Introduction to Julia

Cédric Simal

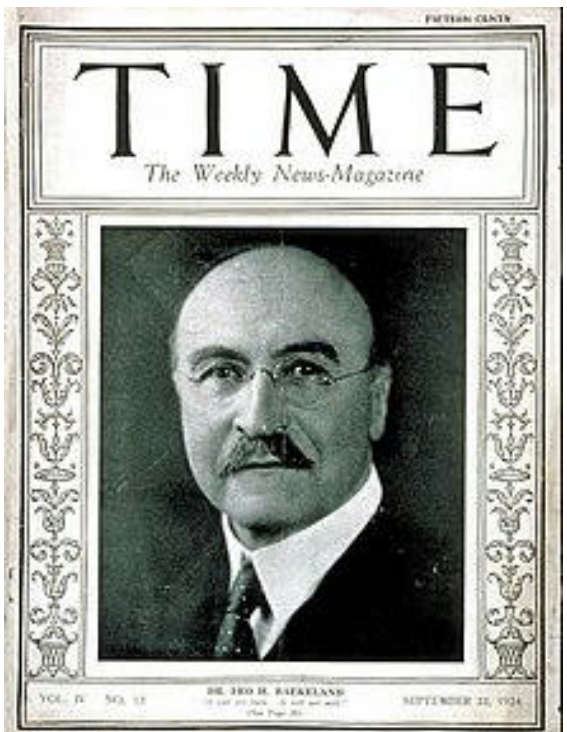
cedric.simal@unamur.be

CéCI Training Sessions

14/11/2023

Science Anniversaries of today

Apollo 12 Launch (1969)



Birth of Leo Baekeland (1863)

image source: Wikipedia

Outline

1. Installing Julia
2. Elevator pitch (while you're installing)
3. The Julia REPL
4. Basics of Julia
5. Multiple Dispatch
6. Package Management

Installing Julia (via Juliaup)

