Pyodide

Release 0.15.0

Aug 25, 2020

Usage

Important: From your browser, you can try Pyodide in an Iodide notebook. The Iodide documentation site provides additional user and developer documentation.

The Python 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, try the experimental feature, Installing packages from PyPI and try to *pip install* the package.

To create a Pyodide package to support and share libraries for new applications, try Creating a Pyodide package.

CHAPTER 1

Using Pyodide

Pyodide may be used in several ways, including in an Iodide notebook, directly from JavaScript, or to execute Python scripts asynchronously in a web worker. Although still experimental, additional packages may be installed from PyPI to be used with Pyodide.

1.1 Using Pyodide from Iodide

This document describes using Pyodide inside Iodide. For information about using Pyodide directly from Javascript, see *Using Pyodide from Javascript*.

1.1.1 Running basic Python

Create a Python chunk, by inserting a line like this:

```
%% py
```

Type some Python code into the chunk, and press Shift+Enter to evaluate it. If the last clause in the cell is an expression, that expression is evaluated, converted to Javascript and displayed in the console like all other output in Javascript. See *type conversions* for more information about how data types are converted between Python and Javascript.

```
%% py
import sys
sys.version
```

1.1.2 Loading packages

Only the Python standard library and `six` are available after importing Pyodide. Other available libraries, such as `numpy` and `matplotlib` are loaded on demand.

If you just want to use the versions of those libraries included with Pyodide, all you need to do is import and start using them:

```
%% py
import numpy as np
np.arange(10)
```

For most uses, that is all you need to know.

However, if you want to use your own custom package or load a package from another provider, you'll need to use the `pyodide.loadPackage` function from a Javascript chunk. For example, to load a special distribution of Numpy from `custom.com`:

```
%% js
pyodide.loadPackage('https://custom.com/numpy.js')
```

After doing that, the numpy you import from a Python chunk will be this special version of Numpy.

1.1.3 Using a local build of Pyodide with Iodide

You may want to build a local copy of Pyodide with some changes and test it inside of Iodide.

By default, Iodide will use a copy of Pyodide deployed to Netlify. However, it will use locally-installed copy of Pyodide if `USE_LOCAL_PYODIDE` is set.

Set that environment variable in your shell:

```
export USE_LOCAL_PYODIDE=1
```

Then follow the building and running instructions for Iodide as usual.

Next, build Pyodide using the regular instructions in `../README.md`. Copy the contents of Pyodide's build directory to your Iodide checkout's `build/pyodide` directory:

```
mkdir $IODIDE_CHECKOUT/build/pyodide
cp $PYODIDE_CHECKOUT/build/* $IODIDE_CHECKOUT/build/pyodide
```

1.2 Using Pyodide from Javascript

This document describes using Pyodide directly from Javascript. For information about using Pyodide from Iodide, see *Using Pyodide from Iodide*.

1.2.1 Startup

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

https://pyodide-cdn2.iodide.io/v0.15.0/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 files 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 global scope: `pyodide`.

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

1.2.2 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, converted to Javascript objects (See *type conversions*).

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

1.2.3 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://pyodide-cdn2.iodide.io/v0.15.0/full/';
      </script>
      <script src="https://pyodide-cdn2.iodide.io/v0.15.0/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>
```

1.2.4 Loading packages

Only the Python standard library and `six` are available after importing Pyodide. To use other libraries, you'll need to load their package using `pyodide.loadPackage`. This downloads the file data over the network (as a `.data` and `.js` index file) and installs the files in the virtual filesystem.

Packages can be loaded by name, for those included in the official pyodide repository (e.g. `pyodide.loadPackage('numpy')`). It is also possible to load packages from custom URLs (e.g. `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.

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
});
```

1.2.5 Serving pyodide files

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'


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!

1.3 Using Pyodide from a web worker

This document describes how to use pyodide to execute python scripts asynchronously in a web worker.

1.3.1 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.

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, and add listeners for `onerror` and `onmessage`.

Call `postMessage` on your web worker, passing an object with the key `python` containing the script to execute as a string. You may pass other keys in the data object. By default the web worker assigns these to its global scope so that they may be imported from python. The results are returned as the `results` key, or if an error was encountered, it is returned in the `error` key.

