PyJulia Documentation

Release 0.6.1

The Julia and IPython development teams

Feb 28, 2023

Contents:

Experimenting with developing a better interface to Julia language that works with Python 3 and Julia v1.0+.

PyJulia is tested against Python versions 3.5+

CHAPTER 1

Installation

tl;dr

1. *Install Julia*.
2. *Install PyJulia* by

```
$ python3 -m pip install --user julia
```

   Remove `--user` if you are using a virtual environment.

3. *Install Julia dependencies of PyJulia* by

```
$ python3
>>> import julia
>>> julia.install()
```

See below for more detailed explanations.

Note: If you are using Python installed with Ubuntu or `conda`, PyJulia may not work with the default setting. For workarounds, see *Troubleshooting*. Same caution applies to any Debian-based and possibly other GNU/Linux distributions.

1.1 Step 1: Install Julia

Get the Julia installer from https://julialang.org/downloads/. See also the Platform Specific Instructions.

Your python installation must be able to call command line program `julia`. If your installer does not add the Julia binary directory to your `PATH`, you will have to add it. *An alias will not work.*

Alternatively, you can pass the file path of the Julia executable to PyJulia functions. See *`julia.install`* and *`Julia`*.

1.2 Step 2: Install PyJulia

Note: If you are not familiar with `pip` and have some troubles with the following installation steps, we recommend going through the Tutorial in Python Packaging User Guide or pip's User Guide.

To get released versions you can use:

```
$ python3 -m pip install --user julia
$ python2 -m pip install --user julia  # If you need Python 2
```

where `--user` should be omitted if you are using virtual environment (`virtualenv`, `venv`, `conda`, etc.).

If you are interested in using the development version, you can install PyJulia directly from GitHub:

```
$ python3 -m pip install --user 'https://github.com/JuliaPy/pyjulia/archive/master.zip
↪#egg=julia'
```

You may clone it directly to (say) your home directory.

```
$ git clone https://github.com/JuliaPy/pyjulia
```

then inside the `pyjulia` directory you need to run the python setup file

```
$ cd pyjulia
$ python3 -m pip install --user .
$ python3 -m pip install --user -e .  # If you want "development install"
```

The `-e` flag makes a development install, meaning that any change to PyJulia source tree will take effect at next python interpreter restart without having to reissue an install command.

See *Testing* for how to run tests.

1.3 Step 3: Install Julia packages required by PyJulia

Launch a Python REPL and run the following code

```
>>> import julia
>>> julia.install()
```

This installs Julia packages required by PyJulia. See also *julia.install*.

Alternatively, you can use Julia's builtin package manager.

```
julia> using Pkg
julia> Pkg.add("PyCall")
```

Note that PyCall must be built with Python executable that is used to import PyJulia. See https://github.com/JuliaPy/PyCall.jl for more information about configuring PyCall.

CHAPTER 2

Usage

PyJulia provides a high-level interface which assumes a "normal" setup (e.g., `julia` program is in your `PATH`) and a low-level interface which can be used in a customized setup.

2.1 High-level interface

To call a Julia function in a Julia module, import the Julia module (say `Base`) with:

```
>>> from julia import Base
```

and then call Julia functions in `Base` from python, e.g.,

```
>>> Base.sind(90)
```

Other variants of Python import syntax also work:

```
>>> import julia.Base
>>> from julia.Base import Enums      # import a submodule
>>> from julia.Base import sin, sind # import functions from a module
```

The global namespace of Julia's interpreter can be accessed via a special module `julia.Main`:

```
>>> from julia import Main
```

You can set names in this module to send Python values to Julia:

```
>>> Main.xs = [1, 2, 3]
```

which allows it to be accessed directly from Julia code, e.g., it can be evaluated at Julia side using Julia syntax:

```
>>> Main.eval("sin.(xs)")
```

2.2 Low-level interface

If you need a custom setup for PyJulia, it must be done *before* importing any Julia modules. For example, to use the Julia executable named `custom_julia`, run:

```
>>> from julia import Julia
>>> jl = Julia(runtime="custom_julia")
```

You can then use, e.g.,

```
>>> from julia import Base
```

See also the API documentation for *Julia*.

