Julia

What is Julia?

- First developed in 2012
- Is a high-level numerical computing library
    - Meant to be general purpose too
- Inspired by Python, R, MATLAB, Java, C++, FORTRAN, LISP, ...
    - Is a functional language underneath it all
- Speed is a high priority
- Has built-in support for distribution and parallelization

Technical Details

- Code is compiled using JIT compilation
  - Compiled to LLVM code (can be seen using the @code_llvm macro)
- Types are very important, even if they are optional
  - Functions are called using multiple-dispatch
  - Think R style OOP on Steriods

In [ ]:

```
methods(+)
```

Popularity

- Since it debuted, Julia has gained a lot of fans in various communities
- Seems to be a large user-base growing in econ
  - The Federal Reserve Bank of New York has modeled the economy of the US using Julia
- Also popular with traditional large scale computing applications like Astrononmy
  - In September 2017, Julia became one of a handful of langauges capable of performing over 1 petaflop per second

Comparison to Python

- Because of its easy of use, Julia is a common language for Python programmers to play with, this usually goes one of two ways
  - It's actually not that fast
  - I can make Python just as fast
- The Julia team has written their own comparison (based on syntax) to many languages, available at https://docs.julialang.org/en/latest/manual/noteworthy-differences/?highlight=differences

Unicode Variable Names

- One of the more unique aspects of Julia is that mathematical symbols and non-Latin characters are very well supported
- To promote this, all Julia REPL systems, and most Julia IDEs allow auto completion to a non unicode character
  - Based on LATEX symbol names
- To type the letter alpha
  - Type `\alpha` followed immediately by a TAB

In [ ]:

```
x = 10
println(x)
```

In [ ]:

```
β = 0.1
println(β)
```

Numbers

- As a numerical computing language, Julia has a very robust number system
  - A large number of types
- Most mathematical functions are built in
- When typing large numbers, the _ (underscore) can be used a separator, it is simply ignored
- If overflow happens, cast the intenger to big using `big(number)`

In [ ]:

```
1_000_000_000 + 1
```

In [ ]:

```
1_00_00_00_00_00 + 1
```

In [ ]:
```
4000000000000000 * 4e300
```

In [ ]:
```
big(4000000000000000) * 4e300
```

Standard Mathematical Operations

- Mathematical functions in Julia are similar to functional programming languages in that they take many arguments

  `1 + 2 + 3 == +(1,2,3)`

- Julia has two divisions
  - `/` which is floating point division
  - ÷ or `div` which is intenger division
- When multipying variables with a number, no symbol is needed, just like in math

In [ ]:
```
3 + 2 + 1
```

In [ ]:

```
3 * 2 * 1
```

In [ ]:
```
4 ^ 5
```

In [ ]:

```
4 % 5
```

In [ ]:
```
@code_native 1 + 2 + 3
```

In [ ]:
```
@code_native +(1,2,3)
```

In [ ]:
```
3/2
```

In [ ]:
```
3 ÷2
```

In [ ]:
```
div(3,2)
```

In [ ]:

```
x = 20
4 * x + 3
```

In [ ]:

```
y = 10
(10)(4)(x * y) + 3
```

Built-In Mathematical Functions

- `sqrt` or $\sqrt{*}$
- `sin`, `cos`, etc.
- `lcm` and `gcd`
- `abs` and `sign`

In [ ]:

```
sqrt(200)
```

In [ ]:

```
√200
```

In [ ]:
```
sin(pi/2)
```

In [ ]:

```
cos(pi/2)
```

In [ ]:
```
lcm(100,5,20,40)
```

In [ ]:
```
gcd(100,5,20,40)
```

In [ ]:

```
abs(10)
```

In [ ]:

```
abs(-10)
```

In [ ]:

```
sign(-10)
```

Built-In Mathematical Constants

- `pi` or π
- `e`
- `golden` or φ (`\varphi` NOT `\phi`)

In [ ]:

```
sin(π/2)
```

In [ ]:
```
log(e)
```

In [ ]:

```
φ
```

User-Defined Functions

- To define a function in Julia, use the keyword `function`
  - The function definition is ended with the keyword `end`
- Short functions can be defined like `f(x) = x * x`
- Julia Functions support default values, named parameters, etc.