For example:

```
var pyodideWorker = new Worker('./webworker.js')

pyodideWorker.onerror = (e) => {
  console.log(`Error in pyodideWorker at ${e.filename}, Line: ${e.lineno}, ${e.
↪message}`)
}

pyodideWorker.onmessage = (e) => {
  const {results, error} = e.data
  if (results) {
    console.log('pyodideWorker return results: ', results)
  } else if (error) {
    console.log('pyodideWorker error: ', error)
  }
}

const data = {
  A_rank: [0.8, 0.4, 1.2, 3.7, 2.6, 5.8],
  python:
    'import statistics\n' +
    'from js import A_rank\n' +
    'statistics.stdev(A_rank)'
}

pyodideWorker.postMessage(data)
```

1.3.2 Loading packages

Packages referenced from your python script will be automatically downloaded the first time they are encountered. Please note that some of the larger packages such as `numpy` or `pandas` may take several seconds to load. Subsequent uses of these packages will not incur the download overhead of the first run, but there is still some time required for the `import` in python itself.

If you would like to pre-load some packages, or the automatic package loading does not work for you for some reason, you may modify the `webworker.js` source to load some specific packages as described in *Using Pyodide directly from Javascript*.

For example, to always load packages `numpy` and `pytz`, you would insert the line `self.pyodide.loadPackage(['numpy', 'pytz']).then(() => {` as shown below:

```
self.languagePluginUrl = 'http://localhost:8000/'
importScripts('./pyodide.js')

var onmessage = function(e) { // eslint-disable-line no-unused-vars
  languagePluginLoader.then(() => {
    self.pyodide.loadPackage(['numpy', 'pytz']).then(() => {
      const data = e.data;
      const keys = Object.keys(data);
      for (let key of keys) {
        if (key !== 'python') {
          // Keys other than python must be arguments for the python script.
          // Set them on self, so that `from js import key` works.
          self[key] = data[key];
        }
      }
      self.pyodide.runPythonAsync(data.python, () => {})
          .then((results) => { self.postMessage({results}); })
          .catch((err) => {
            // if you prefer messages with the error
            self.postMessage({error : err.message});
            // if you prefer onerror events
            // setTimeout(() => { throw err; });
          });
    });
  });
}
```

1.3.3 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.

1.4 Installing packages from PyPI

Pyodide has experimental support for installing pure Python wheels from PyPI.

For use in Iodide:

```
%% py
import micropip
micropip.install('snowballstemmer')

# Iodide implicitly waits for the promise to resolve when the packages have finished
```

(continues on next page)

(continued from previous page)

```
# installing...

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

For use outside of Iodide (just Python), 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.

1.4.1 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'
)
```

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`.

The remote server 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.

1.4.2 Complete 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://pyodide-cdn2.iodide.io/v0.15.0/full/';
  </script>
  <script type="text/javascript" src="https://pyodide-cdn2.iodide.io/v0.15.0/full/
↪pyodide.js"></script>
```

(continues on next page)

(continued from previous page)

```
  <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 API Reference

pyodide version 0.1.0

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

1.5.1 Python API

pyodide.open_url(url)

Fetches a given *url* and returns a `io.StringIO` to access its contents.

Parameters

| name | type | description | |———-|——|—————| | *url* | str | the URL to open |

Returns

A `io.StringIO` object with the URL contents./

pyodide.eval_code(code, ns)

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

This function may be overridden to change how `pyodide.runPython` interprets code, for example to perform some preprocessing on the Python code first.

Parameters

| name | type | description | |———|———-|——————| | *code* | str | the code to evaluate | | *ns* | dict | evaluation name space |

Returns

Either the resulting object or `None`.

pyodide.as_nested_list(obj)

Converts Javascript nested arrays to Python nested lists. This conversion can not be performed automatically, because Javascript Arrays and Objects can be combined in ways that are ambiguous.

Parameters

| name | type | description | |———|——-|—————–| | *obj* | JS Object | The object to convert |

Returns

The object as nested Python lists.

1.5.2 Javascript API

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.

Parameters

| name | type | description | |————-|———-|———————————| | *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

Loading is asynchronous, therefore, this returns a `Promise`.

pyodide.loadedPackages

`Object` with loaded packages.

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

pyodide.pyimport(name)

Access a Python object from Javascript. The object must be in the global Python namespace.

