

Introduction to Probabilistic Machine Learning

Julia

Ralf Herbrich

- 2012 developed by Jeff Bezanson, Alan Edelman, Stefan Karpinski and Viral B. Shah at MIT
- Used for numerical and scientific computing with high performance
 - Execution speed is similar to C and FORTRAN
 - Hierarchical and parameterized type system as well as method overloading („multiple dispatching“) as central concepts
 - Native calls from C-(compiled) code possible (without wrappers)
- Unicode is efficiently supported (e.g., UTF-8)
- Alongside C, C++ and FORTRAN, the only programming language that has entered the “PetaFlop Club”



Jeff Bezanson
(1981–)



Alan Edelman
(1963 –)



Stefan Karpinski
(1981–)



Viral Shah
Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

Features of Julia

- **Just-in-time (JIT) compilation implemented using LLVM**
 - Approaching static compilation such as C!
- **Optional typing and type inference**
 - Every object has a type; type declaration *can* be made
 - Types are run-time objects
 - A rich language of types for constructing and describing objects
- **Multiple dispatch**
 - Functions are uniquely defined by their argument types
 - Alternative to classes in object-oriented programming
- **Very simple core language that imposes very little**
 - Julia Base and the standard library are written in Julia itself, including primitive operations like integer arithmetic
- **Language support for meta-programming**
 - Macros are a typed extensions to pre-processors that modify expression trees

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Overview

1. Installation
2. Variables and Functions
3. Types
4. Methods & Interfaces
5. Modules

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Overview

1. **Installation**
2. Variables and Functions
3. Types
4. Methods & Interfaces
5. Modules

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

- On command line, execute

- This will install the command line interface that can be launched with `julia`

Documentation: <https://docs.julialang.org>

```
Type "?" for help, "]?" for Pkg help.
```

Version 1.10.2 (2024-03-01)

Official <https://julialang.org/> release

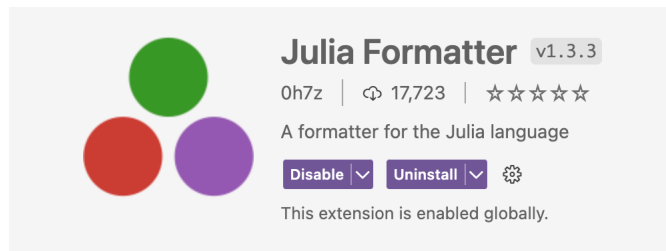
- The installation manager is called `juliaup` and you should regularly run

juliaup update

- `juliaup` can also be used to install several (older) versions of Julia in parallel!

Visual Studio Code Installation

- VS Code has excellent support for Julia but you should install two packages



- Syntax highlighting
- Integrated Julia REPL
- Code completion
- Linter
- Debugger
- Plot gallery
- Grid viewer for tabular data

- Automatic Julia code formatting

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Overview

1. Installation
- 2. Variables and Functions**
3. Types
4. Methods & Interfaces
5. Modules

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

- **Variables:** A name associated (or bound) to a value
 - Variables names can contain any Unicode character (best typed by `\<LaTeX name>-tab`)

```
julia> δ = 0.00001
1.0e-5

julia> 안녕하세요 = "Hello"
"Hello"
```

- A variable assignment with `=` *binds* the value but does not change it

```
julia> a = [1,2,3] # an array of 3 integers
3-element Vector{Int64}:
 1
 2
 3

julia> b = a # both b and a are names for the same array!
3-element Vector{Int64}:
 1
 2
 3

julia> a[1] = 42 # change the first element
42

julia> a = 3.14159 # a is now the name of a different object
3.14159

julia> b # b refers to the original array object, which has been mutated
3-element Vector{Int64}:
 42
 2
 3
```

- **Functions:** An object that maps a tuple of argument values to a return value
 - Julia functions are not pure mathematical functions: they can alter and be affected by the global state of the program

Long syntax

```
• julia> function f(x,y)
    x + y
end
f (generic function with 1 method)
```

