Pyodide

Release 0.17.0a2

Mozilla

Apr 20, 2021

CONTENTS

Python with the scientific stack, compiled to WebAssembly.

Note: Pyodide bundles support for the following packages: numpy, scipy, and many other libraries in the Python scientific stack.

To use additional packages from PyPI, see *Micropip* .

You can also *create a Pyodide package* to support and share libraries for new applications.

CHAPTER

ONE

USING PYODIDE

Pyodide may be used in several ways: directly from JavaScript, or to execute Python scripts asynchronously in a web worker. Additional pure Python packages may be installed from PyPI to be used with Pyodide.

1.1 Using Pyodide from Javascript

This document describes using Pyodide directly from Javascript.

1.1.1 Startup

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

https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/pyodide.js

You can also download a release from Github releases (or build it yourself), include its contents in your distribution, and import the `pyodide.js` file there from a `<script>` tag. See the following section on *Serving Pyodide packages* for more details.

The `pyodide.js` file has a single `Promise` object which bootstraps the Python environment: `languagePluginLoader`. Since this must happen asynchronously, it is a `Promise`, which you must call `then` on to complete initialization. When the promise resolves, Pyodide will have installed a namespace in the global scope called *pyodide*.

```
languagePluginLoader.then(() => {
  // Pyodide is now ready to use...
  console.log(pyodide.runPython(`import sys\nsys.version`));
});
```

1.1.2 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*).

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

After importing Pyodide, only packages from the standard library are available. See *Loading packages* documentation to load additional packages.

1.1.3 Complete example

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

```
<!DOCTYPE html>
<html>
  <head>
      <script type="text/javascript">
          // set the Pyodide files URL (packages.json, pyodide.asm.data etc)
          window.languagePluginUrl = 'https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/
↪';
      </script>
      <script src="https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/pyodide.js"></
↪script>
  </head>
  <body>
    Pyodide test page <br>
    Open your browser console to see Pyodide output
    <script type="text/javascript">
          languagePluginLoader.then(function () {
              console.log(pyodide.runPython(`
                  import sys
                  sys.version
              `));
              console.log(pyodide.runPython(`print(1 + 2)`));
          });
    </script>
  </body>
</html>
```

1.1.4 Alternative Example

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript">
        window.languagePluginUrl = 'https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/';
    </script>
    <script src="https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/pyodide.js"></script>
</head>

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

(continues on next page)

(continued from previous page)

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

    output.value = 'Initializing...\n';
    // init Pyodide
    languagePluginLoader.then(() => { output.value += 'Ready!\n'; });

    function evaluatePython() {
      pyodide.runPythonAsync(code.value)
        .then(output => addToOutput(output))
        .catch((err) => { addToOutput(err) });
    }
  </script>
</body>

</html>
```

1.1.5 Accessing Python scope from Javascript

You can also access from Javascript all functions and variables defined in Python using the *pyodide.pyimport* api or the *pyodide.globals* object.

For example, if you run the code `x = numpy.ones([3,3])` in Python, you can access the variable `x` from Javascript in your browser's developer console as follows: `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:

```
pyodide.runPython(`x=numpy.ones([3, 3])`);
pyodide.globals.get("x");
// >>> [Float64Array(3), Float64Array(3), Float64Array(3)]

// create the same 3x3 ndarray from js
let x = pyodide.globals.get("numpy").ones(new Int32Array([3, 3]));
// x >>> [Float64Array(3), Float64Array(3), Float64Array(3)]
```

Since you have full access to Python global scope, you can also re-assign new values or even Javascript functions to variables, and create new ones from Javascript:

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

// create a new js function that will be available from Python
// this will show a browser alert if the function is called from Python
pyodide.globals.set("alert", alert);

// this new function will also be available in Python and will return the squared␣
↪value.
pyodide.globals.set("square", x => x*x);
```

Feel free to play around with the code using the browser console and the above example.

1.1.6 Accessing Javascript scope from Python

The Javascript scope can be accessed from Python using the `js` module (see *Importing Javascript objects into Python*). This module represents the global object `window` that allows us to directly manipulate the DOM and access global variables and functions from Python.

```
import js

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

See *Serving Pyodide packages* to distribute Pyodide files locally.

1.2 Using Pyodide from a web worker

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

1.2.1 Startup

Setup your project to serve `webworker.js`. You should also serve `pyodide.js`, and all its associated `.asm.js`, `.data`, `.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 `self.languagePluginUrl` 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).

1.2.2 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 **consumer**s 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 wan 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, lets 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 lines `pythonLoading = self.pyodide.loadPackage(['numpy', 'pytz'])` and `await pythonLoading;` 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`, `.data`, `.json`,
// and `.wasm` files as well:
self.languagePluginUrl = 'https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/';
importScripts('https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/pyodide.js');

let pythonLoading;
async function loadPythonPackages(){
    await languagePluginLoader;
    pythonLoading = self.pyodide.loadPackage(['numpy', 'pytz']);
}

self.onmessage = async(event) => {
    await languagePluginLoader;
     // since loading package is asynchronous, we need to make sure loading is done:
    await pythonLoading;
    // Don't bother yet with this line, suppose our API is built in such a way:
    const {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 {
        self.postMessage({
            results: await self.pyodide.runPythonAsync(python)
        });
    }
    catch (error){
        self.postMessage(
            {error : error.message}
        );
    }
}
```

The worker API

Now that we established what the two sides need and how they operate, lets 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('./build/webworker.js')

export function run(script, context, onSuccess, onError){
    pyodideWorker.onerror = onError;
    pyodideWorker.onmessage = (e) => onSuccess(e.data);
    pyodideWorker.postMessage({
        ...context,
```

(continues on next page)

(continued from previous page)

```
        python: script,
    });
}

// Transform the run (callback) form to a more modern async form.
// This is what allows to write:
//    const {results, error} = await asyncRun(script, context);
// Instead of:
//    run(script, context, successCallback, errorCallback);
export function asyncRun(script, context) {
    return new Promise(function(onSuccess, onError) {
        run(script, context, onSuccess, onError);
    });
}
```

1.2.3 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.

1.3 Serving Pyodide packages

If you built your Pyodide distribution or downloaded the release tarball you need to serve Pyodide files with a appropriate headers.

Because browsers require WebAssembly files to have mimetype of `application/wasm` we're unable to serve our files using Python's built-in `SimpleHTTPServer` module.

Let's wrap Python's Simple HTTP Server and provide the appropiate mimetype for WebAssembly files into a `pyodide_server.py` file (in the `pyodide_local` directory):