For example, to access the `foo` Python object from Javascript:

```
var foo = pyodide.pyimport('foo')
```

Parameters

| name | type | description | |———|———|—————–| | *names* | String | Python variable name |

Returns

| name | type | description | |———|———|———————————| | *object* | *any* | If one of the basic types (string, | | | | number, boolean, array, object), the | | | | Python object is converted to | | | | Javascript and returned. For other | | | | types, a Proxy object to the Python | | | | object is returned. |

pyodide.globals

An object whose attributes are members of the Python global namespace. This is a more convenient alternative to `pyodide.pyimport`.

For example, to access the `foo` Python object from Javascript:

```
pyodide.globals.foo
```

pyodide.repr(obj)

Gets the Python's string representation of an object.

This is equivalent to calling `repr(obj)` in Python.

Parameters

| name | type | description | |———|———|—————–| | *obj* | *any* | Input object |

Returns

| name | type | description | |———|———|———————————–| | *str_repr* | String | String representation of the input object |

pyodide.runPython(code)

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

Parameters

| name | type | description | |———|———|———————————| | *code* | String | Python code to evaluate |

Returns

| name | type | description | |———|———|———————————| | *jsresult* | *any* | Result, converted to Javascript |

pyodide.runPythonAsync(code, messageCallback, errorCallback)

Runs Python code, possibly asynchronously loading any known packages that the code chunk imports.

For example, given the following code chunk

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

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

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

Parameters

| name | type | description | |———————-|———-|———————————| | *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) |

Returns

| name | type | description | |———|———|———————————————| | *result* | Promise | Resolves to the result of the code chunk |

pyodide.version()

Returns 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`).

Parameters

None

Returns

| name | type | description | |———|———|———————————————| | *version* | String | Pyodide version string |

1.6 Frequently Asked Questions (FAQ)

1.6.1 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.eval_code(pyodide.open_url('https://some_url/...'))
```

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

1.6.2 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.

CHAPTER 2

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.

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.
- lessc to compile less to css.
- uglifyjs to minify Javascript builds.
- 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`)
- 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. Note that building from 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`
3. Run `make` to build.

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
```

Note that this environment variable must contain both the packages and their dependencies. 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 micropip and package is generally always included for any non empty value of `PYODIDE_PACKAGES`.

If scipy is included in `PYODIDE_PACKAGES`, BLAS/LAPACK must be manually built first with `make -c packages/CLAPACK`.

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.

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/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/ldflags

Extra arguments to pass to the linker when building for WebAssembly.

(This key is not in the Conda spec).

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).

requirements

requirements/run

A list of required packages.

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

2.3 Type conversions

Python to Javascript conversions occur:

- when returning the final expression from a `pyodide.runPython` call (evaluating a Python cell in Iodide)
- using `pyodide.pyimport`
- passing arguments to a Javascript function from Python

Javascript to Python conversions occur:

- when using the `from js import ...` syntax
- returning the result of a Javascript function to Python

2.3.1 Basic types

The following basic types are implicitly converted between Javascript and Python. The values are copied and any connection to the original object is lost.

| Python | Javascript | |————————|————————|| `int`, `float` | `Number` || `str` | `String` || `True` | `true` || `False` | `false` || `None` | `undefined`, `null` || `list`, `tuple` | `Array` || `dict` | `Object` |

2.3.2 Typed arrays

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

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

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

Python `bytes` and `buffer` objects are converted to Javascript as `Uint8ClampedArrays`, without any memory copy at all, and 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.

2.3.3 Class instances

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

Javascript from Python

When passing a Javascript object to Python, an extension type is used to delegate Python operations to the Javascript side. The following operations are currently supported. (More should be possible in the future – work in ongoing to make this more complete):

| Python | Javascript | |————————|————————|| `repr(x)` | `x.toString()` || `x.foo` | `x.foo` || `x.foo = bar` | `x.foo = bar` || `del x.foo` | `delete x.foo` || `x(...)` | `x(...)` || `x.foo(...)` | `x.foo(...)`

| X.new(...) | new X(...) | len(x) | x.length | x[foo] | x[foo] | x[foo] = bar | x[foo] = bar | del x[foo] | delete x[foo] | x == y | x == y | x.typeof | typeof x |

One important difference between Python objects and Javascript objects is that if you access a missing member in Python, an exception is raised. In Javascript, it returns `undefined`. Since we can't make any assumptions about whether the Javascript member is missing or simply set to `undefined`, Python mirrors the Javascript behavior. For example:

```
// Javascript
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
point = new Point(42, 43))
```

```
# python
from js import point
assert point.y == 43
del point.y
assert point.y is None
```

