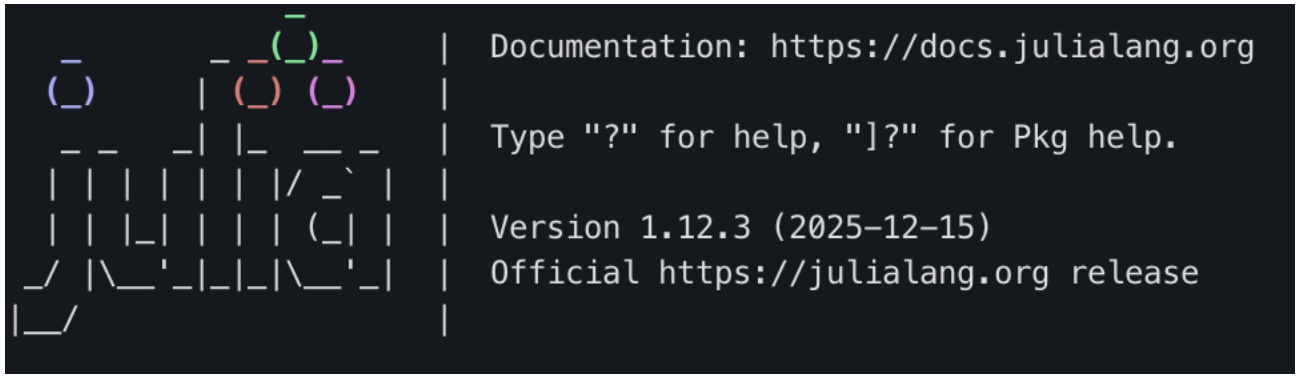
Installing and running Julia

- Download Julia
  - Free and Open Source
  - **https://julialang.org/downloads/**
  - v1.12.3 – the latest stable version
- Programming environment – VS Code
  - https://code.visualstudio.com/download/
- Jupyter notebook
  - Available via IJulia package

Julia Command Line (REPL)

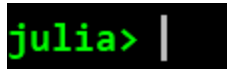


pressing] changes REPL to package installation mode

pressing ; changes REPL to shell mode

pressing ? changes REPL to help mode

to go back to normal mode press **BACKSPACE**



Adding Julia packages

- Start Julia REPL

- Press `]` to start the Julia package manager
  (prompt `(v1.12) pkg>` will be seen)

- Sample package installation command

```
(v1.12) pkg> add DataFrames Distributions
```

to go back to normal mode press **BACKSPACE**

Managing packages (press] for the package management in REPL mode)

