Published in Netherlands eScience Center

Abel Soares Siqueira Follow

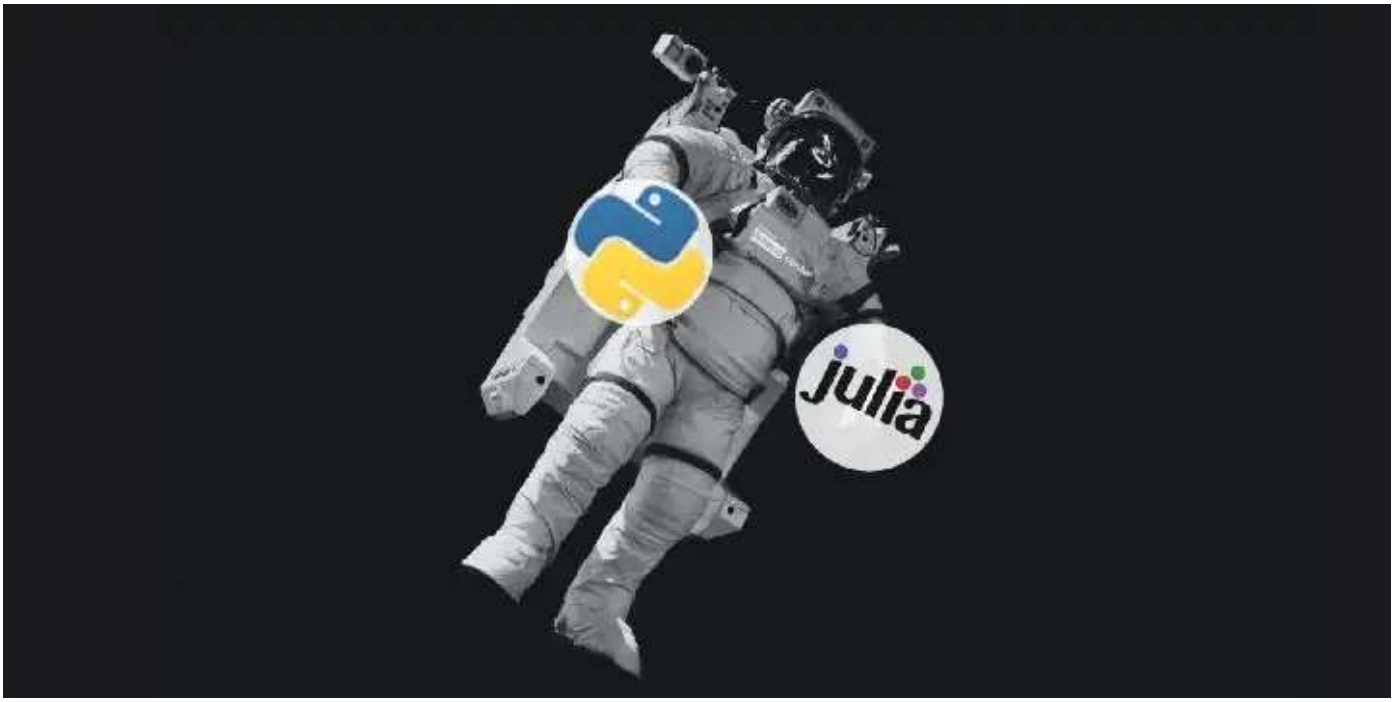
Jan 19 · 6 min read · Listen

Save

How to call Julia code from Python

A three-part series on achieving high performance with high-level code

By Abel Soares Siqueira and Faruk Diblen



Caption: Astronaut carrying Python and Julia. Photo by Brian McGowan on Unsplash (https://unsplash.com/photos/MR9xsNWVKvo), modified by us.

Target audience

This is the first post in a three-part series about achieving high performance with high-level code. This series is aimed at people working with Python who needs better performance but prefers not to develop a low-level performant library.

Introduction

Having recently joined the Netherlands eScience Center after working with research using the Julia language for seven years, I was excited to highlight some of its cool features. At the eScience Center, many of our engineers use Python, some use C or C++, and in some cases, we see Python calling C++ code, to speed up the code. This was the perfect opportunity to introduce Julia's interoperability with Python and to investigate whether we could achieve comparable speed by calling Julia code in Python. This means that for situations where Python's performance is not sufficient, we can speed it up with another high-level language, avoiding the use of a low-level language like C++. This series of three blog posts will investigate these topics.

In this first post, we will learn how to call a Julia code from Python. We will set up the environment and show some examples. In the second post, we will take a problem that was solved by using Python in combination with C++ to speed up the code. We will replace the C++ code with a Julia code and compare the performance. In the third post, we will solve the same problem in Julia, optimize the Julia code to reach its maximum performance and compare it with the implementation in the second post.

What is Julia, and how does it compare to Python?

What is Julia? Julia is a high-performance, high-level programming language. It was created a few years ago with the ambitious goal of being fast with a high-level syntax, and it has been mostly successful. It can, in a few cases, reach the speed of low-level programming languages like C. For more information, the julialang.org site is a great first stop.

