NTNU

Norwegian University of Science and Technology

Introduction to julia

Presentation and Workshop

Ronny Bergmann

Julia Users Group Trondheim
and
Department of Mathematical Sciences, NTNU.

Trondheim, March 20, 2025.

Overview

Norwegian University of Science and Technology

What is Julia?

Goal: Scientific Computing & Fast Prototyping

In scientific computing we need

- high performance to tackle large scale problems
  - $\Rightarrow$ compiled languages (C/C++, Rust)
  - all types are known at compile time
  - static, hence maybe missing flexibility

Goal: Scientific Computing & Fast Prototyping

In scientific computing we need

- high performance to tackle large scale problems
  - $\Rightarrow$ compiled languages (C/C++, Rust)
  - all types are known at compile time
  - static, hence maybe missing flexibility
- high-level dynamic languages (like Python, Matlab, R)
  - $\Rightarrow$ fast prototyping
  - types have to be *inferred* at runtime
  - code is interpreted (slow)

Goal: Scientific Computing & Fast Prototyping

In scientific computing we need

- high performance to tackle large scale problems
  - ⇒ compiled languages (C/C++, Rust)
  - all types are known at compile time
  - static, hence maybe missing flexibility
- high-level dynamic languages (like Python, Matlab, R)
  - ⇒ fast prototyping
  - types have to be *inferred* at runtime
  - code is interpreted (slow)

Often: Fast code is written in C/C++ and is interfaced.

⇒ new users might have to compile the C/C++ (e.g. MEX files)

Combine both: Julia!

Julia is

- dynamic with type inference
- just-in-time (JIT) compiled
- focusses on high-level numerical computing

Combine both: Julia!

Julia is

- dynamic with type inference
- just-in-time (JIT) compiled
- focusses on high-level numerical computing

A short history

2009 Adam Edelman starts the project with
Jeff Bezanson, Stefan Karpinski, Viral B. Shah

2012 first public version

2018 Julia 1.0, i.e. no breaking releases since then

2024 Julia 1.11

Resources

Main homepage `https://julialang.org`

Documentation `https://docs.julialang.org/en/v1/`

Modern Julia Workflows `https://modernjuliaworkflows.org/`

Discourse `https://discourse.julialang.org`

JuliaHub webfrontend for the General Registry
`https://juliahub.com/ui/Packages`

These slides
`https://github.com/Julia-Users-Trondheim/Intro-to-Julia/blob/main/presentation/introduction-to-julia.pdf`

NTNU

Norwegian University of Science and Technology

Installation & REPL

Installation

Windows Install Julia from the Microsoft Store by running this in the command prompt

```
winget install julia -s msstore
```

Mac OS / Linux run the installer for example by

```
curl -fsSL https://install.julialang.org | sh
```

...or install `juliaup` via your favourite package manager

We can take a closer look at your individual installation after this presentation in the workshop.

Read-Eval-Print Loop (REPL)

The Julia command line is called REPL.

- for fast computations
- easily define variables & functions
- `include("script.jl");` to run a script.

Read-Eval-Print Loop (REPL)

The Julia command line is called REPL.

- for fast computations
- easily define variables & functions
- `include("script.jl");` to run a script.

Quick commands

^D Quit

^L Clear console screen

Up Arrow last command

REPL modes

Starting with special characters on REPL enters specific modes

? help mode
quick access to the documentation of a function

Example:
`? sqrt` displays the help for the `sqrt` function on REPL,
see also the (HTML) documentation
`https://docs.julialang.org/en/v1/base/math/#Base.sqrt-Tuple{Number}`

] package mode
quick access to manage packages

; shell mode
quick access to shell without exiting Julia,
e. g. to change folders

NTNU

Norwegian University of Science and Technology

Main features

NTNU

General philosophy & Code format

Philosophy

- Write functions not scripts
- Julia has data types, but not objects
- write generic code “acting” on data
- no need to write “vectorized code”
- avoid global variables

General philosophy & Code format

Philosophy

- Write functions not scripts
- Julia has data types, but not objects
- write generic code “acting” on data
- no need to write “vectorized code”
- avoid global variables

Format

- blocks have an `end`
- Indentation with 4 spaces is recommended but not necessary
- functions that modify their data should be named with an `!`.

Prequel: Packages & Pluto Notebooks

A Package is a module (namespace) providing additional functionality.