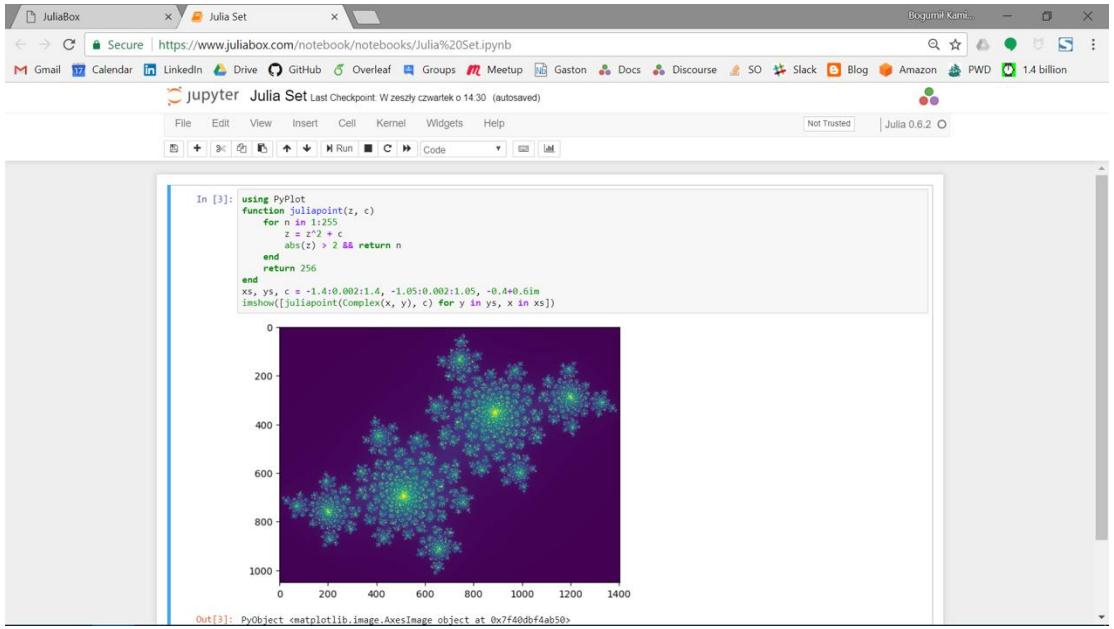
```
(@v1.12) pkg> st
Status `~/.julia/environments/v1.12/Project.to
  [992eb4ea] CondaPkg v0.2.33
  [a93c6f00] DataFrames v1.8.1
  [6099a3de] PythonCall v0.9.31
  [6f49c342] RCall v0.14.10
```

```
(@v1.12) pkg> add RCall
   Resolving package versions...
    Updating `~/.julia/environments/v1.12/Project.toml`
  [6f49c342] + RCall v0.14.10
    Updating `~/.julia/environments/v1.12/Manifest.toml`
  [66dad0bd] + AliasTables v1.1.3
  [324d7699] + CategoricalArrays v1.0.2
  [8f4d0f93] + Conda v1.10.3
  [ffbed154] + DocStringExtensions v0.9.5
  [34004b35] + HypergeometricFunctions v0.3.28
  [92d709cd] + IrrationalConstants v0.2.6
  [692b3bcd] + JLLWrappers v1.7.1
  [682c06a0] + JSON v1.3.0
  [2ab3a3ac] + LogExpFunctions v0.3.29
  [69de0a69] + Parsers v2.8.3
  [43287f4e] + PtrArrays v1.3.0
  [6f49c342] + RCall v0.14.10
  [ae029012] + Requires v1.3.1
  [79098fc4] + Rmath v0.9.0
  [1277b4bf] + ShiftedArrays v2.0.0
  [276daf66] + SpecialFunctions v2.6.1
  [82ae8749] + StatsAPI v1.8.0
  [2913bbd2] + StatsBase v0.34.9
  [4c63d2b9] + StatsFuns v1.5.2
  [3eaba693] + StatsModels v0.7.7
  [ec057cc2] + StructUtils v2.6.0
  [81def892] + VersionParsing v1.3.0
  [1b915085] + WinReg v1.0.0
  [efe28fd5] + OpenSpecFun_jll v0.5.6+0
```

Jupyter notebook

- Jupyter notebook
  - `using Pkg; Pkg.add("IJulia")`
  - `using IJulia`
  - `notebook(dir=".")`

  - Press Ctrl+C to exit



Julia 10,000 feet overview

- Exponential growth, in several areas became a standard for scientific and high performance computing
- “walks like Python runs like C”
- Syntax in-between Python/numpy and Matlab
- Compiles to assembly
- Compiles to GPU
- Distributed computing built into the language

Why another language for data science?

Two language problem of data science – programming languages

- are either fast (C++, Fortran)
- or are convenient (Python, R, Matlab)

