The Julia Programming Language

First steps: Introduction, Installation and Examples

using Julia v1.5.3

Ricardo A. Fernandes

ricardoaf@lccv.ufal.br

Advisor: Adeildo S. Ramos Jr.

January, 2021

Introduction

Julia (Programming Language)

- Homepage (download, docs, community)
  - https://julialang.org
- Open source (MIT license)
  - https://github.com/JuliaLang/julia
- First appeared: 2012
- Current stable release: 1.5.3 (November 9, 2020)
- Multiple Platforms: Windows, Linux, macOS, FreeBSD

History

- 2009: Work started (create a free, high-level and fast language)
- 2012: Julia's launch (website julialang)
- 2014-2020: JuliaCon (academic conference for Julia users and developers)
- 2018: Release candidate for Julia 1.0
- 2020: Julia 1.5 release (significant improvements: debugging, stability and performance)

Numbers since launch

- +20M downloads (at more than 10k companies)
- Used at more than 1500 universities
- 2020 JuliaCon with +28k unique viewers

Notable uses and prizes

- Risk calculations using time-series analytics from investment manager BlackRock
- Models of US economy 10x faster than MATLAB from Federal Reserve Bank of New York
- Three of Julia co-creators received 2019 James H. Wilkinson Prize for Numerical Software
- Alan Edelman, professor of applied mathematics at MIT, received 2019 IEEE Computer Society Sidney Fernbach Award for outstanding breakthroughs in HPC, linear algebra, computational science and for contributions to Julia
- Space mission planning and satellite simulation by NASA and Brazilian INPE

Introduction

"Why we created Julia"

https://julialang.org/blog/2012/02/why-we-created-julia/

- A quote from the creators of Julia from their first official blog article

  "We want a language that's **open source**, with a liberal license. We want the **speed of C** with the **dynamism of Ruby**.

  We want a language that's **homoiconic, with true macros like Lisp**, but with **obvious, familiar mathematical notation like Matlab**.

  We want something as **usable for general programming as Python**, as **easy for statistics as R**, as **natural for string processing as Perl**, as **powerful for linear algebra as Matlab**, as **good at gluing programs together as the shell**.

  Something that is **dirt simple to learn, yet keeps the most serious hackers happy**.

  We want it interactive and we want it compiled.

  (Did we mention it should be as fast as C?)"

2017 article (Julia official reference for citations)

Julia: A Fresh Approach to Numerical Computing (2017) SIAM Review, 59: 65–98.

https://doi.org/10.1038/d41586-019-02310-3


SIAM REVIEW
Vol. 59, No. 1, pp. 65–98

© 2017 Society for Industrial and Applied Mathematics

Julia: A Fresh Approach to Numerical Computing*

Jeff Bezanson†
Alan Edelman‡
Stefan Karpinski§
Viral B. Shah†

2019 article from Nature

Nature 572, 141-142 (2019)

https://doi.org/10.1038/d41586-019-02310-3


TOOLBOX

JULIA: COME FOR THE SYNTAX, STAY FOR THE SPEED

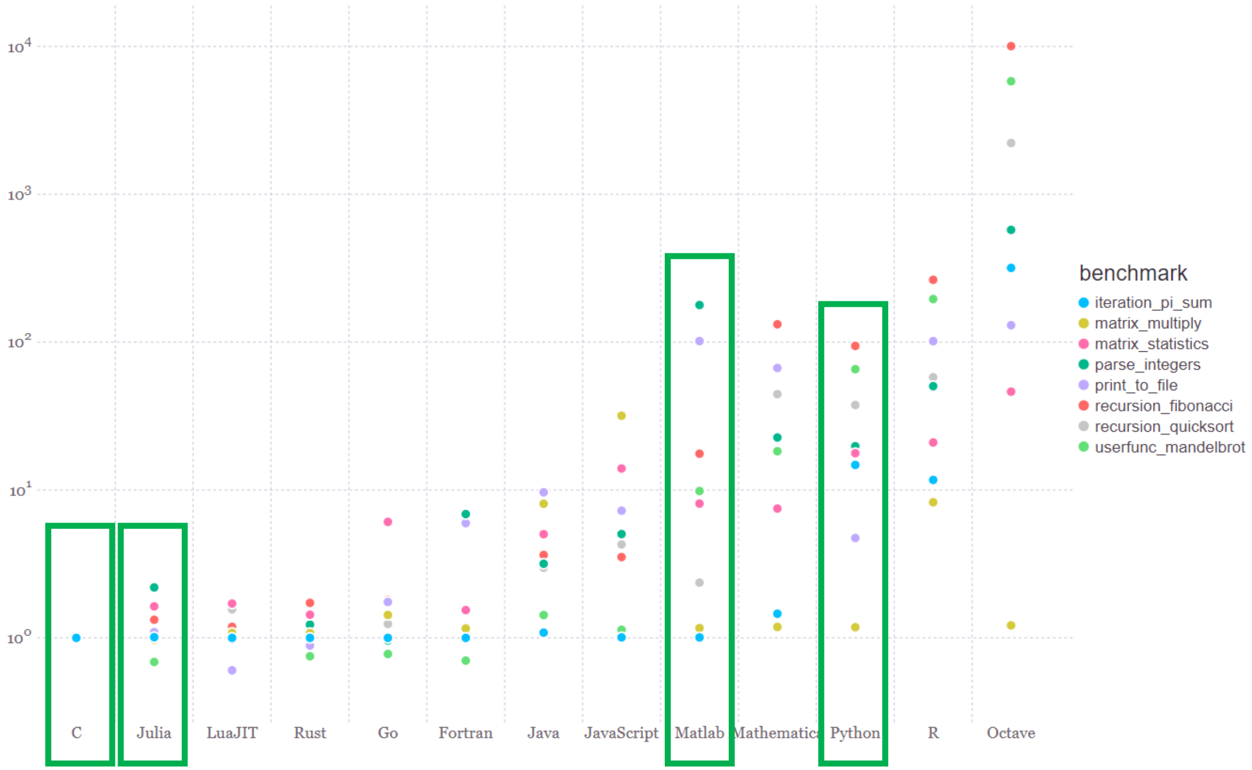
Researchers often find themselves coding algorithms in one programming language, only to have to rewrite them in a faster one. An up-and-coming language could be the answer.

Introduction

Julia Features

- High level
- Dynamic programming language
- *High performance*
- Well suited for numerical analysis and computational science
- Parametric polymorphism (multiple dispatch)
- Supports parallel and distributed computing
- Uses a just-in-time (JIT) compiler
- Garbage-collection (GC)
- A built-in package manager
- Includes efficient libraries
  - Linear algebra, Statistics, Optimization, Machine learning (ML), Plots
- Integrated development environments (IDE) for coding
  - Microsoft Visual Studio Code, Juno/Atom, Jupyter
- Extensions for code debugging and profile

https://julialang.org/benchmarks/



Introduction

Adoption in 2020

Total cumulative numbers from Jan 1, 2020 to Jan 1, 2021

Source: newsletter@juliacomputing.com

- **Number of downloads** (JuliaLang + Docker + JuliaPro)
  - 12,950,630 → **24,205,141 (↑87%)**
- **Number of Packages**
  - 2,787 → **4,809 (↑73%)**
- **GitHub stars**
  - 99,830 → **161,774 (↑62%)**
- **YouTube views** (youtube.com/user/JuliaLanguage)
  - 1,562,223 → **3,320,915 (↑113%)**
- **Published citations** (2012 & 2017 official papers)
  - 1,680 → **2,531 (↑51%)**

TIOBE Index Rank

Julia Rises from #47 to #23 in TIOBE Index

https://www.tiobe.com/tiobe-index/

According to TIOBE CEO Paul Jansen

"The top candidate [to break into the top 20 in 2021] is without doubt Julia, which jumped from position 47 to position 23 in the last 12 months."

Among languages developed on **GitHub**, Julia ranks

- #7 in stars
- #9 in forks

Julia also ranks

- #24 in the **PYPL Index**
  - https://pypl.github.io/PYPL.html
- #19 in the **IEEE Spectrum ranking**
  - https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020

Introduction

Julia in the Classroom

https://julialang.org/learning/classes/

Installation

Installing Julia in Windows

Step #1

- Select and appropriate version and download Julia from https://julialang.org/downloads/

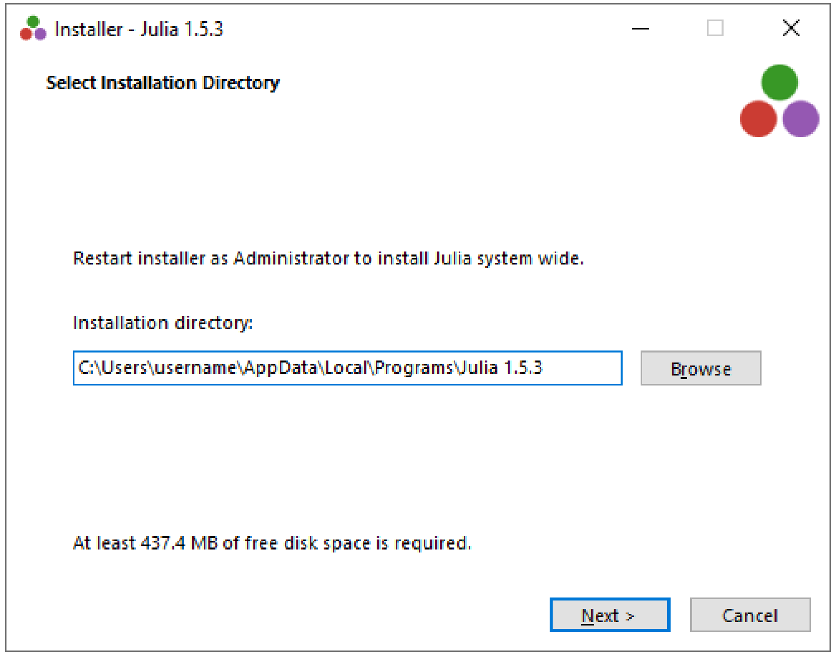
Current stable release: v1.5.3 (Nov 9, 2020)

Checksums for this release are available in both MD5 and SHA256 formats.