2.3 IPython magic

In IPython (and therefore in Jupyter), you can directly execute Julia code using `%julia` magic:

```
In [1]: %load_ext julia.magic
Initializing Julia runtime. This may take some time...

In [2]: %julia [1 2; 3 4] .+ 1
Out[2]:
array([[2, 3],
       [4, 5]], dtype=int64)
```

You can call Python code from inside of `%julia` blocks via `$var` for accessing single variables or `py"..."` for more complex expressions:

```
In [3]: arr = [1, 2, 3]

In [4]: %julia $arr .+ 1
Out[4]:
array([2, 3, 4], dtype=int64)

In [5]: %julia sum(py"[x**2 for x in arr]")
Out[5]: 14
```

Inside of strings and quote blocks, `$var` and `py"..."` don't call Python and instead retain their usual Julia behavior. To call Python code in these cases, you can "escape" one extra time:

```
In [6]: foo = "Python"
        %julia foo = "Julia"
        %julia ("this is $foo", "this is $($foo)")
Out[6]: ('this is Julia', 'this is Python')
```

Expressions in macro arguments also always retain the Julia behavior:

```
In [7]: %julia @eval $foo
Out[7]: 'Julia'
```

Results are automatically converted between equivalent Python/Julia types (should they exist). You can turn this off by appending `o` to the Python string:

```
In [8]: %julia typeof(py"1"), typeof(py"1"o)
Out[8]: (<PyCall.jlwrap Int64>, <PyCall.jlwrap PyObject>)
```

Code inside `%julia` blocks obeys the Python scope:

```
In [9]: x = "global"
   ...: def f():
   ...:     x = "local"
   ...:     ret = %julia py"x"
   ...:     return ret
   ...: f()
Out[9]: 'local'
```

2.3.1 IPython configuration

PyJulia-IPython integration can be configured via IPython's configuration system. For the non-default behaviors, add the following lines in, e.g., `~/.ipython/profile_default/ipython_config.py` (see Introduction to IPython configuration).

To disable code completion in `%julia` and `%%julia` magics, use

```
c.JuliaMagics.completion = False  # default: True
```

To disable code highlighting in `%%julia` magic for terminal (non-Jupyter) IPython, use

```
c.JuliaMagics.highlight = False  # default: True
```

To enable Revise.jl automatically, use

```
c.JuliaMagics.revise = True  # default: False
```

2.4 Virtual environments

PyJulia can be used in Python virtual environments created by `virtualenv`, `venv`, and any tools wrapping them such as `pipenv`, provided that Python executable used in such environments are linked to identical libpython used by PyCall. If this is not the case, initializing PyJulia (e.g., `import julia.Main`) prints an informative error message with detected paths to libpython. See PyCall documentation for how to configure Python executable.

Note that Python environment created by `conda` is not supported.

CHAPTER 3

Troubleshooting

3.1 Your Python interpreter is statically linked to libpython

If you use Python installed with Debian-based Linux distribution such as Ubuntu or install Python by `conda`, you might have noticed that PyJulia cannot be initialized properly out-of-the-box. This is because those Python executables are statically linked to libpython. (See *Limitations* for why that's a problem.)

If you are unsure if your `python` has this problem, you can quickly check it by:

```
$ ldd /usr/bin/python
        linux-vdso.so.1 (0x00007ffd73f7c000)
        libpthread.so.0 => /usr/lib/libpthread.so.0 (0x00007f10ef84e000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f10ef68a000)
        libpython3.7m.so.1.0 => /usr/lib/libpython3.7m.so.1.0 (0x00007f10ef116000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2␣
↪(0x00007f10efaa4000)
        libdl.so.2 => /usr/lib/libdl.so.2 (0x00007f10ef111000)
        libutil.so.1 => /usr/lib/libutil.so.1 (0x00007f10ef10c000)
        libm.so.6 => /usr/lib/libm.so.6 (0x00007f10eef87000)
```

in Linux where `/usr/bin/python` should be replaced with the path to your `python` command (use `which python` to find it out). In macOS, use `otool -L` instead of `ldd`. If it does not print the path to libpython like `/usr/lib/libpython3.7m.so.1.0` in above example, you need to use one of the workaround below.

3.1.1 Turn off compilation cache

New in version 0.3.

The easiest workaround is to pass `compiled_modules=False` to the `Julia` constructor.

```
>>> from julia.api import Julia
>>> jl = Julia(compiled_modules=False)
```

This is equivalent to `julia`'s command line option `--compiled-modules=no` and disables the precompilation cache mechanism in Julia. Note that this option slows down loading and using Julia packages especially for complex and large ones.