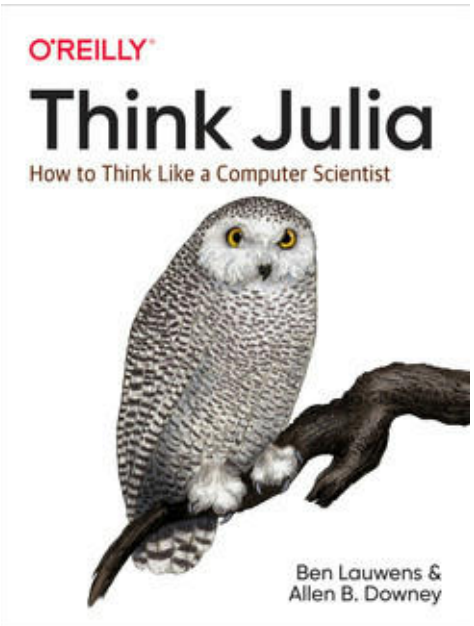
Windows

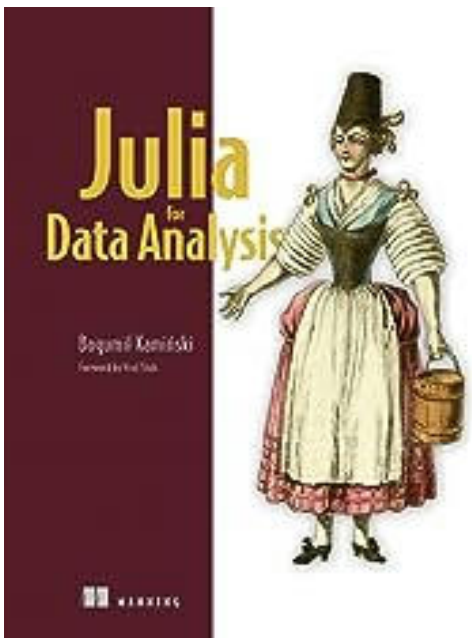
```
winget install julia -s msstore
```

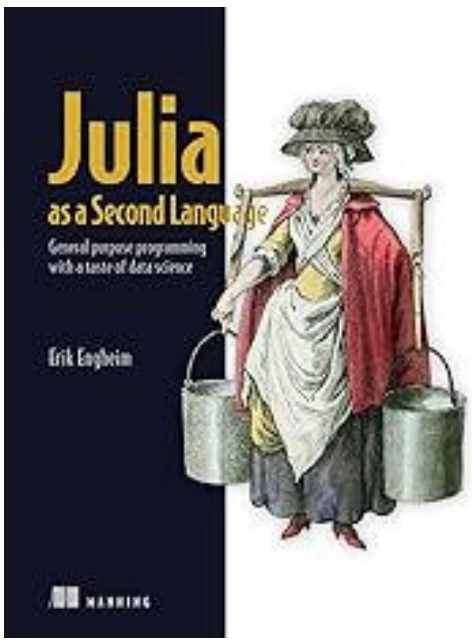
Mac/Linux

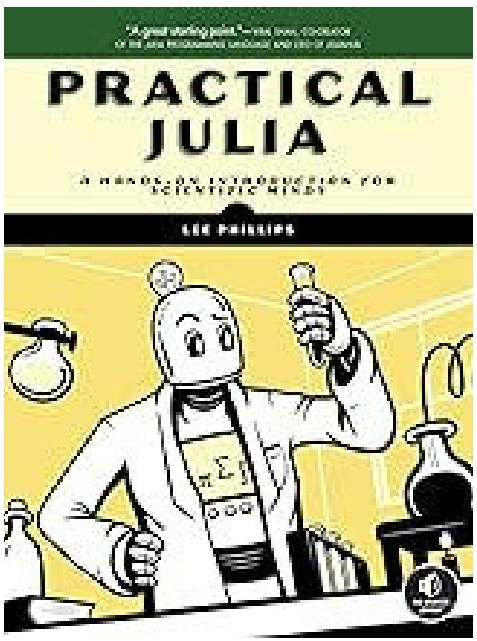
```
curl -fsSL https://install.julialang.
org | sh
```