- To install one for our demos use the package mode

  ```
  ] add Pluto
  ```

  This has only to be done once.

Prequel: Packages & Pluto Notebooks

A Package is a module (namespace) providing additional functionality.

- To install one for our demos use the package mode

  ```
  ] add Pluto
  ```

  This has only to be done once.
- To load a package after starting Julia, use the `using` keyword

  ```
  using Pluto
  ```

Prequel: Packages & Pluto Notebooks

A Package is a module (namespace) providing additional functionality.

- To install one for our demos use the package mode

  ```
  ] add Pluto
  ```

  This has only to be done once.
- To load a package after starting Julia, use the `using` keyword

  ```
  using Pluto
  ```
- we can call a function from the package always by

  ```
  Pluto.run()
  ```

Prequel: Packages & Pluto Notebooks

A Package is a module (namespace) providing additional functionality.

- To install one for our demos use the package mode
  ```
  ] add Pluto
  ```
  This has only to be done once.
- To load a package after starting Julia, use the `using` keyword
  ```
  using Pluto
  ```
- we can call a function from the package always by
  ```
  Pluto.run()
  ```
- the last two can be done in one line, when using `;` as a divider
  ```
  using Pluto; Pluto.run()
  ```

Prequel: Packages & Pluto Notebooks

A Package is a module (namespace) providing additional functionality.

- To install one for our demos use the package mode

  ```
  ] add Pluto
  ```

  This has only to be done once.
- To load a package after starting Julia, use the `using` keyword

  ```
  using Pluto
  ```
- we can call a function from the package always by

  ```
  Pluto.run()
  ```
- the last two can be done in one line, when using ; as a divider

  ```
  using Pluto; Pluto.run()
  ```

Prequel: Packages & Pluto Notebooks

A Package is a module (namespace) providing additional functionality.

- To install one for our demos use the package mode
  ```
  ] add Pluto
  ```
  This has only to be done once.
- To load a package after starting Julia, use the `using` keyword
  ```
  using Pluto
  ```
- we can call a function from the package always by
  ```
  Pluto.run()
  ```
- the last two can be done in one line, when using ; as a divider
  ```
  using Pluto; Pluto.run()
  ```

We will continue command demos in the Pluto notebook
(similar to a Jupyter notebook, but with a persistent state)

Control flow I: for & while

Iterate with for-loops

```
for i=1:4
    print(i," ")
end # prints "1 2 3 4"
```

Control flow I: for & while

Iterate with for-loops

```
for i=1:4
    print(i," ")
end # prints "1 2 3 4"
```

Combine several (and use ∈)

```
for i ∈ 1:3, j ∈ 1:2
    print(i,"×",j,", ")
end # prints 1×1, 1×2, ...
```

Control flow I: for & while

Iterate with for-loops

```
for i=1:4
    print(i," ")
end # prints "1 2 3 4"
```

Combine several (and use ∈)

```
for i ∈ 1:3, j ∈ 1:2
    print(i,"×",j,", ")
end # prints 1×1, 1×2, ...
```

Or through several of same length

```
for (i,j) ∈ zip(1:4, 5:8)
    print(i,"|",j," ")
end # prints 1|5 2|6 3|7 4|8
```

Control flow I: for & while

Iterate with for-loops

```
for i=1:4
    print(i," ")
end # prints "1 2 3 4"
```

Combine several (and use ∈)

```
for i ∈ 1:3, j ∈ 1:2
    print(i,"×",j,", ")
end # prints 1×1, 1×2, ...
```

Or through several of same length

```
for (i,j) ∈ zip(1:4, 5:8)
    print(i,"|",j," ")
end # prints 1|5 2|6 3|7 4|8
```

or as a comprehension for vectors

```
x = [3*s for s ∈ 1:3 ]
```

creates `[3, 6, 9]`

Loops with "unknown end"

```
i = 1;
# do as long as i <= 4
while i <= 4
    print(i," ");
    i += 1
end # also prints "1 2 3 4"
```

NTNU

Control flow II: Conditionals

Conditionals require an expression that evaluates to a `Bool`. Then

```
if (x > 3) || (z < 2) # brackets (x > 3) are optional
    print("x is at least 3")
else
    print("x is 3 or less")
end
```

Control flow II: Conditionals

Conditionals require an expression that evaluates to a `Bool`. Then

```
if (x > 3) || (z < 2) # brackets (x > 3) are optional
    print("x is at least 3")
else
    print("x is 3 or less")
end
```

There is lazy evaluation: the second parts of

```
 (x > 4) && print("x > 4")
(x <= 4) || print("x > 4")
```

are only called/evaluated if $x > 4$.