Short syntax

```
• julia> f(x,y) = x + y
f (generic function with 1 method)
```

```
• julia> Σ(x,y) = x + y
Σ (generic function with 1 method)
```

- **Function call:** A function is called using parantheses syntax

```
• julia> f(2,3)
5
```

```
• julia> Σ(2, 3)
5
```

- **Functions are objects:** A function is an object that can be assigned

```
• julia> g = f
f (generic function with 1 method)
• julia> g(2,3)
5
```

Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

Julia Functions: Arguments and Returns

■ **Arguments:** Values are not copied when they are passed to functions

- Function arguments themselves act as new variable bindings
- Modifications to mutable values made within a function will be visible to the caller

Function definition

```

julia> function f(x, y)
    x[1] = 42      # mutates x
    y = 7 + y      # new binding for y, no mutation
    return y
end
f (generic function with 1 method)
    
```

- A common convention is to put an explanation mark in the function name if the function mutates an argument

■ **Return:** The value returned by a function is the value of the last expression evaluated

- The **return** keyword causes a function to return immediately

```

julia> function g(x,y)
    return x * y
    x + y
end
g (generic function with 1 method)

julia> g(2,3)
6
    
```

Function application

```

julia> a = [4,5,6]
3-element Vector{Int64}:
 4
 5
 6

julia> b = 3
3

julia> f(a, b) # returns 7 + b == 10
10

julia> a # a[1] is changed to 42 by f
3-element Vector{Int64}:
42
 5
 6

julia> b # not changed
3
    
```

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Julia Functions: Operators & Anonymous Functions

- **Operators:** Operators are just functions with support for special syntax

- Can also be applied using parenthesized argument lists
- Can also be assigned to other variables

```
• julia> +(1,2,3)
6
```

```
• julia> f = +
+ (generic function with 189 methods)
• julia> f(1,2,3)
6
```

- **Operators with Special Names:** A few special expressions correspond to calls to functions with special names

- These operators can be made work for custom types by defining type-specific methods (see soon!)

- **Anonymous Functions:** A function object without a name

- In other programming languages, they are called *lambda* functions

Long syntax

```
• julia> function (x)
    x^2 + 2x - 1
end
#5 (generic function with 1 method)
```

Short syntax

```
• julia> x -> x^2 + 2x - 1
#7 (generic function with 1 method)
```

Expression	Calls
[A B C ...]	<code>hcat</code>
[A; B; C; ...]	<code>vcats</code>
[A B; C D; ...]	<code>hvcats</code>
[A; B;; C; D;; ...]	<code>hvnvcats</code>
A'	<code>adjoint</code>
A[i]	<code>getindex</code>
A[i] = x	<code>setindex!</code>
A.n	<code>getproperty</code>
A.n = x	<code>setproperty!</code>

Julia Functions: Control Flow (Conditionals)

- **Compound expressions:** Several sub-expressions that evaluate to the final expression

Long syntax

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
3
```

Short syntax

```
julia> z = (x = 1; y = 2; x + y)
3
```

- **Conditional evaluation:** Allows portions of code to be evaluated or not evaluated depending on the value of a *Boolean* expression

Function definition

```
julia> function test(x, y)
           if x < y
               println("x is less than y")
           elseif x > y
               println("x is greater than y")
           else
               println("x is equal to y")
           end
       end
test (generic function with 1 method)
```

Function application

```
julia> test(1, 2)
x is less than y
julia> test(2, 1)
x is greater than y
julia> test(1, 1)
x is equal to y
```

- **Ternary Operator:** Shorthand for an **if-then-else-end** expression

```
julia> test(x, y) = println(x < y ? "x is less than y" :
                           x > y ? "x is greater than y" : "x is equal to y")
test (generic function with 1 method)
```

