INTRODUCTION TO JULIA

WHO WE ARE

Bjarke Grove Poulsen

PostDoc @ AU

Animal Scientist

Traditionally worked in R

Maintainer of the LocalAncestry.jl

Emre Karaman

Associate Professor @ AU

Data Scientist

Maintainer of NextGP.jl

TODAYS TOPICS

Julia in general

- What it tries to do
- How Julia differs from other languages

Workshop topics

- Project environments
- Simple types
- Composite types and collections
- Functions
- Data wrangling

A NOTE ON TODAY

Learning outcomes

You can use Julia for daily data handling while looking up specifics.

You have familiarity with the functionalities of Julia

You have a familiar reference that eases further learning.

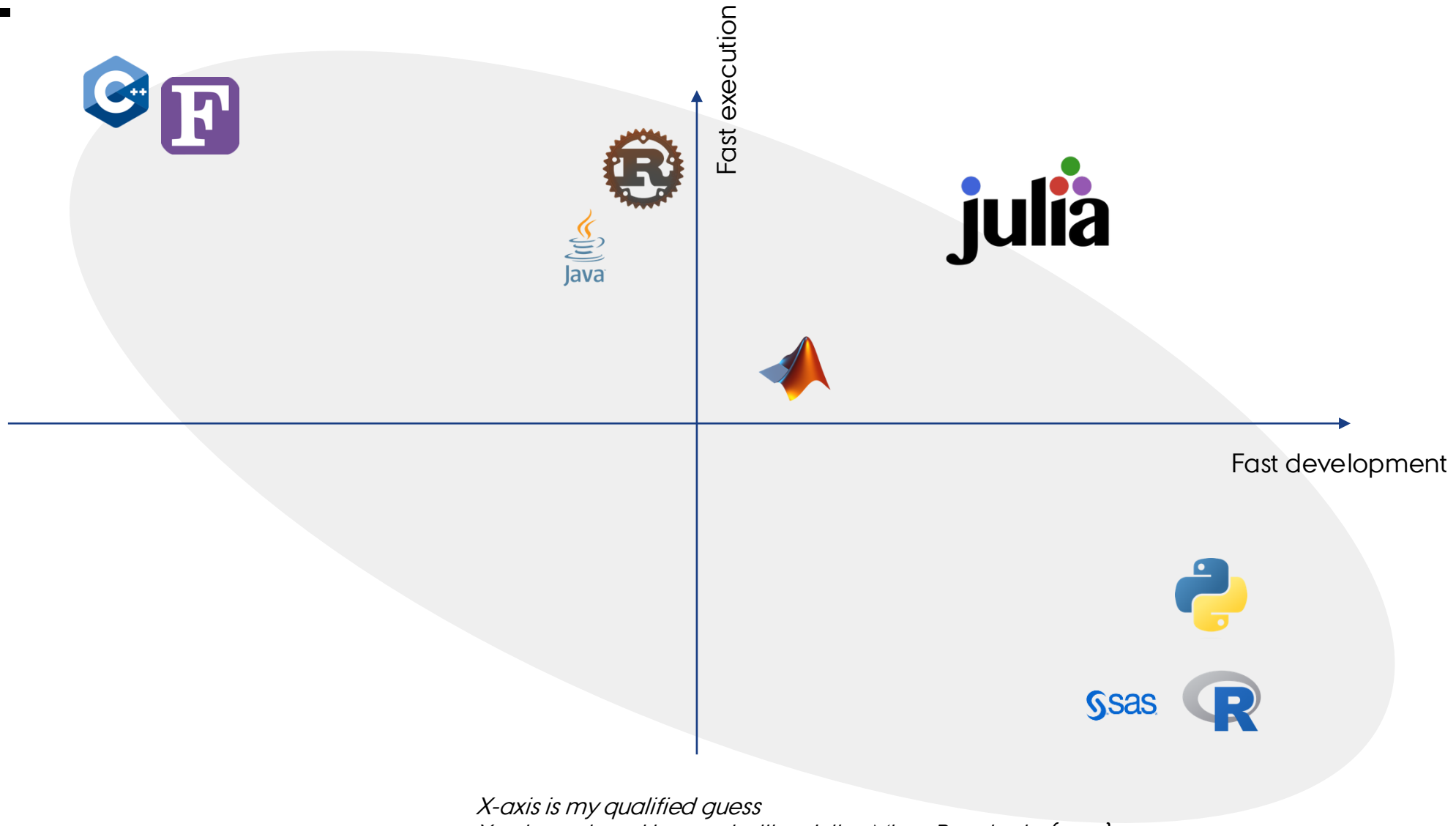
The workshop will have many exercises

Each exercise is small

Solutions is meant to be a reference for later use

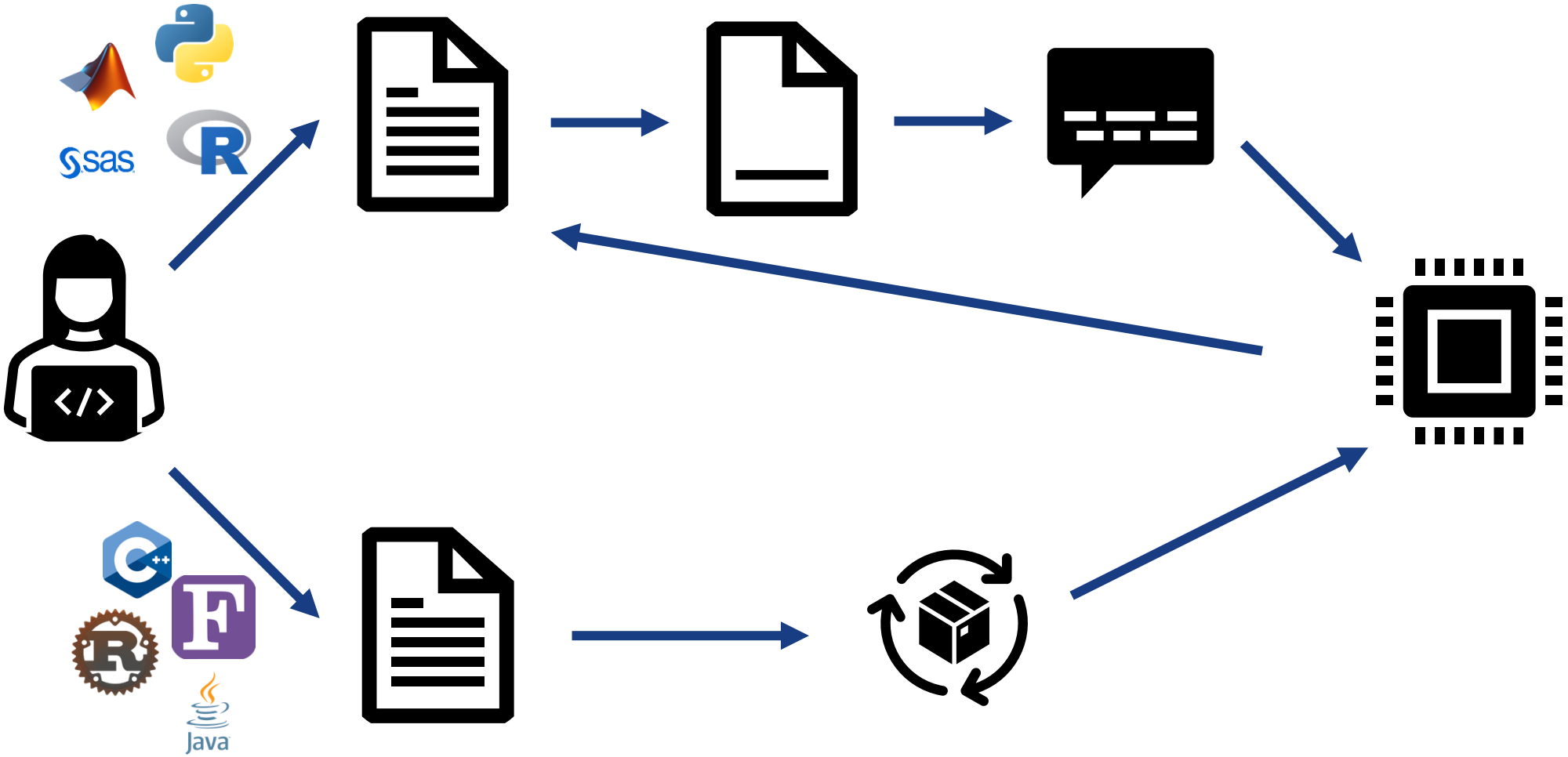
The use of generative AI is OK, if you think it improves your learning