Control flow II: Conditionals

Conditionals require an expression that evaluates to a `Bool`. Then

```
if (x > 3) || (z < 2) # brackets (x > 3) are optional
    print("x is at least 3")
else
    print("x is 3 or less")
end
```

There is lazy evaluation: the second parts of

```
 (x > 4) && print("x > 4")
(x <= 4) || print("x > 4")
```

are only called/evaluated if $x > 4$.

Conditionals can be used inline with

```
y = (x > 4) ? 1 : 3*x
```

Defining functions

```
"""
    phase(z)

Compute the phase of a complex number z
"""
function phase(z)
    return atan(imag(z), real(z))
end
```

- naming convention `snake_case`

Defining functions

```
"""
    phase(z)

Compute the phase of a complex number z
"""
function phase(z)
    return atan(imag(z), real(z))
end
```

- naming convention `snake_case`
- (multiline) `"String"` upfront: doc-string, may use Markdown

Defining functions

```
"""
    phase(z)

Compute the phase of a complex number z
"""
function phase(z)
    return atan(imag(z), real(z))
end
```

- naming convention `snake_case`
- (multiline) `"String"` upfront: doc-string, may use Markdown
- specify type with `z::Number` (but avoid overtyping like `::Float64`)

Defining functions

```julia
"""
    phase(z)

Compute the phase of a complex number z
"""
function phase(z)
    return atan(imag(z), real(z))
end
```

- naming convention `snake_case`
- (multiline) `"String"` upfront: doc-string, may use Markdown
- specify type with `z::Number` (but avoid overtyping like `::Float64`)
- (last) `return` optional, but reommended
  (last evaluated expression returned)

Defining functions

```julia
"""
    phase(z)

Compute the phase of a complex number z
"""
function phase(z)
    return atan(imag(z), real(z))
end
```

- naming convention `snake_case`
- (multiline) `"String"` upfront: doc-string, may use Markdown
- specify type with `z::Number` (but avoid overtyping like `::Float64`)
- (last) `return` optional, but reommended (last evaluated expression returned)

Defining functions

```
"""
    phase(z)

Compute the phase of a complex number z
"""
function phase(z)
    return atan(imag(z), real(z))
end
```

- naming convention `snake_case`
- (multiline) `"String"` upfront: doc-string, may use Markdown
- specify type with `z::Number` (but avoid overtyping like `::Float64`)
- (last) `return` optional, but reommended

  (last evaluated expression returned)

Shorter form

```
magnitude(z) = sqrt(imag(z)^2+real(z)^2)
```

More on functions

- functions are first-class objects (like variables)

More on functions

- functions are first-class objects (like variables)
- anonymous function `(x,y) -> x^y` e.g. to pass as parameter

More on functions

- functions are first-class objects (like variables)
- anonymous function `(x,y) -> x^y` e.g. to pass as parameter
- Broadcast: apply `phase(z)` to a whole vector

  `Z = [1.0im, 2.0, 1.0 + 0.2im]`

  by adding a `.` after the function name: `phase.(Z)`

More on functions

- functions are first-class objects (like variables)
- anonymous function `(x,y) -> x^y` e.g. to pass as parameter
- Broadcast: apply `phase(z)` to a whole vector

  ```
  Z = [1.0im, 2.0, 1.0 + 0.2im]
  ```

  by adding a `.` after the function name: `phase.(Z)`
- broadcast with multiple vectors

  ```
  X = [0.1, 0.2, 0.3]; Y = [1.0, 2.0, 3.0]
  X.^Y # same: [X[i]^Y[i] for i=1:3] or [0.1, 0.04, 0.027]
  ```

More on functions

- functions are first-class objects (like variables)
- anonymous function `(x,y) -> x^y` e.g. to pass as parameter
- Broadcast: apply `phase(z)` to a whole vector

  ```
  Z = [1.0im, 2.0, 1.0 + 0.2im]
  ```

  by adding a `.` after the function name: `phase.(Z)`
- broadcast with multiple vectors

  ```
  X = [0.1, 0.2, 0.3]; Y = [1.0, 2.0, 3.0]
  X.^Y # same: [X[i]^Y[i] for i=1:3] or [0.1, 0.04, 0.027]
  ```
- functions can modify their input

  ```
  function add_scalar!(X, v)
      X .+= v  # X an array, v a scalar: add to every entry
      return X # the X we got passed is now changed
  end
  ```