Python from Javascript

When passing a Python object to Javascript, the Javascript Proxy API is used to delegate Javascript operations to the Python side. In general, the Proxy API is more limited than what can be done with a Python extension, so there are certain operations that are impossible or more cumbersome when using Python from Javascript than vice versa. The most notable limitation is that while Python has distinct ways of accessing attributes and items (`x.foo` and `x[foo]`), Javascript conflates these two concepts. The following operations are currently supported:

| Javascript | Python | |—————-|—————————| | foo in x | hasattr(x, 'foo') | | x.foo | getattr(x, 'foo') | | x.foo = bar | setattr(x, 'foo', bar) | | delete x.foo | delattr(x, 'foo') | | x.ownKeys() | dir(x) | | x(...) | x(...) | | x.foo(...) | x.foo(...) |

An additional limitation is that when passing a Python object to Javascript, there is no way for Javascript to automatically garbage collect that object. Therefore, custom Python objects must be manually destroyed when passed to Javascript, or they will leak. To do this, call `.destroy()` on the object, after which Javascript will no longer have access to the object.

```
var foo = pyodide.pyimport('foo');
foo.call_method();
foo.destroy();
foo.call_method(); // This will raise an exception, since the object has been
                   // destroyed
```

2.3.4 Using Python objects from Javascript

A Python object (in global scope) can be brought over to Javascript using the `pyodide.pyimport` function. It takes a string giving the name of the variable, and returns the object, converted to Javascript (See *type conversions*).

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

2.3.5 Using Javascript objects from Python

Javascript objects can be accessed from Python using the special `js` module. This module looks up attributes of the global (`window`) namespace on the Javascript side.

```
import js
js.document.title = 'New window title'
```

Performance considerations

Looking up and converting attributes of the `js` module happens dynamically. In most cases, where the value is small or results in a proxy, this is not an issue. However, if the value takes a long time to convert from Javascript to Python, you may want to store it in a Python variable or use the `from js import ...` syntax.

For example, given this large Javascript variable:

```
var x = new Array(1000).fill(0)
```

Use it from Python as follows:

```
import js
x = js.x  # conversion happens once here
for i in range(len(x)):
    item = x[i]  # we don't pay the conversion price each time here
```

Or alternatively:

```
from js import x  # conversion happens once here
for i in range(len(x)):
    item = x[i]  # we don't pay the conversion price each time here
```

2.4 How to Contribute

Thank you for your interest in contributing to PYODIDE! There are many ways to contribute, and we appreciate all of them. Here are some guidelines & pointers for diving into it.

2.4.1 Development Workflow

See building from sources and testing 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.

2.4.2 Code of Conduct

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

2.4.3 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, doc/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.

2.4.4 Bugs & Issues

We use Github Issues for announcing and discussing bugs and features. Use this link to report an 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.

2.4.5 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 py.test, 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`.

2.4.6 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>
   ```

2.4.7 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.

2.4.8 Get in Touch

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

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
```

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 test/ packages/
```

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
```

2.6 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 35 packages which are currently available.

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

While closely related to the iodide project, a tool for *literate scientific computing and communication for the web*, Pyodide goes beyond running in a notebook environment. To maximize the flexibility of the modern web, **Pyodide** may be used standalone in any context where you want to **run Python inside a web browser**.

Important: From your browser, you can try Pyodide in an Iodide notebook. The Iodide documentation site provides additional user and developer documentation.

2.7 Release notes

2.7.1 Version 0.16.0

Unreleased

- Pyodide now includes CPython 3.8.2 #712
- FIX Only call `Py_INCREF()` once when proxied by PyProxy #708
- Updated docker image to Debian buster
- FIX Infer package tarball directory from source url #687
- Updated to emscripten 1.38.31 #674
- New packages: freesasa, lxml, python-sat, traits, astropy
- Updated packages: numpy 1.15.4, pandas 1.0.5 among others.

2.7.2 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

2.7.3 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

2.7.4 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

2.7.5 Version 0.13.0

May 31, 2019

- Tagged versions of Pyodide are now deployed to Netlify.

2.7.6 Version 0.12.0

May 3, 2019

User improvements:

- Packages with pure Python wheels can now be loaded directly from PyPI. See `docs/pypi.md` 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.

2.7.7 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`

2.7.8 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.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search