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

NTNU

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

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.

NTNU

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

Main features

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"
```

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

```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

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`)

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)

NTNU

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)

Shorter form

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

More on functions I: positional and keyword args

- positional optional parameters are defined by providing defaults

  ```
  f(a, b=2, c=3) = a*exp(b/c)
  f(1) #equals f(1,2,3)
  f(1,3) #equals f(1,3,3)
  f(1,3,5) #equals f(1,3,5)
  ```

- short to write, but to set `c`, you always have to provide `b`

More on functions I: positional and keyword args

- positional optional parameters are defined by providing defaults

```
f(a, b=2, c=3) = a*exp(b/c)
f(1) #equals f(1,2,3)
f(1,3) #equals f(1,3,3)
f(1,3,5) #equals f(1,3,5)
```

- short to write, but to set `c`, you always have to provide `b`
- keyword arguments are provided after a `;`

```
g(a; b=2, c=3) = a*exp(b/c)
g(1; b=3) #equals g(1; b=3, c=3)
g(1; c=5) #equals g(1; b=2, c=5)
```

- name has to be specified to set a value, order is not important.

More on functions I: positional and keyword args

- positional optional parameters are defined by providing defaults

```
f(a, b=2, c=3) = a*exp(b/c)
f(1) #equals f(1,2,3)
f(1,3) #equals f(1,3,3)
f(1,3,5) #equals f(1,3,5)
```

- short to write, but to set `c`, you always have to provide `b`
- keyword arguments are provided after a `;`

```
g(a; b=2, c=3) = a*exp(b/c)
g(1; b=3) #equals g(1; b=3, c=3)
g(1; c=5) #equals g(1; b=2, c=5)
```

- name has to be specified to set a value, order is not important.
- in `g(1; b=4, b=3)` the last one “wins”, so `b` is 3.

More on functions I: positional and keyword args

- positional optional parameters are defined by providing defaults

```
f(a, b=2, c=3) = a*exp(b/c)
f(1) #equals f(1,2,3)
f(1,3) #equals f(1,3,3)
f(1,3,5) #equals f(1,3,5)
```

- short to write, but to set `c`, you always have to provide `b`
- keyword arguments are provided after a `;`

```
g(a; b=2, c=3) = a*exp(b/c)
g(1; b=3) #equals g(1; b=3, c=3)
g(1; c=5) #equals g(1; b=2, c=5)
```

- name has to be specified to set a value, order is not important.
- in `g(1; b=4, b=3)` the last one "wins", so `b` is 3.
- You can "collect and pass on":
  - `h1(args...) = f(1, args...)`
  - `h2(; kwargs...) = g(1; kwargs...)`
  - or combine both as `h3(args...; kwargs...) = #def here`

More on functions II: broadcast and mutation

- functions are first-class objects (like variables)

More on functions II: broadcast and mutation

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

More on functions II: broadcast and mutation

- 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 II: broadcast and mutation

- 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 II: broadcast and mutation

- 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

There are abstract types to build a type hierarchy.

```
abstract type ExperimentData end
```

Data structures

There are abstract types to build a type hierarchy.

```
abstract type ExperimentData end
```

naming convention:
Types are `CamelCase`.

Data structures

There are abstract types to build a type hierarchy.

```
abstract type ExperimentData end
```

Variant I. default: immutable

```
struct TimeSeries <: ExperimentData
    name::String
    data::Vector
end # default constructor:
ts = TimeSeries("A", [1,2,3])
```

naming convention:
Types are `CamelCase`.

Data structures

There are abstract types to build a type hierarchy.

```
abstract type ExperimentData end
```

naming convention:
Types are `CamelCase`.

Variant I. default: immutable

```
struct TimeSeries <: ExperimentData
    name::String
    data::Vector
end # default constructor:
ts = TimeSeries("A", [1,2,3])
```

- fields can not be (ex)changed: `ts.name="B"` and `ts.data=[4,5]` error.

