



# SKILLPILLS

## Skill Pill: Julia

### Lecture 1: Introduction

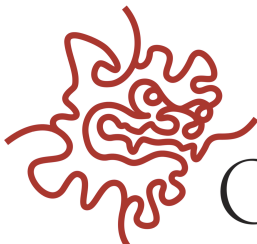
James Schloss    Valentin Churavy

Okinawa Institute of Science and Technology

*[james.schloss@oist.jp](mailto:james.schloss@oist.jp)*

*[valentin.churavy@oist.jp](mailto:valentin.churavy@oist.jp)*

July 4, 2017



OIST

dawn of time

0.1

0.2

0.3

0.4

0.5

## Windows, Linux, and Mac OSX

Download a precompiled version of 0.6 from  
<https://julialang.org/downloads/>

## Linux and Mac OSX

- 1 Use your package manager (Mac OSX: `brew cask install julia`)
- 2 Follow the build instructions from  
<https://github.com/JuliaLang/julia/>

## Sango and Tombo

OIST has Julia installed on Sango and Tombo, in case the version you need is not there let [it-help@oist.jp](mailto:it-help@oist.jp) know.

**Documentation** <https://docs.julialang.org/en/release-0.6/>

**Forum** <https://discourse.julialang.org>

**Issue Tracker** <https://github.com/JuliaLang/julia>

**Downloads** <https://julialang.org/downloads/>

**Packages** <https://pkg.julialang.org/>  
<https://juliaobserver.com/>

Statement: Scientist like high-level programming languages - Why Julia thread on discourse - old blogposts

The typical languages used in science are

- 1 Python
- 2 Matlab
- 3 R

Once a problem is becoming too big we usually move to

- 1 C/C++
- 2 Fortran

This is called the 2+ language problem and Julia is trying to solve that.

- Find that slide

Hard to optimize, JIT limited, GIL, fast code needs to be written in C  
Numpy gets in the way when writing scientific code



Only fast for the subst of operation mathworks deemed important Cost

Slow, don't even try to do numerics

Find slide

## The Read-Eval-Print-Loop

The REPL is a command-line interface to Julia and is ideal for short experiments.

```
      _          _ _(_) _      | A fresh approach to technical computing
    (_)          | (_) ( )      | Documentation: https://docs.julialang.org
      _ _        _| | _ _ _ _   | Type "?help" for help.
    | | | | | | | | / _ ' |      |
    | | | _| | | | ( _| |      | Version 0.6.0 (2017-06-19 13:05 UTC)
  _/ | \_ _ ' _| | | \_ _ ' _| |
| _ _/                          | x86_64-pc-linux-gnu
```

julia>

Juno/Atom VStudioCode

Jupyter is an interactive web-based client for Python, Julia, R and many other languages. It offers a programming environment that is well suited for explorative data analysis or prototyping.

## Installation

```
julia> ENV["JUPYTER"] = ""  
julia> Pkg.add("IJulia")
```

## Starting a Jupyter session

```
julia> using IJulia  
julia> notebook()
```

## JuliaBox

There is an online service provided by JuliaComputing at <https://juliabox.com> that gives you a cloud version of Jupyter.

Julia is a dynamic language and so you can simply create variables in any scope.

```
x = 1    # x will be of type Int64
y = 1.0  # y will be of type Float64
z = 1.0 - 2.0im # z will be an Complex{Float64}
1//2
""
', '
6.1e6
```











# Multiple dispatch in a nutshell









Julia allows you to use other languages (such as Fortran or C) by using the `ccall` function:

```
julia> t = ccall(:clock, "libc"), Int32, ())  
2292761
```

Here, we are calling the `clock` function from the `libc` library in C.



Let's say you want to use a simply multiply function in Fortran:

```
!! We'll be using subroutines instead of functions
subroutine multiply(A, B, C)
    REAL*8 :: A, B, C
    C = A * B
    return
end
```

or C:

```
// Nothing fancy here...
double multiply(double A, double B){
    return A*B;
}
```

In order to use your favorite C or Fortran code in Julia, you need to compile it into a library, like so:

```
gcc -shared -O2 multiply.c -fPIC -o c_multiply.so
gfortran -shared -O2 multiply.f90 -fPIC -o
    fortran_multiply.so
```

In order to use your favorite C or Fortran code in Julia, you need to compile it into a library, like so:

```
gcc -shared -O2 multiply.c -fPIC -o c_multiply.so
gfortran -shared -O2 multiply.f90 -fPIC -o
    fortran_multiply.so
```

These will create libraries with all of the necessary functions you could want, but beware:

**C and Fortran compilers mangle function names!**

There are 3 things to keep. Make sure you

- ① Have the right mangled name
- ② Are using the right type
- ③ Are using the function correctly.

There are 3 things to keep. Make sure you

- ❶ Have the right mangled name
- ❷ Are using the right type
- ❸ Are using the function correctly.

For example, in C:

```
# This function multiplies a and b into c by using the
# created C library
function call_c()
    a = Cdouble(1.0)
    b = Cdouble(3.0)
    c = ccall((:multiply, "/full/path/to/c_multiply"),
              Cdouble, (Cdouble, Cdouble), a, b)
    println(c)
end
```

Pointers are okay! For example, in Fortran:

```
# This function multiplies a and b into c by using
# the created FORTRAN library
function call_fortran()
    a = Cdouble[1.0]
    b = Cdouble[2.0]
    c = Cdouble[0.0]
    ccall((:multiply_, "/full/path/to/fortran_multiply"),
        Void, (Ptr{Float64},Ptr{Float64},Ptr{Float64}),
            a,b,c)
    println(c[1])
end
```

Pointers are okay! For example, in Fortran:

```
# This function multiplies a and b into c by using
# the created FORTRAN library
function call_fortran()
    a = Cdouble[1.0]
    b = Cdouble[2.0]
    c = Cdouble[0.0]
    ccall((:multiply_, "/full/path/to/fortran_multiply"),
        Void, (Ptr{Float64}, Ptr{Float64}, Ptr{Float64}),
            a, b, c)
    println(c[1])
end
```

More information can be found here: <https://docs.julialang.org/en/stable/manual/calling-c-and-fortran-code/>