JULIA TRIES TO SOLVE THE 2 LANGUAGE PROBLEM

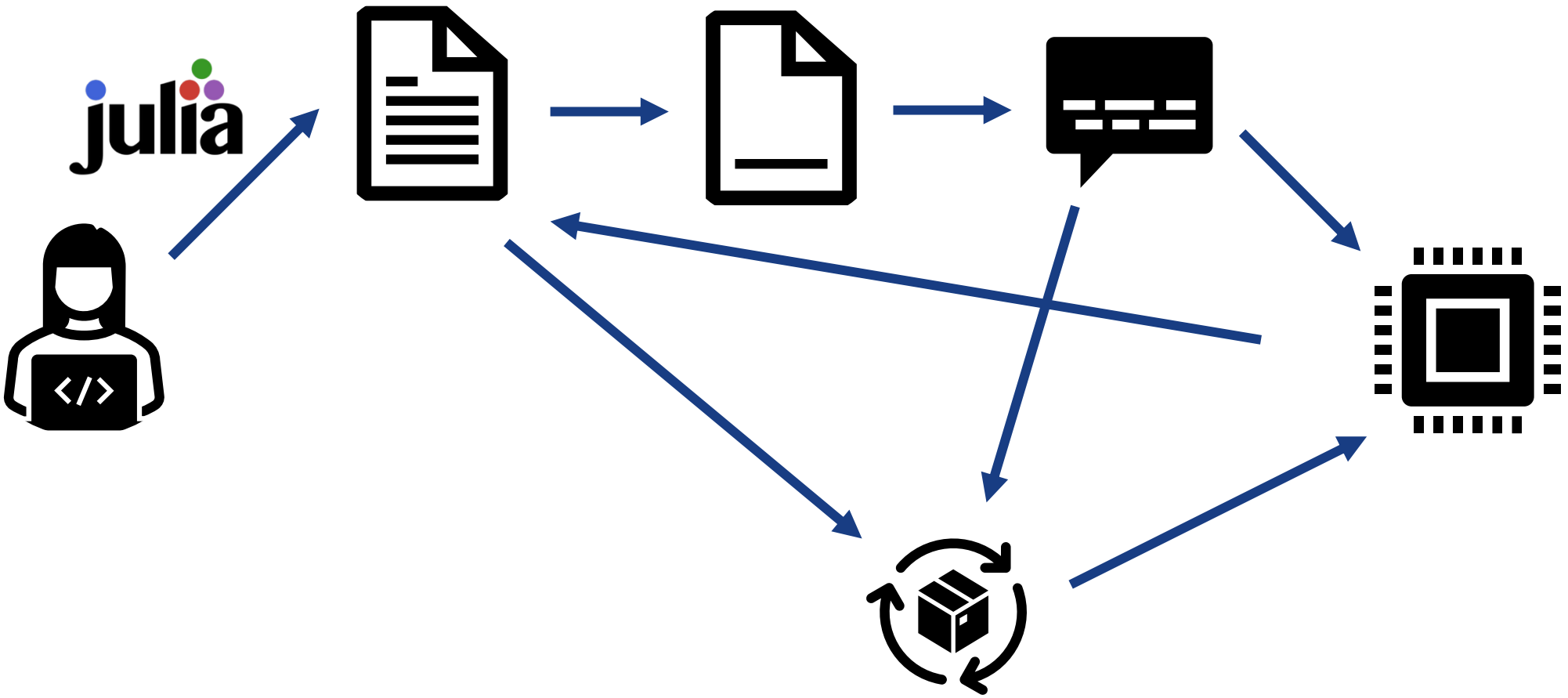


X-axis is my qualified guess
Y-axis produced by eye-balling Julias Micro-Bencharks (+sas)