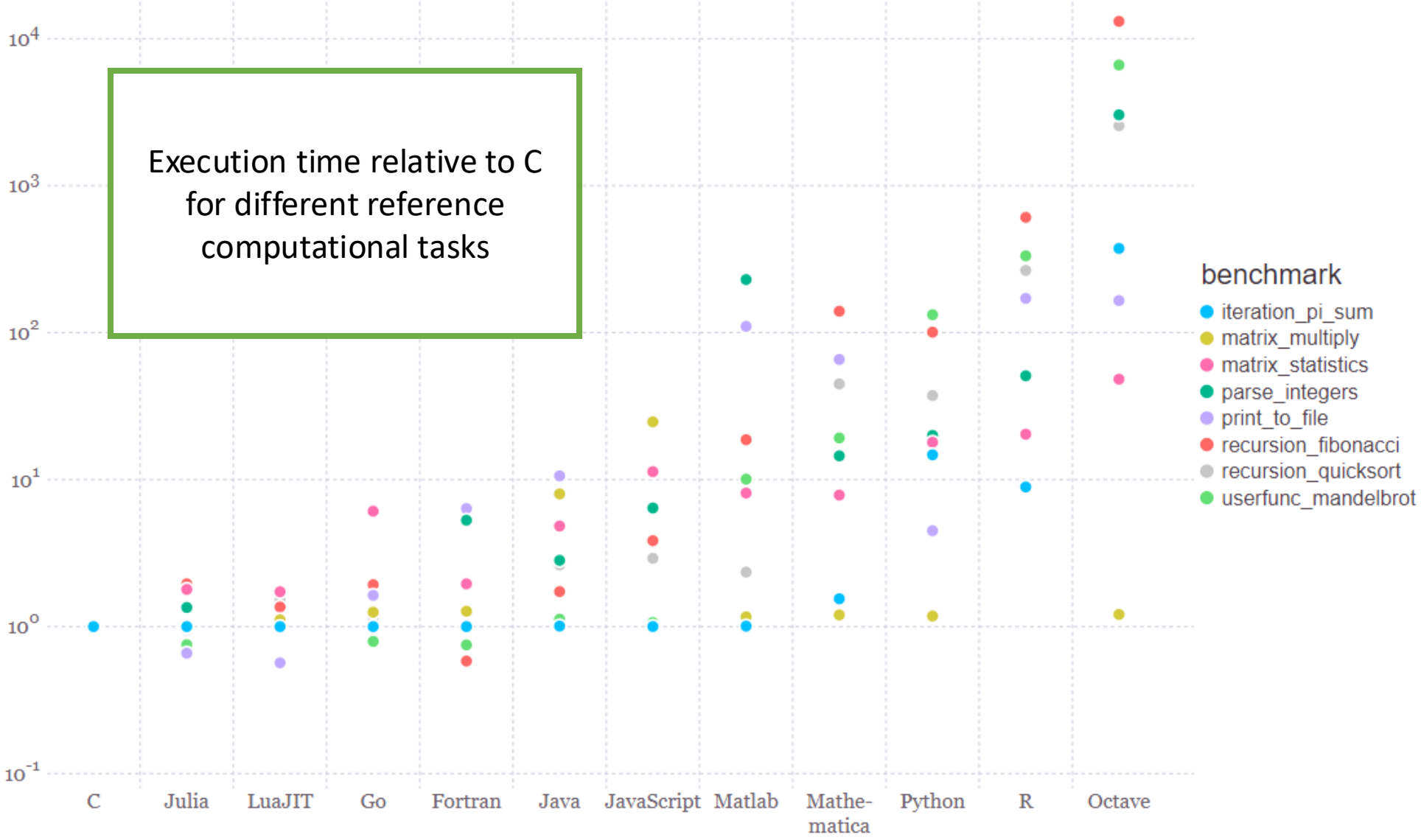
Main features of Julia

1. Efficiency
2. Expressiveness
3. Metaprogramming – DSLs for various data science subproblems
4. Integrations

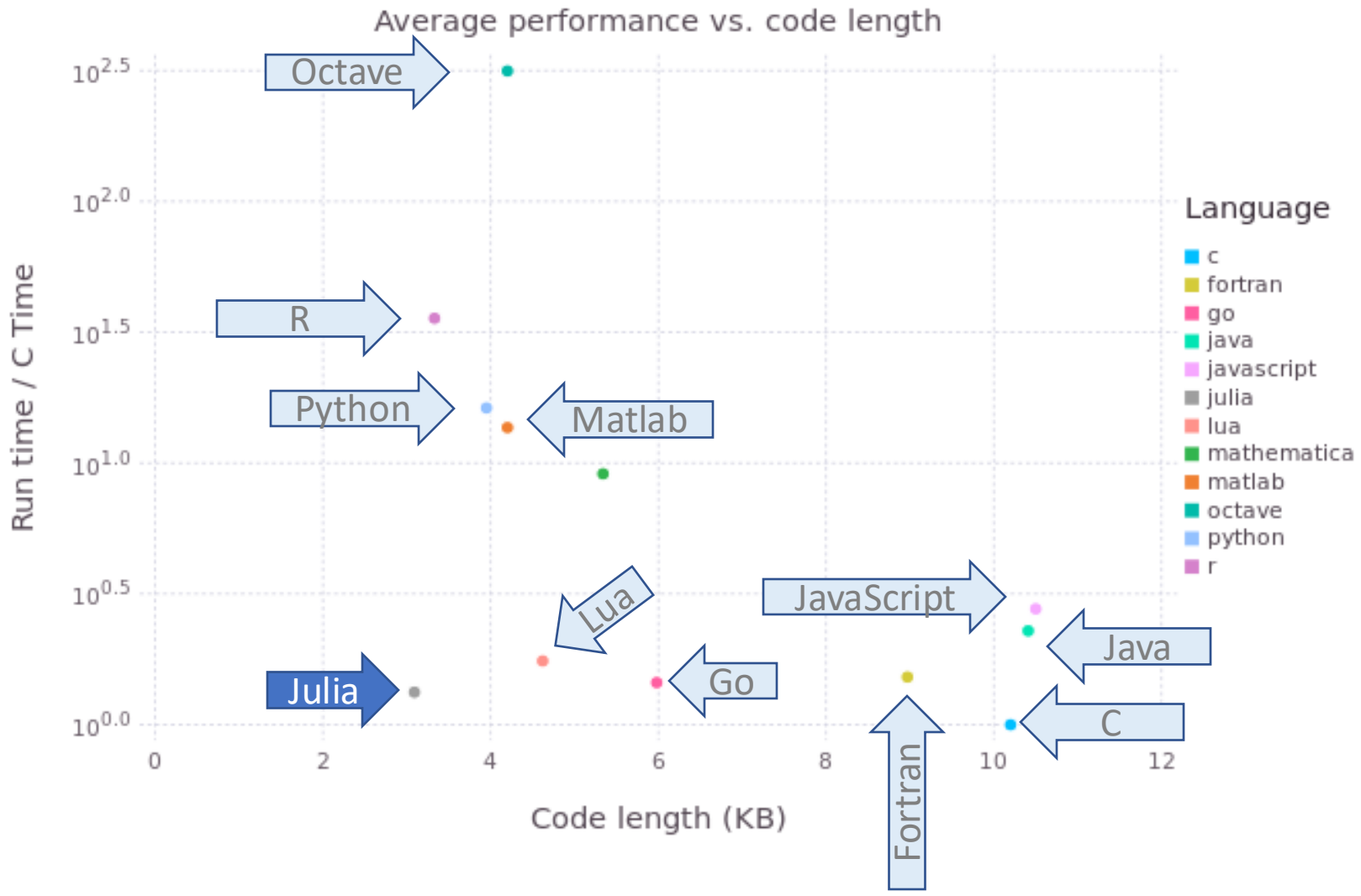
Methods of achieving high performance in different data science environments

Ecosystem	Glue	Hot code	GPU
R-based	**R**	**RCpp**	**C**
Python-based	**Python**	**Numba/Cython/C**	**C**
Julia-based	**Julia**	**Julia**	**Julia**
Matlab-based	**Matlab**	**C**	**GPU coder**

