Introduction to 6.S083 and the Julia language

Welcome!

- Welcome to 6.S083!
- New, pilot alternative class to 6.0002: some overlap, some differences
- Prerequisite: 6.0001 (intro to computer science and programming).
- Corequisite: 18.02 (multivariable calculus), since many applications require **derivatives**.
- 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.
- Computation also **helps** to understand concepts (e.g. mathematical) intuitively and deeply.
- **Performance** (execution speed), e.g. $O(N)$ vs $O(N^2)$, not just computational complexity (“polynomial”)

Syllabus

- https://github.com/dpsanders/6.S083

Goals for today

- See how to use Julia
- Learn basic syntax of Julia
- Revisit code fragments from 6.0001 and rewrite in idiomatic Julia
- Hence start to learn “Julian style” (different from “Pythonic style”)

The Julia language

- Julia language: developed at MIT in Prof. Edelman's group in math / CSAIL; released in 2012.
- Syntax stable since version 1.0 (August 2018).
- Current release is 1.2 (waiting for 1.3 in a few days)
- **Modern, powerful** language:
- **Interactive** but **high performance** (fast) – previously mutually exclusive.
- Free, open source software. Developed by world-wide community on GitHub.

Julia II

- 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** (unlike Python).
- Hence much easier to examine and modify algorithms.
- Example: `@edit sin(3.1)`

Some goals of Julia

- Write 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:

- **Juno IDE** – install Atom editor and `uber-juno` Atom package
- Jupyter notebook via `IJulia.jl` package
- REPL (Read–Eval–Print–Loop) in the terminal

Variables

- Python:

```
pi = 355/113

radius = 2.2

area = pi*(radius**2)

circumference = pi*(radius*2)
```

- 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:

```
@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:

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


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


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

- Functions *behave differently* for different types: `julia 3 * 3`

  `"3" * "3"`

- This is fundamental to how Julia works.

Functions

- Functions are **most important constructs** in any program, since 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 in Julia.
- `A` is **local variable**: exists only inside functiojn
- `"""` 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`

```
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

```
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:

```
x = 0.0

for i in 1:10
    global x += 0.1    # `global` not needed inside a function
    @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)
```

Debugging

- Juno IDE contains *interactive debugger*
- Debug function with arguments by typing in Juno REPL pane: `Juno.@enter f(1, 2)`
- Demo

Recursion

- Long form: “‘julia function fact2(n) if n ≤ 1 return 1 end

```
return n * fact2(n-1)
```

  end “‘

- Short form: `julia fact(n) = (n ≤ 1) ? 1 : ( n * fact(n-1) )`
- **Ternary operator** `a ? b : c` – “if a then b, else c”
- Leads to short but less readable code

```
fact(10), fact2(10)
```

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`
- NB: *Different* from Python; *silent* behaviour
- (Slow) solution: `BigInt` type – arbitrarily large integers

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

- Can catch overflow using *checked arithmetic*: `julia # Base.checked_mul(10^20, 10^20)`

Towers of Hanoi

- Transfer tower of discs from one peg to another; discs must always be in order
- Remember: Don't print, return instead

```
function towers(n, from, to, spare)
    n == 1 && return [ (from, to) ]

    return vcat(towers(n-1, from, spare, to),
                towers(1, from, to, spare),
                towers(n-1, spare, to, from) )
end

towers(4, 1, 2, 3)
```

- `[ (1, 2) ]` creates a `Vector` containing a tuple
- `&&` is like “if-then” (but it “short-circuits”)
- `vcat` concatenates arrays

Hanoi II

- Don't want to write `(1, 2, 3)` each time.
- *Make new version of function* – called a *method*: "'julia

  towers(n) = towers(n, 1, 2, 3) # defines a method

  towers(3) "'
- Same function name, different number of arguments.
- *Extremely common* to use this "pattern" in Julia.
- *No performance loss*!
- **Exercise**: Modify the code to track the number of each disc. Emit moves as `(i, j, k)` to represent "disc i moves from peg j to peg k".

Objects: Composite types

- Define *new types* of object by grouping data (“attributes” / “fields”) that belong together.
- Unlike Python, *functions do not live inside objects* – one of key changes
- Basic syntax for defining new type:

```
struct CoordinatePair
    x::Float64
    y::Float64
end
```

Constructors

- Create objects using **constructor** (function with same name):