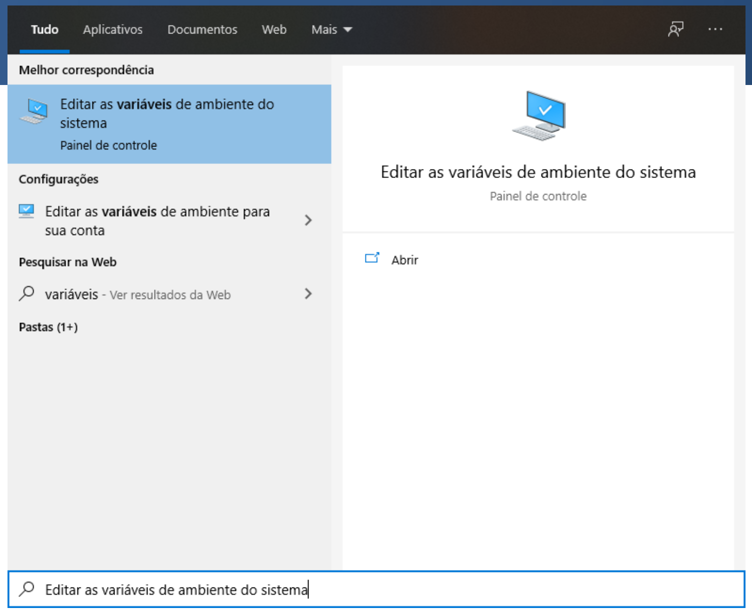
Windows [help]	64-bit (installer), 64-bit (portable)	32-bit (installer), 32-bit (portable)	
macOS [help]	64-bit		
Generic Linux on x86 [help]	64-bit (GPG), 64-bit (musl)[1] (GPG)	32-bit (GPG)	
Generic Linux on ARM [help]	64-bit (AArch64) (GPG)		
Generic FreeBSD on x86 [help]	64-bit (GPG)		
Source	Tarball (GPG)	Tarball with dependencies (GPG)	GitHub

Step #2

- Install Julia



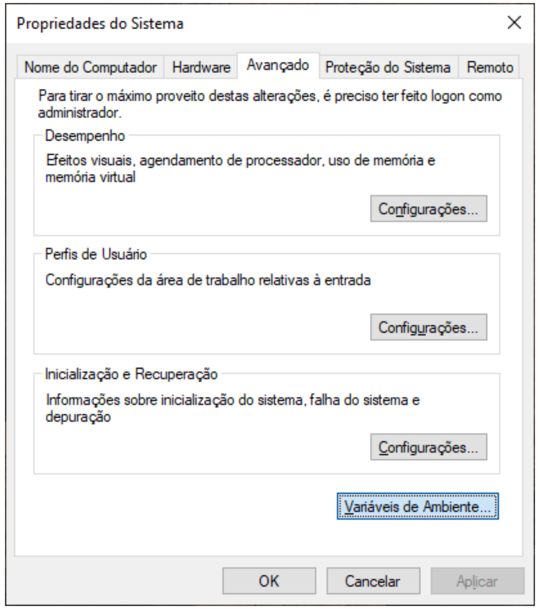
Installation

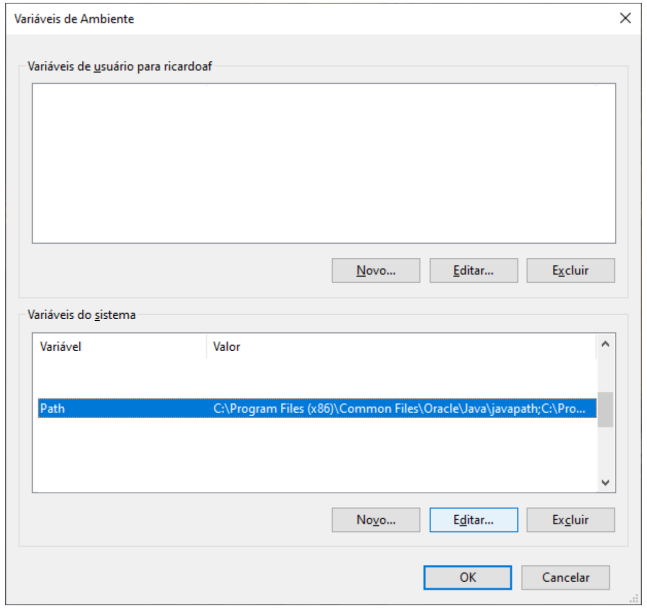


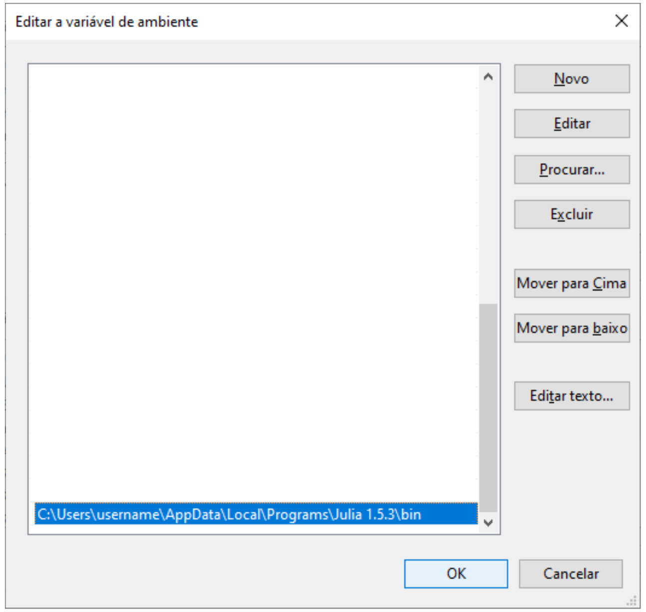
Installing Julia in Windows

Step #3

- Add Julia directory (appended with \bin) into System Path
- Must be consistent with the installation directory in step #2





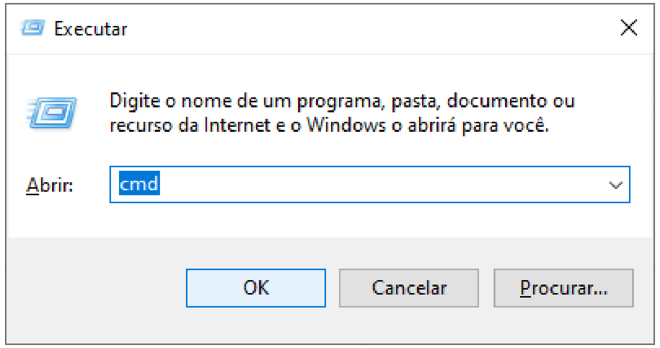


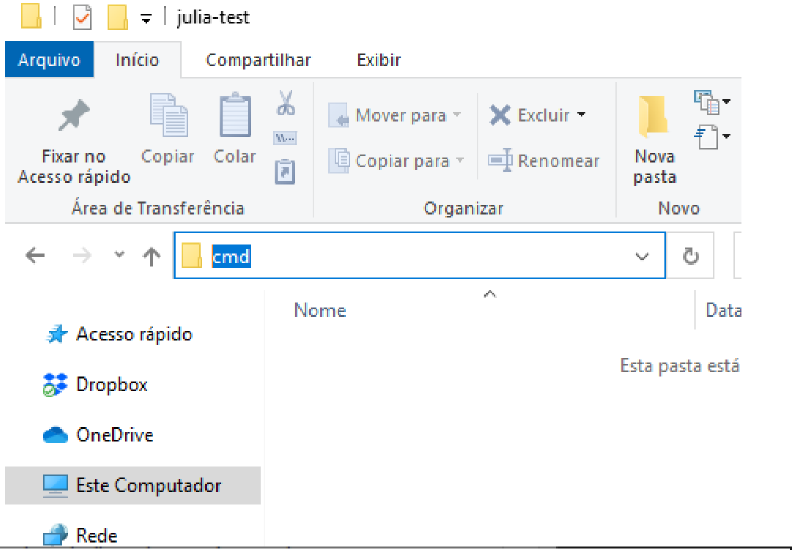
Installation

Installing Julia in Windows

Step #4

- Open Command Prompt
  - Press **Win key + r** and type **cmd**
  - **Or** go to **any folder** in explorer and type **cmd**
    - It will be the current directory

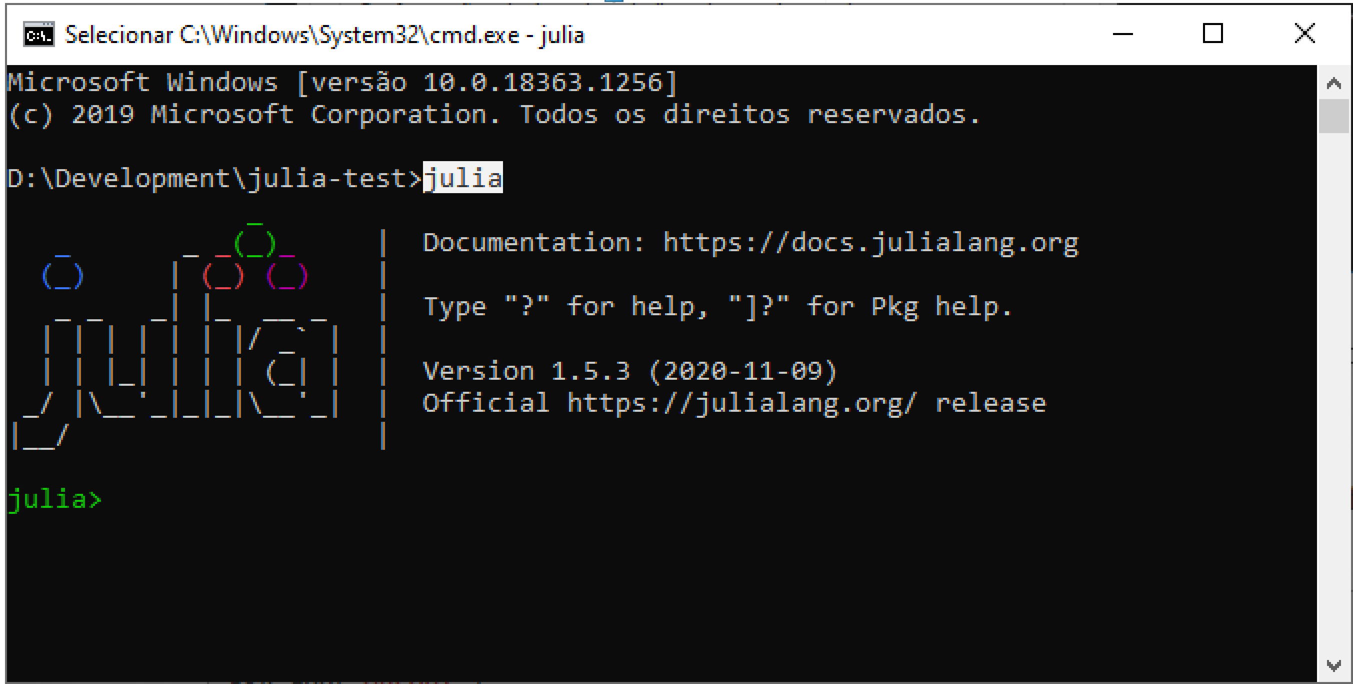




- Run Julia
  - Type **julia** in Command Prompt
  - If Julia **REPL** appears,
    you have successfully installed Julia!

