6.S083 / 18.S190: Introduction to computational thinking with Julia + applications to the COVID-19 pandemic

Welcome!

- Welcome to 6.S083 / 18.S190!
- Class web page: https://github.com/mitmath/6.S083

Welcome!

- Welcome to 6.S083 / 18.S190!
- Class web page: https://github.com/mitmath/6.S083

- Alternative to 6.0002
- Prerequisite: programming at level of 6.0001
- Desirable: 18.02 (multivariable calculus)

Welcome!

- Welcome to 6.S083 / 18.S190!
- Class web page: https://github.com/mitmath/6.S083

- Alternative to 6.0002
- Prerequisite: programming at level of 6.0001
- Desirable: 18.02 (multivariable calculus)

- Language: **Julia** instead of Python

Goals for the class

- **Computational thinking**: applying computational techniques to solve problems
- Use computer as *tool* to investigate and solve problems

Goals for the class

- **Computational thinking**: applying computational techniques to solve problems
- Use computer as *tool* to investigate and solve problems

- Key problem as we speak: **COVID-19 pandemic**

Goals for the class

- **Computational thinking**: applying computational techniques to solve problems
- Use computer as *tool* to investigate and solve problems
- Key problem as we speak: **COVID-19 pandemic**
- Understand **data** and build **models**

Goals for today

- Get hold of some data
- Clean and explore the data
- Learn basic Julia syntax
- Create visualizations

Why Julia?

- Julia: developed at MIT by Prof. Edelman’s group
- Released 8 years ago
- Current release: 1.4; now quite mature

Why Julia?

- Julia: developed at MIT by Prof. Edelman's group
- Released 8 years ago
- Current release: 1.4; now quite mature

- Free, open source software
- Developed by world-wide community on GitHub
- Over 3,000 regiestered packages in wide range of domains

Julia II

- **Modern, powerful** language:
- **Interactive** but **high performance** (fast) – previously mutually exclusive

Julia II

- **Modern, powerful** language:
- **Interactive** but **high performance** (fast) – previously mutually exclusive

- Syntax: similar to Python / MATLAB / R
- But carefully designed for high-performance Computational Science & Engineering applications

Julia II

- **Modern, powerful** language:
- **Interactive** but **high performance** (fast) – previously mutually exclusive

- Syntax: similar to Python / MATLAB / R
- But carefully designed for high-performance Computational Science & Engineering applications

- Design means that **most of Julia is written in Julia itself**
- Hence much easier to examine and modify algorithms

Julia II

- **Modern, powerful** language:
- **Interactive** but **high performance** (fast) – previously mutually exclusive

- Syntax: similar to Python / MATLAB / R
- But carefully designed for high-performance Computational Science & Engineering applications

- Design means that **most of Julia is written in Julia itself**
- Hence much easier to examine and modify algorithms

Some goals of Julia

- Enables and encourages writing code that:
  - is more compact: better **abstractions** (e.g. broadcasting)
  - **looks like maths** (Unicode variable and operator names)
  - **performant** (specialization, compilation)
  - **generic** (specialization, multiple dispatch)
- Enable **code re-use**: see Stefan Karpinski's talk at JuliaCon 2019

How to use Julia:

- **Jupyter Notebook** computational environment
- See `installation.md` for instructions

How to use Julia:

- **Jupyter Notebook** computational environment
- See `installation.md` for instructions

- **Juno IDE** – install Atom editor and `uber-juno` Atom package
- REPL (Read–Eval–Print–Loop) in the terminal

Variables

- Define variables; types are inferred

```julia
r = 2.2   # \scrr<TAB> for italic `r`
area = π * r^2   # \pi<TAB>   # ^ for powers
circumference = π * (2r)   # implicit multiplication
```

- π (or `pi`) pre-defined as special value with special behaviour:

```julia
@show π
```

Types

- Values like `3` stored as bits (`0` / `1`) in memory.
- Julia associates **types** to values: specify **behaviour** of the bits under operations.
- Some basic types:

```julia
x = 3
@show typeof(x)    # Int64

y = -3.1    # Float64
@show typeof(y)

s = "6.S083"  # String
```

Functions and types

- Functions *behave differently* for different types
- E.g. `*` (multplication) is just another function:

```
3 * 3
```

```
"3" * "3"
```

- Fundamental to how Julia works

Functions

- Functions are **most important constructs** in any program
- They enable **abstraction** and **code reuse**
- Short syntax for simple mathematical functions:

```julia
area(r) = π * r^2

A = area(1.0)
```

- Long syntax:

```julia
"""Calculate area of circle of radius `r`."""
function area(r)
    A = π * r^2    function
    return A
end
```

Functions II

- Docstring is written *above* function body
- `A` is **local variable**: exists only inside function
- `"""` denotes multiline string
- Use `?area` from REPL or notebook to see documentation
- Operations with `π` convert to `Float64`
- In Julia: *everything should be in a function*

Conditionals

- `if...then...else`

```julia
a = 5

if a < 4
    s = "small"

elseif a < 6
    s = "medium"

else
    s = "large"
end

s
```

Conditionals II

- No `:`; but needs `end`
- Using `end` means that indentation is *not* significant
- That is, not significant *for the computer*, but still is *for us humans* – make sure to always indent correctly!

Loops

- Again replace `:` by `end`
- Use simple loop to find square root using “guess and check” / exhaustive enumeration:

Loops II

```julia
function square_root(n)
    found = 0

    for i in 1:n
        if i^2 ≥ abs(n)   # \ge<TAB> or >=
            found = i    # i doesn't exist outside loop
            break
        end
    end

    if found^2 == n
        return (found, :exact)
    else
        return (found, :not_exact)
    end
```

Loops III

- Always prefer to *return information* instead of printing
- Julia automatically *displays* last result
- `:a` is a `Symbol`, a type of optimized string
- **Exercise**: Does `square_root` work with `Float64`? Should it?

Floating-point arithmetic

- Recall: floating-point arithmetic gives *approximation* to real numbers:

```julia
x = 0.0

for i in 1:10
    global x += 0.1    # `global` not needed inside a functi
    @show x    # prefer @show instead of print
end

x, (x == 1.0)
```

- `@show` prints name *and* value of a variable; prefer it to `print` for debugging
- Internal representation:

```
bitstring(0.1)
```

Array comprehensions

- Build **array** of values by repeating calculation:

```
factorials = [fact(n) for n in 1:21]
```

- Goes wrong due to **overflow**: result > max value storable in `Int64`
- (Slow) solution: `BigInt` type – arbitrarily large integers

```
fact(big(30))
```

- Can catch overflow using *checked arithmetic*:

```
# Base.checked_mul(10^20, 10^20)
```