```
julia> test(1, 2)
x is less than y
julia> test(2, 1)
x is greater than y
julia> test(1, 1)
x is equal to y
```

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Julia Functions: Control Flow (Loops)

- **while-loop**: Evaluates the condition expression and as long it remains true, keeps also evaluating the body of the loop
- **for-loop**: Iterates through the container provided
 - A **for**-loop always introduces a new iteration variable in its body, regardless of whether a variable of the same name exists in the enclosing scope (a **while**-loop does not do that)

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

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

```
julia> for i in [1,4,0]
    println(i)
end
1
4
0
```

- **break**: Terminate the repetition of a **while**-loop before the test condition is false or stop iterating in a **for**-loop before the end of the iterable object is reached

```
julia> i = 1
1
julia> while true
    println(i)
    if i >= 3
        break
    end
    i += 1
end
1
2
3
```

```
julia> for j = 1:1000
    println(j)
    if j >= 3
        break
    end
end
1
2
3
```

Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

Overview

1. Installation
2. Variables and Functions
- 3. Types**
4. Methods & Interfaces
5. Modules

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Julia's Type System

- Julia's type system is **dynamic** (i.e. types are not known until run time)
 - There is no meaningful concept of a "compile-time type"!
 - It is possible to indicate that certain values are of specific types
 - Explicit types are used for method dispatching (see soon!)
- Julia's type system is also **nominative** (i.e., hierarchical relationships between types are explicitly declared)
 - Concrete types cannot subtype each other: all concrete types are final
 - Only abstract types can be supertypes
- Julia's type system is also **parametric**
 - Types can be parameterized by other types or by symbols (e.g. numbers)
 - Powerful for container types (e.g., lists)
- **Note:** Only values, not variables, have types!
 - Variables are simply names bound to values
 - "type of a variable" is shorthand for "type of the value to which a variable refers".

Julia Types: Type Declarations

- **:: operator:** Can be used to attach type annotations to expressions and variables in programs

```
⊙ julia> (1+2)::AbstractFloat
ERROR: TypeError: in typeassert, expected AbstractFloat, got a value of type Int64
Stacktrace:
 [1] top-level scope
      @ REPL[27]:1
● julia> (1+2)::Int
3
```

- **typeof** function: Returns the type of the value that is passed to it
- **Function return types:** Can be specified at function definition

typeof function

```
● julia> function foo()
    x::Int8 = 100
    x
end
foo (generic function with 1 method)
● julia> x = foo()
100
● julia> typeof(x)
Int8
```

Function return type

```
● julia> function g(x, y)::Int8
    return x * y
end;
● julia> typeof(g(1, 2))
Int8
```

Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

Julia Types: Abstract Types

- **Abstract Types:** Cannot be instantiated, and serve only as nodes in the type graph
 - Abstract types are declared using the **abstract type** keyword

```
abstract type «name» end  
abstract type «name» <: «supertype» end
```

- **:< function:** Allows to specify a supertype relationship or test for it

Type declarations

```
• julia> abstract type Number end  
• julia> abstract type Real          <: Number end  
• julia> abstract type AbstractFloat <: Real end  
• julia> abstract type Integer       <: Real end  
• julia> abstract type Signed        <: Integer end  
• julia> abstract type Unsigned      <: Integer end
```

Supertype check

```
• julia> Integer <: Number  
true  
• julia> Integer <: AbstractFloat  
false
```

Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

- No loss in performance when using function whose arguments are abstract types
 - They are recompiled for each tuple of concrete argument types with which it is invoked!

Julia Types: Primitive Types

- **Primitive Types:** Concrete type whose data consists of plain old bits

- Primitive types are declared using the **primitive type** keyword

```
primitive type «name» «bits» end  
primitive type «name» <: «supertype» «bits» end
```

- The number of bits indicates how much storage the type requires

Type declarations

```
• julia> primitive type F16      <: AbstractFloat 16 end  
• julia> primitive type F32      <: AbstractFloat 32 end  
• julia> primitive type F64      <: AbstractFloat 64 end  
• julia> primitive type Integer8 <: Signed      8 end  
• julia> primitive type UInteger8 <: Unsigned   8 end
```

Supertype check