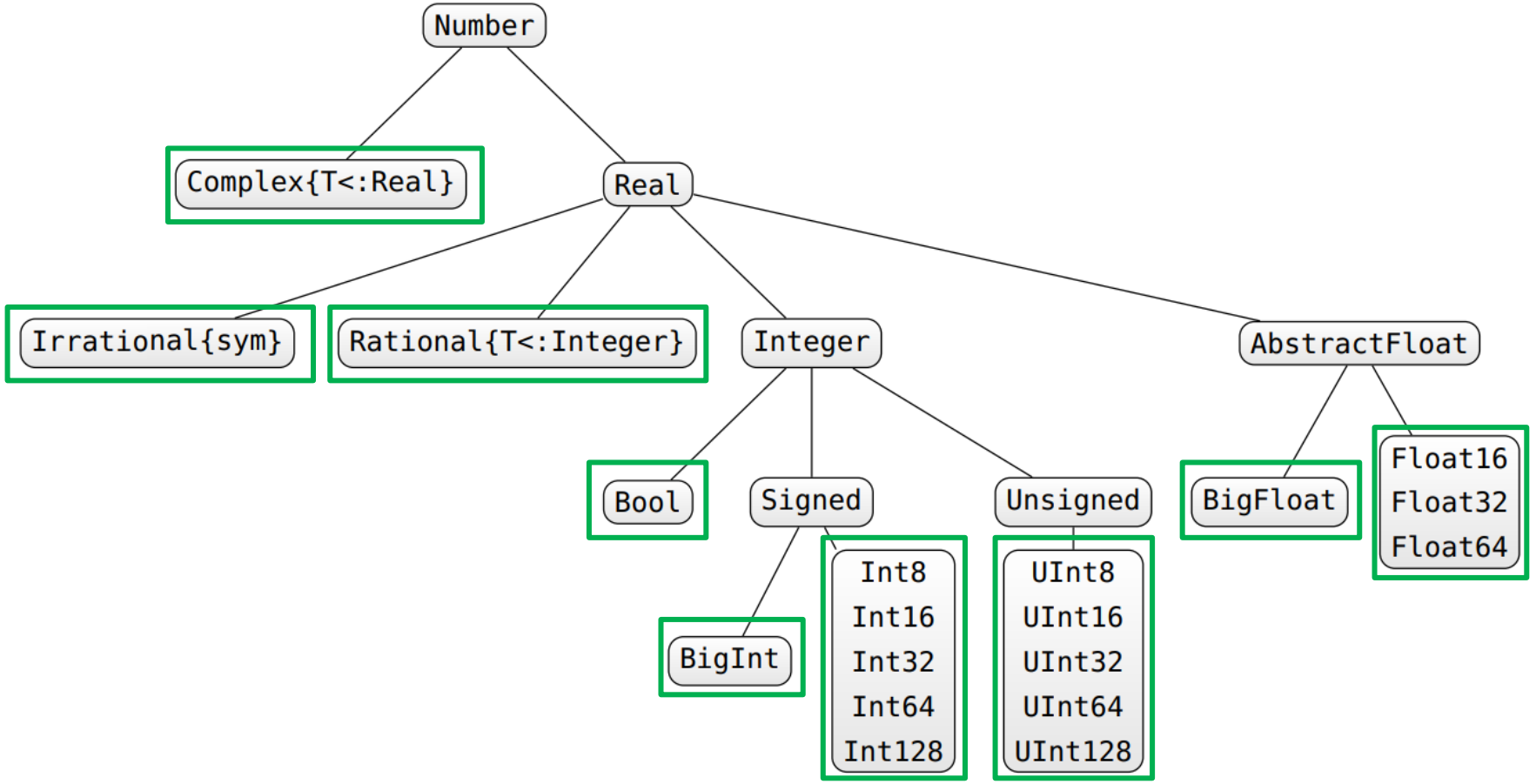
JULIA HAS JUST-IN-TIME COMPILATION



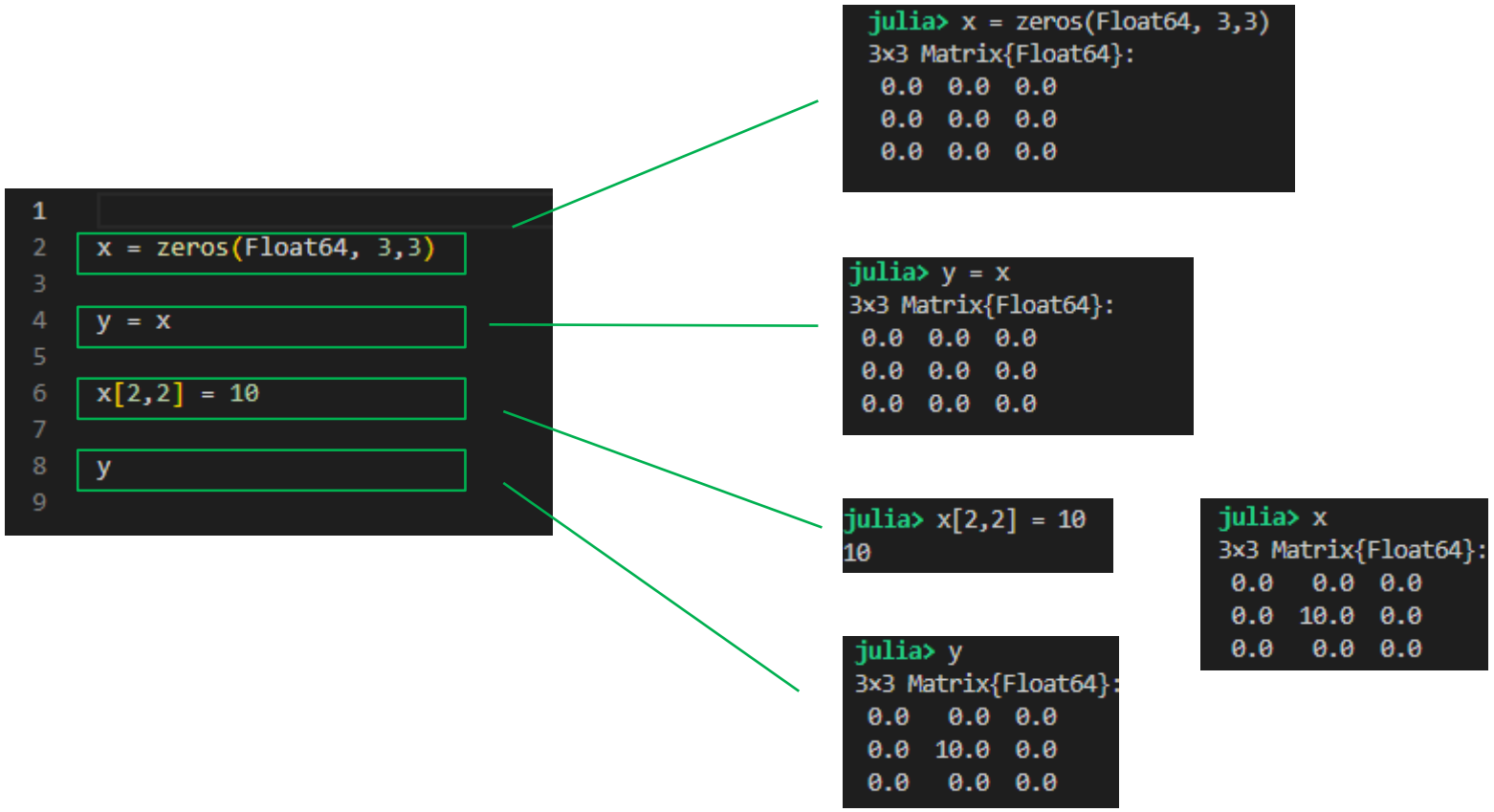
JULIA HAS JUST-IN-TIME COMPILATION



JULIA OFFERS MULTIPLE DISPATCH

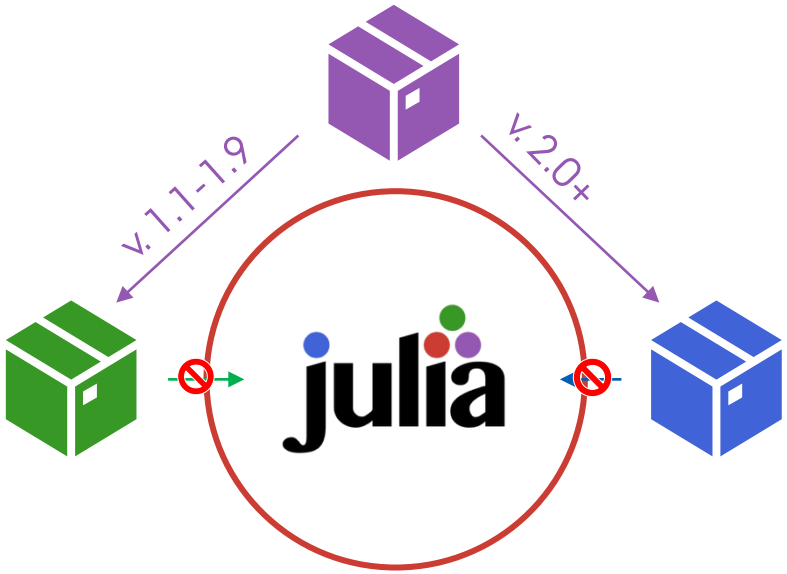


JULIA PASSES BY REFERENCE BY DEFAULT

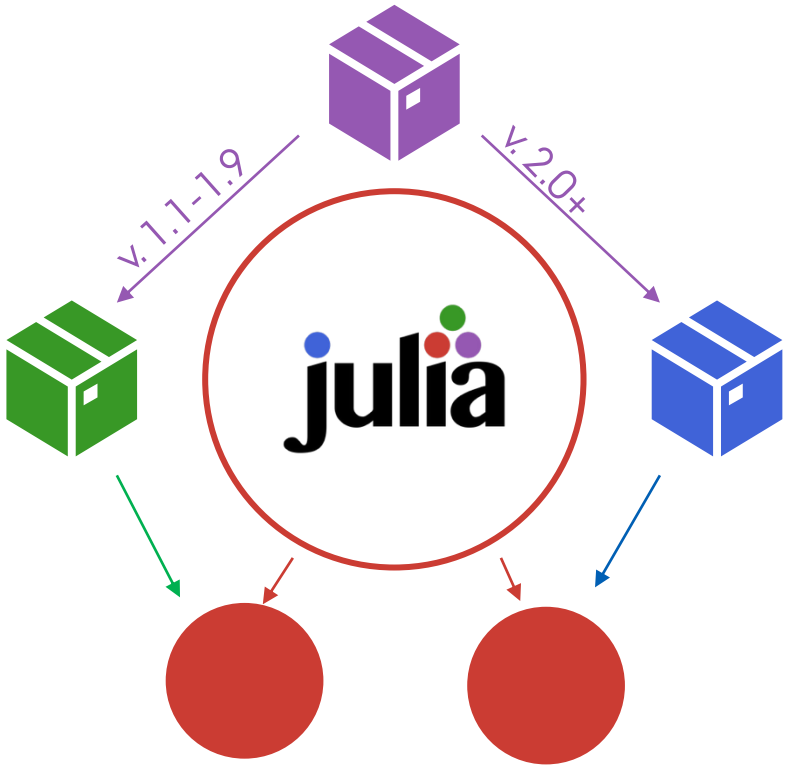


BLOCK 1/5:
PROJECT ENVIRONMENTS

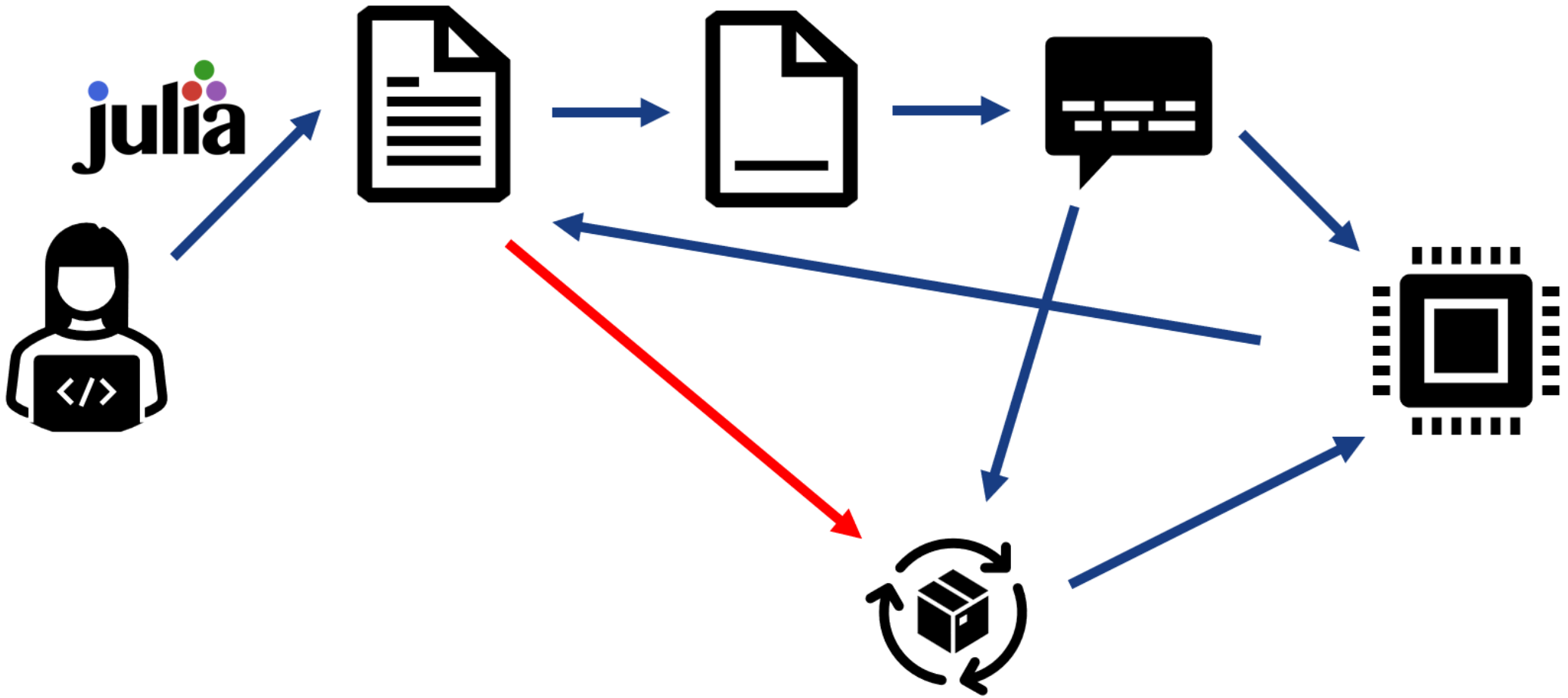
Installing packages into the global environment



Installing into project environments



THEY SPEED UP START-UP TIMES



HOW TO START PROJECT ENVIRONMENTS

```
$ julia --project=.
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.12.5 (2026-02-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

(NADAS Julia) pkg>
```

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.12.5 (2026-02-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

julia> using Pkg

julia> Pkg.activate(".")
  Activating project at `C:\Users\au488376\OneDrive - Aarhus universit