> REPL (Read-Eval-Print-Loop) is the Julia session that runs in the Command Prompt

  - Leave REPL: **Ctrl+D** or **exit()**
  - Clean REPL screen: **Ctrl+L**
  - Interrupt execution: **Ctrl+C**
  - Reverse search: **Ctrl+R**



Package Management

In Julia REPL, type **using Pkg**

Installing Packages

- Type **Pkg.add("PackageName")**

Removing Packages

- Type **Pkg.rm("PackageName")**

Updating Packages

- Type **Pkg.update()**

Check Installed Packages

- Type **Pkg.status()**
- For a specific package, **Pkg.status("PackageName")**

Building Packages

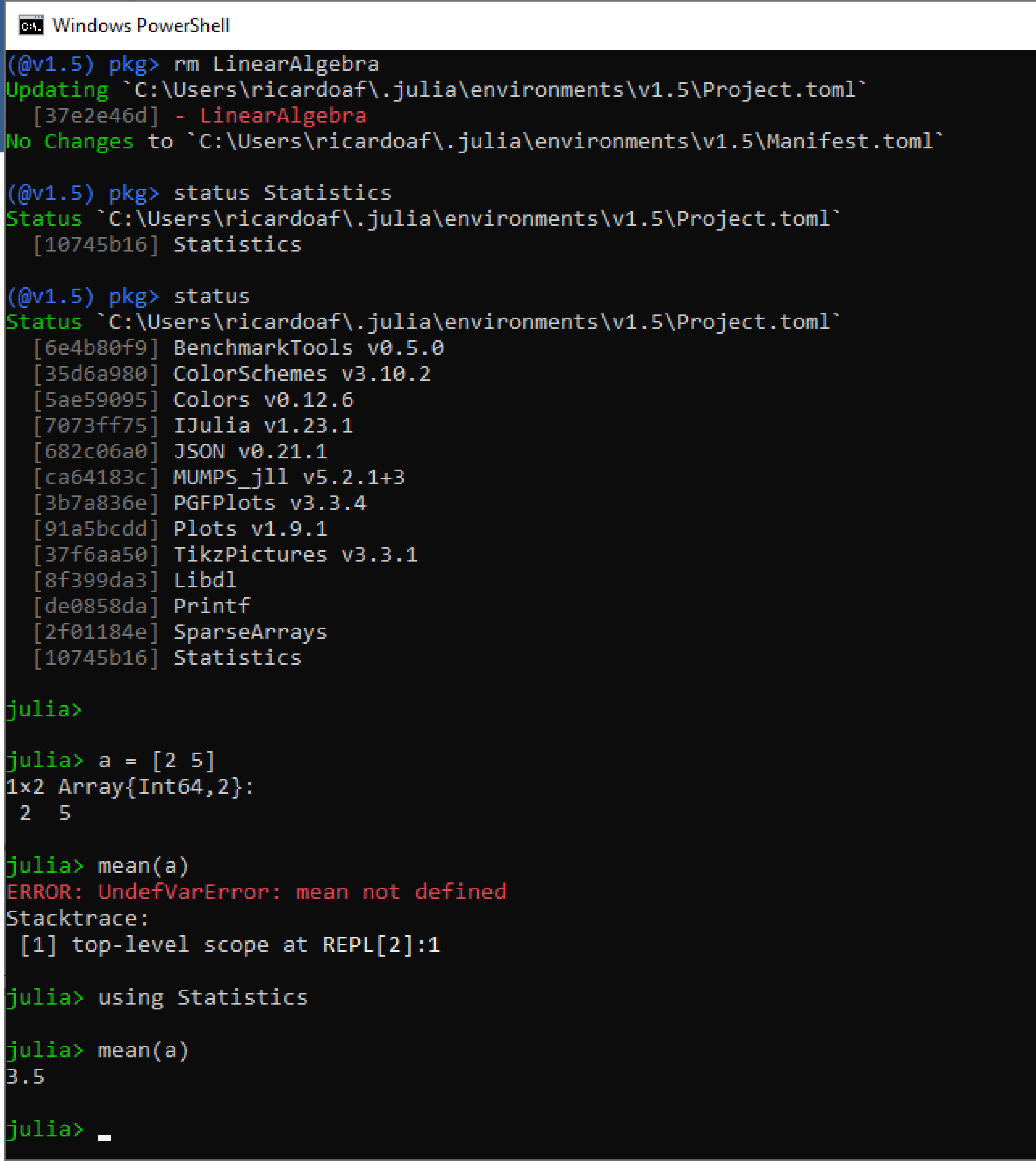
- Used when a package installation fails. Install missing libs and check system path variables. Reboot OS or restart Julia session. Then, type **Pkg.build("PackageName")**

Windows PowerShell

```
julia> using Pkg

julia> Pkg.add("LinearAlgebra")
  Resolving package versions...
Updating `C:\Users\ricardoaf\.julia\environments\v1.5\Project.toml`
  [37e2e46d] + LinearAlgebra
No Changes to `C:\Users\ricardoaf\.julia\environments\v1.5\Manifest.toml`

julia> Pkg.status()
Status `C:\Users\ricardoaf\.julia\environments\v1.5\Project.toml`
  [6e4b80f9] BenchmarkTools v0.5.0
  [35d6a980] ColorSchemes v3.10.2
  [5ae59095] Colors v0.12.6
  [7073ff75] IJulia v1.23.1
  [682c06a0] JSON v0.21.1
  [ca64183c] MUMPS_jll v5.2.1+3
  [3b7a836e] PGFPlots v3.3.4
  [91a5bcdd] Plots v1.9.1
  [37f6aa50] TikzPictures v3.3.1
  [8f399da3] Libdl
  [37e2e46d] LinearAlgebra
  [de0858da] Printf
  [2f01184e] SparseArrays
  [10745b16] Statistics

julia>
```

Package Management

- Alternatively, one can also type **]** in Julia REPL
  - This activates Julia "Package mode"
  - And so, to execute the forementioned commands, you can simply type
    - **add PackageName**
    - **rm PackageName**
    - **update**
    - **status**
    - **status PackageName**
    - **build PackageName**
  - To leave "Package mode", press **backspace**
- To use a package, one must type **using PackageName**



Commands and Pkg help

For **commands help**, one can type **?** in Julia REPL

This activates Julia “Help mode” and you can simply type a command for explanation and examples

It also works on “Package mode”

```
help?> abs
search: abs abs2 abspath AbstractSet abstract type AbstractChar AbstractDict AbstractFloat AbstractArray AbstractRange

  abs(x)

  The absolute value of x.

  When abs is applied to signed integers, overflow may occur, resulting in the return of a negative value. This
  overflow occurs only when abs is applied to the minimum representable value of a signed integer. That is, when x ==
  typemin(typeof(x)), abs(x) == x < 0, not -x as might be expected.

  Examples
  ==========

  julia> abs(-3)
  3

  julia> abs(1 + im)
  1.4142135623730951

  julia> abs(typemin(Int64))
  -9223372036854775808
```

```
(@v1.5) pkg> ?add
  add [--preserve=<opt>] pkg[=uuid] [@version] [#rev] ...

  Add package pkg to the current project file. If pkg could refer to multiple different packages, specifying uuid
  allows you to disambiguate. @version optionally allows specifying which versions of packages to add. Version
  specifications are of the form @1, @1.2 or @1.2.3, allowing any version with a prefix that matches, or ranges
  thereof, such as @1.2-3.4.5. A git revision can be specified by #branch or #commit.

  If a local path is used as an argument to add, the path needs to be a git repository. The project will then track
  that git repository just like it would track a remote repository online. If the package is not located at the top of
  the git repository, a subdirectory can be specified with path:subdir/path.

  Pkg resolves the set of packages in your environment using a tiered approach. The --preserve command line option
  allows you to key into a specific tier in the resolve algorithm. The following table describes the command line
  arguments to --preserve (in order of strictness).

  Argument Description
  -------- ---------------------------------------------------------------------------------
  all      Preserve the state of all existing dependencies (including recursive dependencies)
  direct   Preserve the state of all existing direct dependencies
  semver   Preserve semver-compatible versions of direct dependencies
  none     Do not attempt to preserve any version information
  tiered   Use the tier which will preserve the most version information (this is the default)

  │ Julia 1.5
  │
  │  Subdirectory specification requires at least Julia 1.5.

  Examples

  pkg> add Example
  pkg> add --preserve=all Example
  pkg> add Example@0.5
  pkg> add Example#master
  pkg> add Example#c37b675
  pkg> add https://github.com/JuliaLang/Example.jl#master
  pkg> add git@github.com:JuliaLang/Example.jl.git
  pkg> add Example=7876af07-990d-54b4-ab0e-23690620f79a

(@v1.5) pkg> _
```