```
import sys
import socketserver
from http.server import SimpleHTTPRequestHandler


class Handler(SimpleHTTPRequestHandler):

    def end_headers(self):
        # Enable Cross-Origin Resource Sharing (CORS)
        self.send_header('Access-Control-Allow-Origin', '*')
        super().end_headers()


if sys.version_info < (3, 7, 5):
    # Fix for WASM MIME type for older Python versions
    Handler.extensions_map['.wasm'] = 'application/wasm'
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    port = 8000
    with socketserver.TCPServer(("", port), Handler) as httpd:
        print("Serving at: http://127.0.0.1:{}".format(port))
        httpd.serve_forever()
```

Let's test it out. In your favourite shell, let's start our WebAssembly aware web server:

```
python pyodide_server.py
```

Point your WebAssembly aware browser to http://localhost:8000/index.html and open your browser console to see the output from Python via Pyodide!

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:

- *pyodide.loadPackage* for packages built with Pyodide, or
- *micropip.install* for pure Python packages with wheels available on PyPi or from other URLs.

Note: `micropip` can also be used to load packages built in Pyodide (in which case it relies on *pyodide.loadPackage*).

Alternatively you can run Python code without manually pre-loading packages. You can do this with *pyodide.runPythonAsync* which will automatically download all packages that the code snippet imports. It only supports packages included in Pyodide (not on PyPi) at present.

1.4.1 Loading packages with `pyodide.loadPackage`

Packages can be loaded by name, for those included in the official Pyodide repository using e.g.,

```
pyodide.loadPackage('numpy')
```

It is also possible to load packages from custom URLs,

```
pyodide.loadPackage('https://foo/bar/numpy.js')
```

in which case the URL must end with `<package-name>.js`.

When you request a package from the official repository, all of that package's dependencies are also loaded. Dependency resolution is not yet implemented when loading packages from custom URLs.

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.

Multiple packages can also be loaded in a single call,

```
pyodide.loadPackage(['cycler', 'pytz'])
```

pyodide.loadPackage returns a `Promise`.

```
pyodide.loadPackage('matplotlib').then(() => {
  // matplotlib is now available
});
```

1.4.2 Micropip

Installing packages from PyPI

Pyodide supports installing pure Python wheels from PyPI with `micropip`. You can use the `then` method on the `Promise` that *`micropip.install()`* returns to do work once the packages have finished loading:

```
def do_work(*args):
    import snowballstemmer
    stemmer = snowballstemmer.stemmer('english')
    print(stemmer.stemWords('go goes going gone'.split()))

import micropip
micropip.install('snowballstemmer').then(do_work)
```

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`).

All required dependencies need also to be previously installed with `micropip` or *`pyodide.loadPackage`*.

If the file is on a remote server, it must set Cross-Origin Resource Sharing (CORS) headers to allow access. Otherwise, you can prepend a CORS proxy to the URL. Note however 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.

1.4.3 Example

Adapting the setup from the section on *Using Pyodide from Javascript* a complete example would be,

```
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <script type="text/javascript">
      // set the Pyodide files URL (packages.json, pyodide.asm.data etc)
      window.languagePluginUrl = 'https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/';
```

(continues on next page)

(continued from previous page)

```
  </script>
  <script type="text/javascript" src="https://cdn.jsdelivr.net/pyodide/v0.17.0a2/full/
↪pyodide.js"></script>
  <script type="text/javascript">
    pythonCode = `
      def do_work(*args):
          import snowballstemmer
          stemmer = snowballstemmer.stemmer('english')
          print(stemmer.stemWords('go goes going gone'.split()))

      import micropip
      micropip.install('snowballstemmer').then(do_work)
    `

    languagePluginLoader.then(() => {
      return pyodide.loadPackage(['micropip'])
    }).then(() => {
      pyodide.runPython(pythonCode);
    })
  </script>
</body>
</html>
```

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 using *pyodide.pyimport*,
- 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 using the `from js import ...` syntax
- passing arguments to a Python function called from Javascript
- 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 result and destroy it when you are done with it. Unfortunately, we currently provide no convenient way to do this, particularly when calling Javascript functions from Python.

1.5.1 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 (with the exception of `nan` because `nan != nan`). 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 (with the exception of `NaN` because `NaN !== NaN`, and of `null` which after a round trip is converted to `undefined`). 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).

1.5.2 Implicit conversions

We only implicitly convert immutable types. This is to ensure that a mutable type in Python can be modified in 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. Proxying tuples also has the benefit of ensuring that implicit conversions take a constant amount of time.

Python to Javascript

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

Python	Javascript
`int`	`Number`
`float`	`Number`
`str`	`String`
`bool`	`Boolean`
`None`	`undefined`

Javascript to Python

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

Javascript	Python
`Number`	`int` or `float` as appropriate
`String`	`str`
`Boolean`	`bool`
`undefined`	`None`
`null`	`None`

1.5.3 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`. (More should be possible in the future – work is ongoing to make this more complete):

Python	Javascript
`str(proxy)`	`x.toString()`
`proxy.foo`	`x.foo`
`proxy.foo = bar`	`x.foo = bar`
`del proxy.foo`	`delete x.foo`
`hasattr(proxy, "foo")`	`"foo" in x`
`proxy(...)`	`x(...)`
`proxy.foo(...)`	`x.foo(...)`
`proxy.new(...)`	`new X(...)`
`len(proxy)`	`x.length` or `x.size`
`foo in proxy`	`x.has(foo)`
`proxy[foo]`	`x.get(foo)`
`proxy[foo] = bar`	`x.set(foo, bar)`
`del proxy[foo]`	`x.delete(foo)`
`proxy1 == proxy2`	`x === y`
`proxy.typeof`	`typeof x`
`iter(proxy)`	`x[Symbol.iterator]()`
`next(proxy)`	`x.next()`
`await proxy`	`await x`
`proxy.object_entries()`	`Object.entries(x)`

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
`proxy[idx]`	`x.toString()`
`proxy[idx] = val`	`x.foo`
`idx in proxy`	`idx in array`
`del proxy[idx]`	`proxy.splice(idx)`

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 currently supported:

Javascript	Python
`foo in proxy`	`hasattr(x, 'foo')`
`proxy.foo`	`x.foo`
`proxy.foo = bar`	`x.foo = bar`
`delete proxy.foo`	`del x.foo`
`Object.getOwnPropertyNames(proxy)`	`dir(x)`
`proxy(...)`	`x(...)`
`proxy.foo(...)`	`x.foo(...)`
`proxy.length` or `x.size`	`len(x)`
`proxy.has(foo)`	`foo in x`
`proxy.get(foo)`	`x[foo]`
`proxy.set(foo, bar)`	`x[foo] = bar`
`proxy.delete(foo)`	`del x[foo]`
`x.type`	`type(x)`
`x[Symbol.iterator]()`	`iter(x)`
`x.next()`	`next(x)`
`await x`	`await x`
`Object.entries(x)`	`repr(x)`

Memory Leaks and PyProxy

When proxying a Python object into Javascript, there is no way for Javascript to automatically garbage collect the Proxy. The `PyProxy` must be manually destroyed when passed to Javascript, or the proxied Python object will leak. To do this, call `PyProxy.destroy()` on the `PyProxy`, after which Javascript will no longer have access to the Python object. If no references to the Python object exist in Python either, then the Python garbage collector can eventually collect it.

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

Memory Leaks and PyProxy method calls

Every time you access a Python method on a `PyProxy`, it creates a new temporary `PyProxy` of a Python bound method. If you do not capture this temporary and destroy it, you will leak the Python object.

Here's an example:

```
pyodide.runPython(`
    class Test(dict):
        def __del__(self):
            print("destructed!")
    d = Test(a=2, b=3)
    import sys
    print(sys.getrefcount(d)) # prints 2
`);
let d = pyodide.pyimport("d");
// Leak three temporary bound "get" methods!
let l = [d.get("a", 0), d.get("b", 0), d.get("c", 0)];
d.destroy(); // Try to free dict
// l is [2, 3, 0].
pyodide.runPython(`
    print(sys.getrefcount(d)) # prints 5 = original 2 + leaked 3
    del d # Destructor isn't run because of leaks
`);
```

Here is how we can do this without leaking:

```
let d = pyodide.pyimport("d");
let d_get = d.get; // this time avoid the leak
let l = [d_get("a", 0), d_get("b", 0), d_get("c", 0)];
d.destroy();
d_get.destroy();
// l is [2, 3, 0].
pyodide.runPython(`
    print(sys.getrefcount(d)) # prints 2
    del d # runs destructor and prints "destructed!".
`);
```

Another exciting inconsistency is that `d.set` is a **Javascript** method not a PyProxy of a bound method, so using it has no effect on refcounts or memory reclamation and it cannot be destroyed.

```
let d = pyodide.pyimport("d");
let d_set = d.set;
d_set("x", 7);
pyodide.runPython(`
    print(sys.getrefcount(d)) # prints 2, d_set doesn't hold an extra reference to d
`);
d_set.destroy(); // TypeError: d_set.destroy is not a function
```

1.5.4 Explicit Conversion of Proxies

Python to Javascript

Explicit conversion of a `PyProxy` into a native Javascript object is done with the `toJs` method. By default, the `toJs` method does a recursive "deep" conversion, to do a shallow conversion use `proxy.toJs(1)`. The `toJs` method performs the following explicit conversions:

Python	Javascript
`list, tuple`	`Array`
`dict`	`Map`
`set`	`Set`

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.

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, but we provide no way to manage this.

To ensure that no `PyProxy` is leaked, the following code suffices:

```
function destroyToJsResult(x){
    if(!x){
        return;
    }
    if(x.destroy){
        x.destroy();
        return;
    }
    if(x[Symbol.iterator]){
        for(let k of x){
            freeToJsResult(k);
        }
    }
}
```

Javascript to Python

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

Javascript	Python
`Array`	`list`
`Object**`	`dict`
`Map`	`dict`
`Set`	`set`

** `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.

1.5.5 Buffers

Converting Javascript Typed Arrays to Python

Javascript typed arrays (`Int8Array` and friends) are translated to Python `memoryviews`. This happens with a single binary memory copy (since Python can't directly access arrays if they are outside of the wasm heap), and the data type is preserved. This makes it easy to correctly convert the array to a Numpy array using `numpy.asarray`:

```
let array = new Float32Array([1, 2, 3]);
```

```
from js import array
import numpy as np
numpy_array = np.asarray(array)
```

Converting Python Buffer objects to Javascript

Python `bytes` and `buffer` objects are translated to Javascript as `TypedArrays` without any memory copy at all. This conversion is thus very efficient, but be aware that any changes to the buffer will be reflected in both places.

Numpy arrays are currently converted to Javascript as nested (regular) Arrays. A more efficient method will probably emerge as we decide on an ndarray implementation for Javascript.

1.5.6 Importing Python objects into Javascript

A Python object in the `__main__` global scope can imported into Javascript using the *pyodide.pyimport* function. Given the name of the Python object to import, `pyimport` returns the object translated to Javascript.

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

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.

1.5.7 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. You can create your own custom Javascript modules using *pyodide.registerJsModule*.

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

1.6 API Reference

1.6.1 Python API

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

pyodide.eval_code(code[, globals, locals, ...])	Runs a code string.
pyodide.find_imports(code)	Finds the imports in a string of code
pyodide.open_url(url)	Fetches a given URL
pyodide.JsException	A wrapper around a Javascript Error to allow the Error to be thrown in Python.
pyodide.register_js_module(name, jsproxy)	Registers `jsproxy` as a Javascript module named `name`.
pyodide.unregister_js_module(name)	Unregisters a Javascript module with given name that has been previously registered with *pyodide.registerJsModule* or *pyodide.register_js_module*.
pyodide.console.InteractiveConsole(locals, ...)	Interactive Pyodide console
pyodide.console.repr_shorten(value[, limit, ...])	Compute the string representation of *value* and shorten it if necessary.
pyodide.console.displayhook(value, repr)	A displayhook with custom *repr* function.
pyodide.webloop.WebLoop()	A custom event loop for use in Pyodide.
pyodide.create_proxy(obj)	Create a *JsProxy* of a *PyProxy*.
pyodide.create_once_callable(obj)	Wrap a Python callable in a Javascript function that can be called once.

pyodide.eval_code

pyodide.**eval_code**(*code: str*, *globals: Optional[Dict[str, Any]] = None*, *locals: Optional[Dict[str, Any]] = None*, *return_mode: str = 'last_expr'*, *quiet_trailing_semicolon: bool = True*, *filename: str = '<exec>'*) → Any

Runs a code string.

Parameters

- **code** (str) – The Python code to run.
- **globals** (dict) – The global scope in which to execute code. This is used as the globals parameter for exec. See the exec documentation for more info. If the globals is absent, it is set equal to a new empty dictionary.
- **locals** (dict) – The local scope in which to execute code. This is used as the locals parameter for exec. As with exec, if locals is absent, it is set equal to globals. See the exec documentation for more info.
- **return_mode** (str) – Specifies what should be returned, must be one of 'last_expr', 'last_expr_or_assign' or 'none'. On other values an exception is raised.
  - '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) – Whether a trailing semicolon should 'quiet' the result or not. Setting this to True (default) mimic the CPython's interpreter behavior ; whereas setting it to False mimic the IPython's interpreter behavior.
- **filename** (str) – The file name to use in error messages and stack traces

Returns If the last nonwhitespace character of code 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.)

Return type Any

pyodide.find_imports

pyodide.**find_imports**(*code: str*) → List[str]

Finds the imports in a string of code

Parameters **code** (*str*) – the Python code to run.

Returns A list of module names that are imported in the code.

Return type List[str]

Examples

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

pyodide.open_url

pyodide.**open_url**(*url: str*) → _io.StringIO

Fetches a given URL

Parameters **url** (*str*) – URL to fetch

Returns the contents of the URL.

Return type io.StringIO

pyodide.JsException

exception pyodide.**JsException**

A wrapper around a Javascript Error to allow the Error to be thrown in Python.

pyodide.register_js_module

pyodide.**register_js_module**(*name: str*, *jsproxy: pyodide._core.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** (*JsProxy*) – Javascript object backing the module

pyodide.unregister_js_module

pyodide.**unregister_js_module**(*name: str*)

Unregisters a Javascript module with given name that has been previously registered with *pyodide.registerJsModule* or *pyodide.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 js module

pyodide.console.InteractiveConsole

class pyodide.console.**InteractiveConsole**(*locals: Optional[dict] = None*, *stdout_callback: Optional[Callable[[str], None]] = None*, *stderr_callback: Optional[Callable[[str], None]] = None*, *persistent_stream_redirection: bool = False*)

Interactive Pyodide console

Base implementation for an interactive console that manages stdout/stderr redirection. Since packages are loaded before running code, *runcode* returns a JS promise. Override *sys.displayhook* to catch the result of an execution.

self.stdout_callback and *self.stderr_callback* can be overloaded.

Parameters

- **locals** – Namespace to evaluate code.
- **stdout_callback** – Function to call at each *sys.stdout* flush.
- **stderr_callback** – Function to call at each *sys.stderr* flush.
- **persistent_stream_redirection** – Wether or not the std redirection should be kept between calls to *runcode*.

__init__(*locals: Optional[dict] = None*, *stdout_callback: Optional[Callable[[str], None]] = None*, *stderr_callback: Optional[Callable[[str], None]] = None*, *persistent_stream_redirection: bool = False*)

Constructor.

The optional locals argument will be passed to the InteractiveInterpreter base class.

The optional filename argument should specify the (file)name of the input stream; it will show up in tracebacks.

Methods

__init__([locals, stdout_callback, ...])	Constructor.
banner()	A banner similar to the one printed by the real Python interpreter.
complete(source)	Use CPython's rlcompleter to complete a source from local namespace.
flush_all()	Force stdout/stderr flush.
interact([banner, exitmsg])	Closely emulate the interactive Python console.
push(line)	Push a line to the interpreter.
raw_input([prompt])	Write a prompt and read a line.
redirect_stdstreams()	Toggle stdout/stderr redirections.
resetbuffer()	Reset the input buffer.
restore_stdstreams()	Restore stdout/stderr to the value it was before the creation of the object.
runcode(code)	Load imported packages then run code, async.
runsource(*args, **kwargs)	Force streams redirection.
showsyntaxerror([filename])	Display the syntax error that just occurred.
showtraceback()	Display the exception that just occurred.
stdstreams_redirections()	Ensure std stream redirection.

continues on next page

Table 2 – continued from previous page

write(data)	Write a string.

pyodide.console.repr_shorten

pyodide.console.**repr_shorten**(*value: Any*, *limit: int = 1000*, *split: Optional[int] = None*, *separator: str = '...'*)

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

If it is longer than *limit* then return the firsts *split* characters and the last *split* characters seperated by '...'. Default value for *split* is *limit // 2*.

pyodide.console.displayhook

pyodide.console.**displayhook**(*value*, *repr: Callable[[Any], str]*)

A displayhook with custom *repr* function.

It is intendend to overload *sys.displayhook*. Note that monkeypatch *builtins.repr* does not work in *sys.displayhook*. The pointer to *repr* seems hardcoded in default *sys.displayhook* version (which is written in C).

pyodide.webloop.WebLoop

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.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
add_reader(fd, callback, *args)	
add_signal_handler(sig, callback, *args)	
add_writer(fd, callback, *args)	
call_at(when, callback, *args[, context])	Like call_later(), but uses an absolute time.
call_exception_handler(context)	
call_later(delay, callback, *args[, context])	Arrange for a callback to be called at a given time.

continues on next page

Table 3 – continued from previous page

call_soon(callback, *args[, context])	Arrange for a callback to be called as soon as possible.
call_soon_threadsafe(callback, *args[, context])	Like call_soon(), but thread-safe.
close()	Close the loop.
connect_read_pipe(protocol_factory, pipe)	Register read pipe in event loop.
connect_write_pipe(protocol_factory, pipe)	Register write pipe in event loop.
create_connection(protocol_factory[, host, ...])	
create_datagram_endpoint(protocol_factory[, ...])	A coroutine which creates a datagram endpoint.
create_future()	Create a Future object attached to the loop.
create_server(protocol_factory[, host, ...])	A coroutine which creates a TCP server bound to host and port.
create_task(coro, *[, name])	Schedule a coroutine object.
create_unix_connection(protocol_factory[, ...])	
create_unix_server(protocol_factory[, path, ...])	A coroutine which creates a UNIX Domain Socket server.
default_exception_handler(context)	
get_debug()	
get_exception_handler()	
get_task_factory()	Return a task factory, or None if the default one is in use.
getaddrinfo(host, port, *[, family, type, ...])	
getnameinfo(sockaddr[, flags])	
is_closed()	Returns True if the event loop was closed.
is_running()	Returns True if the event loop is running.
remove_reader(fd)	
remove_signal_handler(sig)	
remove_writer(fd)	
run_forever()	Run the event loop forever.
run_in_executor(executor, func, *args)	
run_until_complete(future)	Run until future is done.
sendfile(transport, file[, offset, count, ...])	Send a file through a transport.
set_debug(enabled)	
set_default_executor(executor)	
set_exception_handler(handler)	

continues on next page

Table 3 – continued from previous page

set_task_factory(factory)	Set a task factory that will be used by loop.create_task().
shutdown_asyncgens()	Shutdown all active asynchronous generators.
sock_accept(sock)	
sock_connect(sock, address)	
sock_recv(sock, nbytes)	
sock_recv_into(sock, buf)	
sock_sendall(sock, data)	
sock_sendfile(sock, file[, offset, count, ...])	
start_tls(transport, protocol, sslcontext, *)	Upgrade a transport to TLS.
stop()	Stop the event loop as soon as reasonable.
subprocess_exec(protocol_factory, *args[, ...])	
subprocess_shell(protocol_factory, cmd, *[, ...])	
time()	Return the time according to the event loop's clock.

pyodide.create_proxy

pyodide.**create_proxy**(*obj: Any*) → pyodide._core.JsProxy

Create a *JsProxy* of a *PyProxy*.

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

pyodide.create_once_callable

pyodide.**create_once_callable**(*obj: Callable*) → pyodide._core.JsProxy

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.

1.6.2 Javascript API

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

pyodide.**runPython**(*code*)

Runs a string of Python code from Javascript.

The last part of the string may be an expression, in which case, its value is returned.

Arguments

- **code** (*string*) – Python code to evaluate

Returns The result of the python code converted to Javascript

pyodide.**runPythonAsync**(*code*, *messageCallback*, *errorCallback*)

Runs Python code, possibly asynchronously loading any known packages that the code imports. For example, given the following code

