Photo by Randy Tarampi (https://unsplash.com/@randytarampi) on Unsplash (https://unsplash.com/)

Interoperability in Julia

5 minute read

> ***Language interoperability*** *is the capability of two different programming languages to natively interact as part of the same system.*

> *~ From Wikipedia (https://en.wikipedia.org/wiki/Language_interoperability), the free encyclopedia*

In this lesson we will give you some hints on how to use several programming languages directly from within Julia.

You can find the code examples for this lesson here (https://github.com/aurelio-amerio/techytok-examples/blob/master/lesson-other-languages/other-languages.jl).

Python

We have already seen how to install and call Python from Julia in this lesson (https://techytok.com/lesson-interacting-with-python/). It is possible to call Julia code from Python using PyJulia (https://github.com/JuliaPy/pyjulia). In order to use some Julia code from Python, please first install and configure `PyCall`, as seen in the Python guide, and then install `PyJulia`, typing the following command in the Conda environment shell:

```
pip install julia
```

You can now setup the `julia` package in Python:

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

If the Julia executable is not in path, you will receive an error message. In order to let Python know where to find the Julia executable, please set an environment variable called `julia` which should contain the path to the Julia executable.

In order to set an environment variable in Windows 10, open the anaconda prompt and type:

```
set julia=C:/path/to/julia.exe
```

If you use Linux, type instead inside the Anaconda prompt:

```
export julia=/path/to/julia
```

Now, without closing the command prompt, open python and run again:

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

If everything is alright, this time you should be able to call Julia from Python:

```
from julia import Base
Base.sind(90)
```

You can find additional information on how to use `PyJulia` at the official repository (https://github.com/JuliaPy/pyjulia).

C++

It is possible to call C++ code and libraries from Julia using the `Cxx` package. In this lesson we will see only a brief introduction to its capability and I refer you to the official repository (https://github.com/JuliaInterop/Cxx.jl) for the documentation and more examples.

In order to install `Cxx` please type in the REPL:

```
using Pkg
Pkg.add("Cxx")
```

> At the time of writing, version 0.4.0 of `Cxx` has not been merged into the Julia registry yet, so you will have to type `Pkg.add("Cxx#master")` for it to work properly on Windows.

It is possible to create simple C++ functions and call them directly from Julia:

```
# include headers
using Cxx
cxx""" #include <iostream> """

# Declare the function
cxx"""
   void mycppfunction() {
      int z = 0;
      int y = 5;
      int x = 10;
      z = x*y + 2;
      std::cout << "The number is " << z << std::endl;
   }
"""

# Convert C++ to Julia function
>>>julia_function() = @cxx mycppfunction()
julia_function (generic function with 1 method)

# Run the function
>>>julia_function()
The number is 52
```

For more examples, please see the official repository page (https://github.com/JuliaInterop/Cxx.jl#using-cxxjl).

Wolfram Mathematica

In order to be able to call the Wolfram Language, you need to have installed an updated version of Mathematica (paid) or Wolfram Engine (https://www.wolfram.com/engine/) (free). Once you have logged in, you can proceed with the `MathLink` installation.

First we need to add some environmental variables:

fails, you will need to set the following environment variables:

- `JULIA_MATHKERNEL` : the path of the MathKernel executable

- `JULIA_MATHLINK` : the path of the MathLink dynamic library named

- `libML64i4.so` / `libML32i4.so` on Linux
    - `libML64.dll` / `libML32.dll` on Windows
    - `mathlink` on macOS

To add an environmental variable, type the following code in the Julia REPL:

```
ENV["JULIA_MATHKERNEL"]="/path/to/MathKernel/executable"
ENV["`JULIA_MATHLINK`"]="/path/to/libML64..."
```

Now we can install the `MathLink` package:

```
using Pkg
Pkg.add("MathLink")
```

And we can run Mathematica code in the following way:

```
using MathLink

>>>W"Sin"
W"Sin"

>>>sin1 = W"Sin"(1.0)
W"Sin(1.0)"

>>>sinx = W"Sin"(W"x")
W"Sin"(W"x")

>>>weval(sin1)
0.8414709848078965

>>>weval(sinx)
W"Sin"(W"x")

>>>weval(W"Integrate"(sinx, (W"x", 0, 1)))
W"Plus"(1, W"Times"(-1, W"Cos"(1)))
```

For more information, please take a look at the official repository (https://github.com/JuliaInterop/MathLink.jl).