Editors and IDEs

These are some editors and IDE that can be used for Julia coding

Juno

Atom Plugin

VS Code

VS Code Extension

Jupyter

jupyter

Jupyter kernel

Pluto.jl

Simple reactive notebooks

Vim

Vim plugin

Emacs

Emacs plugin

SublimeText

Sublime Text

NotePad++

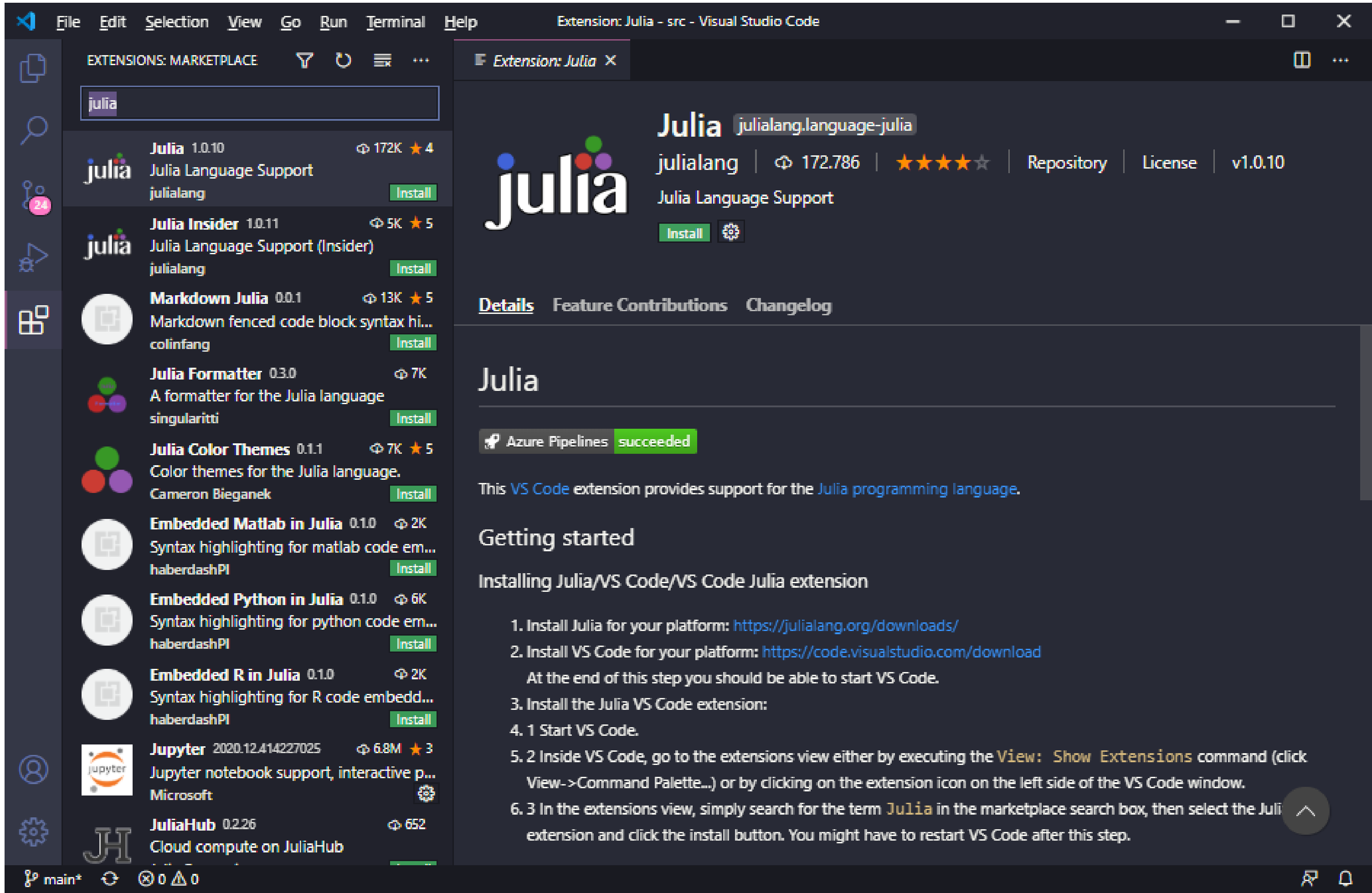
Notepad++

Using a IDE for Julia

Using VS Code for Julia

- **Install VS Code** based on the platform you are using
  https://code.visualstudio.com/

- **Install Julia extension**
  - Open VS Code
  - Select **View** and then click **Extensions**
  - Type **julia** in the search box
  - Click the green **Install** button
  - **Restart** VS Code after installation



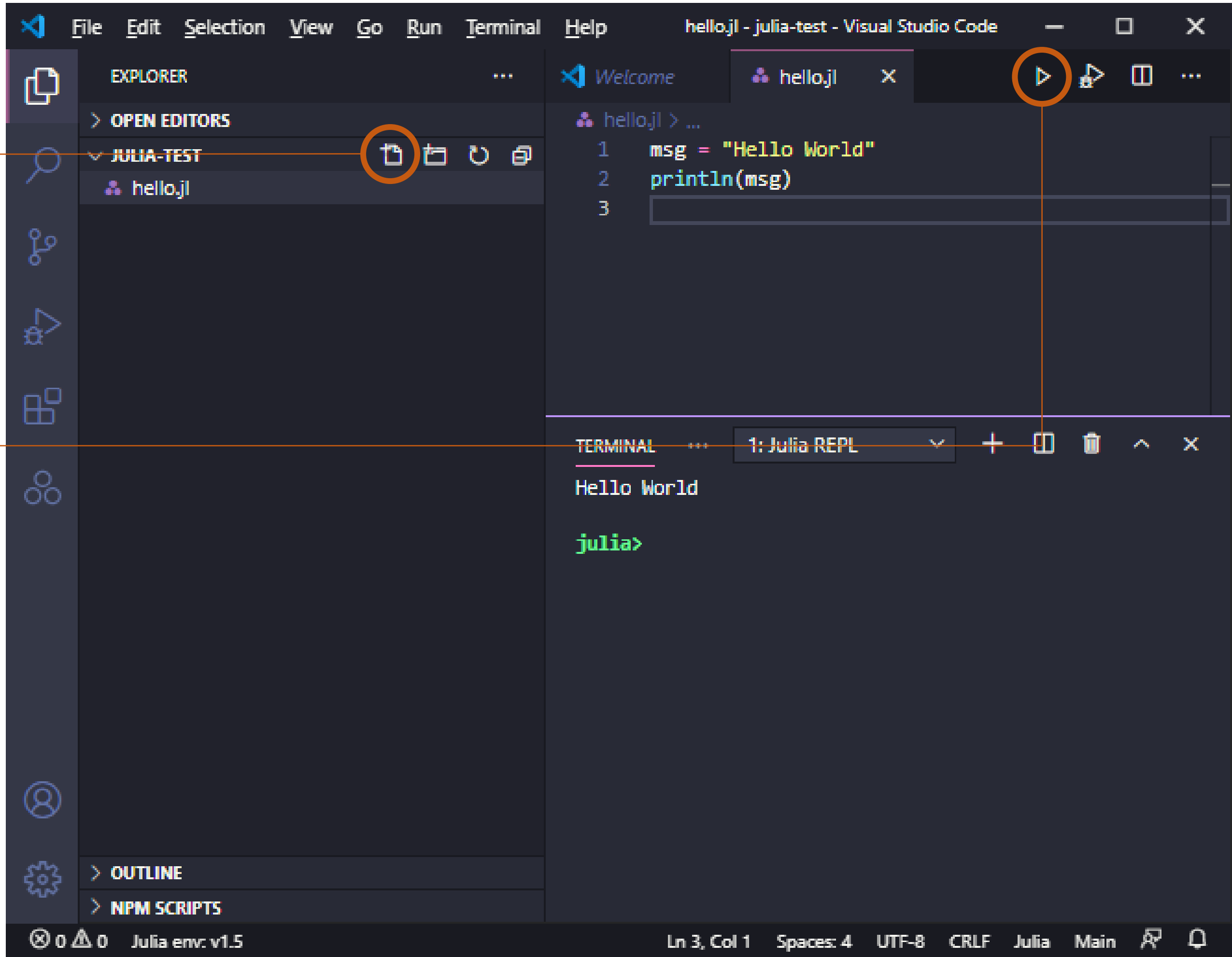
More details:
https://www.julia-vscode.org/docs/stable/

Using a IDE for Julia

Using VS Code for Julia

- Creating your Julia Hello World program
  - Click **File > Open Folder…**
    - Select a folder for your program
  - On Explorer panel, under your folder tree, click **New File**
    - Name the file with .jl extension (**hello.jl**)
  - Enter **Hello World code** in hello.jl

- Running Hello World program
  - Click **Julia: Execute File** in terminal play button
  - See the corresponding output in the terminal



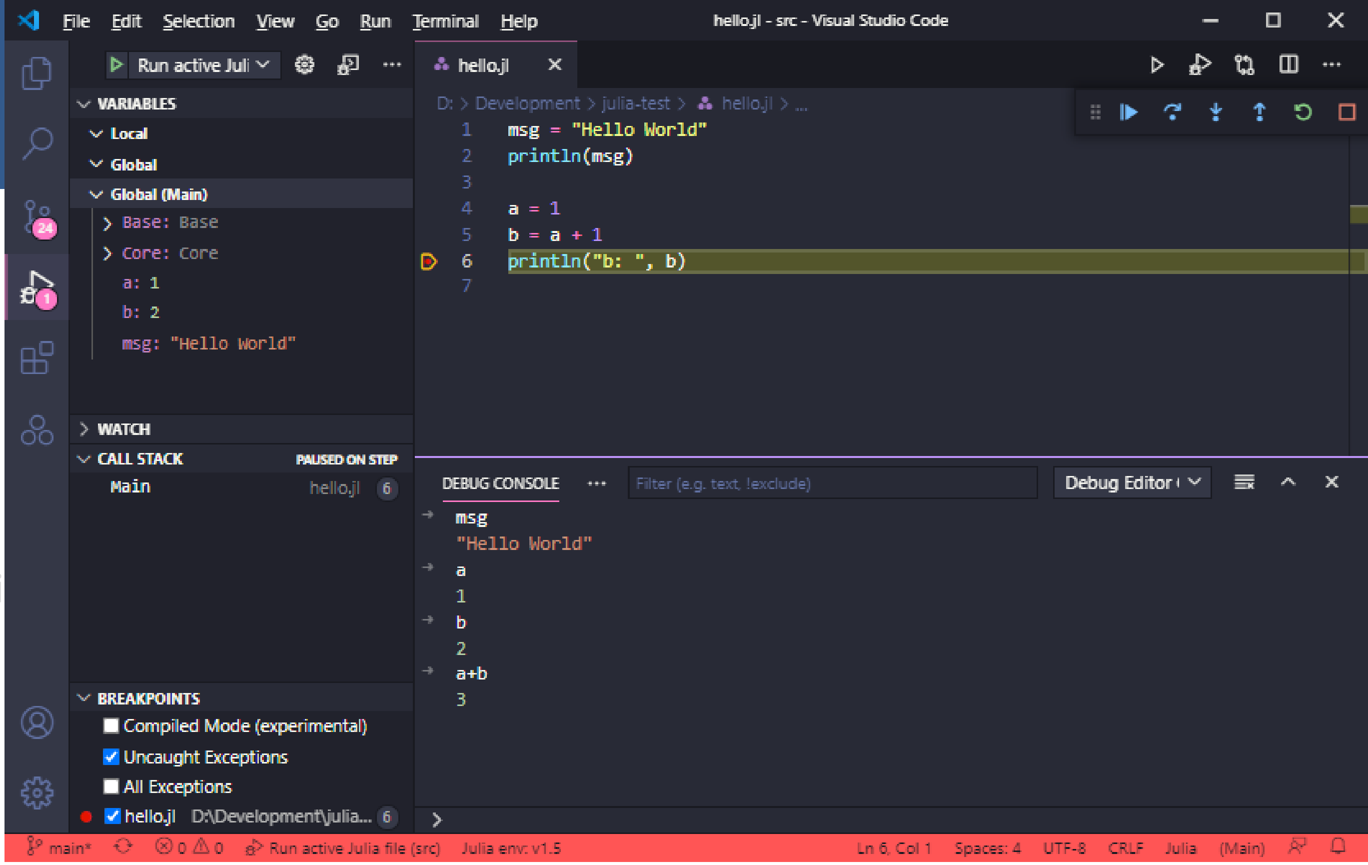
More details:
https://www.julia-vscode.org/docs/stable/