```
import numpy as np
x = np.array([1, 2, 3])
```

pyodide will first call `pyodide.loadPackage(['numpy'])`, and then run the code, returning the result. Since package fetching must happen asynchronously, this function returns a *Promise* which resolves to the output. For example:

```
pyodide.runPythonAsync(code, messageCallback)
       .then((output) => handleOutput(output))
```

Arguments

- **code** (*string*) – Python code to evaluate
- **messageCallback** (*function*) – A callback, called with progress messages. (optional)
- **errorCallback** (*function*) – A callback, called with error/warning messages. (optional)

pyodide.**globals**

type: PyProxy

An alias to the global Python namespace.

An object whose attributes are members of the Python global namespace. This is an alternative to `pyimport()`. For example, to access the `foo` Python object from Javascript use `pyodide.globals.get("foo")`

pyodide.**pyodide_py**

type: PyProxy

An alias to the Python pyodide package.

pyodide.**version**

type: string

The pyodide version.

It can be either the exact release version (e.g. `0.1.0`), or the latest release version followed by the number of commits since, and the git hash of the current commit (e.g. `0.1.0-1-bd84646`).

pyodide.**pyimport**(*name*)

Access a Python object in the global namespace from Javascript.

Arguments

- **name** (*string*) – Python variable name

Returns If the Python object is an immutable type (string, number, boolean), it is converted to Javascript and returned. For other types, a `PyProxy` object is returned.

pyodide.**loadPackage**(*names*, *messageCallback*, *errorCallback*)

Load a package or a list of packages over the network. This makes the files for the package available in the virtual filesystem. The package needs to be imported from Python before it can be used.

Arguments

- **names** (*String|Array*) – package name, or URL. Can be either a single element, or an array
- **messageCallback** (*function*) – A callback, called with progress messages (optional)
- **errorCallback** (*function*) – A callback, called with error/warning messages (optional)

Returns **Promise** – Resolves to `undefined` when loading is complete

`pyodide.`**loadedPackages**
type: object

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.`**loadPackagesFromImports**(*code*, *messageCallback*, *errorCallback*)
Inspect a Python code chunk and use *pyodide.loadPackage()* to load any known packages that the code chunk imports. Uses *pyodide_py.find_imports* to inspect the code.

For example, given the following code as input

```
import numpy as np
x = np.array([1, 2, 3])
```

loadPackagesFromImports() will call `pyodide.loadPackage(['numpy'])`. See also *runPythonAsync()*.

Arguments

- **code** (*) –
- **messageCallback** (*) –
- **errorCallback** (*) –

`pyodide.`**registerJsModule**(*name*, *module*)
Registers the Js object `module` as a Js module with `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 the `pyodide_py` api *pyodide.register_js_module()*.

Arguments

- **name** (*string*) – Name of js module to add
- **module** (*object*) – Javascript object backing the module

`pyodide.`**unregisterJsModule**(*name*)
Unregisters a Js module with given name that has been previously registered with *pyodide.registerJsModule()* or *pyodide.register_js_module()*. If a Js 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 the `pyodide_py` api *pyodide.unregister_js_module()*.

Arguments

- **name** (*string*) – Name of js module to remove

1.6.3 Micropip API

micropip.install(requirements)	Install the given package and all of its dependencies.

micropip.install

micropip.**install**(*requirements: Union[str, List[str]]*)

Install the given package and all of its dependencies.

See *loading packages* for more information.

This only works for packages that are either pure Python or for packages with C extensions that are built in Pyodide. If a pure Python package is not found in the Pyodide repository it will be loaded from PyPi.

Parameters **requirements** (str | List[str]) – A requirement or list of requirements to install. Each requirement is a string, which should be either a package name or URL to a wheel:

- If the requirement ends in .whl it will be interpreted as a URL. The file must be a wheel named in compliance with the PEP 427 naming convention.
- If the requirement does not end in .whl, it will interpreted as the name of a package. A package by this name must either be present in the Pyodide repository at languagePluginUrl or on PyPi

Returns A Future that resolves to None when all packages have been downloaded and installed.

Return type Future

1.7 Frequently Asked Questions (FAQ)

1.7.1 How can I load external Python files in Pyodide?

The two possible solutions are,

- include these files in a Python package, build a pure Python wheel with `python setup.py bdist_wheel` and *load it with micropip*.
- fetch the Python code as a string and evaluate it in Python,

```
pyodide.runPython(await fetch('https://some_url/...'))
```

In both cases, files need to be served with a web server and cannot be loaded from local file system.

1.7.2 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. You need to serve them with a web-browser. Recently there is a Native File System API supported in Chrome but not in Firefox. There is a discussion about implementing it for Firefox here.

1.7.3 How can I change the behavior of `runPython` and `runPythonAsync`?

The definitions of *runPython* and *runPythonAsync* are very simple:

```
function runPython(code){
  pyodide.pyodide_py.eval_code(code, pyodide.globals);
}
```

```
async function runPythonAsync(code, messageCallback, errorCallback) {
  await pyodide.loadPackagesFromImports(code, messageCallback, errorCallback);
  return pyodide.runPython(code);
};
```

To make your own version of *runPython*:

```
pyodide.runPython(`
  import pyodide
  def my_eval_code(code, ns):
    extra_info = None
    result = pyodide.eval_code(code, ns)
    return ns["extra_info"], result]
`)

function myRunPython(code){
  return pyodide.globals.get("my_eval_code")(code, pyodide.globals);
}

function myAsyncRunPython(code){
  await pyodide.loadPackagesFromImports(code, messageCallback, errorCallback);
  return pyodide.myRunPython(code, pyodide.globals);
}
```

Then `pyodide.myRunPython("2+7")` returns `[None, 9]` and `pyodide.myRunPython("extra_info='hello' ; 2 + 2")` returns `['hello', 4]`. If you want to change which packages *pyodide.loadPackagesFromImports* loads, you can monkey patch *pyodide.find_imports* which takes `code` as an argument and returns a list of packages imported.

1.7.4 How can I execute code in a custom namespace?

The second argument to *pyodide.eval_code* is a global namespace to execute the code in. The namespace is a Python dictionary.