Data structures

There are abstract types to build a type hierarchy.

```
abstract type ExperimentData end
```

naming convention:
Types are `CamelCase`.

Variant I. default: immutable

```
struct TimeSeries <: ExperimentData
    name::String
    data::Vector
end # default constructor:
ts = TimeSeries("A", [1,2,3])
```

- fields can not be (ex)changed: `ts.name="B"` and `ts.data=[4,5]` error.
- but `ts.data[2]=4` works (modified in-place)

Data structures

There are abstract types to build a type hierarchy.

```julia
abstract type ExperimentData end
```

naming convention:
Types are `CamelCase`.

Variant I. default: immutable

```julia
struct TimeSeries <: ExperimentData
    name::String
    data::Vector
end # default constructor:
ts = TimeSeries("A", [1,2,3])
```

- fields can not be (ex)changed: `ts.name="B"` and `ts.data=[4,5]` error.
- but `ts.data[2]=4` works (modified in-place)
- more efficient

Data structures

There are abstract types to build a type hierarchy.

```julia
abstract type ExperimentData end
```

naming convention:
Types are `CamelCase`.

Variant I. default: immutable

```julia
struct TimeSeries <: ExperimentData
    name::String
    data::Vector
end # default constructor:
ts = TimeSeries("A", [1,2,3])
```

- fields can not be (ex)changed: `ts.name="B"` and `ts.data=[4,5]` error.
- but `ts.data[2]=4` works (modified in-place)
- more efficient

Data structures

There are abstract types to build a type hierarchy.

```
abstract type ExperimentData end
```

naming convention:
Types are `CamelCase`.

Variant I. default: immutable

```
struct TimeSeries <: ExperimentData
    name::String
    data::Vector
end # default constructor:
ts = TimeSeries("A", [1,2,3])
```

- fields can not be (ex)changed: `ts.name="B"` and `ts.data=[4,5]` error.
- but `ts.data[2]=4` works (modified in-place)
- more efficient

Variant II. mutable – reassign fields:

```
mutable struct Measurement <: ExperimentData
    name::String
    value::Float64
end # same constructor
m = Measurement("B", 3.1415)
```

Data structures

There are abstract types to build a type hierarchy.

```
abstract type ExperimentData end
```

naming convention:
Types are `CamelCase`.

Variant I. default: immutable

```
struct TimeSeries <: ExperimentData
    name::String
    data::Vector
end # default constructor:
ts = TimeSeries("A", [1,2,3])
```

- fields can not be (ex)changed: `ts.name="B"` and `ts.data=[4,5]` error.
- but `ts.data[2]=4` works (modified in-place)
- more efficient

Variant II. mutable – reassign fields:

```
mutable struct Measurement <: ExperimentData
    name::String
    value::Float64
end # same constructor
m = Measurement("B", 3.1415)
```

- `m.name="B"; m.value=4.5` both work (if same type)

Data structures

There are abstract types to build a type hierarchy.

```
abstract type ExperimentData end
```

naming convention:
Types are `CamelCase`.

Variant I. default: immutable

```
struct TimeSeries <: ExperimentData
    name::String
    data::Vector
end # default constructor:
ts = TimeSeries("A", [1,2,3])
```

- fields can not be (ex)changed: `ts.name="B"` and `ts.data=[4,5]` error.
- but `ts.data[2]=4` works (modified in-place)
- more efficient

Variant II. mutable – reassign fields:

```
mutable struct Measurement <: ExperimentData
    name::String
    value::Float64
end # same constructor
m = Measurement("B", 3.1415)
```

- `m.name="B"; m.value=4.5` both work (if same type)
- slightly less efficient

Paremetric types & functions

- ensure two fields have exactly the same type

Paremetric types & functions

- ensure two fields have exactly the same type
- to avoid abstract types in concrete instances (reduce performance)

Paremetric types & functions

- ensure two fields have exactly the same type
- to avoid abstract types in concrete instances (reduce performance)
- stay flexible to for new use cases