Using a IDE for Julia

Using VS Code for Julia

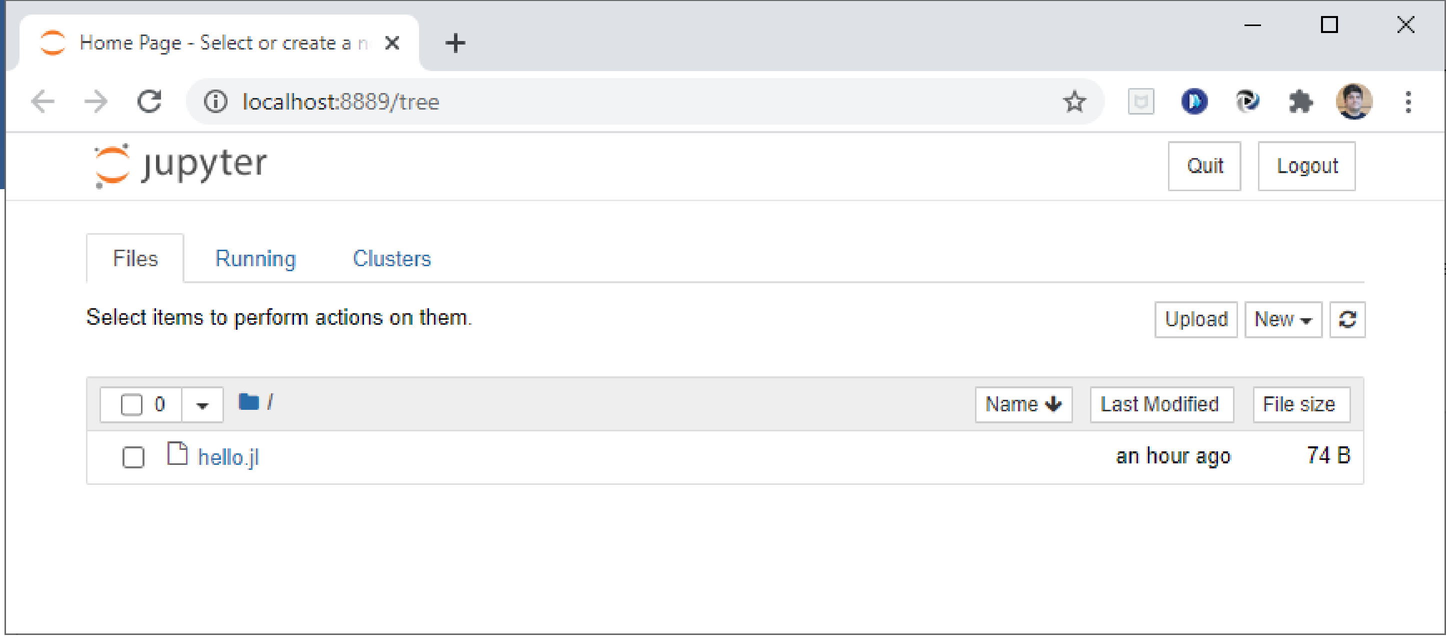
- Creating your Julia Hello World program
  - Click **File > Open Folder…**
    - Select a folder for your program
  - On Explorer panel, under your folder tree, click **New Fi**
    - Name the file with .jl extension (**hello.jl**)
  - Enter **Hello World code** in hello.jl

- Running Hello World program
  - Click **Julia: Execute File** in terminal play button
  - See the corresponding output in the terminal



- Debugging your code
  - Put **breakpoints** into the code (click near to the line number)
  - Click **Julia: Debug File** in terminal play button with a bug
  - See variables and stack on debug panel on the left
  - Manipulate variables on **Debug console** near to terminal

Using a Notebook for Julia



Using Jupyter for Julia

- Install IJulia (Interactive Julia)
  - In Julia REPL, type] for "Package Mode"
  - Type **add IJulia**

- Install Jupyter
  - In Julia REPL, type **using IJulia**
  - Then, type **notebook()**
  - As first execution, proceed with Jupyter installation using Conda.jl
  - Then a IJulia notebook will launch in your browser

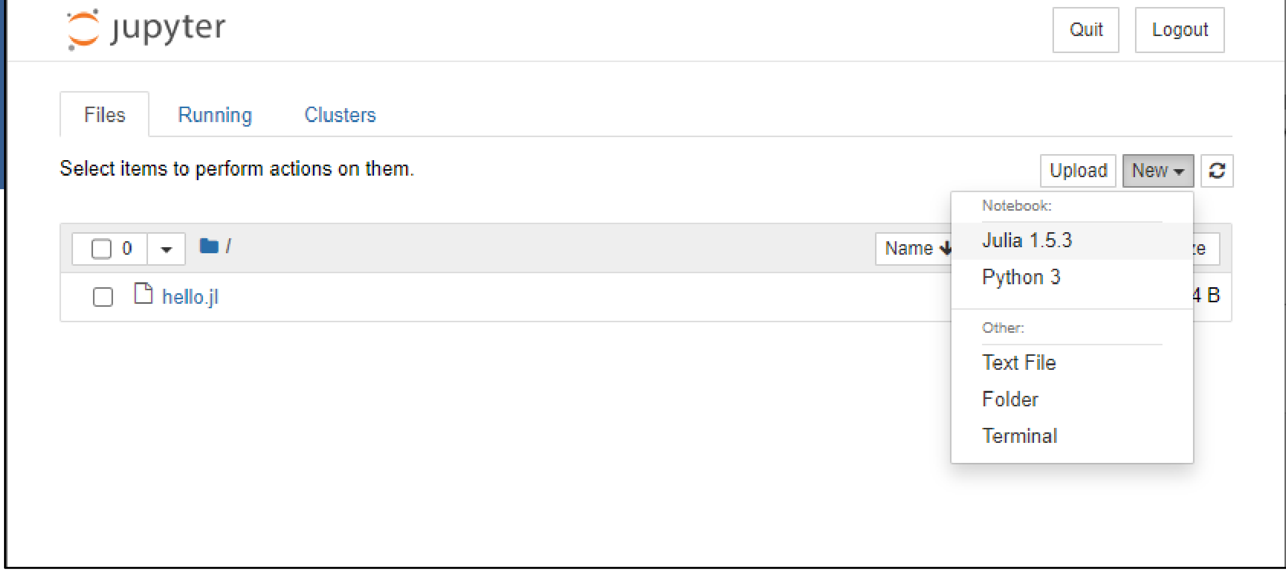
- Run Jupyter
  - In Julia REPL, type **using IJulia**
  - Then, type **notebook()**
  - Use **notebook(dir=pwd())** to launch Jupyter in current directory

More details:
https://github.com/JuliaLang/IJulia.jl/

Using a Notebook for Julia

Using <u>Jupyter</u> for Julia

- Creating a IJulia Notebook
  - In Jupyter session, click on **New**
  - Select Julia installed version



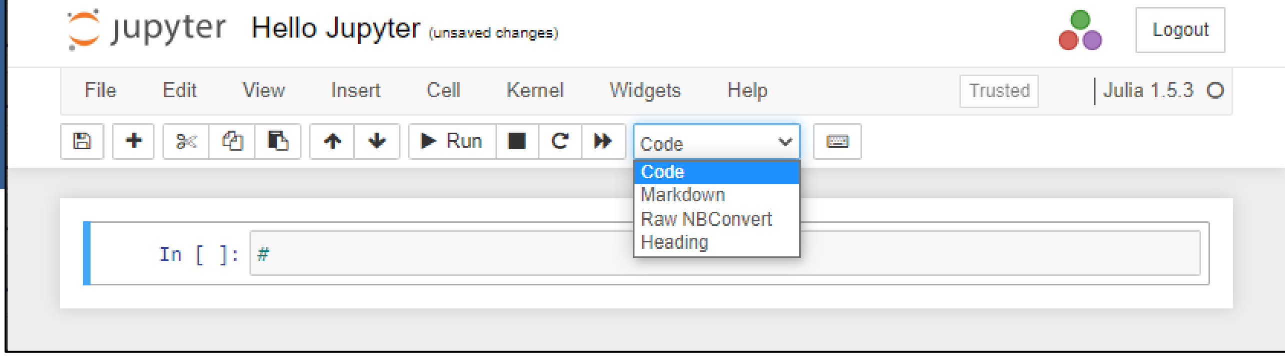
Using a Notebook for Julia

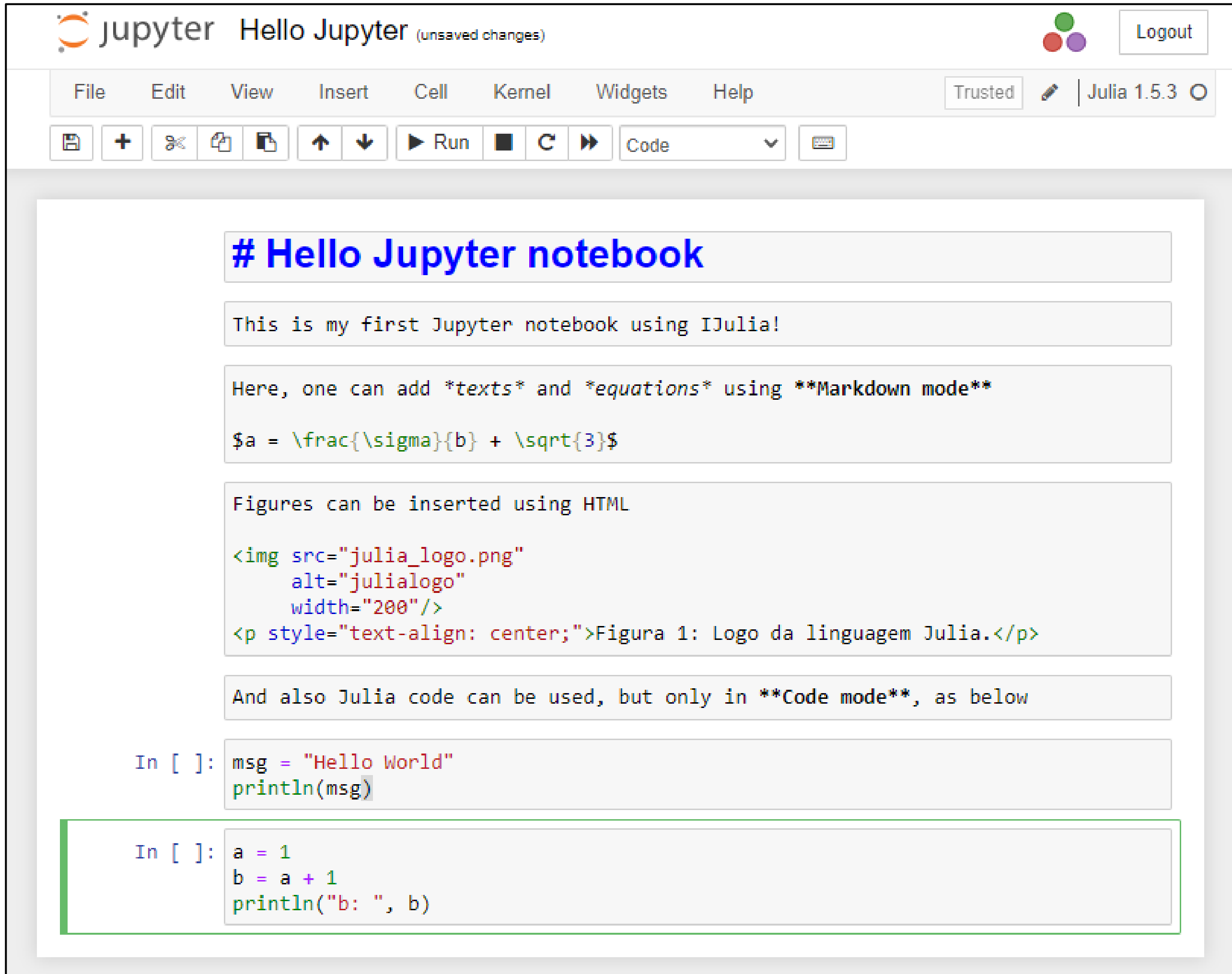
Using Jupyter for Julia

- Creating a IJulia Notebook
  - In Jupyter session, click on New
  - Select Julia installed version

- Editing a IJulia Notebook
  - Click on Untitled to rename the notebook
  - Essentially, one can use **Code** or **Markdown** modes
  - Markdown accept
    - Heading levels
    - Text and figures (using HTML)
    - Equations (using LaTeX syntax)
  - Code accept REPL Julia syntax

  - **Ctrl+Enter** runs the selected cell





Using a Notebook for Julia

Using Jupyter for Julia

- Creating a IJulia Notebook
  - In Jupyter session, click on **New**
  - Select Julia installed version

- Editing a IJulia Notebook
  - Click on Untitled to rename the notebook
  - Essentially, one can use **Code** or **Markdown** modes
  - Markdown accept
    - Heading levels
    - Text and figures (using HTML)
    - Equations (using LaTeX syntax)
  - Code accept REPL Julia syntax

  - **Ctrl+Enter** runs the selected cell
  - All cells can be executed in **Kernel** > **Restart & Run All**

Hello Jupyter notebook

This is my first Jupyter notebook using IJulia!