```
let my_namespace = pyodide.globals.dict();
pyodide.pyodide_py.eval_code(`x = 1 + 1`, my_namespace);
pyodide.pyodide_py.eval_code(`y = x ** x`, my_namespace);
my_namespace.y; // ==> 4
```

This effectively runs the code in "module scope". Like the Python `eval` function you can provide a third argument to `eval_code` to specify a separate `locals` dict to run code in "function scope".

1.7.5 How to detect that code is run with Pyodide?

At run time, you can detect that a code is running with Pyodide using,

```
import sys

if "pyodide" in sys.modules:
   # running in Pyodide
```

More generally you can detect Python built with Emscripten (which includes Pyodide) with,

```
import platform

if platform.system() == 'Emscripten':
    # running in Pyodide or other Emscripten based build
```

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 environement.

To detect Pyodide, **at build time** use,

```
import os

if "PYODIDE" in os.environ:
    # building for Pyodide
```

We used to use the environment variable `PYODIDE_BASE_URL` for this purpose, but this usage is deprecated.

1.7.6 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
```

CHAPTER

TWO

DEVELOPING PYODIDE

The Development section help Pyodide contributors to find information about the development process including making packages to support third party libraries and understanding type conversions between Python and JavaScript.

The Project section helps contributors get started and gives additional information about the project's organization.

2.1 Building from sources

Building is easiest on Linux and relatively straightforward on Mac. For Windows, we currently recommend using the Docker image (described below) to build Pyodide. Another option for building on Windows is to use WSL2 to create a Linux build environment.

2.1.1 Build 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.

Additional build prerequisites are:

- A working native compiler toolchain, enough to build CPython.
- A native Python 3.8 to run the build scripts.
- CMake
- PyYAML
- FreeType 2 development libraries to compile Matplotlib.
- Cython to compile SciPy
- SWIG to compile NLopt
- gfortran (GNU Fortran 95 compiler)
- f2c
- ccache (optional) *highly* recommended for much faster rebuilds.

On Mac, you will also need:

- Homebrew for installing dependencies
- System libraries in the root directory (`sudo installer -pkg /Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.14.pkg -target /` should do it, see https://github.com/pyenv/pyenv/issues/1219#issuecomment-428305417)

- coreutils for md5sum and other essential Unix utilities (`brew install coreutils`)
- cmake (`brew install cmake`)
- Cython to compile SciPy (`brew install cython`)
- SWIG to compile NLopt (`brew install swig`)
- pkg-config (`brew install pkg-config`)
- openssl (`brew install openssl`)
- gfortran (`brew cask install gfortran`)
- f2c: Install wget (`brew install wget`), and then run the buildf2c script from the root directory (`sudo ./tools/buildf2c`)

After installing the build prerequisites, run from the command line:

```
make
```

2.1.2 Using Docker

We provide a Debian-based Docker image on Docker Hub with the dependencies already installed to make it easier to build Pyodide. On top of that we provide a pre-built image which can be used for fast custom and partial builds of pyodide. Note that building from the non pre-built the Docker image is *very* slow on Mac, building on the host machine is preferred if at all possible.

1. Install Docker
2. From a git checkout of Pyodide, run `./run_docker` or `./run_docker --pre-built`
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 one point in each subsequent try, increasing the maximum RAM usage available to the docker container might help [This is different from the physical RAM capacity inside the system]. 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 your source checkout on your host machine, and then repeatedly run `make` inside the Docker environment to test your changes.

2.1.3 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.

To build a minimal version of Pyodide, set `PYODIDE_PACKAGES="micropip"`. The packages micropip and distutils are always automatically included (but an empty `PYODIDE_PACKAGES` is interpreted as unset). As a shorthand for this, one can say `make minimal`.

2.1.4 Environment variables

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/dev/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 of 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.

2.2 Creating a Pyodide package

Pyodide includes a toolchain to make it easier to add new third-party Python libraries to the build. We automate the following steps:

- Download a source tarball (usually from PyPI)
- Confirm integrity of the package by comparing it to a checksum
- Apply patches, if any, to the source distribution
- Add extra files, if any, to the source distribution
- If the package includes C/C++/Cython extensions:
  - Build the package natively, keeping track of invocations of the native compiler and linker
  - Rebuild the package using emscripten to target WebAssembly
- If the package is pure Python:
  - Run the `setup.py` script to get the built package
- Package the results into an emscripten virtual filesystem package, which comprises:
  - A `.data` file containing the file contents of the whole package, concatenated together
  - A `.js` file which contains metadata about the files and installs them into the virtual filesystem.

Lastly, a `packages.json` file is output containing the dependency tree of all packages, so *`pyodide.loadPackage`* can load a package's dependencies automatically.

2.2.1 mkpkg

If you wish to create a new package for Pyodide, the easiest place to start is with the `mkpkg` tool. If your package is on PyPI, just run:

```
bin/pyodide mkpkg $PACKAGE_NAME
```

This will generate a `meta.yaml` (see below) that should work out of the box for many pure Python packages. This tool will populate the latest version, download link and sha256 hash by querying PyPI. It doesn't currently handle package dependencies, so you will need to specify those yourself.

2.2.2 The meta.yaml file

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, keeping the file format as close as possible to conda's should make it easier 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. (In the longer term, Pyodide may use conda as its packaging system, and this should hopefully ease that transition.)

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 alpha-numeric characters and `-`, `_`.

package/version

The version of the package.

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/md5

The MD5 checksum of the tarball. It is recommended to use SHA256 instead of MD5. At most one checksum entry should be provided per package.

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/skip_host

Skip building C extensions for the host environment. Default: `True`.

Setting this to `False` will result in ~2x slower builds for packages that include C extensions. It should only be needed when a package is a build time dependency for other packages. For instance, numpy is imported during installation of matplotlib, importing numpy also imports included C extensions, therefore it is built both for host and target.