See also API documentation of *`Julia`*.

3.1.2 Create a custom system image

New in version 0.4.

A very powerful way to avoid this the issue due to precompilation cache is to create a custom system image. This also has an additional benefit that initializing PyJulia becomes instant. See *Custom Julia system image* for how to create and use a custom system image.

3.1.3 `python-jl`: an easy workaround

New in version 0.2.

Another easy workaround is to use the `python-jl` command bundled in PyJulia. This can be used instead of normal `python` command for basic use-cases such as:

```
$ python-jl your_script.py
$ python-jl -c 'from julia.Base import banner; banner()'
$ python-jl -m IPython
```

See `python-jl --help` for more information.

How `python-jl` works

Note that `python-jl` works by launching Python interpreter inside Julia. Importantly, it means that PyJulia has to be installed in the Python environment with which PyCall is configured. That is to say, following commands must work for `python-jl` to be usable:

```
julia> using PyCall

julia> pyimport("julia")
PyObject <module 'julia' from '/.../julia/__init__.py'>
```

In fact, you can simply use PyJulia inside the Julia REPL, if you are comfortable with working in it:

```
julia> using PyCall

julia> py"""
       from julia import Julia
       Julia(init_julia=False)
       # Then use your Python module:
       from your_module_using_pyjulia import function
       function()
       """
```

3.1.4 Ultimate fix: build your own Python

Alternatively, you can use pyenv to build Python with `--enable-shared` option (see their Wiki page). Of course, manually building from Python source distribution with the same configuration also works.

```
$ PYTHON_CONFIGURE_OPTS="--enable-shared" pyenv install 3.6.6
Downloading Python-3.6.6.tar.xz...
-> https://www.python.org/ftp/python/3.6.6/Python-3.6.6.tar.xz
Installing Python-3.6.6...
Installed Python-3.6.6 to /home/USER/.pyenv/versions/3.6.6

$ ldd ~/.pyenv/versions/3.6.6/bin/python3.6 | grep libpython
        libpython3.6m.so.1.0 => /home/USER/.pyenv/versions/3.6.6/lib/libpython3.6m.so.
↪1.0 (0x00007fca44c8b000)
```

For more discussion, see: https://github.com/JuliaPy/pyjulia/issues/185

3.2 Segmentation fault in IPython

You may experience segmentation fault when using PyJulia in old versions of IPython. You can avoid this issue by updating IPython to 7.0 or above. Alternatively, you can use IPython via Jupyter (e.g., `jupyter console`) to workaround the problem.

3.3 Error due to `libstdc++` version

When you use PyJulia with another Python extension, you may see an error like `version `GLIBCXX_3.4.22' not found` (Linux) or `The procedure entry point ... could not be located in the dynamic link library libstdc++6.dll` (Windows). In this case, you might have observed that initializing PyJulia first fixes the problem. This is because Julia (or likely its dependencies like LLVM) requires a recent version of `libstdc++`.

Possible fixes:

- Initialize PyJulia (e.g., by `from julia import Main`) as early as possible. Note that just importing PyJulia (`import julia`) does not work.
- Load `libstdc++.so.6` first by setting environment variable `LD_PRELOAD` (Linux) to `/PATH/TO/JULIA/DIR/lib/julia/libstdc++.so.6` where `/PATH/TO/JULIA/DIR/lib` is the directory which has `libjulia.so`. macOS and Windows likely to have similar mechanisms (untested).
- Similarly, set environment variable `LD_LIBRARY_PATH` (Linux) to `/PATH/TO/JULIA/DIR/lib/julia` directory. Using `DYLD_LIBRARY_PATH` on macOS and `PATH` on Windows may work (untested).

See: https://github.com/JuliaPy/pyjulia/issues/180, https://github.com/JuliaPy/pyjulia/issues/223

CHAPTER 4

API

4.1 Utility functions

julia.**install**(*, *julia="julia"*, *color="auto"*)

Install Julia packages required by PyJulia in *julia*.

This function installs and/or re-builds PyCall if necessary. It also makes sure to build PyCall in a way compatible with this Python executable (if possible).

Keyword Arguments

- **julia** (*str*) – Julia executable (default: "julia")
- **color** (*"auto", False or True*) – Use colorful output if True. "auto" (default) to detect it automatically.