Paremetric types & functions

- ensure two fields have exactly the same type
- to avoid abstract types in concrete instances (reduce performance)
- stay flexible to for new use cases

Paremetric types & functions

- ensure two fields have exactly the same type
- to avoid abstract types in concrete instances (reduce performance)
- stay flexible to for new use cases

```
mutable struct TimeSeries2{T} <: ExperimentData
    param::T          # maybe some concentration
    data::Vector{T} # actually parametrized by element-type
end # Constructor now maybe a bit clumsy:
ts2 = TimeSeries2{Float64}(3.1415, [1.2, 1.3])
```

Paremetric types & functions

- ensure two fields have exactly the same type
- to avoid abstract types in concrete instances (reduce performance)
- stay flexible to for new use cases

```
mutable struct TimeSeries2{T} <: ExperimentData
    param::T         # maybe some concentration
    data::Vector{T} # actually parametrized by element-type
end # Constructor now maybe a bit clumsy:
ts2 = TimeSeries2{Float64}(3.1415, [1.2, 1.3])
```

- makes the previous (implicit) `Vector{Any}` to a concrete type

NTNU

Parametric types & functions

- ensure two fields have exactly the same type
- to avoid abstract types in concrete instances (reduce performance)
- stay flexible to for new use cases

```
mutable struct TimeSeries2{T} <: ExperimentData
    param::T          # maybe some concentration
    data::Vector{T} # actually parametrized by element-type
end # Constructor now maybe a bit clumsy:
ts2 = TimeSeries2{Float64}(3.1415, [1.2, 1.3])
```

- makes the previous (implicit) `Vector{Any}` to a concrete type
- nicer constructor: Define a parametric function

```
function TimeSeries2(c::T, v::Vector{T}) where {T}
    return TimeSeries2{T}(c, v)
end # Then we have back
ts2 = TimeSeries2(3.1415, [1.2, 1.3])
```

Multiple Dispatch

Dispatch: "finding" the "best fitting version" of a function.

Multiple Dispatch

Dispatch: "finding" the "best fitting version" of a function. For

```julia
f(x) = "A"
f(x::Number) = "B"
f(x::Float64) = "C"
```

NTNU

Multiple Dispatch

Dispatch: "finding" the "best fitting version" of a function. For

```
f(x) = "A"
f(x::Number) = "B"
f(x::Float64) = "C"
```

We get that

```
f.(["a", 1, 1.0im, 2.0])
```

is `["A", "B", "B", "C"]`

Multiple Dispatch

Dispatch: "finding" the "best fitting version" of a function. For

```
f(x) = "A"
f(x::Number) = "B"
f(x::Float64) = "C"
```

We get that

```
f.(["a", 1, 1.0im, 2.0])
```

is `["A", "B", "B", "C"]`

⇒ dispatch to "most fitting" method of a function

```
function g(a::Number, t::TimeSeries)
  TimeSeries(t.name, a .* t.data)
end
function g(a::String, t::TimeSeries)
  TimeSeries("$(a) $(t.name)", t.data)
end
function g(a::Number, ts::TimeSeries2)
  TimeSeries2(a*t.param, a .* t)
end
```

Multiple Dispatch

Dispatch: "finding" the "best fitting version" of a function. For

```julia
f(x) = "A"
f(x::Number) = "B"
f(x::Float64) = "C"
```

We get that

```julia
f.(["a", 1, 1.0im, 2.0])
```

is `["A", "B", "B", "C"]`

⇒ dispatch to "most fitting" method of a function

```julia
function g(a::Number, t::TimeSeries)
  TimeSeries(t.name, a .* t.data)
end
function g(a::String, t::TimeSeries)
  TimeSeries("$(a) $(t.name)", t.data)
end
function g(a::Number, ts::TimeSeries2)
  TimeSeries2(a*t.param, a .* t)
end
```

Avoid ambiguities. Defining

```julia
g(a::Float64, b) = 2*a+b
g(a, b::Float64) = a+2*b
```

makes `g(1.0,2.0)` ambiguous.