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 clfags 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/library

Should be set to true for library packages. Library packages are packages that are needed for other packages but are not Python packages themselves. For library packages, the script specified in the `build/script` section is run to compile the library. See the zlib meta.yaml for an example of a library package specification.

build/sharedlibrary

Should be set to true for shared library packages. Shared library packages are packages that are needed for other packages, but are loaded dynamically when Pyodide is run. For shared library packages, the script specified in the `build/script` section is run to compile the library. The script should build the shared library and copy into into a subfolder of the source folder called `install`. Files or folders in this install folder will be packaged to make the Pyodide package. See 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.

build/post

Shell commands to run after building the library. These are run inside of `bash`, and there are two special environment variables defined:

- `$SITEPACKAGES`: The `site-packages` directory into which the package has been installed.
- `$PKGDIR`: The directory in which the `meta.yaml` file resides.

(This key is not in the Conda spec).

build/replace-libs

A list of strings of the form `<old_name>=<new_name>`, to rename libraries when linking. This in particular might be necessary when using emscripten ports. For instance, `png16=png` is currently used in matplotlib.

requirements

requirements/run

A list of required packages.

(Unlike conda, this only supports package names, not versions).

test

test/imports

List of imports to test after the package is built.

2.2.3 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. This directory should contain (at least) two files:

- `Makefile` that specifies how the library should be be built. Note that the build system will call `make`, not `emmake make`. The convention is that the source for the library is downloaded by the Makefile, as opposed to being included in the Pyodide repository.
- `meta.yaml` that specifies metadata about the package. For C libraries, only three options are supported:
  - `package/name`: The name of the library, which must equal the directory name.
  - `requirements/run`: The dependencies of the library, which can include both C libraries and Python packages.
  - `build/library`: This must be set to `true` to indicate that this is a library and not an ordinary package.

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.

2.2.4 Structure of a Pyodide package

This section describes the structure of a pure Python package, and how our build system creates it (In general, it is not recommended, to construct these by hand; instead create a Python wheel and install it with micropip)

Pyodide is obtained by compiling CPython into web assembly. As such, it loads packages the same way as CPython — it looks for relevant files `.py` files in `/lib/python3.x/`. When creating and loading a package, our job is to put our `.py` files in the right location in emscripten's virtual filesystem.

Suppose you have a Python library that consists of a single directory `/PATH/TO/LIB/` whose contents would go into `/lib/python3.8/site-packages/PACKAGE_NAME/` under a normal Python installation.

The simplest version of the corresponding Pyodide package contains two files — `PACKAGE_NAME.data` and `PACKAGE_NAME.js`. The first file `PACKAGE_NAME.data` is a concatenation of all contents of `/PATH/TO/LIB`. When loading the package via `pyodide.loadPackage`, Pyodide will load and run `PACKAGE_NAME.js`. The script then fetches `PACKAGE_NAME.data` and extracts the contents to emscripten's virtual filesystem. Afterwards, since the files are now in `/lib/python3.8/`, running `import PACKAGE_NAME` in Python will successfully import the module as usual.

To construct this bundle, we use the `file_packager.py` script from emscripten. We invoke it as follows:

```
$ ./file_packager.py PACKAGE_NAME.data \
    --js-output=PACKAGE_NAME.js \
    --export-name=pyodide._module \
    --use-preload-plugins \
    --preload /PATH/TO/LIB/@/lib/python3.8/site-packages/PACKAGE_NAME/ \
    --exclude "*__pycache__*" \
    --lz4
```

The arguments can be explained as follows:

- The `--preload` argument instructs the package to look for the file/directory before the separator `@` (namely `/PATH/TO/LIB/`) and place it at the path after the `@` in the virtual filesystem (namely `/lib/python3.8/site-packages/PACKAGE_NAME/`).
- The `--exclude` argument specifies files to omit from the package.
- The `--lz4` argument says to use LZ4 to compress the files

2.3 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.

2.3.1 Development Workflow

See *Building from sources* and *Testing and benchmarking* documentation.

For code-style the use of pre-commit is also recommended,

```
pip install pre-commit
pre-commit install
```

This will run a set of linters at each commit. Currently it runs yaml syntax validation and is removing trailing whitespaces.

2.3.2 Code of Conduct

Pyodide has adopted a *Code of Conduct* that we expect all contributors and core members to adhere to.

2.3.3 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 `master` in a non-breaking state, ideally with tests always passing. The unfortunate reality of software development is sometimes things break. As such, `master` cannot be expected to remain reliable at all times. We recommend using the latest stable version of Pyodide.

Pyodide follows semantic versioning (http://semver.org/) - 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.

2.3.4 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.

2.3.5 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 `master` 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 the default settings of `flake8`. C and Javascript are checked against the Mozilla style in `clang-format`.

2.3.6 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`.

2.3.7 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>
   ```

2.3.8 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 tl;drLegal 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.

2.3.9 Get in Touch

- **Gitter:** Pyodide currently shares the #iodide channel over at gitter.im

2.4 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`.

2.4.1 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 wasm 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 `Module.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 `Module.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 types from Python to Javascript, implicitly converting basic types and creating pyproxies for others. It also implements explicity 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.

2.4.2 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`.

2.4.3 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 adapators

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 = Module.hiwire.get_value(idfunc);
  let jsargs = Module.hiwire.get_value(idargs);
  return Module.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 would return `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), {
  Module.hiwire.get_value(idobj)[idx] = Module.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 Module.hiwire.get_value(idobj).length;
});
```

These wrappers enable the following sort of code:

```
try:
  jsfunc()
except JsException:
  print("Caught an exception thrown in Javascript!")
```

2.4.4 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 propogation, there will be many different execution paths, roughly linearly many in the length of the function. For example, the above psuedocode 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 `JsRef`s 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;
  }
```

1. 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 of 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;
```

1. 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);
  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();
```

(continues on next page)

(continued from previous page)

```
  }
// ... [SNIP]
```

1. 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);
```

2.4.5 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.

2.5 Testing and benchmarking

2.5.1 Testing

Requirements

Install the following dependencies into the default Python installation:

```
pip install pytest selenium pytest-instafail pytest-httpserver
```

Install geckodriver and chromedriver and check that they are in your `PATH`.

Running the test suite

To run the pytest suite of tests, type on the command line:

```
pytest src/ pyodide_build/ packages/*/test_*
```

There are 3 test locations,

- `src/tests/`: general Pyodide tests and tests running the CPython test suite
- `pyodide_build/tests/`: tests related to Pyodide build system (do not require selenium to run)
- `packages/*/test_*`: package specific tests.