4.2 Low-level API

class julia.api.**Julia**(*init_julia=True*, *jl_init_path=None*, *runtime=None*, *jl_runtime_path=None*, *debug=False*, ***julia_options*)

Implements a bridge to the Julia runtime. This uses the Julia PyCall module to perform type conversions and allow full access to the entire Julia runtime.

__init__(*init_julia=True*, *jl_init_path=None*, *runtime=None*, *jl_runtime_path=None*, *debug=False*, ***julia_options*)

Create a Python object that represents a live Julia runtime.

Note: Use *LibJulia* to fully control the initialization of the Julia runtime.

Parameters

- **init_julia** (*bool*) – If True, try to initialize the Julia runtime. If this code is being called from inside an already running Julia, the flag should be passed as False so the interpreter isn't re-initialized.

Note that it is safe to call this class constructor twice in the same process with *init_julia* set to True, as a global reference is kept to avoid re-initializing it. The purpose of the flag is only to manage situations when Julia was initialized from outside this code.

- **runtime** (*str*) – Custom Julia binary, e.g. "/usr/local/bin/julia" or "julia-1.0.0".
- **debug** (*bool*) – If True, print some debugging information to STDERR
- **bindir** (*str*) – Set location of *julia* executable relative to which we find system image (`sys.so`). It is inferred from `runtime` if not given. Equivalent to `--home` of the Julia CLI.
- **check_bounds** (*{True, False, 'yes', 'no'}*) – Emit bounds checks always or never (ignoring declarations). `True` and `False` are synonym of `'yes'` and `'no'`, respectively. This applies to all other options.
- **compile** (*{True, False, 'yes', 'no', 'all', 'min'}*) – Enable or disable JIT compiler, or request exhaustive compilation.
- **compiled_modules** (*{True, False, 'yes', 'no'}*) – Enable or disable incremental precompilation of modules.
- **depwarn** (*{True, False, 'yes', 'no', 'error'}*) – Enable or disable syntax and method deprecation warnings ("error" turns warnings into errors).
- **inline** (*{True, False, 'yes', 'no'}*) – Control whether inlining is permitted, including overriding @inline declarations.
- **optimize** (*{0, 1, 2, 3}*) – Set the optimization level (default level is 2 if unspecified or 3 if used without a level).
- **sysimage** (*str*) – Start up with the given system image file.
- **warn_overwrite** (*{True, False, 'yes', 'no'}*) – Enable or disable method overwrite warnings.
- **min_optlevel** (*{0, 1, 2, 3}*) – Lower bound on the optimization level.
- **threads** (*{int, 'auto'}*) – How many threads to use.

eval(*src*)
Execute code in Julia, then pull some results back to Python.

help(*name*)
Return help string for function by name.

using(*module*)
Load module in Julia by calling the `using module` command

class julia.api.**LibJulia**(*libjulia_path*, *bindir*, *sysimage*)
Low-level interface to `libjulia` C-API.

Examples

```
>>> from julia.api import LibJulia, JuliaInfo
```

An easy way to create a *LibJulia* object is *LibJulia.load*:

```
>>> api = LibJulia.load()                                    # doctest: +SKIP
```

Or, equivalently,

```
>>> api = LibJulia.load(julia="julia")                    # doctest: +SKIP
>>> api = LibJulia.from_juliainfo(JuliaInfo.load())      # doctest: +SKIP
```

You can pass a path to the Julia executable using *julia* keyword argument:

```
>>> api = LibJulia.load(julia="PATH/TO/CUSTOM/julia")   # doctest: +SKIP
```

Path to the system image can be configured before initializing Julia:

```
>>> api.sysimage                                         # doctest: +SKIP
'/home/user/julia/lib/julia/sys.so'
>>> api.sysimage = "PATH/TO/CUSTOM/sys.so"              # doctest: +SKIP
```

Finally, the Julia runtime can be initialized using *LibJulia.init_julia*. Note that only the first call to this function in the current Python process takes effect.

```
>>> api.init_julia()
```

Any command-line options supported by Julia can be passed to *init_julia*:

```
>>> api.init_julia(["--compiled-modules=no", "--optimize=3"])
```

Once *init_julia* is called, any subsequent use of *Julia* API (thus also `from julia import <JuliaModule>` etc.) uses this initialized Julia runtime.