```
• julia> Integer8 <: Integer  
true  
• julia> Integer8 <: Real  
true  
• julia> Integer8 <: AbstractFloat  
false
```

Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

- Unusual to redefine primitive types as Julia's primitive types are highly performant

Julia Types: Composite Types

■ **Composite Types:** A collection of named fields

- Composite types are declared using the **struct** keyword
- New objects are created by applying the type name as a function (*default constructor*)
- Fields can be accessed with the property notation `<variable> . <fieldname>`

```
julia> struct Foo
    bar
    baz::Int
    qux::Float64
end
```

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo
```

```
julia> foo.bar
"Hello, world."

julia> foo.baz
23

julia> foo.qux
1.5
```

■ Composite objects declared with **struct** are immutable!

- Efficient and easier to reason about for the (just-in-time) compiler (can use registers!)

■ Composite objects declared with **mutable struct** are mutable!

- Such objects are generally allocated on the heap and have stable memory addresses

```
julia> mutable struct Bar
    baz
    qux::Float64
end

julia> bar = Bar("Hello", 1.5)
Bar("Hello", 1.5)
```

```
julia> bar.qux = 2.0
2.0

julia> bar.baz = 1//2
1//2
```

Julia Types: Parametric Types

- **Parametric Types:** Types that have parameters that introduce a whole family of new types – one for each possible combination of parameter values
 - Type parameters are introduced after the type name, surrounded by curly braces
 - The type parameter can be any type at all (or a value of any primitive type!)

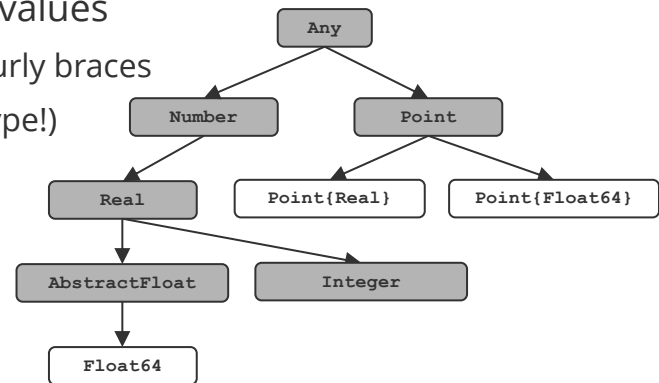
```
julia> struct Point{T}
    x::T
    y::T
end
```

```
julia> p1 = Point{Float64}(1.0, 2.0)
Point{Float64}(1.0, 2.0)

julia> p2 = Point(1,2)
Point{Int64}(1, 2)

julia> typeof(p1)
Point{Float64}

julia> typeof(p2)
Point{Int64}
```



- The non-parametric type is a super-type of any specialized parametric type!

```
julia> Point{Float64} <: Point
true

julia> Point{Float64} <: Point{Real}
false

julia> Point{Float64} <: Point{<:Real}
true
```

Any type that is a subtype of **Real**

Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

Julia Types: Tuple Types

- **Tuple Types:** Parameterized immutable type where each parameter is the type of one field

- Tuple types may have any number of type parameters
- Tuples do not have field names; fields are only accessed by index
- Tuple types are *covariant* in their parameters

```
julia> struct Tuple2{A,B}
    a::A
    b::B
end
```

```
julia> x = (1,"foo",2.5)
(1, "foo", 2.5)
```

```
julia> typeof(x)
Tuple{Int64, String, Float64}

julia> x[1]
1
```

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,Any}
true

julia> Tuple{Int,AbstractString} <: Tuple{Real,Real}
false
```

- **Named Tuple Types:** Tuple types with field names

- Field names can be used to access elements