In [ ]:

```
function my_first_function(a,b,c)
    a + b * c
end
```

In [ ]:
```
my_first_function(1,2,4)
```

In [ ]:
```
my_first_function(1,2,3.5)
```

In [ ]:
```
my_first_function(1,4//5,4)
```

In [ ]:

```
function defaults(a,b=10,c=20)
    a + b + c
end
```

In [ ]:

```
methods(defaults)
```

In [ ]:

```
defaults(10)
```

In [ ]:

```
@code_native defaults(10)
```

In [ ]:

```
z(x) = 4x + 3
z(10)
```

Lambda

- Julia supports anonymous functions through a syntax similar to Java

```
arguments -> function_body
```

- Lambdas with multiple parameters should be wrapped up in a tuple

```
(a,b,c) -> function_body
```

In [ ]:

```
x = y->10y
x(10)
```

In [ ]:

```
map(x -> x*x , [1 2 3 4])
```

Arrays

- Arrays are one of the primary datatypes of Julia
  - Created using `[]`
  - Indexing starts at 1
  - Negative indexing is replaced with the `end` keyword
- All operators can be used on arrays as well
  - Special functions like `mean`, `median`, etc exist for arrays

In [ ]:

```
my_array = [1 2 3 4 5 6]
my_array[0]
```

In [ ]:
```
my_array[1]
```

In [ ]:

```
my_array[end]
```

In [ ]:

```
my_array[end-1]
```

In [ ]:

```
my_array = [1,2,3,4]
my_array + 4
```

In [ ]:

```
my_array * 4
```

In [ ]:

```
√my_array
```

In [ ]:

```
mean(my_array)
```

Multi-Dimensional Arrays

- To create a 2 dimensional array in Julia
  - Separate the elements by spaces
  - Separate the rows by semicolons

    ```
    [1 2 3 4; 5 6 7 8]
    ```

- Comma separated elements creates a column vector, space separated elements a row vector

In [ ]:

```
[1 2 3 4;
    5 6 7 8]
```

In [ ]:

```
[1 2 3 4; 5 6 7 8; 9 10 11 12]
```

In [ ]:

```
[ 1 2 3 4]
```

In [ ]:

```
[1,2,3,4]
```

Dot Notation on Function Names

- Any function you write can automatically be applied element-wise to an array
- Just append a dot `.` after the function name but before the parentheses
- This is faster than using `map` or a for loop most of the time

In [ ]:

```
c = 123456
test = [1 8 1 9 0 4 8 1 6 3]
f(x) = x + c
```

In [ ]:

```
@time map(f,test)
```

In [ ]:

```
@time test + c
```

In [ ]:
```
@time f.(test) #Only works in Julia 0.5 + :(
```

In [ ]:
```
two_d = [1 8 1; 9 0 4; 8 1 6]
@time map(f,two_d)
```

In [ ]:
```
@time f.(two_d)
```

In [ ]:
```
@time two_d + c
```

Types

- Much of Julia's speed comes from Type system
  - By Dynamically inferring types, the most optimized version (both in algorithm and in assembly) can be called
- To specify a type for a variable, use the syntax

```
name::TYPE
```

- To look at the built in type heirarchy, use the functions `subtypes` and `super` or `supertype` in new versions of Julia

In [ ]:

```
supertype(Number)
```

In [ ]:
```
supertype(Float64)
```

In [ ]:
```
subtypes(AbstractString)
```

In [ ]:

```
subtypes(Number)
```

In [ ]:
```
subtypes(Any)
```

In [ ]:

```
function typed(x)
    x ^ 3
end

function typed(x::Integer)
    x ^ 3
end
```

In [ ]:
```
@time typed(10.0)
```

In [ ]:
```
@time typed(10)
```

In [ ]:

```
## From https://en.wikibooks.org/wiki/Introducing_Julia/Types
function t1(n)
           s  = 0
           for i in 1:n
               s += s/i
           end
       end
```