LibJulia can be used to access Julia's C-API:

```
>>> ret = api.jl_eval_string(b"Int64(1 + 2)")
>>> int(api.jl_unbox_int64(ret))
3
```

However, a proper use of the C-API is more involved and presumably very challenging without C macros. See also: https://docs.julialang.org/en/v1/manual/embedding/.

libjulia_path
Path to libjulia.

Type str

bindir
`Sys.BINDIR` of *julia*. This is passed to `jl_init_with_image` unless overridden by argument `option` to *init_julia*.

Type str

sysimage
Path to system image. This is passed to `jl_init_with_image` unless overridden by argument `option` to *init_julia*.

If *Custom Julia system image* is a relative path, it is interpreted relative to the current directory (rather than relative to the Julia *bindir* as in the `jl_init_with_image` C API).

Type str

init_julia(*options=None*)
Initialize `libjulia`. Calling this method twice is a no-op.

It calls `jl_init_with_image` (or `jl_init_with_image__threading`) but makes sure that it is called only once for each process.

Parameters **options** (sequence of `str` or `JuliaOptions`) – This is passed as command line options to the Julia runtime.

> **Warning:** Any invalid command line option terminates the entire Python process.

classmethod load(***kwargs*)

Create *LibJulia* based on information retrieved with *JuliaInfo.load*.

This classmethod runs *JuliaInfo.load* to retrieve information about *julia* runtime. This information is used to intialize *LibJulia*.

class julia.api.**JuliaInfo**(*julia*, *version_raw*, *version_major*, *version_minor*, *version_patch*, *bindir=None*, *libjulia_path=None*, *sysimage=None*, *python=None*, *libpython_path=None*)

Information required for initializing Julia runtime.

Examples

```
>>> from julia.api import JuliaInfo
>>> info = JuliaInfo.load()
>>> info = JuliaInfo.load(julia="julia")  # equivalent
>>> info = JuliaInfo.load(julia="PATH/TO/julia")       # doctest: +SKIP
>>> info.julia
'julia'
>>> info.sysimage                                      # doctest: +SKIP
'/home/user/julia/lib/julia/sys.so'
>>> info.python                                        # doctest: +SKIP
'/usr/bin/python3'
>>> info.is_compatible_python()                        # doctest: +SKIP
True
```

julia

Path to a Julia executable from which information was retrieved.

Type str

bindir

`Sys.BINDIR` of *julia*.

Type str

libjulia_path

Path to libjulia.

Type str

sysimage

Path to system image.

Type str

python

Python executable with which PyCall.jl is configured.

Type str

libpython_path

libpython path used by PyCall.jl.

Type str

is_compatible_python()
Check if python used by PyCall.jl is compatible with `sys.executable`.

classmethod load(*julia='julia'*, ***popen_kwargs*)
Get basic information from `julia`.

class julia.api.**JuliaError**
Wrapper for Julia exceptions.

CHAPTER 5

Custom Julia system image

New in version 0.4.

If you use standard `julia` program, the basic functionalities and standard libraries of Julia are loaded from so called *system image* file which contains the machine code compiled from the Julia code. The Julia runtime can be configured to use a customized system image which may contain non-standard packages. This is a very effective way to reduce startup time of complex Julia packages such as PyCall. Furthermore, it can be used to workaround the problem in statically linked Python executable if you have the problem described in *Your Python interpreter is statically linked to libpython*.

5.1 How to use a custom system image

To compile a custom system image for PyJulia, run

```
$ python3 -m julia.sysimage sys.so
```

where `sys.dll` and `sys.dylib` may be used instead of `sys.so` in Windows and macOS, respectively.

The command line interface *`julia.sysimage`* will:

- Install packages required for compiling the system image in an isolated Julia environment.
- Install PyCall to be compiled into the system image in an isolated Julia environment.
- Create the system image at the given path (`./sys.so` in the above example).

To use this system image with PyJulia, you need to specify its path using `sysimage` keyword argument of the *`Julia`* constructor. For example, if you run `python3` REPL at the directory where you ran the above *`julia.sysimage`* command, you can do

```
>>> from julia import Julia
>>> jl = Julia(sysimage="sys.so")
```

to initialize PyJulia. To check that this Julia runtime is using the correct system image, look at the output of `Base.julia_cmd()`

```
>>> from julia import Base
>>> Base.julia_cmd()
<PyCall.jlwrap `/PATH/TO/bin/julia-py -Cnative -J/PATH/TO/sys.so -g1`>
```

5.2 Limitations

- `PyCall` and its dependencies cannot be updated after the system image is created. A new system image has to be created to update those packages.
- The system image generated by *`julia.sysimage`* uses a different set of precompilation cache paths for each pair of `julia-py` executable and the system image file. Precompiled cache files generated by `julia` or a different `julia-py` executable cannot be reused by PyJulia when using the system image generated by *`julia.sysimage`*.
- The absolute path of `julia-py` is embedded in the system image. This system image is not usable if `julia-py` is removed.