(NADAS Julia) pkg>
```

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.12.5 (2026-02-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

(@v1.12) pkg> activate .
  Activating project at `C:\Users\au488376\OneDrive - Aarhus universit

(NADAS Julia) pkg>
```

WHY YOU SHOULD USE THEM

Separation of packages across projects

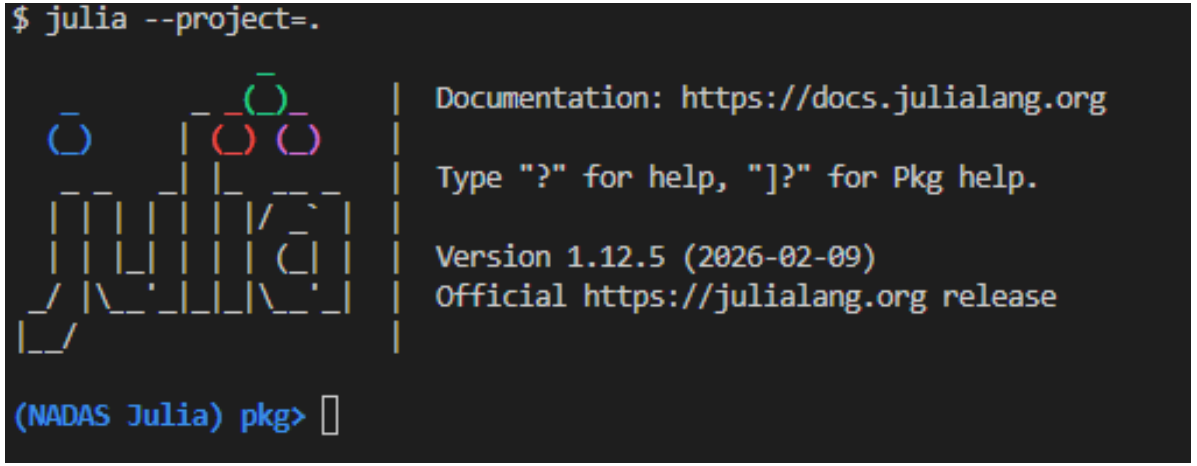
- Less risk of conflicts between packages
- Faster start-up

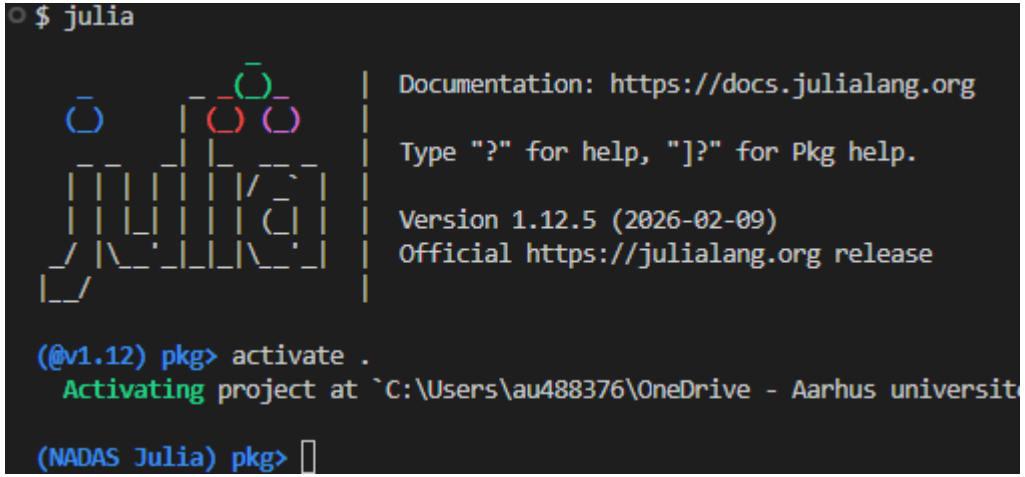
Not a new concept

- Python: virtual environment
- R: renv

What you need to know

1. How to make, use, move, remove





BLOCK 2/5: SIMPLE TYPES

THE BUILDING BLOCKS OF MOST OBJECTS

Primitive types

Data type	Julia names	Memory size (bytes)	Example
Signed integers	Int8-128	1-16	1
Unsigned integers	UInt8-128	1-16	0x0000000000000001
Decimal numbers	Float8-64	1-8	1.0
Characters	Char	4	'1'
Boolean	Bool	1	true