Multiple Dispatch

Dispatch: "finding" the "best fitting version" of a function. For

```julia
f(x) = "A"
f(x::Number) = "B"
f(x::Float64) = "C"
```

We get that

```julia
f.(["a", 1, 1.0im, 2.0])
```

is `["A", "B", "B", "C"]`

⇒ dispatch to "most fitting" method of a function

```julia
function g(a::Number, t::TimeSeries)
  TimeSeries(t.name, a .* t.data)
end
function g(a::String, t::TimeSeries)
  TimeSeries("$(a) $(t.name)", t.data)
end
function g(a::Number, ts::TimeSeries2)
  TimeSeries2(a*t.param, a .* t)
end
```

Avoid ambiguities. Defining

```julia
g(a::Float64, b) = 2*a+b
g(a, b::Float64) = a+2*b
```

makes `g(1.0,2.0)` ambiguous. Resolve by

```julia
g(a::Float64, b::Float64) = 2*a + 2*b
```

Operators are Functions

Operators like `+`, `*`, `^` are functions. Add a method to `+` via

```
function Base.:+(t::TimeSeries, s::TimeSeries)
    if !(length(t.data)==length(s.data))
        error("Time series not of same length")
    end
    return TimeSeries(
        "$(t.name) and $(s.name)",
        t.data .+ s.data
    )
end
```

Operators are Functions

Operators like +, *, ^ are functions. Add a method to + via

```
function Base.:+(t::TimeSeries, s::TimeSeries)
    if !(length(t.data)==length(s.data))
        error("Time series not of same length")
    end
    return TimeSeries(
        "$(t.name) and $(s.name)",
        t.data .+ s.data
    )
end
```

Then

```
u = TimeSeries("A", [1,2]) + TimeSeries("B", [3,4])
```

returns `TimeSeries("A and B", [4, 6])`.

Operators are Functions

Operators like `+`, `*`, `^` are functions. Add a method to `+` via

```
function Base.:+(t::TimeSeries, s::TimeSeries)
    if !(length(t.data)==length(s.data))
        error("Time series not of same length")
    end
    return TimeSeries(
        "$(t.name) and $(s.name)",
        t.data .+ s.data
    )
end
```

Then

```
u = TimeSeries("A", [1,2]) + TimeSeries("B", [3,4])
```

returns `TimeSeries("A and B", [4, 6])`.

To ensure same type parameter, define a function with

```
Base.:+(t::TimeSeries2{T}, s::TimeSeries2{T}) where {T}
```

Functors: function-like structures

Consider (actually taken from the Julia documentation)

```julia
struct Polynomial{R}
    coeffs::Vector{R}
end
```

Functors: function-like structures

Consider (actually taken from the Julia documentation)

```
struct Polynomial{R}
    coeffs::Vector{R}
end
```

We can turn a `Polynomial` into a function as well definiing

```
function (p::Polynomial)(x)
    v = p.coeffs[end] # Horner Schema, (a_2x + a_1)x + a_0
    for i = (length(p.coeffs)-1):-1:1
        v = v*x + p.coeffs[i]
    end
    return v
end
```

Functors: function-like structures

Consider (actually taken from the Julia documentation)

```
struct Polynomial{R}
    coeffs::Vector{R}
end
```

We can turn a `Polynomial` into a function as well definiing

```
function (p::Polynomial)(x)
    v = p.coeffs[end] # Horner Schema, (a_2x + a_1)x + a_0
    for i = (length(p.coeffs)-1):-1:1
        v = v*x + p.coeffs[i]
    end
    return v
end
```

For `p = Polynomial([1, 10, 100]); p(3)` we get
$100 \cdot 3^2 + 10 \cdot 3 + 1 = 931$

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

Packages

Namespaces & Modules

```julia
module MyModule #Same naming convention as types: CamelCase
    f(x) = x^2          # is exported
    struct MyField end # is not exported
    export f
end
```

- introduces a namespace, loaded with `using .MyModule`

  (the `.` necessary for modules defined in scripts/REPL)

Namespaces & Modules

```julia
module MyModule #Same naming convention as types: CamelCase
    f(x) = x^2          # is exported
    struct MyField end # is not exported
    export f
end
```

- introduces a namespace, loaded with `using .MyModule`
  (the `.` necessary for modules defined in scripts/REPL)
- a module can internally also use other (dependent) packages.