Here, one can add *texts* and *equations* using **Markdown mode**

$a = \frac{\sigma}{b} + \sqrt{3}$

Figures can be inserted using HTML

Figura 1: Logo da linguagem Julia.

And also Julia code can be used, but only in **Code mode**, as below

In [1]:
```
msg = "Hello World"
println(msg)
```
```
Hello World
```

In [2]:
```
a = 1
b = a + 1
println("b: ", b)
```
```
b: 2
```

Basic Concepts: Variables

Julia Basic Concepts

Ricardo A. Fernandes ricardoaf@lccv.ufal.br

This notebook presents main Julia basic concepts and serves as a introductory guide to the language. It is based on the two following references:

- Kochenderfer, M. J. & Wheeler, T. A. (2019) Algorithms for Optimization, MIT Press
- Banas D. (2018) Julia Tutorial, http://www.newthinktank.com/2018/10/julia-tutorial/

Variables

- Types are dynamically assigned and can be changed
- Variables start with _, letters and then numbers, or !
- Unicode characters are also allowed, with a few restrictions

In [1]:
```
β = 3  # unicode chars: \beta(tab)
s = "Dogs: "
print(s, β)
```

```
Dogs: 3
```

Basic Concepts: Types: Booleans

Types

Booleans

The Boolean type in Julia, written *Bool*, includes the values **true** and **false**

In [2]:
```
x = true
y = false
typeof(x)
```
Out[2]:
```
Bool
```

In [3]:
```
!x  # not
```
Out[3]:
```
false
```

In [4]:
```
x && y  # and
```
Out[4]:
```
false
```

In [5]:
```
x || y  # or
```
Out[5]:
```
true
```

Basic Concepts: Types: Numbers

Numbers

Julia supports integer and floating point numbers

```
In [6]: typeof(42), typeof(42.0)
Out[6]: (Int64, Float64)
```

```
In [7]: x = 4
        y = 2
        (x^2 + 2y) / (y + 1)
Out[7]: 6.666666666666667
```

```
In [8]: x % y  # x mod y
Out[8]: 0
```

```
In [9]: x += 1  # shortcut for x = x + 1
Out[9]: 5
```

```
In [10]: 3 > 4
Out[10]: false
```

```
In [11]: 3 ≤ 4  # unicode also works
Out[11]: true
```

```
In [12]: 3 < 4 < 5
Out[12]: true
```

Basic Concepts: Types: Strings

Strings

A string is an array of characteres

In [13]:
```
s1 = "Just some random words"
length(s1)
```
Out[13]:
```
22
```

In [14]:
```
s1[1], s1[end], s1[1:4]
```
Out[14]:
```
('J', 's', "Just")
```

In [15]:
```
s2 = string("Hey", " you!")  # concatenation
s2 == "Hey" * " you!"
```
Out[15]:
```
true
```

In [16]:
```
a = 2
b = 3
"$a + $b = $(a + b)"  # interpolation
```
Out[16]:
```
"2 + 3 = 5"
```

Basic Concepts: Types: Strings


```
In [17]: s3 = "I
         have
         many
         lines"
         print(s3)

         I
         have
         many
         lines
```

```
In [18]: "House" > "Home"   # string comparison (== > < !=)

Out[18]: true
```

```
In [19]: occursin("key", "monkey")   # substring

Out[19]: true
```

Basic Concepts: Types: Vectors

Vectors

- A *vector* is a 1D-array that stores a sequence of values
- Can be constructed using square brackets, separating elements by commas

In [20]:
```
x = [];                        # empty vector
x = trues(3);                  # Boolean vector containing three trues
x = ones(3);                   # vector of three ones
x = zeros(3);                  # vector of three zeros
x = rand(3);                   # vector of three random numbers between 0 and 1
x = [3, 1, 4];                 # vector of integers
x = [3.1415, 1.618, 2.7182]; # vector of floats
```

Array comprehension can also be used to create vectors

In [21]:
```
print([sin(x) for x = 1:5])
```

```
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672, -0.7568024953079282, -0.9589242746631385]
```

In [22]:
```
typeof([3, 1, 4]), typeof([3.1415, 1.618, 2.7182])
```

Out[22]:
```
(Array{Int64,1}, Array{Float64,1})
```

In [23]:
```
println(x[1])       # first element is indexed by 1
println(x[3])       # third element
println(x[end])     # last element
println(x[end - 1]) # second to last element)
```

```
3.1415
2.7182
2.7182
1.618
```

Basic Concepts: Types: Vectors

In [24]:
```
x = [1, 1, 2, 3, 5, 8, 13]
println("len(x): ", length(x))  # vector length
println(x[1:3])                 # first three elements
println(x[1:2:end])             # elements with odd indices
println(x[end:-1:1])            # reverse order
```

```
len(x): 7
[1, 1, 2]
[1, 2, 5, 13]
[13, 8, 5, 3, 2, 1, 1]
```

In [25]:
```
println(sum(x))      # sum of vector elements
println(maximum(x))  # max value
println(minimum(x))  # min value
```

```
33
13
1
```

In [26]:
```
using Statistics
println(mean(x))  # mean of vector elements
```

```
4.714285714285714
```

Basic Concepts: Types: Vectors

In [27]:
```julia
println([x, x])             # concatenation
println(push!(x, -1))       # add an element to the end
println(pop!(x))            # remove and element from the end
println(append!(x, [2, 3])) # append to the end of x
println(sort!(x))           # sort vector elements
x[1] = 2; println(x)        # change first element
```

```
[[1, 1, 2, 3, 5, 8, 13], [1, 1, 2, 3, 5, 8, 13]]
[1, 1, 2, 3, 5, 8, 13, -1]
-1
[1, 1, 2, 3, 5, 8, 13, 2, 3]
[1, 1, 2, 2, 3, 3, 5, 8, 13]
[2, 1, 2, 2, 3, 3, 5, 8, 13]
```

In [28]:
```julia
x = [1, 2]
y = [3, 4]
println(x + y)        # add vectors
println(3x - [1, 2])  # multiply by a scalar and subtract
```

```
[4, 6]
[2, 4]
```

In [29]:
```julia
using LinearAlgebra
println(dot(x,y))  # dot product
println(x⋅y)       # dot product using unicode character
```

```
11
11
```

Basic Concepts: Types: Vectors

In [30]:
```
# element-wise operations
println(x .* y)        # multiplication
println(x .^ 2)        # squaring
println(sin.(x))       # application of sin
println(sqrt.(x))      # application of sqrt
println(max.(x, 1.5))  # application of max{xᵢ,1.5}
```

```
[3, 8]
[1, 4]
[0.8414709848078965, 0.9092974268256817]
[1.0, 1.4142135623730951]
[1.5, 2.0]
```

Basic Concepts: Types: Matrices

Matrices

- A *matrix* is a 2D-array
- Like a vector, can be constructed using square brackets
- Use spaces to delimit elements in the same row
- Use semicolons to delimit rows

In [31]:
```
X = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
println(typeof(X))
size(X)
```

```
Array{Int64,2}
```

Out[31]:
```
(4, 3)
```

In [32]:
```
println(X[2])         # 2nd element using column-major ordering
println(X[3,2])       # element in 3rd row and 2nd column
println(X[1,:])       # 1st row
println(X[:,2])       # 2nd column
println(X[:,1:2])     # first two columns
println(X[1:2,1:2])   # top left 2x2 submatrix
```

```
4
8
[1, 2, 3]
[2, 5, 8, 11]
[1 2; 4 5; 7 8; 10 11]
[1 2; 4 5]
```

Basic Concepts: Types: Matrices

In [33]:
```
println(Matrix(1.0I, 3, 3))          # 3x3 identity matrix
println(Matrix(Diagonal([3, 2, 1]))) # Diagonal matrix
println(rand(3,2))                   # Random matrix
println(zeros(3,2))                  # Matrix of zeros
println([sin(x + y) for x=1:3, y=1:2]) # array comprehension
```

```
[1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0]
[3 0 0; 0 2 0; 0 0 1]
[0.10073073367249985 0.6128712205592737; 0.532513242861034 0.13013717690737714; 0.7904904927101746 0.6373919171245033]
[0.0 0.0; 0.0 0.0; 0.0 0.0]
[0.9092974268256817 0.1411200080598672; 0.1411200080598672 -0.7568024953079282; -0.7568024953079282 -0.9589242746631385]
```

In [34]:
```
println(X')       # complex conjugate transpose
println(3X .+ 2)  # multiplying by scalar and adding scalar
X = [1 3; 3 1]    # invertible matrix
println(inv(X))   # invsersion
println(det(X))   # determinant
println([X X])    # horizontal concatenation
println([X; X])   # vertical concatenation
println(sin.(X))  # element-wise application of sin
```

```
[1 4 7 10; 2 5 8 11; 3 6 9 12]
[5 8 11; 14 17 20; 23 26 29; 32 35 38]
[-0.125 0.375; 0.375 -0.125]
-8.0
[1 3 1 3; 3 1 3 1]
[1 3; 3 1; 1 3; 3 1]
[0.8414709848078965 0.1411200080598672; 0.1411200080598672 0.8414709848078965]
```

Basic Concepts: Types: Matrices

In [35]:
```
# Solving a system of linear equations
# 3x + 2y - z = 1
# 2x - 2y + 4z = -2
# -x + 1/2*y -z = 0

A = [3 2 -1; 2 -2 4; -1 1/2 -1]
b = [1, -2, 0]
x = A\b
print(x)
```

```
[0.9999999999999994, -1.9999999999999984, -1.9999999999999984]
```

Basic Concepts: Types: Tuples

Tuples

- A *tuple* is an ordered list of values (can be of different types)
- Similar to arrays, but **can't be mutated!**
- Constructed with parentheses