Book Recommendations





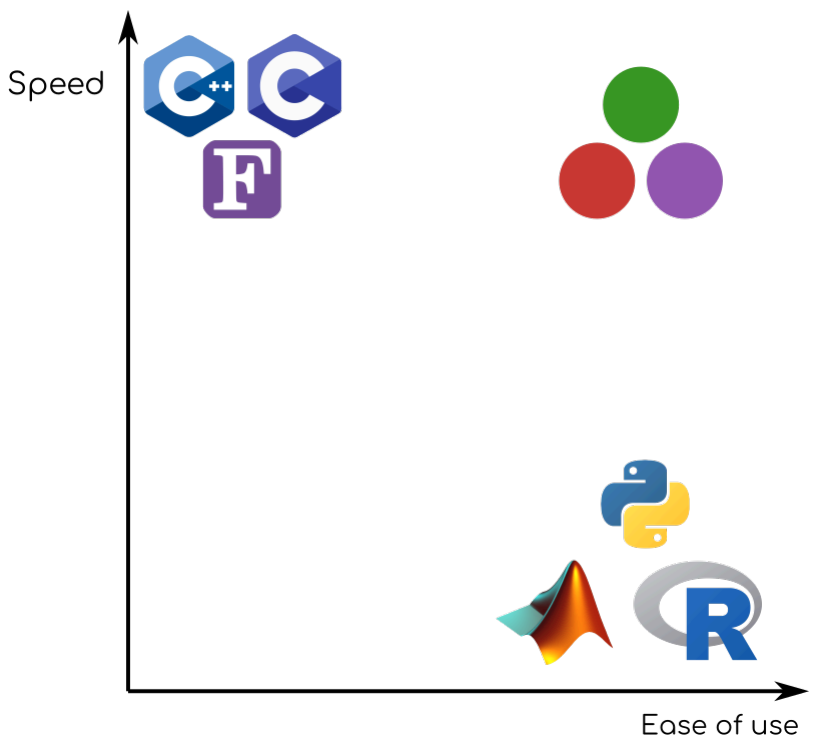




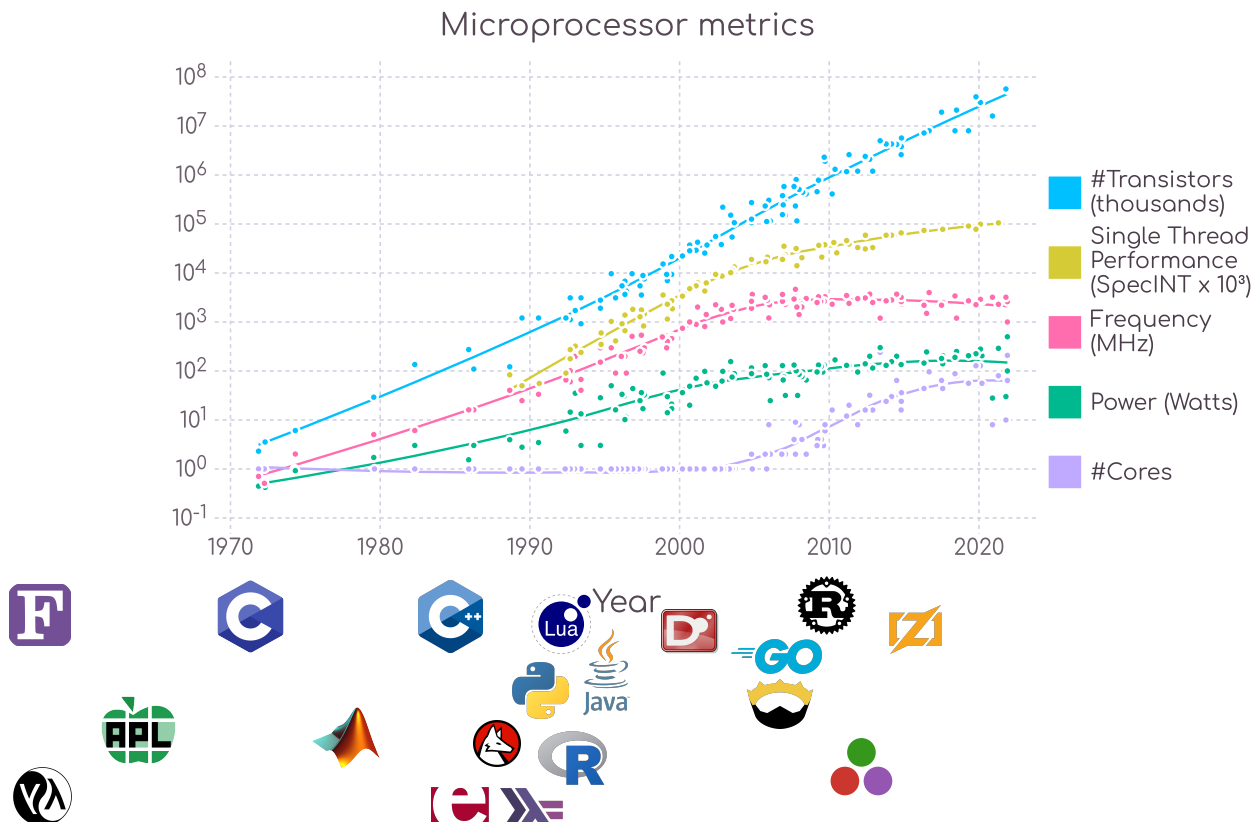
https://benlauwens.github.io/ThinkJulia.jl/latest/book