Namespaces & Modules

```julia
module MyModule #Same naming convention as types: CamelCase
    f(x) = x^2          # is exported
    struct MyField end # is not exported
    export f
end
```

- introduces a namespace, loaded with `using .MyModule`
  (the `.` necessary for modules defined in scripts/REPL)
- a module can internally also use other (dependent) packages.
- anything it `exports` is available in global namespace

Namespaces & Modules

```julia
module MyModule #Same naming convention as types: CamelCase
    f(x) = x^2          # is exported
    struct MyField end # is not exported
    export f
end
```

- introduces a namespace, loaded with `using .MyModule`
  (the `.` necessary for modules defined in scripts/REPL)
- a module can internally also use other (dependent) packages.
- anything it `export`s is available in global namespace
- other functions/structs via `MyModule.local_function`

Namespaces & Modules

```julia
module MyModule #Same naming convention as types: CamelCase
    f(x) = x^2          # is exported
    struct MyField end # is not exported
    export f
end
```

- introduces a namespace, loaded with `using .MyModule`
  (the `.` necessary for modules defined in scripts/REPL)
- a module can internally also use other (dependent) packages.
- anything it `export`s is available in global namespace
- other functions/structs via `MyModule.local_function`

! if two modules `A` and `B` exort `f`, one also has to use `A.f` and `B.f` or specify which one to use with `using A: f`

Namespaces & Modules

```
module MyModule #Same naming convention as types: CamelCase
    f(x) = x^2          # is exported
    struct MyField end # is not exported
    export f
end
```

- introduces a namespace, loaded with `using .MyModule`
  (the `.` necessary for modules defined in scripts/REPL)
- a module can internally also use other (dependent) packages.
- anything it **exports** is available in global namespace
- other functions/structs via `MyModule.local_function`

! if two modules `A` and `B` exort `f`, one also has to use `A.f` and `B.f` or specify which one to use with `using A: f`

- Default packages are among others Base (loaded on start) `LinearAlgebra`, `Random`, `Statistics`, ...

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`
- Shortcut: Package mode in REPL; Start command with `]`

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`
- Shortcut: Package mode in REPL; Start command with `]`
- `] add PackageName` installs a package

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`
- Shortcut: Package mode in REPL; Start command with `]`
- `] add PackageName` installs a package
  - including all packages `PackageName` depends on.

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`
- Shortcut: Package mode in REPL; Start command with `]`
- `] add PackageName` installs a package
  - including all packages `PackageName` depends on.
  - resolves versions to "fit" to all already installed ones

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`
- Shortcut: Package mode in REPL; Start command with `]`
- `] add PackageName` installs a package
  - including all packages `PackageName` depends on.
  - resolves versions to “fit” to all already installed ones
- `] status` lists all installed packages with their versions

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`
- Shortcut: Package mode in REPL; Start command with]
-] `add PackageName` installs a package
  - including all packages `PackageName` depends on.
  - resolves versions to “fit” to all already installed ones
-] `status` lists all installed packages with their versions
-] `update` update all packages to newest version

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`
- Shortcut: Package mode in REPL; Start command with `]`
- `] add PackageName` installs a package
  - including all packages `PackageName` depends on.
  - resolves versions to "fit" to all already installed ones
- `] status` lists all installed packages with their versions
- `] update` update all packages to newest version

Installing & Using Packages

- modules that come from a Registry, package manager: `Pkg.jl`
- default: `https://github.com/JuliaRegistries/General`
- Shortcut: Package mode in REPL; Start command with]
-] `add PackageName` installs a package
  - including all packages `PackageName` depends on.
  - resolves versions to “fit” to all already installed ones
-] `status` lists all installed packages with their versions
-] `update` update all packages to newest version