One of the most frequently asked questions is: "how does it compare to Python or some other programming language in terms of performance?". The short answer: Julia is generally faster than Python and many other programming languages.

The performance of a Python code can be optimized, but even the optimized code usually underperforms compared to a Julia version of the same code. The performance

increase of a Python code can be achieved in a few ways, but a frequent one is to call a code written in a low-level language, such as C, C++ and Fortran, like NumPy which does its calculations mostly in those low-level languages. What is less common, but also possible, is to call Julia from Python. In this post, we are going to show you how to do that!

Before we forget, all the code used in this post can be found in our GitHub repository. We have also created a Docker image that includes a ready-to-use environment to run both Julia and Python. To run that environment with Python 3.10 and Julia 1.6, install Docker and run the following in your terminal:

```
$ docker pull abelsiqueira/python-and-julia:py3.10-jl1.6
$ docker run -it exec abelsiqueira/python-and-julia:py3.10-jl1.6
/bin/bash
```

Preparation

In the following steps, we will configure our system to execute Julia code from Python. To learn more about this topic, the documentation for the packages we describe below is a great starting point. You will need four things:

1. **Python** distribution compiled with shared libpython option. There are workarounds, but this is the most straightforward way.
2. **Julia**, the executable that runs the Julia language.
3. **PyCall**, the Julia package that defines the conversions between Julia and Python.
4. **PyJulia**, the Python package to access Julia from Python.

We are going to go through the installation and configuration of these steps on a Linux system. It will be very similar on MacOS or with WSL for Windows once the required tools are installed.

Step 1: Python with shared libpython

To check whether the Python distribution is compiled with **--enable-shared** option, we run:

```
ldd $(which python3) | grep libpython
```

If the output is something like:

```
libpython3.10.so.1.0 => /usr/local/lib/libpython3.10.so.1.0
(0x00007f567e548000)
```

... then we are good to go! If we get nothing, that means that the Python distribution has not been compiled with the desired flags. In this case, we can compile our own Python distribution with the flag --**enable-shared**, which takes some time but is mostly straightforward. This Dockerfile has the instructions. Remember that if you just want to test it out, you can run the Docker image as mentioned in the previous section.

Step 2: Julia and PyCall

Now, we will install Julia. We recommend using jill, a script I created, which downloads and installs a specific version of Julia, but Julia can also be installed via the official binaries or package managers. In this post, we use version 1.6.5, which is the current Long Term Support version at the time of writing. Most likely this will work with a newer version as well. To install Julia 1.6.5 using jill, we run:

```
$ wget
https://raw.githubusercontent.com/abelsiqueira/jill/v0.4.0/jill.sh
$ sudo bash jill.sh -y -v 1.6.5
```

Now, we will install PyCall and configure it to use the correct Python version. We start Julia by running `julia` in the terminal, and then we set the `ENV["PYTHON"]` variable:

```
$ julia
julia> ENV["PYTHON"] = "PATH/TO/python"
```

Here we use the full path to Python's executable. In our case, it is the Python distribution we compiled from the source code. You could change the path according to your configuration.

Now, we will install PyCall using Pkg, Julia's package manager:

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

Step 3: PyJulia

As the last step, we must install the Python package to talk with Julia. First, use pip, Python's package manager, to install the package PyJulia — remember to use the same Python passed to `ENV["PYTHON"]`:

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

To finalize configuring the communication between Julia and Python, we run the following in the Python interpreter:

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

If we had more than one Julia version on our system, we could specify it with an argument:

```
>>> julia.install(julia='julia-1.6.5')
```

We test the installation running the following in the Python interpreter run:

```
>>> from julia import Main
>>> Main.eval('[x^2 for x in 0:4]')
```

Showcasing PyJulia

Basics

- To use a Julia module, use `from julia import MODULE`
- To evaluate a command, import `Main` and use `Main.eval("...")`
- To create and use variables, use `Main.VARIABLE`
- To install Julia packages, `import Pkg` and use `Pkg.add("Package")`
- Use `%load_ext julia.magic` to add a IPython's magic command called `%julia`. Just prepend `%julia` to Julia commands. In this case, use `$var` to access python variables

Example: Linear Algebra

In this short example, we can see one of the strengths of Julia syntax for Linear Algebra. A random linear system is created and solved. The result is checked with NumPy, so we can see the compatibility.

```
In [1]: from julia import Main
In [2]: import numpy as np
In [3]: %load_ext julia.magic
Initializing Julia interpreter. This may take some time...

In [4]: Main.eval('A = rand(3, 3)')
Out[4]:
array([[0.27076591, 0.11216354, 0.96836277],
       [0.62722533, 0.37383451, 0.81030294],
       [0.61029488, 0.7961276 , 0.93911852]])

In [5]: Main.b = Main.eval('A * ones(3)')
In [6]: %julia x = A \ b
Out[6]: array([1., 1., 1.])

In [7]: np.linalg.norm(np.matmul(Main.A, Main.x) - Main.b)
Out[7]: 0.0
```

linear-algebra-01.py hosted with ❤ by GitHub view raw

We have chosen to define `A`, `b` and `x` in three different ways, to show the different syntaxes. The definition of **A** occurs completely inside the eval block. The variable `A` is created and is available inside the Julia scope, or as `Main.A`. The definition of `b` uses the `Main.b` access directly and uses the result of `Main.eval`. Finally, `%julia` is the magic IPython command to simply use Julia syntax directly.