Modern Problems in Scientific Computing

The two languages problem



The rise of parallel computing

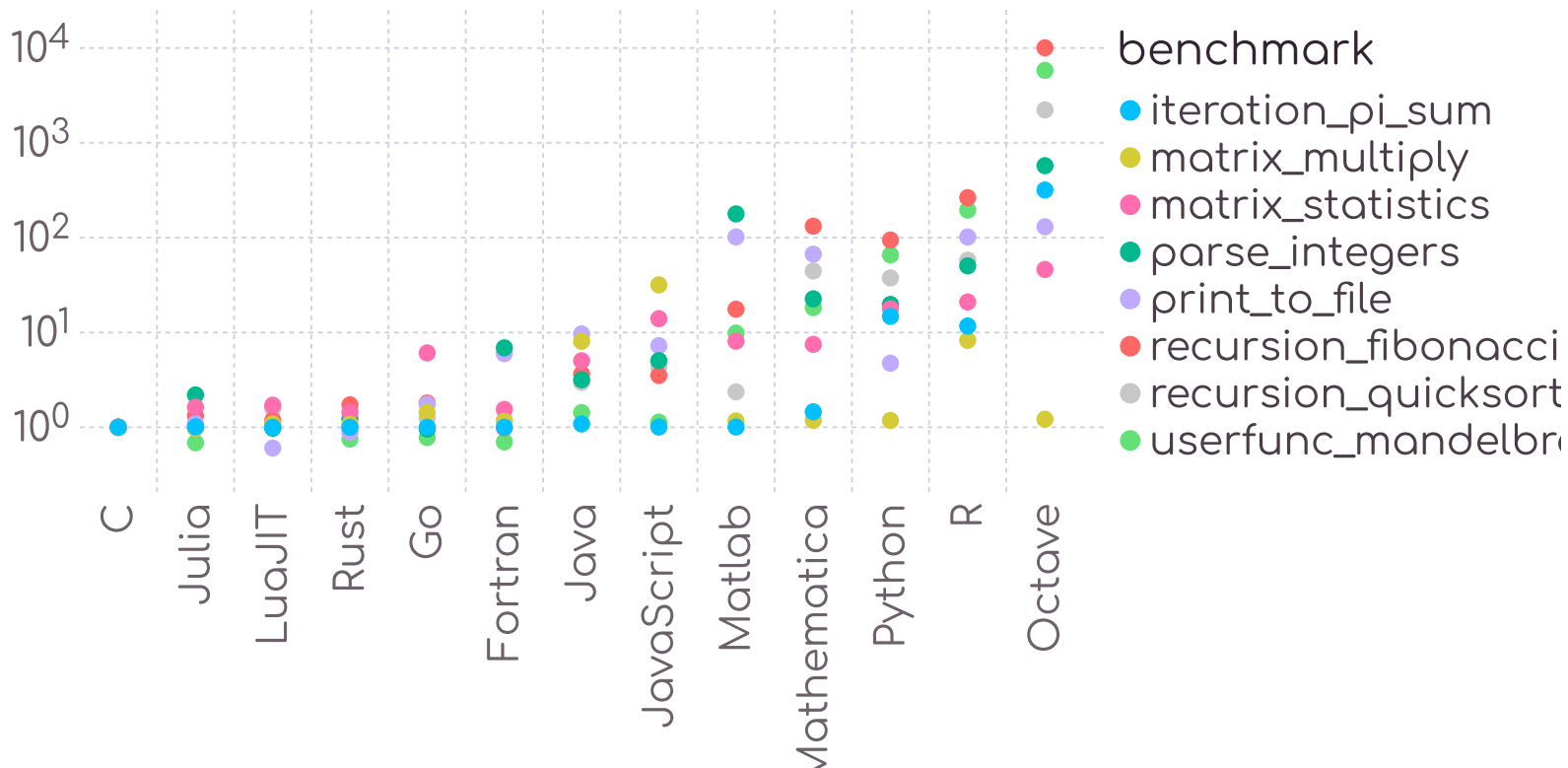


data: https://github.com/karlrupp/microprocessor-trend-data

Meet Julia

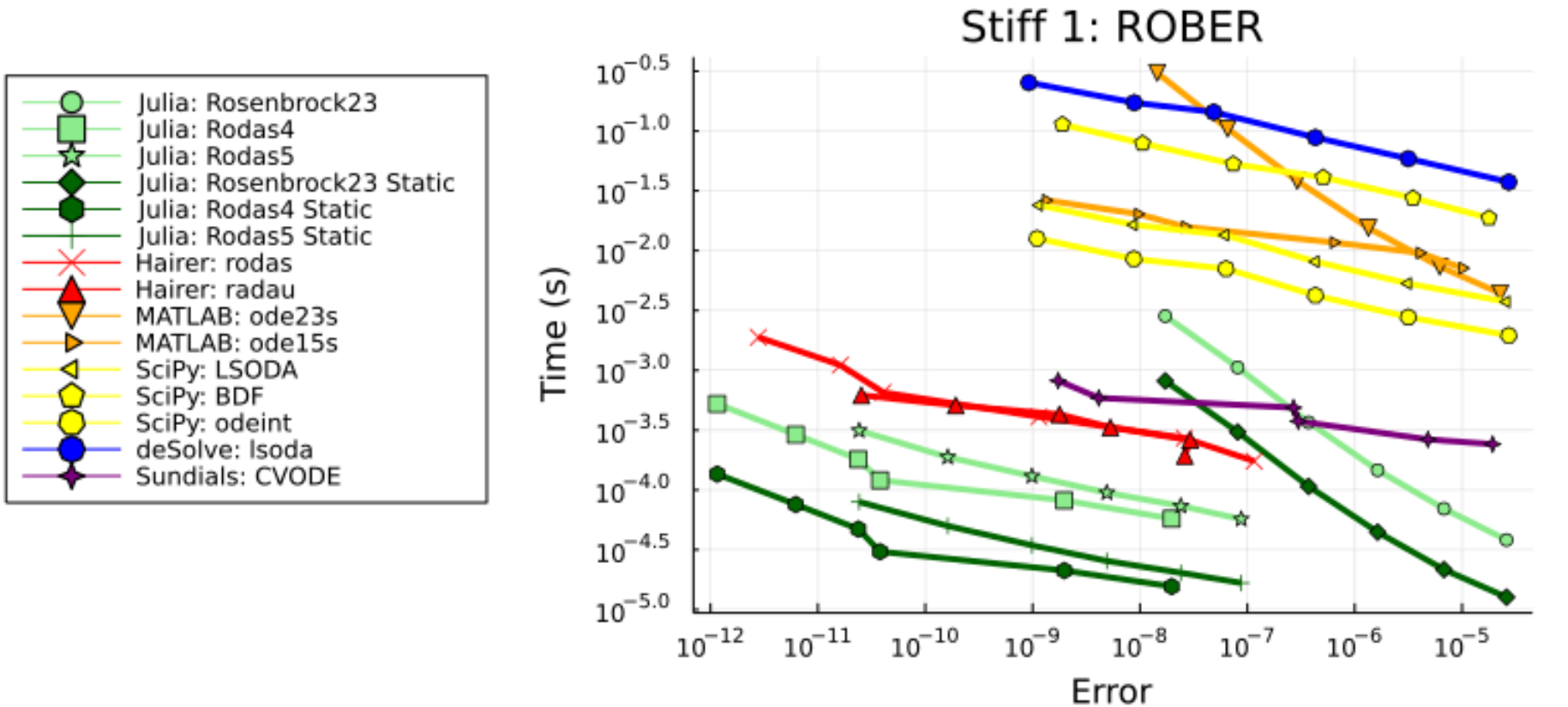
A short history

Julia is fast



https://julialang.org/benchmarks/

Julia is fast (2)



https://benchmarks.sciml.ai/stable/MultiLanguage/ode_wrapper_packages/

Why you might want to use Julia

- Open Source (MIT license)
  - Free as in Free Beer *and* Free Speech
  - Faster development than proprietary languages
- Made for numerical computing
- Amazing Interop with C/C++, Fortran, Python, Matlab...
- Unicode
  - Code looks like math
- Secretly a LISP

Why you might NOT want to use Julia

If you want to evangelize a tool or process or technology, you should be able to answer two questions:

1. What negative thing do you have to deal with?
2. What positive thing do you have to give up?[1]

My personal takes for Julia:

1. Time to first X; Poorly documented/Undocumented features.
2. No formal Interfaces/Traits/Typeclasses. Types don't help with correctness.

[1] https://hillelwayne.com/hate-your-tools/

Why you might NOT want to use Julia

- Open Source