MATLAB

It is possible to use functions from MATLAB using the `MATLAB` package. In order to install it you must follow a different procedure depending on the OS. The following instructions are taken from the official package repository (https://github.com/JuliaInterop/MATLAB.jl)

Installation

Important: The procedure to setup this package consists of the following steps.

By default, `MATLAB.jl` uses the MATLAB installation with the greatest version number. To specify that a specific MATLAB installation should be used, set the environment variable `MATLAB_HOME` .

Windows

1. Start a Command Prompt as an Administrator and enter `matlab /regserver` .
2. From Julia run: `Pkg.add("MATLAB")`

Linux

1. Make sure `matlab` is in executable path.

2. Make sure `csh` is installed. (Note: MATLAB for Linux relies on `csh` to open an engine session.)

   To install `csh` in Debian/Ubuntu/Linux Mint, you may type in the following command in terminal:

   ```
   sudo apt-get install csh
   ```

3. From Julia run: `Pkg.add("MATLAB")`

Mac OS X

1. Ensure that MATLAB is installed in `/Applications` (for example, if you are using MATLAB R2012b, you may add the following command to `.profile` : `export MATLAB_HOME=/Applications/MATLAB_R2012b.app`).
2. From Julia run: `Pkg.add("MATLAB")`

Usage

For the usage, please refer to the official documentation (https://github.com/JuliaInterop/MATLAB.jl#usage). For example, we can create a MATLAB variable in Julia and retrieve its content in this way:

```
using MATLAB

x = mxarray(Float64, 42)   # creates a 42-by-1 MATLAB zero array of double valued type

j = jarray(x) # converts x to a Julia array
```

R

In order to inter-operate with the **R language** we can use the `RCall` package. To install `RCall` please type:

```
using Pkg
Pkg.add("RCall")
```

RCall.jl will automatically install R for you using Conda (https://github.com/JuliaPy/Conda.jl) if it doesn't detect that you have R 3.4.0 or later installed already. For more information on how to install `RCall` and further customisation options, please refer to this (http://juliainterop.github.io/RCall.jl/stable/installation.html) documentation page.

You can access the R prompt from Julia (once you have loaded `RCall` via `using RCall`) by typing `$` in the REPL.

Furthermore you can transfer data from Julia to R and vice versa using the `@rput` and `@rget` macros:

```
using RCall

julia> z = 1
1

julia> @rput z
1

R> z
[1] 1

R> r = 2

julia> @rget r
2.0

julia> r
2.0
```

For more information, please take a look at the official getting started (http://juliainterop.github.io/RCall.jl/stable/gettingstarted) page.

FORTRAN

Although it is not as straight forward as with other languages, it is possible to call compiled FORTRAN libraries using the `ccall` function. The following example, taken from the official documentation (https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/index.html#Fortran-Wrapper-Example-1), utilises `ccall` to call a function in a common FORTRAN library (libBLAS) to computes a dot product.

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    @assert length(DX) == length(DY)
    n = length(DX)
    incx = incy = 1
    product = ccall((:ddot_, "libLAPACK"),
                    Float64,
                    (Ref{Int32}, Ptr{Float64}, Ref{Int32}, Ptr{Float64}, Ref{Int32}),
                    n, DX, incx, DY, incy)
    return product
end
```

If you are interested in calling FORTRAN from Julia, please take a look at this Julia wrapper (https://github.com/JuliaLinearAlgebra/Arpack.jl) of the arpack (https://github.com/opencollab/arpack-ng/) library.

Other languages

Please take a look at JuliaInterop (https://github.com/JuliaInterop) for more packages to integrate Julia with different languages or frameworks.

Conclusions

In this lesson we have seen how it is possible to call some programming languages from Julia. If you are interested in any of them, I advise you to take a look at the official documentation for such packages for more information and examples.

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

Tags: Julia

Categories: Lessons Tutorial

Updated: February 7, 2020

LEAVE A COMMENT

ALSO ON TECHYTOK

Code optimisation in Julia

9 months ago • 7 comments

Tips and Tricks to optimise your Julia code and achieve the maximum possible …

Variable Scope

7 months ago • 4 comments

From zero to Julia Lesson 6. Variable scope

Parallel computin

7 months ago • 4 comm

From zero to Julia Le 15. Parallel computin

0 Comments **techytok** **Privacy Policy** 1 **Login**

♡ **Recommend** Tweet Share **Sort by Best**

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

✉ **Subscribe** **Add Disqus to your siteAdd DisqusAdd** ⚠ **Do Not Sell My Data**