Reference benchmarks from Julia website



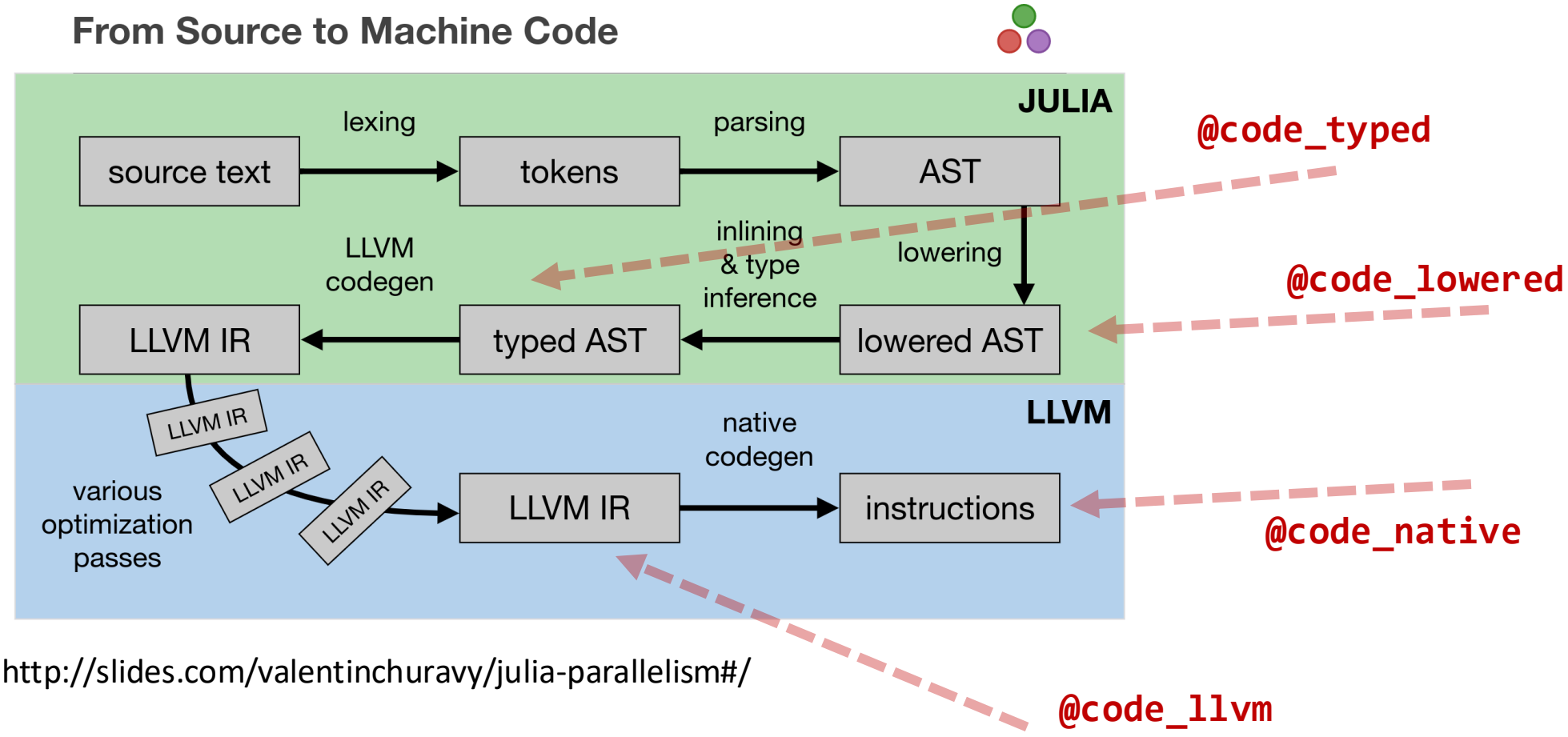
Language Code Complexity vs Execution Speed