  - Documentation can be lackluster
  - Projects with only one maintainer
- Time to first X
  - Parts of your program have to be recompiled on startup
- Error messages are not beginner-friendly
- Correctness bugs
- Tiny compared to Python
- Array indices start at 1

Basics of Julia

Variables and comments

Variable declaration:

```
foo = 1
```

Multiple declarations:

```
a, b, c = 1, 2, 3 # a = 1; b = 2; c = 3
```

Multiline comment:

```
#= This comment
spans multiple
lines
=#
```

Expressions

```
a = 1; a += 1; a *= 2
# equivalent to
a = 1
a += 1 # a = a + 1
a *= 2 # a = a * 2

begin
    a = 1
    a += 1
    a *= 2
end
```

Elementary Types

Integers

```
a = 1
```

Floating point numbers (double)

```
b = 0.2
```

Booleans

```
e = true; f = false
```

Strings and characters

```
c = 'c'
d = "Hello, world\n" # NB C-style character literals work
```

Arrays, tuples and Dicts

```
x = [1, 2, 3]
y = ("Waffle", 2)
z = Dict(
    "Frites" ⇒ 2.0,
    "Fricandelle" ⇒ 1.5,
    "Sauce Andalouse" ⇒ 0.3
)
```

Conditionals

```
# get a random number between -30 and
150
water_temperature = rand(-30:150)
if water_temperature < 0
    print("Ice")
elseif water_temperature < 100
    print("Liquid Water")
else
    print("Steam")
end
```