```
julia> x = (a=1,b="hello")
(a = 1, b = "hello")

julia> y = NamedTuple{(:a, :b)}((1, ""))
(a = 1, b = "")
```

```
julia> typeof(x)
@NamedTuple{a::Int64, b::String}

julia> typeof(y)
@NamedTuple{a::Int64, b::String}
```

```
julia> x.b
"hello"

julia> y.a
1
```

Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

- Tuple types are an abstraction of the arguments of a function

- [illegible]

23/31

Julia Types: Strings

- Strings are finite sequences of characters

- The built-in concrete type used for strings (and string literals) in Julia is **String**
- It supports the full range of Unicode characters via the UTF-8 encoding
- String literals are delimited by double quotes or triple double quotes
- Strings are arrays of characters with the special indexers **begin** and **end**

```
julia> s2 = "\u2200 x \u2203 y"
"∀ x ∃ y"
```

```
julia> str = "Hello, world.\n"
"Hello, world.\n"

julia> s1 = """Contains "quote" characters"""
"Contains \"quote\" characters"
```

```
julia> str[begin]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[1]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[end]
'\n': ASCII/Unicode U+000A (category Cc: Other, control)

julia> str[end-1]
'.': ASCII/Unicode U+002E (category Po: Punctuation, other)
```

- Strings are concatenated with ***** and interpolated with **\$**

- The shortest complete expression after the **\$** is taken as the expression whose value is to be interpolated into the string

```
julia> greet * ", " * whom * ".\n"
"Hello, world.\n"

julia> "$greet, $whom.\n"
"Hello, world.\n"
```

```
julia> greet = "Hello"
"Hello"

julia> whom = "world"
"world"
```

- Julia supports regular expression with the **r** string prefix for the regex:

```
julia> m = match(r"^s*(?:#s*(.*)s*$|s)", "# a comment ")
RegexMatch("# a comment ", 1="a comment")
```

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Overview

1. Installation
2. Variables and Functions
3. Types
- 4. Methods & Interfaces**
5. Modules

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

- A function is an object that maps a tuple of arguments to a return value
 - It is common for the same conceptual function or operation to be implemented quite differently for different types of arguments

- **Method:** A definition of one possible behavior for a function

```
• julia> f(x::Float64, y::Float64) = 2x + y
f (generic function with 1 method)
```

```
• julia> f(x::Number, y::Number) = 2x - y
f (generic function with 2 methods)
```

```
• julia> methods(f)
# 2 methods for generic function "f" from Main:
[1] f(x::Float64, y::Float64)
    @ REPL[100]:1
[2] f(x::Number, y::Number)
    @ REPL[101]:1
```

- **Method dispatch:** Choice of method to execute when a function is applied

- Multiple dispatch: Using all of a function's arguments to choose which method

```
• julia> f(2.0, 3.0)
7.0
• julia> f(2.0, 3)
1.0
```

```
• julia> f("foo", 3)
ERROR: MethodError: no method matching f(::String, ::Int64)
Closest candidates are:
  f(::Number, ::Number)
    @ Main REPL[101]:1
```

```
• julia> f(x,y) = println("Whoa there, Nelly.")
f (generic function with 3 methods)
• julia> f("foo", 1)
Whoa there, Nelly.
```

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Julia Parametric Methods

- Method definitions can have type parameters qualifying the signature

```
• julia> same_type(x::T, y::T) where {T} = true
  same_type (generic function with 1 method)

• julia> same_type(x,y) = false
  same_type (generic function with 2 methods)
```

```
• julia> same_type(1, 2)
  true

• julia> same_type(1.0, 2.0)
  true
```

```
• julia> same_type(1, 2.0)
  false

• julia> same_type(Int32(1), Int64(2))
  false
```

- Similar to subtype constraints on type parameters in type declarations one can also constrain type parameters of methods using **where**:

```
• julia> same_type_numeric(x::T, y::T) where {T<:Number} = true
  same_type_numeric (generic function with 1 method)

• julia> same_type_numeric(x::Number, y::Number) = false
  same_type_numeric (generic function with 2 methods)
```