Notes

All have exact bit-representations

One byte gives large range – often no need to overcomplicate typing

Most conversions are intuitive

INTUITIVE CONVERSION

From integer to decimal point

```
julia> x::Int = 1
1
julia> Float64(x)
1.0
```

From boolean to integer

```
julia> z::Bool = true
true
julia> Int(z)
1
```

From decimal point to integer

```
julia> y::Float64 = 1.0
1.0
julia> Int(y)
1
```

From boolean to integer

```
julia> x::Int = 1
1
julia> Bool(1)
true
```

MORE INTUITIVE CONVERSION

From integer to unsigned integer and back

```
julia> u1 = UInt(10)
0x000000000000000a

julia> u2 = UInt(100)
0x0000000000000064

julia> u3 = u1*u2
0x00000000000003e8

julia> Int(u3)
1000
```

LESS INTUITIVE CONVERSION

From integer to character

```
julia> x::Int = 49
49

julia> Char(x)
'1': ASCII/Unicode U+0031 (category N
```

From character to integer (parsed)

```
julia> parse(Int, '1')
1
```

010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:

kaaDBqX4&t=119s

STRINGS ARE JUST CHARACTERS

Strings are very useful

- Human-readable
- Can be concatenated, split, altered
- Uses less memory than a collection of characters (approx. 1/4)

Downsides

- Strings can have any size (Fixed-size string exist)
- Changing a character in a string remakes the whole string
- Can be slower than simpler datatypes

STRING OPERATIONS

Concatenate two strings

```
julia> "abc" * "def"
"abcdef"

julia> join(["abc", "def"], "")
"abcdef"
```

Split a string

```
julia> split("abcXdef", "X")
2-element Vector{SubString{String}}:
 "abc"
 "def"
```

Replace characters in a string

```
julia> replace("abcXef", "X" => "d")
"abcdef"
```

Replace characters using regex

```
julia> replace("abcXsfa221ef", r"X.+1" => "d")
"abcdef"
```

MORE STRING OPERATIONS

Find characters in a string

```
julia> occursin("abc","abcdef")
true

julia> findfirst("abc", "abcdef")
1:3
```

EXERCISES

BLOCK 3/5: COLLECTIONS

COLLECTIONS IN GENERAL

Tuples: ***(1,'a', "a")*** **is a Tuple{Int64, Char, String}**

Immutable object with positional, but not named elements

Named tuples: ***(x1 = 1, x2 = 'a', x3 = "a")*** **is a NamedTuple{x1::Int64, x2::Char, x3::String}**

Immutable object with positional and named elements

Arrays: ***[1,2,3]*** **is a Vector{Int64} (alias for Array{Int64, 1})**

Mutable object with positional elements (multidimensional)

Dictionaries: ***Dict(['a','b'] .=> [1,2])*** **is a Dict{Char, Int64} with 2 entries: 'a' => 1, 'b' => 2**

Mutable object with named elements.

TUPLES AND NAMED TUPLES

Accessing

Individual access: `(1,2,3)[1]` `(1,2,3)[1:2]`

Unpacking

Explicit unpacking: *a,b = (1,2)* (tedious)

Using the @unpack macro (outside the scope of this course)

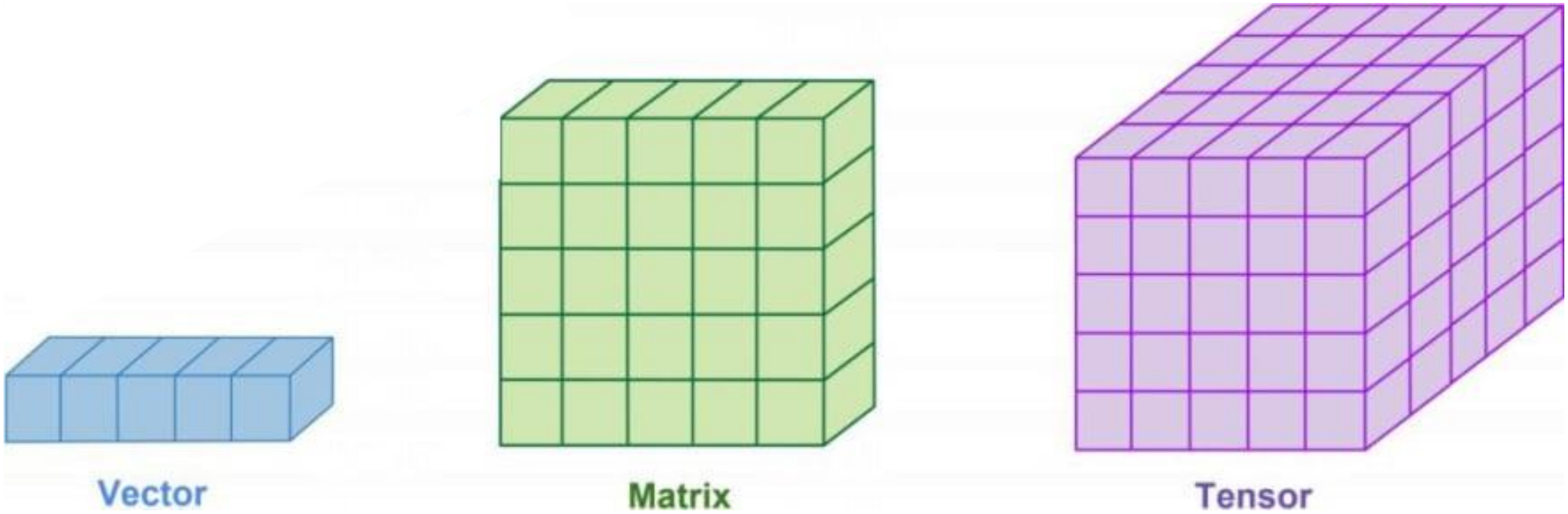
Splatting

A tuple can be ”splatted”/unpacked using ”...”

replace(("string", "ing" => "")...) is the same as replace("string", "ing" => "")

Often used for named arguments

ARRAYS



ARRAYS

Probably the collection you will use the most

Object type for vectors and matrices

Initializing arrays

a1 = []	Vector{Any}
a2 = zeros(Int, 10)	Vector{Int64}
a3 = [1 2 4; 4 5 6; 7 8 9]	2×2 Matrix{Int64}

Can have any number of dimensions

x = zeros(Int, repeat([2], 4)...)	2×2×2×2 Array{Int64, 4}

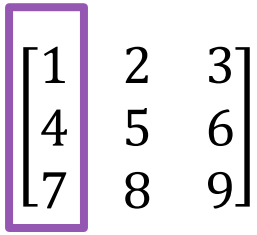
ACCESSING ARRAYS (INCLUDE END-1)

Accessing

Can be done continuously

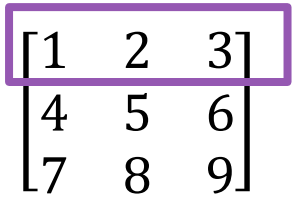
Slice whole column

x[: , 1]



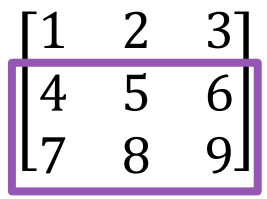
Slice whole row

x[1, :] returns [1, 2, 3]



Slice part of a column

x[2:end, :] returns



DICTIONARIES

Key to value mapping

Efficient for value-based look-up

Not ordered

Initializing dictionaries

d1 = Dict() Dict{Any, Any}()

Accessing

Cannot be done continuously

WHEN TO USE WHICH COLLECTION

Tuples and named tuples

- For performance and no slicing needed

Arrays

- Continuous slicing
- Fast and memory efficient access
- Mutable

Dictionaries

- Key-based index is preferred
- Mutable
- Easy to change due to no explicit position

EXERCISES

BLOCK 4/5: FUNCTIONS AND FLOW CONTROL

FUNCTIONS ARE VERY USEFUL

Areas

Divide and conquer, split code up into meaningful segments

Hide more detailed functionality in a large script.