After a package is installed, it can be used with

```julia
using PackageName, PackageA, PackageB,
```

Package environments

- in package mode: prefix `(@v1.11) pkg>` refers to the environment: by default the (global, version based) environment
- an environment is a set of packages and their versions
- use `] activate Name` to activate a new environment
- use `] activate .` to turn the current folder into an environment.
  $\Rightarrow$ This is easy to activate for a set of scripts

Package environments

- in package mode: prefix `(@v1.11) pkg>` refers to the environment: by default the (global, version based) environment
- an environment is a set of packages and their versions
- use `] activate Name` to activate a new environment
- use `] activate .` to turn the current folder into an environment.
  - ⇒ This is easy to activate for a set of scripts
  - ⇒ reproducible: in the environment, we always have the same set of packages in their versions

Package environments

- in package mode: prefix `(@v1.11) pkg>` refers to the environment: by default the (global, version based) environment
- an environment is a set of packages and their versions
- use `] activate Name` to activate a new environment
- use `] activate .` to turn the current folder into an environment.
  - ⇒ This is easy to activate for a set of scripts
  - ⇒ reproducible: in the environment, we always have the same set of packages in their versions
  - ⇒ file `Project.toml` allows others to install that environment as well: After activation to `] instantiate` to install all packages in exactly the specified versions

Package environments

- in package mode: prefix `(@v1.11) pkg>` refers to the environment: by default the (global, version based) environment
- an environment is a set of packages and their versions
- use `] activate Name` to activate a new environment
- use `] activate .` to turn the current folder into an environment.
  - ⇒ This is easy to activate for a set of scripts
  - ⇒ reproducible: in the environment, we always have the same set of packages in their versions
  - ⇒ file `Project.toml` allows others to install that environment as well: After activation to `] instantiate` to install all packages in exactly the specified versions
  - even safer: `Manifest.toml` all packages and their dependencies in exact versions resolved

Package environments

- in package mode: prefix `(@v1.11) pkg>` refers to the environment: by default the (global, version based) environment
- an environment is a set of packages and their versions
- use `] activate Name` to activate a new environment
- use `] activate .` to turn the current folder into an environment.
  - ⇒ This is easy to activate for a set of scripts
  - ⇒ reproducible: in the environment, we always have the same set of packages in their versions
  - ⇒ file `Project.toml` allows others to install that environment as well: After activation to `] instantiate` to install all packages in exactly the specified versions
  - even safer: `Manifest.toml` all packages and their dependencies in exact versions resolved

Package environments

- in package mode: prefix `(@v1.11) pkg>` refers to the environment: by default the (global, version based) environment
- an environment is a set of packages and their versions
- use `] activate Name` to activate a new environment
- use `] activate .` to turn the current folder into an environment.
  - ⇒ This is easy to activate for a set of scripts
  - ⇒ reproducible: in the environment, we always have the same set of packages in their versions
  - ⇒ file `Project.toml` allows others to install that environment as well: After activation to `] instantiate` to install all packages in exactly the specified versions
  - even safer: `Manifest.toml` all packages and their dependencies in exact versions resolved

Overall: **Reproducible** environment / setup to run your experiments in

Pluto Notebooks

Pluto.jl – Motivation

Similarities & differentes to Jupyter

Live Demo

NTNU

Further topics

- further default data structures
  - `Dict` dictionaries
  - `NamedTuple`s as “lightweight, flexible” struct
  - `IO` reading/writing files
  - further packages from the Standard Library
- `@macro`s – rewriting code
- VS Code extension & the debugger
- specific packages for your concrete problems
- `Test.jl` and running tests on your own package
- `Documenter.jl` and creating a documentation for your own package
- package extensions and weak dependencies

Thanks for your attention!

Are there (further) questions?

Workshop: Let's get you started with Julia!

NTNU
Norwegian University of Science and Technology