  Convention: such a functions name ends in `!`, it returns the modified

Data structures

Multiple Dispatch

Operators are Functions

Functors

Scripts

TLDR: Main differences to Python

- `for`, `if`, `while` etc. blocks are terminated by `end`
- indentation is nice, but not mandatory
- Julia is 1-indexed
- Strings have single `"quotation marks"`, multiline strings three

TLDR: Main differences to Python

- `for`, `if`, `while` etc. blocks are terminated by `end`
- indentation is nice, but not mandatory
- Julia is 1-indexed
- Strings have single `"quotation marks"`, multiline strings three
- loops amd vectors are fast (no need for vectorized code)
- abstract arrays allow arbitrary indexing ⇒ `a[-1]` is in Julia `a[end-1]`
- Julias range `1:5` includes the end and has the general form `start:step:stop` (instead of `start:(stop+1):step`)
- the imaginary unit is `im` (not `j`)

TLDR: Main differences to Python

- `for`, `if`, `while` etc. blocks are terminated by `end`
- indentation is nice, but not mandatory
- Julia is 1-indexed
- Strings have single `"quotation marks"`, multiline strings three
- loops amd vectors are fast (no need for vectorized code)
- abstract arrays allow arbitrary indexing ⇒ `a[-1]` is in Julia `a[end-1]`
- Julias range `1:5` includes the end and has the general form `start:step:stop` (instead of `start:(stop+1):step`)
- the imaginary unit is `im` (not `j`)
- Matrix multiplication is `A * B`, element wise multiplication `A .* B`
- Julia has no objects/classes

TLDR: Main differences to R

- 'single' quotation marks are for characters
- vectors are constructed with square brackets `v = [1,2,3]`
- operations on vectors of different length are not allowed
- `<-`, `<<-` and `->` are not assignment operators
- `->` creates an anonymous function

TLDR: Main differences to R

- 'single' quotation marks are for characters
- vectors are constructed with square brackets `v = [1,2,3]`
- operations on vectors of different length are not allowed
- `<-`, `<<-` and `->` are not assignment operators
- `->` creates an anonymous function
- matrix multiplication is just `A * B`
- function arguments are not copied when calling a function
- `1:5` is an `AbstractRange`, use `collect(1:5)` to create the vector

TLDR: Main differences to R

- `'single'` quotation marks are for characters
- vectors are constructed with square brackets `v = [1,2,3]`
- operations on vectors of different length are not allowed
- `<-`, `<<-` and `->` are not assignment operators
- `->` creates an anonymous function
- matrix multiplication is just `A * B`
- function arguments are not copied when calling a function
- `1:5` is an `AbstractRange`, use `collect(1:5)` to create the vector
- you do not need vectorization for performance
- logical indexing: in R `x[x>3]` has two alternatives in Julia
  - `x[ x .> 3]` (uses a temporary vector memory)
  - `filter(z->z>3, x)` might be nicer to read
  - `filter!(z->z>3, x)` updates `x` inplace (avoids the temporary memory)

TLDR: Main differences to Matlab

- array indexing uses square brackets `A[i,j]`
- Arrays are not copied by default `A=B` references the same, do `A=copy(B)` for an actual copy
- *similarly* function arguments are references, input variables can be modified
- 1-dimensional vectors exist and are not `Nx1` matrices
- `42` is an integer, not a float, use `42.0` for the float.
- `A == B` does not return a matrix of booleans but `true` or `false` use `A .== B` to get such a matrix
- dimensions are not “constant-broadcasted”:
  - `[1:10] + [1:10]'` creates a 10×10 matrix in Matlab
  - `[1:10] + [1:10]'` is a dimension mismatch, because a column vector can not be added to a row vector

NTNU

Norwegian University of Science and Technology

Packages

Installing & Using Pacakges

Package versions & Updating

Package environments

NTNU

Norwegian University of Science and Technology

Pluto Notebooks

Pluto.jl – Motivation

Similarities & differentes to Jupyter

Live Demo

Thanks for your attention!

Are there (further) questions?

NTNU

Norwegian University of Science and Technology

Workshop: Let’s get you started with Julia!