Key features

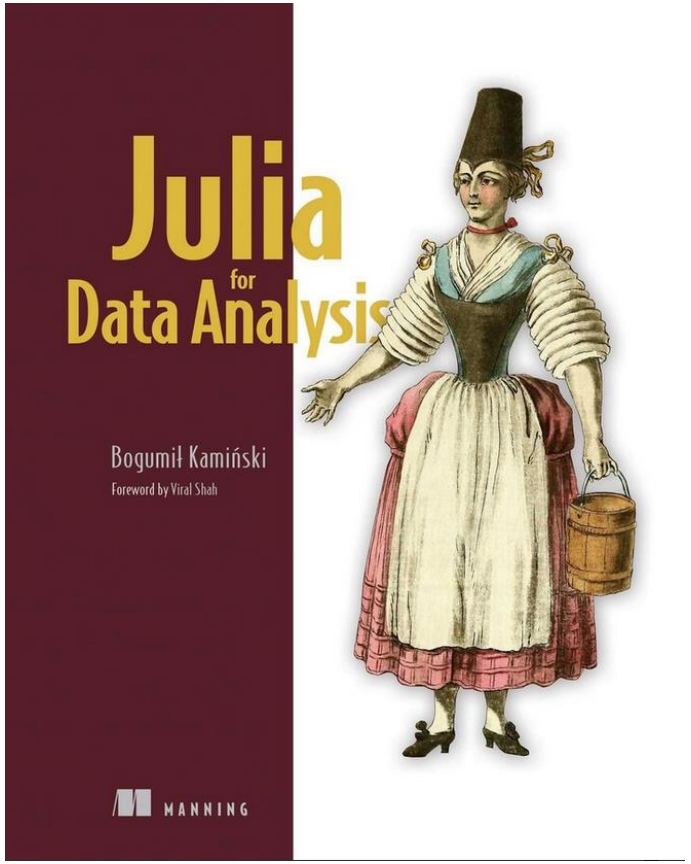
- Performance
  - Dynamically compiled to optimized native machine code
- Scalability
  - SIMD, Threading, Distributed computing
- Modern design of the language
  - multiple dispatch, metaprogramming, dynamic type system
- MIT License
  - corporate-use friendly (also package ecosystem)

Julia code compilation process



http://slides.com/valentinchuravy/julia-parallelism#/

Learning more about Julia



- Website: https://julialang.org/
- Learning materials: https://julialang.org/learning/
- Blogs about Julia: https://www.juliabloggers.com/
- Videos: https://juliaacademy.com/
- Julia forum: https://discourse.julialang.org/
- Q&A for Julia: https://stackoverflow.com/questions/tagged/julia-lang

Julia installation and virtual environment

`JULIA_HOME` *(system environment variable)*

- Julia binaries
- julia.exe
- julia system image

`JULIA_DEPOT_PATH` *(system env variable)*

(defaults to `~/.julia`)

- packages
  (multiple versions)
- precompiled files
  (multiple versions)
- artifacts
  (multiple versions)
- default virtual environment

- `package1`
  - `v1.2.0`
  - `v1.4.0`

Virtual environment is just a folder!

```
Pkg.activate() # default in-built
Pkg.activate("/some/path/")
Pkg.status()
```