Julia has builtin tools for optimizing functions

- Compiler and macros

HOW TO DEFINE FUNCTIONS

```
49  function f(x::Int)
50      y = 2*x
51      return y
52  end
53
```

POSITIONAL AND NAMED ARGUMENTS

Positional arguments

The default

```
function gp(x::Int, z::Int)
    println("x is $(x) and z is $(z)")
    return nothing
end
```

```
julia> gp(1,2)
x is 1 and z is 2
```

Named arguments

All arguments after a semi-colon are keyword arguments

They cant be called with their position

Generally they have default values, but its optional

```
function gn(x::Int; z::Int)
    println("x is $(x) and z is $(z)")
    return nothing
end
```

OPTIONAL ARGUMENTS

Positional arguments

Optional arguments must be placed after positional

```
function gp2(x::Int, z::Int = 10)
    println("x is $(x) and z is $(z)")
    return nothing
end
```

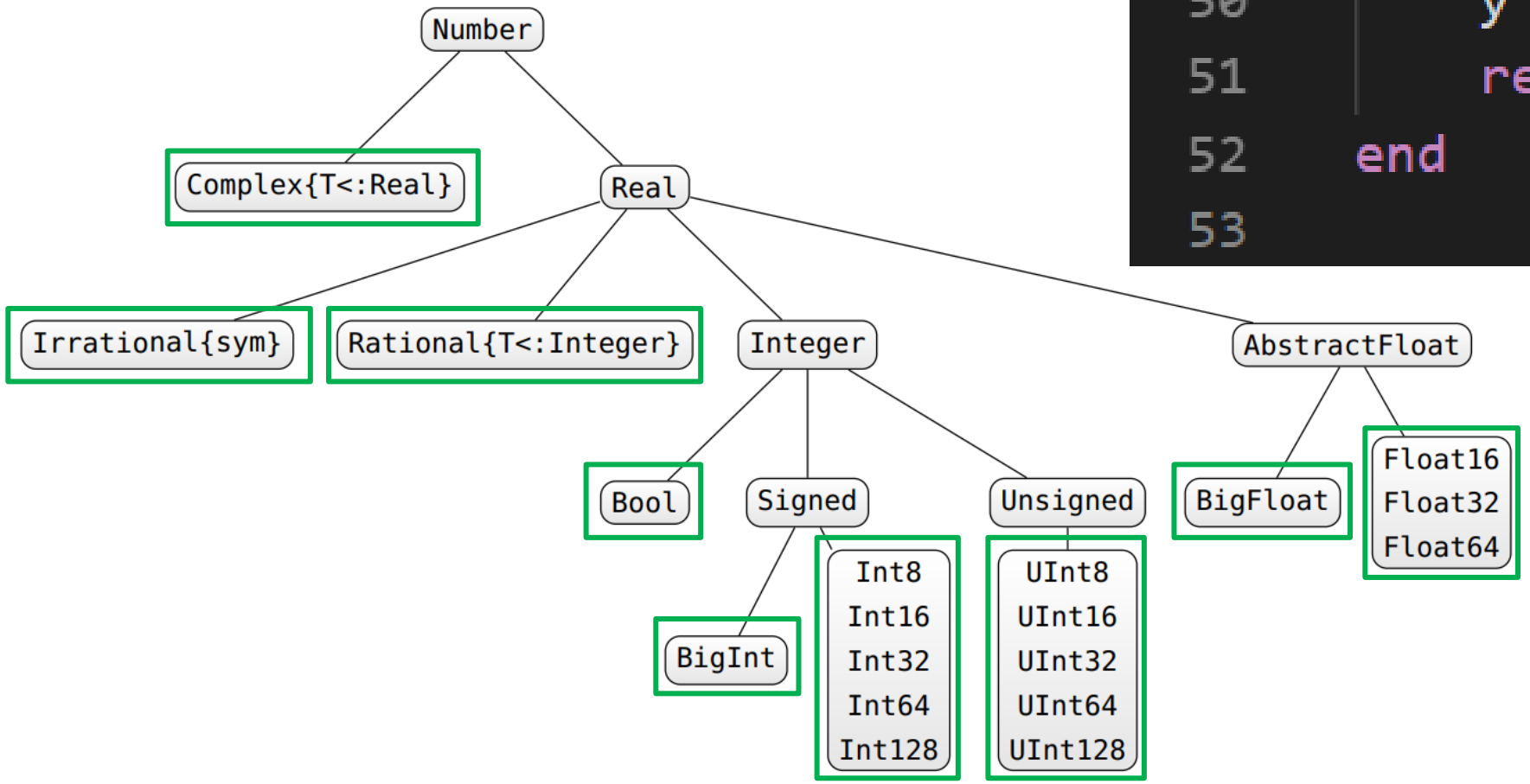
Named arguments

Any order is ok

```
function gn(x::Int; w::Int = 2, z::Int, )
    println("x is $(x) and z is $(z)")
    return nothing
end
```

FUNCTIONS AND MULTIPLE DISPATCH

```
49  function f(x::    )
50      y = 2*x
51      return y
52  end
53
```



BROADCASTING FUNCTIONS

Alternative to vectorizing functions

- Sometimes vectorized functions are slower in Julia
- You don't have to be a vectorization mastermind