Note that an if bloc is an expression that returns a value:

```
age = rand(1:40)
status = if age < 18
    "minor"
else
    "adult"
end
```

Ternary operator:

```
status = age < 18 ? "minor" : "adult"
```

Loops

Python style for-loops with iterators

```
k = 0
for k in 1:10
    k += i
end
```

```
n = 10
while n != 1
    n = iseven(n) ? div(n,2) : 3*n+1
end
```

Special Syntax for nested for loops

```
for i in 1:5
    for j in 1:3
        print(i+j)
    end
end
# This can be rewritten as
for i in 1:5, j in 1:3
    print(i+j)
end
```

Arrays

```
# a 1 dimensional vector with 10 elements of type 'double'
a = Vector{Float64}(undef, 10)
# equivalent to previous line
b = Array{Float64,1}(undef, 10)
# A 2 dimensional array of integers with 3 rows and 4 columns
c = Array{Int,2}(undef, 3,4)
d = Array{Int,2}(undef, (3,4))
```

Special constructors for arrays of zeroes/ones/arbitary value

```
e = zeros(3,4)
f = ones(3)
g = fill("Hello", 4)
```

Array Constructors

Explicit enumeration

```
a = [1,2,3,4] # 1D array (column vector)
b = [1 2 3 4] # 1 x 4 matrix (row vector)
c = [1 2; 3 4; 5 6] # 3 x 2 matrix
d = [1; 2;; 3; 4;;; 5; 6;; 7; 8] # 2 x 2 x 2 3D array
```

Ranges

```
i = 1:10 # integers from 1 to 10
x = 0.0:0.01:1.0 # from 0.0 to 1.0 by steps of 0.1
y = LinRange(0.0,1.0,100)
```

Array Comprehensions

```
# Apply arbitrary function to a collection
a = [i^2 for i in 1:10]
# Use multiple indices to generate higher dimensional arrays
b = [i+j for i in 1:5, j in 1:5]
# Filter elements using a condition
c = [i/(i^2) for i in 1:10 if iseven(i)]
# special syntax for functions taking a collection
p = 4 * sum((-1)^k / (2k+1) for k in 0:10000)
```

Array indexing

```
# default arrays start at 1
a[1]
# use an array of indices
a[[1,3,5]]
# ranges work too
a[1:5]
a[begin] # first element
a[end] # last element
```

Iterate over an arbitrary array

```
for i in eachindex(a)
    print(a[i])
end
```

Array slicing

```
b = rand(3,4) # 3 x 4 matrix of random numbers
b[1,:] # first row of b
b[:,2] # second column of b
```

Iterate over array dimension

```
for j in axes(b,2)
    print(b[:,j])
end
```

Broadcasting

Function calls work as usual

```
cos(π/2) # NB. use "\pi<TAB>" for unicode π
```

Special syntax for "vectorizing" function calls

```
xs = LinRange(0.0,2π,100)
cos.(xs)
# equivalent to
[cos(x) for x in xs]
# It also applies to infix operators
xs .* cos.(xs)
```

Modifying arrays

```
a = zeros(5)
a .= [1,2,3,4,5] # NB. without the dot,
this would create a new array
a .+= 1.0
a .+= [6,7,8,9,10]
```

Multiple broadcasted calls

```
a .= a.^3 .- a.^2 .+ a .- 1
# equivalent to
for i in eachindex(a)
    a[i] = a[i]^3 - a[i]^2 + a[i] - 1
end
# short-hand
@. a = a^3 - a^2 + a - 1
```

Functions

```
function collatz(n)
    if iseven(n)
        return div(n,2)
    else
        return 3n+1
    end
end
# More compact version
function collatz(n)
    return iseven(n) ? div(n,2) : 3n+1
end
# implicit return
function collatz(n)
    iseven(n) ? div(n,2) : 3n+1
end
```

Inline definition

```
square(x) = x^2
```

Lambda expressions/Anonymous functions

```
square = x -> x^2
square = function(x)
    x^2
end
```

Default values and keyword arguments

```
function say_hello(x = "World"; hello_str="Hello", end_str="!")
    # NB. '*' concatenates strings
    print(hello_str * ", " * x * end_str)
end
```

```
julia> say_hello()
Hello, World!

julia> say_hello("Alice")
Hello, Alice!

julia> say_hello("Alice", hello_str="Bonjour", end_str="?")
```

The Type System