`Project.toml` *(file)*

- List of packages
- Ranges of package versions

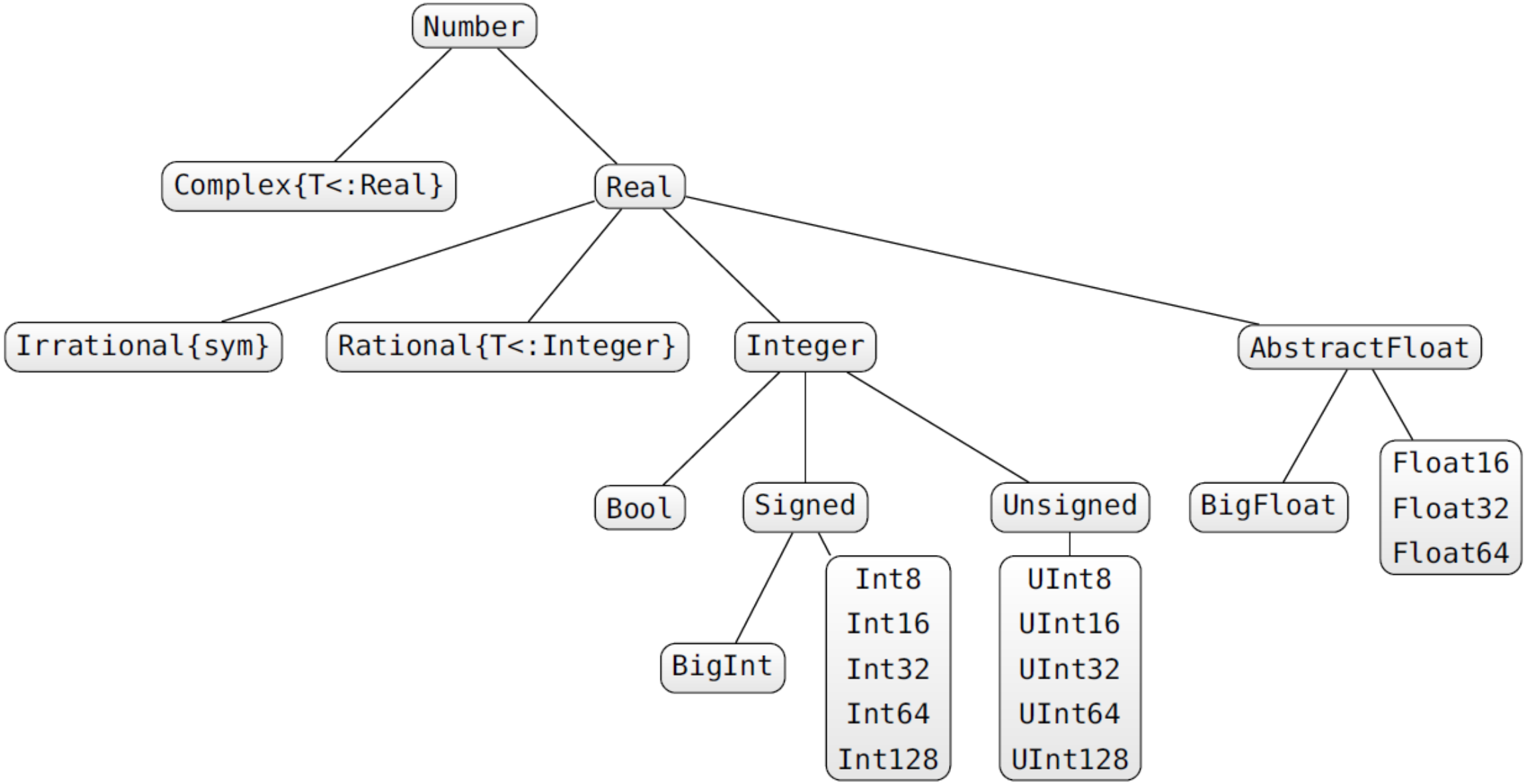
`Manifest.toml` *(file)*

- Links to defined versions of packages
- Resolved dependencies
- Use to exactly replicate venv

Some basic commands

```
@less(max(1,2))
     # show function source code

cd("D:/") # change working directory to D:/
cd(raw"c:\temp")

pwd() # current directory

include("file.jl") # run file

exit() # end your Julia session
```

Numeric type hierarchy



Type conversion functions

- `Int64('a')` *# character to integer*
- `Int64(2.0)` *# float to integer*
- `Int64(1.3)` *# inexact error*
- `Int64("a")` *# error no conversion possible*
- `Float64(1)` *# integer to float*
- `Bool(1)` *# converts to boolean true*
- `Bool(0)` *# converts to boolean false*
- `Char(89)` *# integer to char*
- `zero(10.0)` *# zero of type of 10.0*
- `one(Int64)` *# one of type Int64*
- `convert(Int64, 1.0)` *# convert float to integer*
- `parse(Int64, "1")` *# parse "1" string as Int64*

Special types

- `Any # all objects are of this type`
- `Union{} # subtype of all types, no object can have this type`
- `Nothing # type indicating nothing, subtype of Any`
- `nothing # only instance of Nothing`

Tuples – just like in Python

```
• ()                # empty tuple
• (1,)              # one element tuple
• ("a", 1)          # two element tuple
• ('a', false)::Tuple{Char, Bool} # tuple type assertion
• x = (1, 2, 3)
• x[1]              # first element
• x[1:2]            # (1, 2) (tuple)
• x[4]              # bounds error
• x[1] = 1          # error - tuple is not mutable
• a, b = x          # tuple unpacking a==1, b==2
```

Tuples are immutable, and the Julia compiler makes a good use of that!