```julia
o = CoordinatePair(0, 0)

x = CoordinatePair(1, 2)
```

- Julia *automatically* defines sensible constructor and display methods ()`show` methods.
- Add constructor by adding new method `julia`

```julia
CoordinatePair() = CoordinatePair(0, 0)
```

Functions on types

- Add function acting on type using **type annotation**, `::`

```
distance(a::CoordinatePair, b::CoordinatePair) = √( (a.x - b.x)^2 + (a.y - b.y)
```

```
distance(o, x)
```

- `origin() = CoordinatePair()`

```
distance(a::CoordinatePair) = distance(origin(), a)
```

```
distance(x)
```

New types of number

- Julia is very well suited to defining new types of number:

```
struct Fraction
    num::Int
    denom::Int
end

Base.show(io::IO, x::Fraction) = print(io, x.num, " // ", x.denom)

x = Fraction(3, 4)
```

Arithmetic

- We will *extend* Julia's arithmetic operations to work on our new type by `import`ing the relevant operations:

```julia
import Base: *

*(x::Fraction, y::Fraction) = Fraction(x.num * y.num, x.denom * y.denom)

x = Fraction(3, 4)
y = Fraction(6, 5)

x * y   # result not in lowest terms
```

Methods

- Alternate syntax:

```
*(x, y)
```

- In Julia, `*` is *just a normal function*
- But has large number of *methods* for **objects of different types**:

```
methods(*)
```

- Have not yet defined multiplication of `Fraction` by `Int`, so it throws an error:

```
Fraction(3, 4) * 1
```

- So just *add another method*!
- Implement in terms of functionality we already defined:

```
*(x::Fraction, y::Int) = x * Fraction(y, 1)
Fraction(3, 4) * 1
```

- Or make specialized version instead (may be more efficient):

```
*(x::Fraction, y::Int) = Fraction(x.num * y, x.denom)
```

- **Exercise**: Implement `+`

Introspection: Asking Julia what it is thinking

- `@which`: find *which* method Julia will call for certain combination of types.
- `@edit`: view and edit source code of a method.

```
@which sin(3.1)
```

```
@edit sin(3.1)
```

- May need to set `EDITOR` environment variable.

Interpreters

- Python is an **interpreted** language (as are R, MATLAB):
  - A program ("CPython interpreter") is constantly running.
  - Examines a version ("bytecode") of Python source code over and over again as program is running
  - When sees `a + b`, examines "boxes" `a` and `b` to find what *types* of values they contain.
  - Then chooses correct version of `+` operator and executes it on copies of variables that "live inside" the interpreter.

- It's a program that "behaves as if it were a computer".
- Layer in between your code and the hardware.
- Possible to implement Python interpreter in 500 lines of Python.
- Python is slow because it does those operations *at every single step*, *over and over again*.

Compilers

- Julia can be fast since it is a **compiled** language (as are C, C++):
  - Tries to do lookups once before program runs
  - Turns whole program into machine code that controls hardware (CPU etc.) directly
  - No middle layer that gets in the way

- More time spent compiling at start, but overall gain, provided prog ram runs for long enough.
- Python can be quicker for *small enough* tasks.
- Julia can be slow if has to fall back to Python-like behaviour when it can't work everything out at start.

Levels of compilation

- Code passes through several levels in compilation process
- We can inspect result at each level:
- Lowered code – lower-level Julia code
- Typed code – after type inference
- LLVM code – translated into an “intermediate representation” (IR) for the LLVM compiler
- Native code – machine code to run directly on CPU

```julia
f(x) = 3x


@code_lowered f(3)

@code_typed optimize=false f(3)

@code_typed f(3)

@code_llvm optimize=false f(3)

@code_llvm f(3)

@code_native f(3)
```

Why can Julia be fast?

- `@code_typed f(3)` and `@code_typed f(3.0)` give different results:
  Julia infers *types for all variables throughout function*

- `@code_native f(3)` and `@code_native f(3.0)` give *different* code
  Julia creates specialized versions *of functions for different input argument* types

- Type inference + specialization are key mechanisms for Julia's speed
- If type inference fails, Julia reverts to Python-style mechanism and can be slow
- Solution: Make sure type inference succeeds!
- Tools: `@code_warntype`, `Traceur.jl` package

Review

- Julia is similar to Python
  - Readable, compact
  - Compact
  - Fun
- Julia is *very different* from Python
  - Emphasis on **performance**: compiled
  - Code looks like math

- Methods live *outside* objects
- Not traditional OOP
- New paradigm based around **multiple dispatch**