In [36]:
```
x = (1,)  # a single element
x = (1, 0, [1, 2], 2.5, 4.66)  # third element is a vector
length(x)
```

Out[36]:
```
5
```

In [37]:
```
x[2], x[end], x[4:end]
```

Out[37]:
```
(0, 4.66, (2.5, 4.66))
```

In [38]:
```
t1 = ((1, 2), (3, 4))  # multidimensional tuple
println("t1[1][1] = ", t1[1][1])

t2 = (sue=("Sue", 100), paul=("Paul", 23)) # named tuple
println(t2.sue)
```

```
t1[1][1] = 1
("Sue", 100)
```

Dictionaries

- A *dictionary* is a collection of key-value pairs
- Key-value pairs are indicated with a double arrow operator
- One can index into a dictionary using square brackers as arrays/tuples

In [39]:
```
x = Dict();  # empty dictionary
x[3] = 4     # associate value 4 with key 3
```

Out[39]:
```
4
```

In [40]:
```
x = Dict(3=>4, 5=>1)  # create dictionary with 2 key-value pairs
```

Out[40]:
```
Dict{Int64,Int64} with 2 entries:
  3 => 4
  5 => 1
```

In [41]:
```
println(x[5])           # return value associated with key 5
println(haskey(x, 3))  # check if dict has key 3
println(haskey(x, 4))  # check if dict has key 4
```

```
1
true
false
```

Basic Concepts: Types: Dictionaries

In [42]:
```
d1 = Dict("pi"=>3.14, "e"=>2.718)  # new dict
println(d1["pi"])                  # print value of "pi" key
d1["golden"] = 1.618               # add a key-value
delete!(d1, "pi")                  # delete a key-value
println(keys(d1))                  # display all keys
println(values(d1))                # display all keys
```

```
3.14
["golden", "e"]
[1.618, 2.718]
```

Basic Concepts: Types: Composite Types

Composite types

- A *composite type* is a collection of named fields
- Use **struct** keyword. By default, it is immutable
- Adding keyword **mutable** makes an instance mutable

- Double-colon operator can be used to annotate types (any variable)
- Annotation requires that one pass correct types for fields/variables
- Type annotations alow runtime improvements (compiler optimization)

In [43]:
```julia
struct Customer
    name::String
    balance::Float32
    id::Int
end

# Create a Customer object
bob = Customer("Bob Smith", 10.50, 123)
println(bob.name)

# Change bob name
bob.name = "Sue Smith"   # ERROR!
```

```
Bob Smith

setfield! immutable struct of type Customer cannot be changed

Stacktrace:
 [1] setproperty!(::Customer, ::Symbol, ::String) at .\Base.jl:34
 [2] top-level scope at In[43]:12
 [3] include_string(::Function, ::Module, ::String, ::String) at .\loading.jl:1091
```

Basic Concepts: Types: Composite Types

In [44]:
```
mutable struct MCustomer
    name::String
    balance::Float32
    id::Int
end
bob = MCustomer("Bob Smith", 10.50, 123)

# Change bob name
bob.name = "Sue Smith"
println(bob.name)
```

```
Sue Smith
```

Basic Concepts: Types: Abstract Types

Abstract Types

- Analog to classes in object-oriented languages (but without methods)

In [45]:
```
# Float64 hierarchy
println(supertype(Float64))
println(supertype(AbstractFloat))
println(supertype(Real))
println(supertype(Number))
println(supertype(Any))
```

```
AbstractFloat
Real
Number
Any
Any
```

In [46]:
```
println(subtypes(AbstractFloat))  # different types of AbstractFloats
println(subtypes(Float64))        # Float64 doesn't have any subtypes
```

```
Any[BigFloat, Float16, Float32, Float64]
Type[]
```

Basic Concepts: Types: Abstract Types

We can define our own abstract types

- They can't be instantiated like Structs, but can have subtypes

In [47]:
```julia
abstract type Animal end

struct Dog <: Animal
    name::String
    bark::String
end

struct Cat <: Animal
    name::String
    meow::String
end

bowser = Dog("Bowser", "Ruff")
muffin = Cat("Muffin", "Meow")

println(bowser.name, " sound: ", bowser.bark)
println(muffin.name, " sound: ", muffin.meow)
```

```
Bowser sound: Ruff
Muffin sound: Meow
```

Functions

A *function* is an object that maps a tuple of argument values to a return value

In [48]:
```
# Functions can be named
function f(x, y)
    return x + y
end

f(x, y) = x + y
f(3, 0.1415)
```

Out[48]:
```
3.1415
```

In [49]:
```
# Or anonymous
h = x -> x^2 + 1           # assign anonymous function to a variable
g(f, a, b) = [f(a), f(b)]  # applies function f to a and b
println(g(h, 5, 10))
```

```
[26, 101]
```

In [50]:
```
# map applies a function to each item
println(map(x -> x * x, [1, 2, 3]))
println(map((x,y) -> x + y, [1,2], [3,4]))
```

```
[1, 4, 9]
[4, 6]
```

Basic Concepts: Functions

In [51]:
```julia
# arguments are passed by value
v1 = 5
function changeV1(v1); v1 = 10; end
changeV1(v1); println(v1)

# So, you can use globals inside functions
function changeV12(); global v1 = 10; end
changeV12(); println(v1)

# or return modified parameter
function changeV13(v1); v1 = 10; return v1; end
v1 = changeV13(v1); println(v1)
```

```
5
10
10
```

In [52]:
```julia
# values are not copied when passed to a function
# If a function modifies an array, the changes will be visible in the caller

# ! denotes function argument will be modified (good practice in Julia)
function changeArray!(x)
    x[end] += 1
end

a = [1, 2, 3]
changeArray!(a)
println(a)
```

```
[1, 2, 4]
```

Basic Concepts: Functions

In [53]:
```
# Variable arguments
function getSum(args...)
    sum = 0
    for a in args
        sum += a
    end
    return sum
end
println(getSum(1,2,3,4,5))
```

```
15
```

In [54]:
```
# Return multiple values
function next2(val)
    return val + 1, val + 2
end
println(next2(4))
```

```
(5, 6)
```

In [55]:
```
# Functions that return functions
function makeMultiplier(num)
    return function(x); return x*num; end
end

mult3 = makeMultiplier(3)
println(mult3(6))
```

```
18
```

Basic Concepts: Functions

In [56]:
```
# Optional arguments can be specified setting default values
f(x, y, z=1) = x*y + z
println(f(3, 2, 1))
println(f(3, 2))
```

```
7
7
```

In [57]:
```
# Keyword arguments are defined using a semicolon
f(x, y=10; z=2) = (x+y)*z

println(f(1))          # x=1, y=10, z=2
println(f(2, z=3))     # x=2, y=10, z=3
println(f(2, 3))       # x=2, y=3,  z=2
println(f(2, 3, z=1))  # x=2, y=3,  z=1
```

```
22
36
10
5
```

In [58]:
```
# Function arguments can also handle different data types
function getSum2(num1::Number, num2::Number)
    return num1 + num2
end
println(1, 2.0*5)  # Integer and Float arguments
```

```
110.0
```

Basic Concepts: Functions

In [59]:
```
# Function overloading
F(x::Int64) = x + 10
F(x::Float64) = x + 3.1415

println(F(1))
println(F(1.0))
```

```
11
4.141500000000001
```

Basic Concepts: Important Note!

Important note

In [60]:
```
# Julia arrays are not copied when assigned to another variable
# After A = B, changing elements of B will modify A as well
B = [1, 2, 3, 4]
A = B
C = copy(B)

B[2] = 20
println("A: ", A)
println("C: ", C)

# Updating operators like += do not operate in-place,
# they are equivalent to A = A + B which rebinds the left-hand side to the result of the right-hand side expression
A += B
println("A-2B: ", A-2B)
```

```
A: [1, 20, 3, 4]
C: [1, 2, 3, 4]
A-2B: [0, 0, 0, 0]
```

Basic Concepts: Control Flow: Conditional Evaluation

Control flow

Conditional Evaluation

In [61]:
```
age = 12
if age >= 5 && age <= 6
    println("You're in Kindergarten")
elseif age >= 7 && age <= 13
    println("You're in Middle School")
elseif age >= 14 && age <= 18
    println("You're in High School")
else
    println("Stay Home")
end
```

```
You're in Middle School
```

In [62]:
```
f(x) = x > 0.0 ? x : 0
println(f(-10))
println(f(+10))
```

```
0
10
```

Loops

In [63]:
```
# using while
x = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
while x != []
    s += pop!(x)
end
println(s)
```

```
82
```

In [64]:
```
# using for
x = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for i = 1:length(x)
    s += x[i]
end
println(s)

# or
s = 0
for i in x
    s += i
end
println(s)
```

```
82
82
```

File Input/Output

In [65]:
```
# Open file for writing
open("random.txt", "w") do file
    write(file, "Here is some random text\nIt is great\n")
end

# Open a file for reading
open("random.txt") do file
    # Read whole file into a string
    data = read(file, String)
    println(data)
end

open("random.txt") do file
    # Read each line 1 at a time
    for line in eachline(file)
        println(line)
    end
end
```

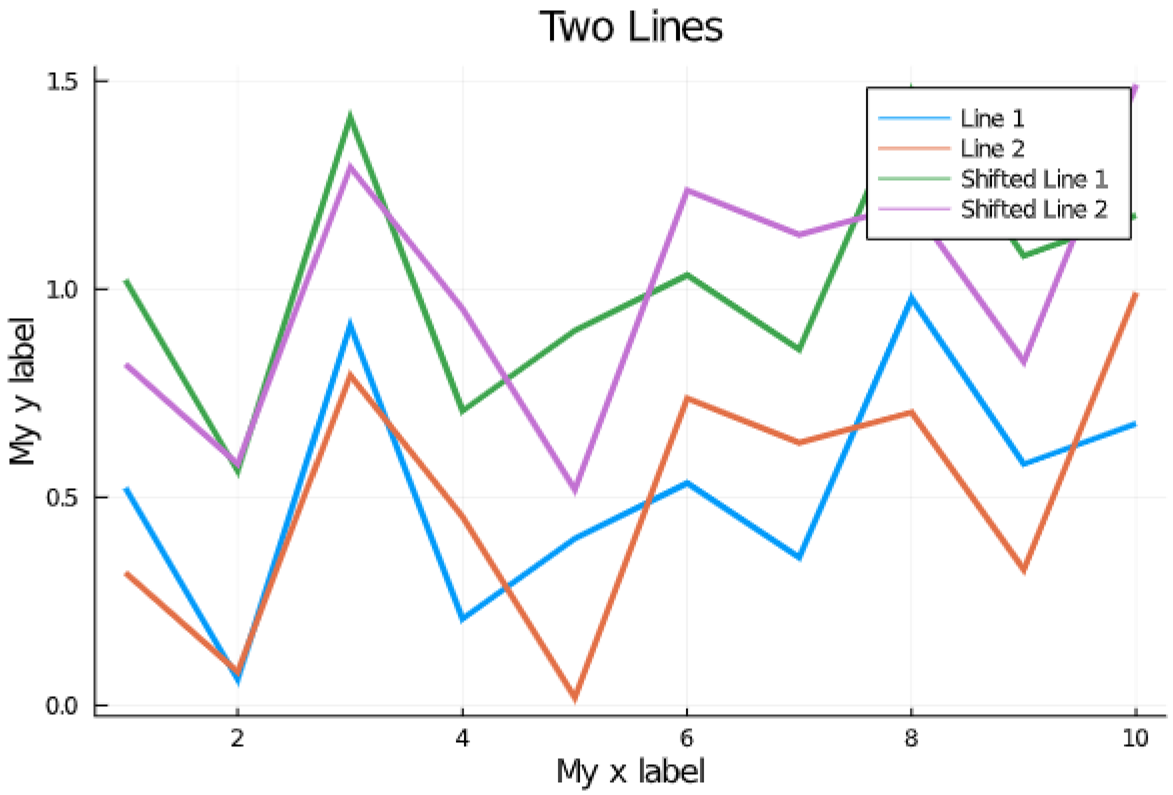
```
Here is some random text
It is great

Here is some random text
It is great
```

