55
`unwrap()` (*`pyodide.ffi.JsDoubleProxy` method*), 85
`update()` (*`pyodide.ffi.JsMutableMap` method*), 88
`url` (*`pyodide.http.FetchResponse` property*), 99

V

`values()` (*`pyodide.ffi.JsMap` method*), 88
`version` (*`None` attribute*), 51

W

`WebLoop` (*class in `pyodide.webloop`*), 100
`WebLoopPolicy` (*class in `pyodide.webloop`*), 101
`WHEEL_FILE`
 `pyodide-auditwheel-copy` command line
 option, 102
 `pyodide-auditwheel-repair` command line
 option, 103
`WHEEL_OR_SO_FILE`
 `pyodide-auditwheel-exports` command line
 option, 102
 `pyodide-auditwheel-imports` command line
 option, 103
 `pyodide-auditwheel-show` command line
 option, 104