Manual interactive testing

To run manual interactive tests, a docker environment and a webserver will be used.

1. Bind port 8000 for testing. To automatically bind port 8000 of the docker environment and the host system, run: `./run_docker`
2. Now, this can be used to test the Pyodide builds running within the docker environment using external browser programs on the host system. To do this, run: `./bin/pyodide serve`
3. This serves the `build` directory of the Pyodide project on port 8000.
   - To serve a different directory, use the `--build_dir` argument followed by the path of the directory.
   - To serve on a different port, use the `--port` argument followed by the desired port number. Make sure that the port passed in `--port` argument is same as the one defined as `DOCKER_PORT` in the `run_docker` script.
4. Once the webserver is running, simple interactive testing can be run by visiting this URL: http://localhost:8000/console.html

2.5.2 Benchmarking

To run common benchmarks to understand Pyodide's performance, begin by installing the same prerequisites as for testing. Then run:

```
make benchmark
```

2.5.3 Linting

Python is linted with `flake8`. C and Javascript are linted with `clang-format`.

To lint the code, run:

```
make lint
```

2.5.4 Testing framework

run_in_pyodide

Many tests simply involve running a chunk of code in Pyodide and ensuring it doesn't error. In this case, one can use the `run_in_pyodide` decorate from `pyodide_build/testing.py`, e.g.

```
from pyodide_build.testing import run_in_pyodide

@run_in_pyodide
def test_add():
    assert 1 + 1 == 2
```

In this case, the body of the function will automatically be run in Pyodide. The decorator can also be called with arguments. It has two configuration options — standalone and packages.

Setting `standalone = True` starts a standalone browser session to run the test (the session is shared between tests by default). This is useful for testing things like package loading.

The `packages` option lists packages to load before running the test. For example,

```
from pyodide_build.testing import run_in_pyodide

@run_in_pyodide(standalone = True, packages = ["regex"])
def test_regex():
    import regex
    assert regex.search("o", "foo").end() == 2
```

2.6 About the Project

The Python scientific stack, compiled to WebAssembly.

Pyodide brings the Python runtime to the browser via WebAssembly, along with the Python scientific stack including NumPy, Pandas, Matplotlib, parts of SciPy, and NetworkX. The packages directory lists over 75 packages which are currently available. In addition it's possible to install pure Python wheels from PyPi.

Pyodide provides transparent conversion of objects between Javascript and Python. When used inside a browser, Python has full access to the Web APIs.

2.6.1 History

Pyodide was created in 2018 by Michael Droettboom at Mozilla as part of the iodide project, a set of experiments around scientific computing and communication for the web.

At present Pyodide is an independent and community driven open-source project. The decision making process is outlined in the *project governance*. Pyodide may be used standalone in any context where you want to run Python inside a web browser.

2.6.2 Infrastructure support

We would also like to thank,

- Mozilla and CircleCl for Continuous Integration resources
- JsDelivr for providing a CDN for pyodide packages
- ReadTheDocs for hosting the documentation.

2.7 Code of Conduct

2.7.1 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.
- On IRC & any other IODIDE communication venue, please avoid using overtly sexual nicknames or other nicknames that 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 a trade-off and numerous costs. There is seldom a right answer.
- There are many unproductive habits that should be avoided in communication. We borrow the Recurse Center's "social rules": no feigning surprise, no well-actually's, no backseat driving, and no subtle -isms. Read the preceding link for further discussion.
- 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.
- We will exclude you from interaction if you insult, demean or harass anyone. That is not welcome behavior. We interpret the term "harassment" as including the definition in the Citizen Code of Conduct; if you have any lack of clarity about what might be included in that concept, 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, if you feel you have been or are being harassed or made uncomfortable by a community member, please contact one of the channel ops or any of the IODIDE core team members immediately. Whether you're a regular contributor or a newcomer, we care about making this community a safe place for you and we've got your back.
- Likewise any spamming, trolling, flaming, baiting or other attention-stealing behavior is not welcome.

2.7.2 Moderation

These are the policies for upholding our community's standards of conduct. If you feel that a thread needs moderation, please contact the IODIDE core team.

1. Remarks that violate the IODIDE standards of conduct, including hateful, hurtful, oppressive, or exclusionary remarks, are not allowed. (Cursing is allowed, but never targeting another user, and never in a hateful manner.)
2. Remarks that moderators find inappropriate, whether listed in the code of conduct or not, are also not allowed.
3. Moderators will first respond to such remarks with a warning.
4. If the warning is unheeded, the user will be "kicked," i.e., kicked out of the communication channel to cool off.
5. If the user comes back and continues to make trouble, they will be banned, i.e., indefinitely excluded.
6. Moderators may choose at their discretion to un-ban the user if it was a first offense and 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 IODIDE community we strive to go the extra step 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. And 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've communicated better — remember that it's your responsibility to make your fellow IODIDE 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 IODIDE venues. If you wish to use this code of conduct for your own project, consider explicitly mentioning your moderation policy or 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._

2.8 Pyodide 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.

2.8.1 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.

2.8.2 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.

2.9 Release notes

2.9.1 Version [Unreleased]

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 `toJs` API to `PyProxy` 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
- Feature A `JsProxy` of a Javascript `Promise` or other awaitable object is now a Python awaitable. #880
- 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
- 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

Fixed

- 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

pyodide-py package

- Feature Added an `InteractiveConsole` with completion support to ease the integration of Pyodide REPL in webpages (used in console.html) #1125 and #1155
- 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 interupt 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

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`.

Build system

- Enhancement Updated to latest emscripten 2.0.13 with the updstream 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.

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

Packages

- six, jedi and parso are no longer vendored in the main Pyodide package, and need to be loaded explicitly #1010, #987.
- Updated packages: bleach 3.3.0, packaging 20.8

2.9.2 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` 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

- Modifiy 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 descripion 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

2.9.3 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

2.9.4 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

2.9.5 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

2.9.6 Version 0.13.0

May 31, 2019

- Tagged versions of Pyodide are now deployed to Netlify.

2.9.7 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.

2.9.8 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`

2.9.9 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.

2.10 Related Projects

2.10.1 WebAssembly ecosystem

- emscripten is the compiler toolchain for WebAssembly that made Pyodide possible.

2.10.2 Notebook environements / IDEs / REPLs

- Iodide is a notebook-like environment for literate scientific computing and communication for the web. It is no longer actively developed. 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).

2.10.3 Other projects

- wc-code is a library to run Javascript, Python and Theme the browser with inline code blocks. It uses Pyodide to execute Python code.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

C

D

E

F

G

I

J

L

O

P

R

U

V

W