We can quickly compare the timing of solving the system with Julia's backslash command and Numpy's `linalg.solve`:

```
In [8]: Main.eval('A = rand(1000, 1000)')
Out[8]:
array([[0.73190831, 0.54695026, 0.01693976, ..., 0.85186758, 0.06950126,
        0.98981994],
...

In [9]: Main.eval('b = A * ones(1000)')
Out[9]:
array([492.21711238, 504.74394262, 504.00070608, 501.81290299,
...

In [10]: %timeit Main.eval('x = A \ b')
11.7 ms ± 101 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [11]: %timeit np.linalg.solve(Main.A, Main.b)
```


Example: Automatic differentiation

The next example installs and uses the package called **ForwardDiff**, which performs automatic differentiation. ForwardDiff defines a Julia type called **Dual** internally, so we can't use it with Python functions because Python functions are not compatible with that type. However, we can define Julia functions and use them.

```
In [1]: from julia import Main
In [2]: from julia import Pkg
In [3]: Pkg.add('ForwardDiff')
    Updating registry at `~/.julia/registries/General`
   Resolving package versions...
...

In [4]: from julia import ForwardDiff
In [5]: f = Main.eval('x -> x^2 - 5x + 6')
In [6]: ForwardDiff.derivative(f, 2.5)
Out[6]: 0.0
```

forwarddiff-01.py hosted with ❤ by GitHub view raw

The local minimum of the quadratic occurs at 2.5, so the derivative at 2.5 is 0.0.

Another, more interesting interaction is below, in which we create a function g inside Julia, and define functions for its derivatives there. Then we create a Python function with the Taylor expansion around the value a. Furthermore, we use Matplotlib, Python's plotting library to visualize the results coming from Julia. Pretty neat, right?

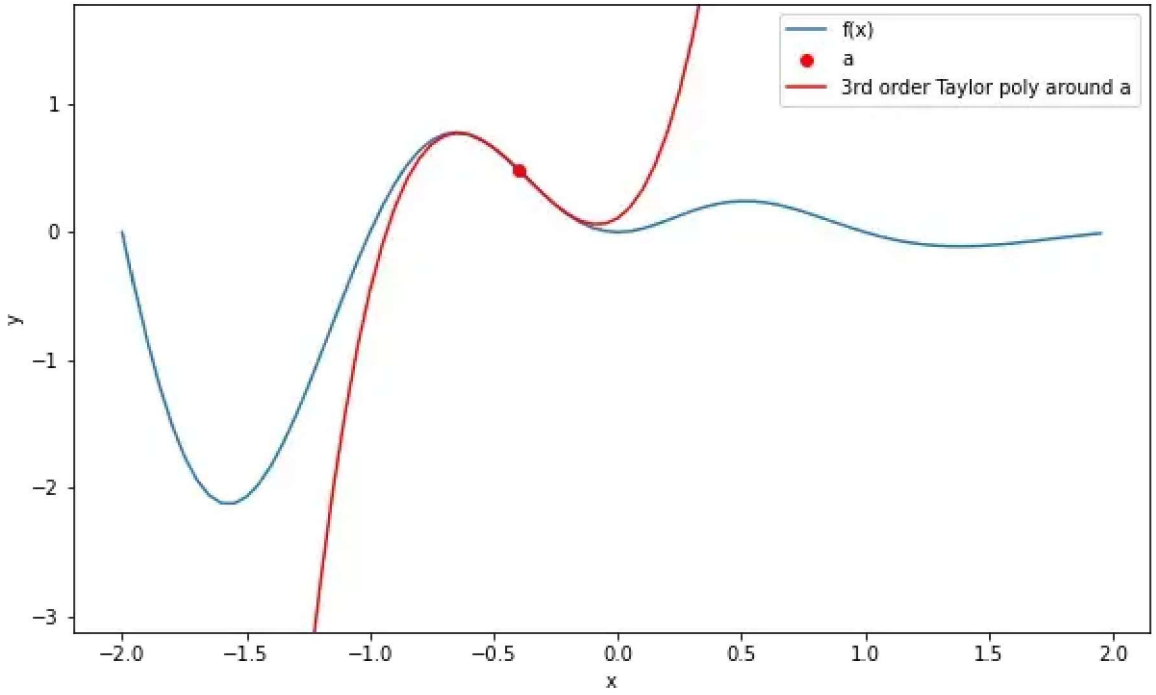


Image generated above, showing the function f and its third-order Taylor approximation.

Next episodes

Now that we can call Julia code in Python, we are prepared to move to our next adventure: improve the speed of a Python code by calling Julia from it. Follow our medium account to get notified when Part 2 goes live.

Many thanks to our proofreaders and reviewers, Elena Ranguelova, Jason Maassen, Jurrian Spaaks, Patrick Bos, Rob van Nieuwpoort, Stefan Verhoeven, and Veronica Pang.

Python Julia Performance Code Interoperability