5.3 Command line interfaces

5.3.1 `python3 -m julia.sysimage`

Build system image.

Example:

```
python3 -m julia.sysimage sys.so
```

Generated system image can be passed to `sysimage` option of *`julia.api.Julia`*.

Note: This script is not tested on Windows.

5.3.2 `julia-py`

Launch Julia through PyJulia.

Currently, `julia-py` is primary used internally for supporting *`julia.sysimage`* command line interface. Using `julia-py` like normal Julia program requires `--sysimage` to be set to the system image created by *`julia.sysimage`*.

Example:

```
$ python3 -m julia.sysimage sys.so
$ julia-py --sysimage sys.so
```

CHAPTER 6

pytest plugin

PyJulia automatically installs a pytest plugin. It takes care of tricky aspects of PyJulia initialization:

- It loads `libjulia` as early as possible to avoid incompatibility of shared libraries such as `libstdc++` (assuming that the ones bundled with `julia` are newer than the ones otherwise loaded).
- It provides a way to succinctly mark certain tests require Julia runtime (see *Fixture* and *Marker*).
- The tests requiring Julia can be skipped with *`--no-julia`*.
- It enables debug-level logging. This is highly recommended especially in CI setting as miss-configuration of PyJulia may result in segmentation fault in which Python cannot provide useful traceback.

To activate PyJulia's pytest plugin[1] add `-p julia.pytestplugin` to the command line option. There are several ways to do this by default in your project. One option is to include this using `addopts` setup of `pytest.ini` or `tox.ini` file. See How to change command line options defaults:

```
[pytest]
addopts =
    -p julia.pytestplugin
```

6.1 Options

Following options can be passed to **`pytest`**

`--no-julia`
: Skip tests that require julia.

`--julia`
: Undo `--no-julia`; i.e., run tests that require julia.

`--julia-runtime`
: Julia executable to be used. Defaults to environment variable *`PYJULIA_TEST_RUNTIME`*.

[1] This plugin is not activated by default (as in normal `pytest-*` plugin packages) to avoid accidentally breaking user's `pytest` setup when PyJulia is included as a non-test dependency.

--julia-<julia_option>
: Some `<julia_option>` that can be passed to `julia` executable (e.g., `--compiled-modules=no`) can be passed to `pytest` plugin by `--julia-<julia_option>` (e.g., `--julia-compiled-modules=no`). See `pytest -p julia.pytestplugin --help` for the actual list of options.

6.2 Fixture

PyJulia's pytest plugin includes a pytest fixture `julia` which is set to an instance of *`Julia`* that is appropriately initialized. Example usage:

```
def test_eval(julia):
    assert julia.eval("1 + 1") == 2
```

This fixture also "marks" that this test requires a Julia runtime. Thus, the tests using `julia` fixture are not run when *`--no-julia`* is passed.

6.3 Marker

PyJulia's pytest plugin also includes a pytest marker `julia` which can be used to mark that the test requires PyJulia setup. It is similar to `julia` fixture but it does not instantiate the actual *`Julia`* object.

Example usage:

```
import pytest

@pytest.mark.julia
def test_import():
    from julia import MyModule
```

CHAPTER 7

How it works

PyJulia loads the `libjulia` library and executes the statements therein. To convert the variables, the `PyCall` package is used. Python references to Julia objects are reference counted by Python, and retained in the `PyCall.pycall_gc` mapping on the Julia side (the mapping is removed when reference count drops to zero, so that the Julia object may be freed).

CHAPTER 8

Limitations

8.1 Mismatch in valid set of identifiers

Not all valid Julia identifiers are valid Python identifiers. Unicode identifiers are invalid in Python 2.7 and so PyJulia cannot call or access Julia methods/variables with names that are not ASCII only. Although Python 3 allows Unicode identifiers, they are more aggressively normalized than Julia. For example, ℇ (GREEK LUNATE EPSILON SYMBOL) and ϵ (GREEK SMALL LETTER EPSILON) are identical in Python 3 but different in Julia. Additionally, it is a common idiom in Julia to append a `!` character to methods which mutate their arguments. These method names are invalid Python identifers. PyJulia renames these methods by subsituting `!` with `_b`. For example, the Julia method `sum!` can be called in PyJulia using `sum_b(...)`.