Arrays

```
Array{Char}(undef, 2, 3, 4)      # 2x3x4 array of Chars
Array{Any}(undef, 2, 3)          # 2x3 array of Any
zeros(5)                      # vector of Float64 zeros
ones(Int64, 2, 1)             # 2x1 array of Int64 ones
trues(3), falses(3)           # tuple of vector of trues and of falses

x = range(1, stop=2, length=5)
  # iterator having 5 equally spaced elements (1.0:0.25:2.0)
collect(x)                    # converts iterator to vector
1:10                          # iterable from 1 to 10
1:2:10                        # iterable from 1 to 9 with 2 skip
reshape(1:12, 3, 4)           # 3x4 array filled with 1:12 values
```

Data structures

```
mutable struct Point
      x::Int64
      y::Float64
      meta
end
p = Point(0, 0.0, "Origin")
println(p.x)                        # access field
p.meta = 2                      # change field value
fieldnames(typeof(p))     # get names of instance fields
fieldnames(Point)           # get names of type fields
```

Julia is not object oriented language – multiple dispatch is used instead

Default values require a macro

```
Base.@kwdef struct A
    a::Int = 6
    b::Float64 = -1.1
    c::UInt8 = 1
end
A()
A(a=2, c=4)
```

Dictionaries

```
x = Dict{Int, Float64}()
      # empty dictionary (types for keys and values are defined)
y = Dict(1=>5.5, 2=>4.5)          # dictionary
y[2]                              # return element
y[3] = 30.0                       # add element

keys(y), values(y)                # iterators
haskey(y)
```

Texts and interpolations

```
"Hi " * "there!"           # concatenation
string("a= ", 123.3)       # concatenation

x = 123
"$x + 3 = $(x+3)"          # $ is used for interpolation
"\$199"                    # and needs to be escaped with a `\`

occursin("CD", "ABCD")     # occurrence
occursin(r"A|B", "ABCD")  # occurrence with RegExp
```

Functions

```
f(x, y = 10) = x + y
                # default value for y is 10


function g(x::Int, y::Int)     # ograniczenie typu
     return y, x # yields a tuple
end

g(x::Int, y::Bool) = x * y     # multiple dispatch
g(2, true)                     # 2nd definition will be called
methods(g)                     # list of methods for g
```

Operators

```
true || false    # binary or operator (singeltons only)
1 < 2 < 3        # condition chaining
[1 2] .< [2 1]   # vectorization with a dot "."
a = 5
2a + 2(a+1) # multiplication “*” can be ommited

x = [1 2 3]         #matrix 1×3 Array{Int64,2}
y = [1, 2, 3]      #vector of 3-elements Array{Int64,1}
# Vectors are vertical and algebra rules apply
x + y # error
x .+ y # 3x3 matrix, dimension broadcasting
x + y' # 1x3 matrix
x * y # array multiplication, 1-element vector (not scalar)
```

BigFloat

```
julia> our_pi(1000, BigFloat)-π
1.0363402266113333550463622235360479485339200437323537662028444164202
31e-76

julia> setprecision(1000) do
    our_pi(1000, BigFloat)-π
end
3.7330544740128755159603581788952686784683657854868320984868573591838
67643903102537817761308391524409438379959721296970496861950085416129
57936608326881572302493764266455330060109598030394360732604440196318
50604524729620500591837351632207130845016604152427935154177059244778
7925691464383688807065164177119e-301
```

Rational numbers

```
julia> [our_pi(n, Rational) for n in 1:10]
10-element Array{Rational{Int64},1}:
8//3
44//15 64//21
976//315
10816//3465
141088//45045
47104//15015
2404096//765765
45693952//14549535
45701632//14549535
```

Julia IO – writing files

- In Julia the open command can be used to read and write to a particular file stream.

```
julia> f = open("some_name.txt","w")
IOStream(<file some_name.txt>)
```

- The write command takes a stream handle as the first parameter accepts a wide range of additional parameters.

```
write(f, "first line\nsecond line\n")
```

- Close the stream

```
close(f)
```

Julia IO – reading files

```
f = open("some_name.txt")
```

In order to read a single line from a file use the readline function.

```
julia> readline(f)
"first line"
julia> readline(f)
"second line“
julia> eof(f)
true
julia> close(f)
```