Improves both syntax and performance

Can be used for any function

- Incl. custom functions and operators

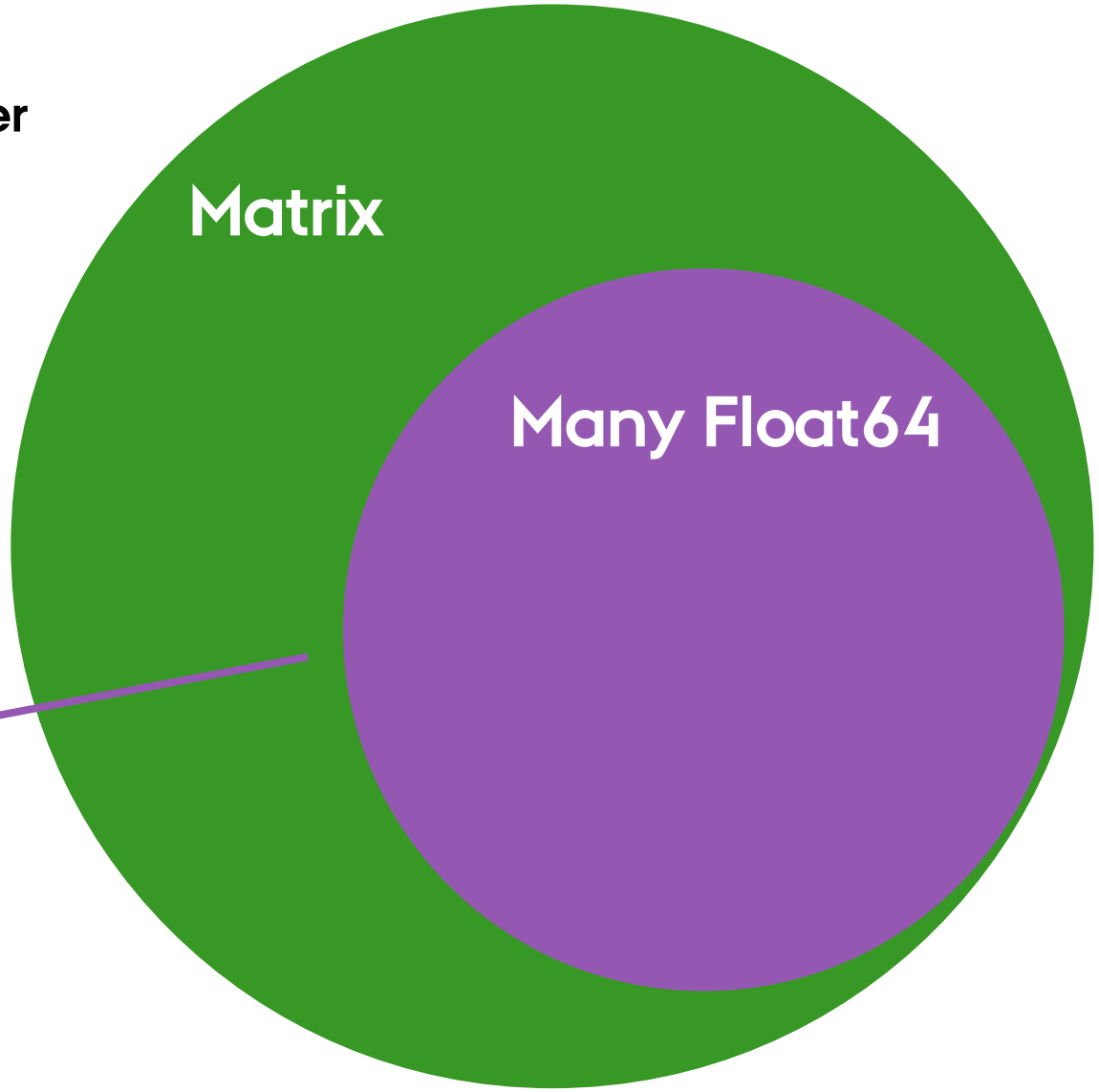
BROADCASTING

Performing the function one layer deeper

M is an array of Float64 objects

log(M)

log.(M)



BROADCASTING EXAMPLES

```
julia> string.([1,2,3,4]) .* " is a number"
4-element Vector{String}:
 "1 is a number"
 "2 is a number"
 "3 is a number"
 "4 is a number"
```

```
julia> M = rand(Float64, 2, 2)
2×2 Matrix{Float64}:
 0.772169  0.985642
 0.430591  0.476756

julia> M .+ 1
2×2 Matrix{Float64}:
 1.77217  1.98564
 1.43059  1.47676
```

```
julia> M .= 2.0
2×2 Matrix{Float64}:
 2.0  2.0
 2.0  2.0
```

CONTROL FLOW

Useful to be able to control the flow through your program and functions

Small example

```
function safelog(x::Float64)
    y = 0.0
    s = 10^-20
    if x > s
        y = log(x)
    elseif x < 0.0
        error("The input was negative")
    else
        y = log(s)
    end
    return y
end
```

TWO EQUIVALENT CONTROL FLOWS

```
if condition1
    z = x
else
    z = y
end
```

```
z = condition ? x : y
```

FOR AND WHILE LOOPS

Julia is optimized around loops

Use them!

```
julia> for i in 1:3
           println(i)
       end
1
2
3
```

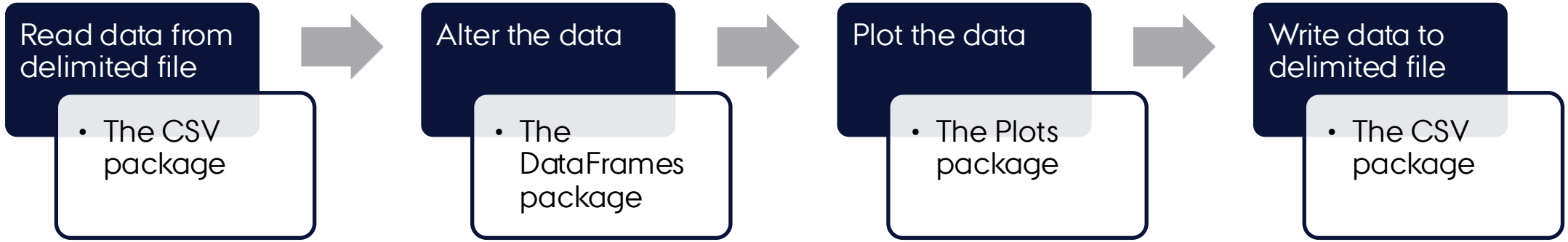
```
julia> i = 1
1

julia> while i <= 3
           println(i)
           i += 1
       end
1
2
3
```

EXERCISES

BLOCK 5/5: GENERAL USE

GENERAL WORKFLOW



THE DATAFRAMES PACKAGE

Functionalities

The DataFrame object: Column-wise storage of equal-size arrays.

Utility functions for DataFrame objects

Fast

Other-language analogues

Rs dplyr

Pythons pandas

DATAFRAME ACCESSING

Access

Index-based and name-based (for columns)

```
julia> x[1,:]
DataFrameRow
 Row │ x     y
     │ Char  Float64
─────┼──────────────
   1 │ a     0.0162386
```

```
julia> x.y[1:2]
2-element Vector{Float64}:
 0.016238571206872 22
 0.8196873048005432
```

```
julia> x
9×2 DataFrame
 Row │ x     y
     │ Char  Float64
─────┼──────────────
   1 │ a     0.0162386
   2 │ a     0.819687
   3 │ a     0.156813
   4 │ b     0.315422
   5 │ b     0.670448
   6 │ b     0.547139
   7 │ c     0.347273
   8 │ c     0.971369
   9 │ c     0.0346189
```

DATAFRAME ADDING COLUMNS

```
julia> x.z = x.y + rand(Float64, 9)
9-element Vector{Float64}:
 0.8673496556227477
 1.4548284928911657
 0.38356877616595675
 0.6914565214856258
 0.9047871602606975
 1.4915004814716224
 0.7334946987610261
 1.100887674892046
 0.20928133623354928
```

```
julia> x
9×3 DataFrame
 Row │ x     y          z
     │ Char  Float64    Float64
─────┼──────────────────────────
   1 │ a     0.0162386  0.86735
   2 │ a     0.819687   1.45483
   3 │ a     0.156813   0.383569
   4 │ b     0.315422   0.691457
   5 │ b     0.670448   0.904787
   6 │ b     0.547139   1.4915
   7 │ c     0.347273   0.733495
   8 │ c     0.971369   1.10089
   9 │ c     0.0346189  0.209281
```

DATAFRAME UTILITY FUNCTIONS

DATAFRAMES GROUPBY+COMBINE

```
julia> combine(groupby(x, "x"), "y" => sum => "sum")
3×2 DataFrame
 Row │ x     sum
     │ Char  Float64
─────┼──────────────
   1 │ a     0.992739
   2 │ b     1.53301
   3 │ c     1.35326
```

```
julia> x
9×3 DataFrame
 Row │ x     y          z
     │ Char  Float64    Float64
─────┼──────────────────────────
   1 │ a     0.0162386  0.86735
   2 │ a     0.819687   1.45483
   3 │ a     0.156813   0.383569
   4 │ b     0.315422   0.691457
   5 │ b     0.670448   0.904787
   6 │ b     0.547139   1.4915
   7 │ c     0.347273   0.733495
   8 │ c     0.971369   1.10089
   9 │ c     0.0346189  0.209281
```

DATAFRAMES GROUPBY+TRANSFORM

```
julia> transform(groupby(x, "x"), "y" => sum => "sum")
9×4 DataFrame
 Row │ x     y          z         sum
     │ Char  Float64    Float64   Float64
─────┼──────────────────────────────────────
   1 │ a     0.0162386  0.86735   0.992739
   2 │ a     0.819687   1.45483   0.992739
   3 │ a     0.156813   0.383569  0.992739
   4 │ b     0.315422   0.691457  1.53301
   5 │ b     0.670448   0.904787  1.53301
   6 │ b     0.547139   1.4915    1.53301
   7 │ c     0.347273   0.733495  1.35326
   8 │ c     0.971369   1.10089   1.35326
   9 │ c     0.0346189  0.209281  1.35326
```

```
julia> x
9×3 DataFrame
 Row │ x     y          z
     │ Char  Float64    Float64
─────┼──────────────────────────
   1 │ a     0.0162386  0.86735
   2 │ a     0.819687   1.45483
   3 │ a     0.156813   0.383569
   4 │ b     0.315422   0.691457
   5 │ b     0.670448   0.904787
   6 │ b     0.547139   1.4915
   7 │ c     0.347273   0.733495
   8 │ c     0.971369   1.10089
   9 │ c     0.0346189  0.209281
```