Basic Concepts: Plots

Simple plot example

In [66]:
```
using Plots
x = 1:10; y = rand(10, 2) # 2 columns means two lines
plot(x, y, title = "Two Lines", label = ["Line 1" "Line 2"], lw = 3)
plot!(x, y.+0.5, label = ["Shifted Line 1" "Shifted Line 2"], lw = 3)
xlabel!("My x label"); ylabel!("My y label")
```

Out[66]:



Julia vs MATLAB benchmark

Copying multidimensional arrays

- Given an arbitrary n x n x 3 matrix A, perform the operations:
  - $A_{ij1} = A_{ij2}$
  - $A_{ij3} = A_{ij1}$
  - $A_{ij2} = A_{ij3}$
- Use loop and vectorization strategies

Obtained elapsed times [s]

	$n = 5k$	$n = 7k$	$n = 9k$
MATLAB loop	0.48	1.03	2.00
Julia loop	***0.046***	***0.091***	***0.15***
MATLAB vector	*0.39*	*0.78*	*1.29*
Julia vector	0.40	0.75	1.19
	8.47x	*8.57x*	*8.60x*

Other benchmarks
https://modelingguru.nasa.gov/docs/DOC-2783

MATLAB

```matlab
function copy_multidimensional_arrays (n_rep, N)
if nargin<1 || isempty(n_rep), n_rep = 5; end
if nargin<2 || isempty(N), N = [5000 7000 9000]; end

loop_time = zeros(size(N));
vec_time = zeros(size(N));
for i = 1:length(N), n = N(i);

    A_ = rand(n, n, 3);

    A = A_; tic;
    for k = 1:n_rep, A = loop_strategy(A, n); end
    loop_time(i) = toc/n_rep;

    A = A_; tic;
    for k = 1:n_rep, A = vec_strategy(A, n); end
    vec_time(i) = toc/n_rep;
end
disp('Loop time [s]:'); disp(loop_time)
disp(' Vec time [s]:'); disp(vec_time)


function A = loop_strategy(A, n)
for i = 1:n
    for j = 1:n
        A(i,j,1) = A(i,j,2);
        A(i,j,3) = A(i,j,1);
        A(i,j,2) = A(i,j,3);
    end
end

function A = vec_strategy(A, ~)
A(:,:,[1 3 2]) = A(:,:,[2 1 3]);
```

Julia

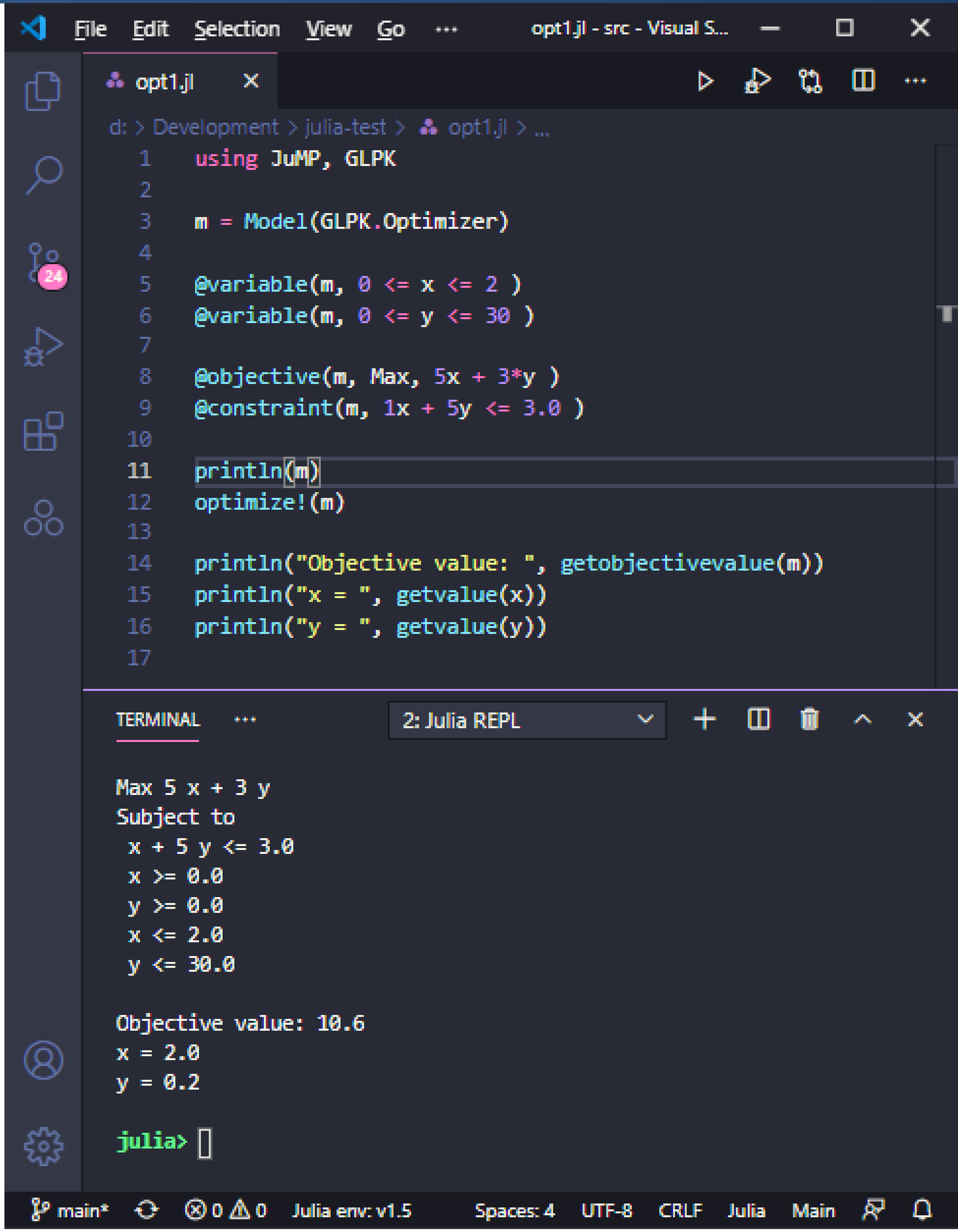
```julia
function copy_multidimensional_arrays(n_rep=5, N=[5000 7000 9000])

    loop_time = zeros(size(N))
    vec_time = zeros(size(N))

    for i = 1:length(N); n = N[i]

        A_ = rand(n, n, 3)

        A = A_;
        loop_time[i] = @elapsed begin
            for k = 1:n_rep; A = loop_strategy(A, n); end
        end
        loop_time[i] /= n_rep;

        A = A_;
        vec_time[i] = @elapsed begin
            for k = 1:n_rep; A = vec_strategy(A); end
        end
        vec_time[i] /= n_rep;
    end
    println("Loop time [s]:"); println(loop_time)
    println(" Vec time [s]:"); println(vec_time)

end

function loop_strategy(A, n)
    for j = 1:n, i = 1:n
        A[i,j,1] = A[i,j,2]
        A[i,j,3] = A[i,j,1]
        A[i,j,2] = A[i,j,3]
    end
    return A
end

function vec_strategy(A)
    A[:,:,[1 3 2]] = A[:,:,[2 1 3]];
    return A
end

copy_multidimensional_arrays()
```

Optimization example

JuMP

- Modeling language and packages for mathematical optimization in Julia
  - https://jump.dev/
- Makes it **easy** to formulate and solve optimization problems
  - Linear programming
  - Semidefinite programming
  - Integer programming
  - Convex optimization
  - Constrained nonlinear optimization
  - Other related optimization problems

- Install JuMP
  - In Julia REPL, type] for “Package Mode”
  - Type **add JuMP**



More …

Research: Publications and Prizes/Awards
https://julialang.org/research/

Julia by Example
https://juliabyexample.helpmanual.io/

Main differences to other programming languages
https://docs.julialang.org/en/v1/manual/noteworthy-differences/

Performance Tips
https://docs.julialang.org/en/v1/manual/performance-tips/

Creating a binary from Julia code
https://julialang.github.io/PackageCompiler.jl/dev/devdocs/binaries_part_2/

Finite Element Method code
http://www.juliafem.org/

Material Point Method codes
https://github.com/vinhphunguyen/MPM-Julia [out of date]
https://github.com/pxl-th/MPM

Conclusions

Matlab vs. Julia vs. Python

https://tobydriscoll.net/blog/matlab-vs.-julia-vs.-python/

- I’ve used MATLAB for over 25 years. Knowing MATLAB has been very good to my career.
- However, it’s impossible to ignore the rise of Python in scientific computing.
- Julia has the advantages and disadvantages of being a latecomer.
- I applaud the Julia creators for thinking they could do better and, to a great extent, I believe they have succeeded.

MATLAB is the corporate solution, especially for engineering. It’s probably still the easiest to learn for basic numerical tasks. Meticulous documentation and decades of contributed learning tools definitely matter.

MATLAB is the BMW sedan of the scientific computing world. It’s expensive, and that’s before you start talking about accessories (toolboxes). You’re paying for a rock-solid, smooth performance and service. It also attracts a disproportionate amount of hate.

Python is a Ford pickup. It’s ubiquitous and beloved by many (in the USA). It can do everything you want, and it’s built to do some things that other vehicles can’t. Chances are you’re going to want to borrow one now and then. But it doesn’t offer a great pure driving experience.

Julia is a Tesla. It’s built with an audacious goal of changing the future, and it might. It may also become just a footnote. But in the meantime you’ll get where you are going in style, and with power to spare.

Questions? Comments?

The Julia Programming Language

First steps: Introduction, Installation and Examples

using Julia v1.5.3

Ricardo A. Fernandes

ricardoaf@lccv.ufal.br

Advisor: Adeildo S. Ramos Jr.

January, 2021