In [ ]:

```
## From https://en.wikibooks.org/wiki/Introducing_Julia/Types
function t2(n)
           s  = 0.0
           for i in 1:n
               s += s/i
           end
       end
```

In [ ]:

```
@time t1(10000000)
```

In [ ]:

```
@time t2(10000000)
```

User Defined Types

- User defined types are just structs, similar to typedef
  - The functions that operate on them will be written separately, like in R
  - The constructor needs to have the same name as the type, and should call new() at the end,

```
struct name #(type in older versions)
  member1::type1
  member2::type2
end
```

- These user defined types can then be used to make new methods or overload existing ones

In [ ]:

```
type TIME #struct TIME in Julia > 0.5
    hour::Integer
    minute::Integer
end
```

In [ ]:
```
x = TIME(10,30)
```

In [ ]:
```
x.hour
```

Overloading Methods

- Now that we know about types, we can overload existing functions like +
- Define a function as you normally would, using the appropriate function name
  - `+` is properly known as `Base.:+`
- Specify your specific types as the parameters

In [ ]:

```
importall Base.Operators
function Base.:+(a::TIME, b::TIME) #New style is just Base.:+
    TIME(a.hour + b.hour, a.minute + b.minute)
end
```

In [ ]:

```
x + TIME(11,30)
```

Strings

- While Julia was concieved as a numerical computation langauge, processing strings is an important part of any language
- Strings must be delimited using double quotes
  - Single quotes indicate a character, which is a different data type
- Numerous string functions are available in the base class, including regular expression support
  - The concatentation operator is $*$ **NOT** $+$
  - Strings can be accessed like arrays

In [ ]:

```
string = "Hello"
uni = "Helαβ"
println(typeof(string) , " ", typeof(uni))
```

In [ ]:

```
string * uni
```

In [ ]:

```
string ^ 3
```

In [ ]:

```
another="Hello is $string and $uni"
```

For Loops

- For loops must be ended with the keyword `end`
- For loops always use the `in` keyword
  - To make a count style loop, use the array creation shortcut of start:step:end

In [ ]:

```
for x in [1 2 3 4 5]
    println(x)
end
```

In [ ]:

```
for x in 1:1:5
   println(x)
end
```

In [ ]:

```
for x in 1:5
    println(x)
end
```

Parallel For Loops

- The `@parallel` macro turns any loop into a parallel loop
  - Data is not shared between iterations of the loop by default
    - Can declare variables as shared
  - The `@parallel` macro can take one argument, which will be the reduce function

In [ ]:
```
nworkers()
```

In [ ]:
```
addprocs(4)
```

In [ ]:
```
nworkers()
```

In [ ]:

```
a = SharedArray{Float64}(10)
@parallel for i = 1:10
    print(i)
    a[i] = i
end

print(a)
```

In [ ]:

```
a = SharedArray{Float64}(10)
@sync @parallel for i = 1:10
    print(i)
    a[i] = i
end

print(a)
```

In [ ]:

```
a = SharedArray{Float64}(10)
fut = @parallel for i = 1:10
    print(i)
    a[i] = i
end
for x in fut
    fetch(x)
end
print(a)
```

In [ ]:
```
@time nheads = @parallel (+) for i = 1:200000000
    Int(rand(Bool))
end
```

In [ ]:
```
nheads_old = 0
@time for i = 1:200000000
    nheads_old += Int(rand(Bool))
end
```

If Statement

- If statements use the keywords `if`, `elseif`, `else`, and `end`
- The `end` keyword goes at the end of the entire block
- There is no special braces, colons, parentheses or anything else

In [ ]:

```
x = 20 + 4
if x > 50
    println("GOOD")
elseif x < 20
    println("BAD")
else
    println("OK")
end
```

In [ ]:

```
(2+3)::Float64
```

Modules

- Julia has a robust module system, and packages can be seend at https://pkg.julialang.org/
- To install new packages use the command `Pkg.add(PACKAGENAME)`
- To use the newly installed package use
  - `using` - Places functions in global namespace
  - `import` - Need to access using module name
- You can also include a file directly using `include()`, and `require()`