8.2 Pre-compilation mechanism in Julia 1.0

There was a major overhaul in the module loading system between Julia 0.6 and 1.0. As a result, the "hack" supporting the PyJulia to load PyCall stopped working. For the implementation detail of the hack, see: https://github.com/JuliaPy/pyjulia/tree/v0.3.0/src/julia/fake-julia

For the update on this problem, see: https://github.com/JuliaLang/julia/issues/28518

8.3 Ctrl-C does not work / terminates the whole Python process

Currently, initializing PyJulia (e.g., by `from julia import Main`) disables `KeyboardInterrupt` handling in the Python process. If you are using normal `python` interpreter, it means that canceling the input by Ctrl-C does not work and repeatedly providing Ctrl-C terminates the whole Python process with the error message `WARNING: Force throwing a SIGINT`. Using IPython 7.0 or above is recommended to avoid such accidental shutdown.

It also means that there is no safe way to cancel long-running computations or I/O at the moment. Sending SIGINT with Ctrl-C will terminate the whole Python process.

For the update on this problem, see: https://github.com/JuliaPy/pyjulia/issues/211

8.4 No threading support

PyJulia cannot be used in different threads since libjulia is not thread safe. However, you can use multiple threads within Julia. For example, start IPython by `JULIA_NUM_THREADS=4 ipython` and then run:

```
In [1]: %load_ext julia.magic
Initializing Julia runtime. This may take some time...

In [2]: %%julia
   ...: a = zeros(10)
   ...: Threads.@threads for i = 1:10
   ...:     a[i] = Threads.threadid()
   ...: end
   ...: a
Out[3]: array([1., 1., 1., 2., 2., 2., 3., 3., 4., 4.])
```

8.5 PyJulia does not release GIL

PyJulia does not release the Global Interpreter Lock (GIL) while calling Julia functions since PyCall expects the GIL to be acquired always. It means that Python code and Julia code cannot run in parallel.

CHAPTER 9

Testing

PyJulia can be tested by simply running `tox`.

```
$ tox
```

The full syntax for invoking `tox` is

```
$ [PYJULIA_TEST_REBUILD=yes] \
  [PYJULIA_TEST_RUNTIME=<julia>] \
  tox [options] [-- pytest options]
```

`PYJULIA_TEST_REBUILD`
: *Be careful using this environment variable!* When it is set to `yes`, your `PyCall.jl` installation will be rebuilt using the Python interpreter used for testing. The test suite tries to build back to the original configuration but the precompilation would be in the stale state after the test. Note also that it does not work if you unconditionally set `PYTHON` environment variable in your Julia startup file.

`PYJULIA_TEST_RUNTIME`
: `julia` executable to be used for testing. See also *`pytest --julia-runtime`*.

`[-- pytest options]` Positional arguments after `--` are passed to `pytest`.

For example,

```
$ PYJULIA_TEST_REBUILD=yes \
  PYJULIA_TEST_RUNTIME=~/julia/julia \
  tox -e py37 -- -s
```

means to execute tests with

- PyJulia in shared-cache mode
- `julia` executable at `~/julia/julia`
- Python 3.7
- `pytest`'s capturing mode turned off

CHAPTER 10

Development

10.1 Release

10.1.1 Step 1: Release

Bump the version number and push the change to `release/main` branch in https://github.com/JuliaPy/pyjulia. This triggers a CI that:

1. releases the package on `test.pypi.org`,
2. installs the released package,
3. runs the test with the installed package and then
4. re-releases the package on `pypi.org`.

10.1.2 Step 2: Tag

Create a Git tag with the form `vX.Y.Z`, merge `release/main` to `master` branch, and then push the tag and `master` branch.

10.2 Special branches

`release/main` Push to this branch triggers the deploy to `test.pypi.org`, test the uploaded package, and then re-upload it to `pypi.org`.

`release/test` Push to this branch triggers the deploy to `test.pypi.org` and test the uploaded package.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

Symbols

B

E

H

I

J

L

P

S

U