JOINS

```
julia> leftjoin(x,y, on = "x")
9×3 DataFrame
 Row │ x     y          sum
     │ Char  Float64    Float64?
─────┼──────────────────────────
   1 │ a     0.0162386  0.992739
   2 │ a     0.819687   0.992739
   3 │ a     0.156813   0.992739
   4 │ b     0.315422   1.53301
   5 │ b     0.670448   1.53301
   6 │ b     0.547139   1.53301
   7 │ c     0.347273   1.35326
   8 │ c     0.971369   1.35326
   9 │ c     0.0346189  1.35326
```

```
julia> x
9×2 DataFrame
 Row │ x     y
     │ Char  Float64
─────┼──────────────
   1 │ a     0.0162386
   2 │ a     0.819687
   3 │ a     0.156813
   4 │ b     0.315422
   5 │ b     0.670448
   6 │ b     0.547139
   7 │ c     0.347273
   8 │ c     0.971369
   9 │ c     0.0346189
```

```
julia> y
3×2 DataFrame
 Row │ x     sum
     │ Char  Float64
─────┼──────────────
   1 │ a     0.992739
   2 │ b     1.53301
   3 │ c     1.35326
```

PIVOTS

```
julia> y
3×2 DataFrame
 Row │ x     sum
     │ Char  Float64
─────┼──────────────
   1 │ a     0.992739
   2 │ b     1.53301
   3 │ c     1.35326

julia> unstack(y, "x", "sum")
1×3 DataFrame
 Row │ a          b          c
     │ Float64?   Float64?   Float64?
─────┼──────────────────────────────
   1 │ 0.992739   1.53301    1.35326
```

```
julia> stack(x, ["y", "z"])
18×3 DataFrame
 Row │ x     variable  value
     │ Char  String    Float64
─────┼─────────────────────────
   1 │ a     y         0.0162386
   2 │ c     y         0.0346189
   3 │ a     y         0.156813
   4 │ b     y         0.315422
   5 │ c     y         0.347273
   ⋮ │   ⋮         ⋮         ⋮
  15 │ b     z         0.658364
  16 │ b     z         0.750682
  17 │ a     z         1.72994
  18 │ c     z         1.1291
                  9 rows omitted
```

THE CSV PACKAGE

Functionalities

- Provides high-level functions for reading and writing delimited files
- Can read any delimiter
- Can read per-row
- Well-integrated with DataFrames
- And much more.

This is the only package I use for reading delimited files.

- Others exist

CSV EXAMPLES

```
julia> CSV.write("test.csv", x)
"test.csv"
```

```
julia> x
9×2 DataFrame
 Row │ x     y
     │ Char  Float64
─────┼──────────────
   1 │ a     0.0162386
   2 │ a     0.819687
   3 │ a     0.156813
   4 │ b     0.315422
   5 │ b     0.670448
   6 │ b     0.547139
   7 │ c     0.347273
   8 │ c     0.971369
   9 │ c     0.0346189
```

```
julia> CSV.read("test.csv", DataFrame)
9×2 DataFrame
 Row │ x       y
     │ String1 Float64
─────┼─────────────────
   1 │ a       0.0162386
   2 │ a       0.819687
   3 │ a       0.156813
   4 │ b       0.315422
   5 │ b       0.670448
   6 │ b       0.547139
   7 │ c       0.347273
   8 │ c       0.971369
   9 │ c       0.0346189
```

THE PLOTS PACKAGE

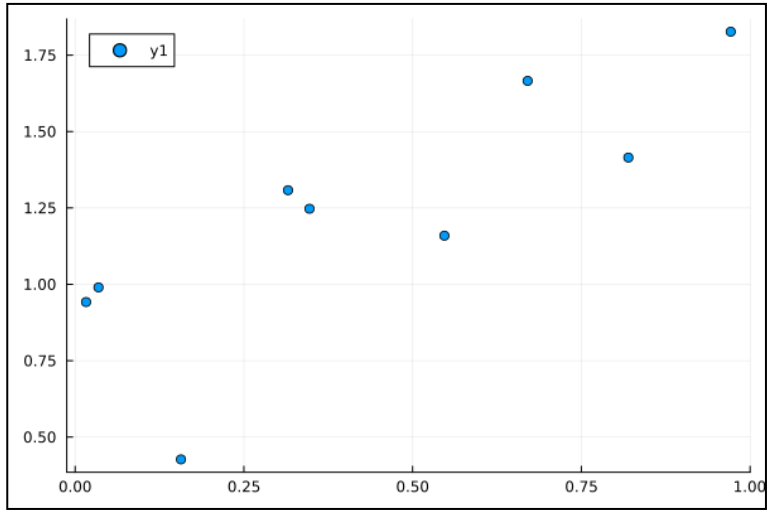
There are many competitive packages

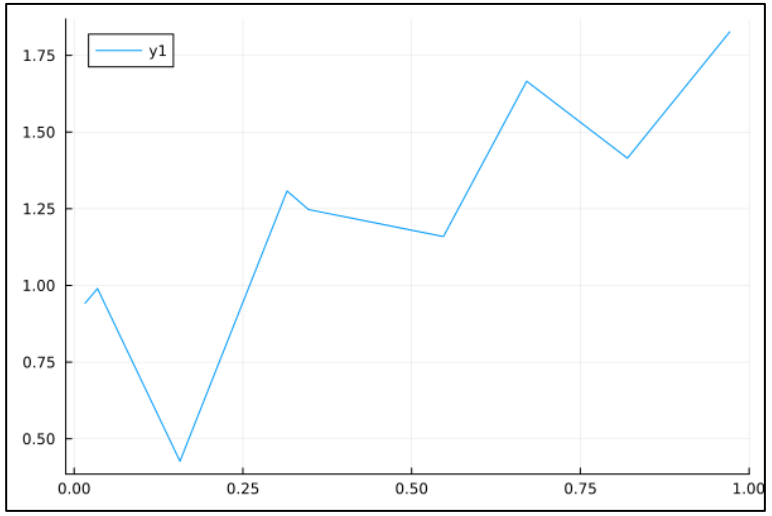
- Plots
- Makie
- UnicodePlots
- AlgebraOfGraphics

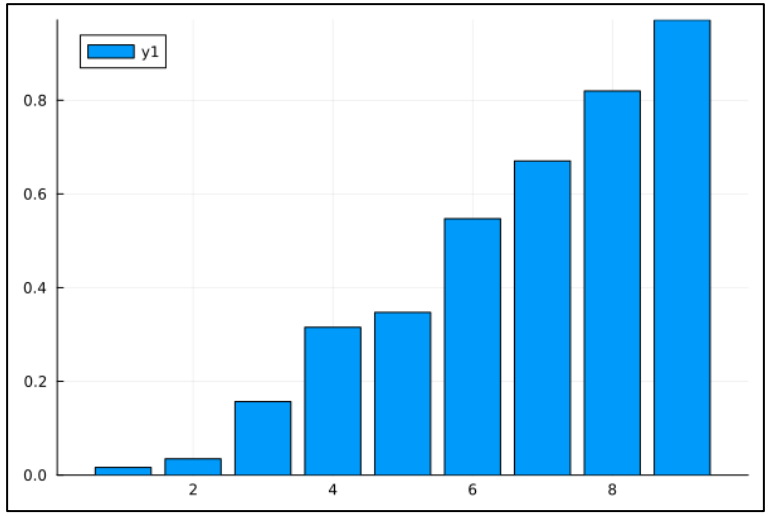
The plots package is one of the simpler packages

- Lightweight

PLOTS


y1
1.75
1.50
1.25
1.00
0.75
0.50
0.00
0.25
0.50
0.75
1.00


y1
1.75
1.50
1.25
1.00
0.75
0.50
0.00
0.25
0.50
0.75
1.00


y1
0.8
0.6
0.4
0.2
0.0
2
4
6
8

EXERCISES

ACKNOWLEDGEMENTS

DANMARKS FRIE FORSKNINGSFOND
INDEPENDENT RESEARCH FUND DENMARK