```
• julia> same_type_numeric(1, 2)
  true

• julia> same_type_numeric(1.0, 2.0)
  true
```

```
• julia> same_type_numeric(1, 2.0)
  false

• julia> same_type_numeric("foo", 2.0)
  ERROR: MethodError: no method matching same_type_numeric(::String, ::Float64)
```

- Constructor methods:** Constructor is just like any other function in Julia

- Defining a method with the same name as composite type is called an *outer constructor*

```
• julia> struct OrderedPair
    x::Real
    y::Real
    OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
end
```

```
• julia> OrderedPair(a) = OrderedPair(a,a)
  OrderedPair

• julia> OrderedPair(10)
  OrderedPair{Int64}(10, 10)
```

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Julia Interfaces

- **Interface:** By extending a few specific methods to work for a custom type, objects of that type can be used in other methods that are written to generically build upon those behaviors.
- **Example:** Iterator interface

```
for item in iter # or "for item = iter"
    # body
end
```



```
next = iterate(iter)
while next != nothing
    (item, state) = next
    # body
    next = iterate(iter, state)
end
```

Custom type

```
julia> struct Squares
    count::Int
end
```

Interface definition

```
julia> Base.iterate(S::Squares, state=1) = state > S.count ? nothing : (state*state, state+1)
```

Interface usage

```
julia> for item in Squares(7)
    println(item)
end

1
4
9
16
25
36
49
```

```
julia> 25 in Squares(10)
true

julia> sum(Squares(100))
338350
```

Introduction to
Probabilistic Machine
Learning

Week 1 - Julia

Julia Interfaces (ctd)

- **Indexing** interface allows any composite type to behave like an array
 - This is true even if the type never explicitly stores an array!
- **Example:** Array of square numbers

```
julia> function Base.getindex(S::Squares, i::Int)
    1 <= i <= S.count || throw(BoundsError(S, i))
    return i*i
end

julia> Base.length(S::Squares) = S.count
julia> Base.firstindex(S::Squares) = 1
julia> Base.lastindex(S::Squares) = length(S)
```

```
julia> Squares(100)[23]
529

julia> Squares(23)[end]
529
```

Methods to implement	Brief description
<code>getindex(X, i)</code>	<code>X[i]</code> , indexed element access
<code>setindex!(X, v, i)</code>	<code>X[i] = v</code> , indexed assignment
<code>firstindex(X)</code>	The first index, used in <code>X[begin]</code>
<code>lastindex(X)</code>	The last index, used in <code>X[end]</code>

- The interface even allows to support new indexing modes!

```
julia> Base.getindex(S::Squares, i::Number) = S[convert{Int, Int}(i)]
julia> Base.getindex(S::Squares, I) = [S[i] for i in I]
```

```
julia> Squares(10)[[3,4,5]]
3-element Vector{Int64}:
 9
16
25
```

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

Overview

1. Installation
2. Variables and Functions
3. Types
4. Methods & Interfaces
- 5. Modules**

**Introduction to
Probabilistic Machine
Learning**

Week 1 - Julia

- **Modules:** Help organize code into coherent units
 - Modules are separate namespaces, each introducing a new global scope
 - Same name can be used for different functions or global variables without conflict
 - Can be precompiled for faster loading, and may contain code for runtime initialization
- **Module Definition:** Using the keywords `module` ... `end`
 - Explicit exports of symbols are done with `export`

```
julia> module NiceStuff
    export nice, DOG
    struct Dog end # singleton type, not exported
    const DOG = Dog() # named instance, exported
    nice(x) = "nice $x" # function, exported
end
Main.NiceStuff
```

- **Module Usage:** Using the keywords `using`
 - Local defined modules start with a dot

```
julia> using .NiceStuff
julia> nice("house 🏠")
"nice house 🏠"
```

■ Julia Concepts

- Runs programs via JIT compilation implemented using LLVM (nearly as fast as C!)
- Julia is 12 years old and has reached wide-spread adoption in scientific computing
- Extensibility comes from using *method dispatch* and unbundling functions and data
- Julia provides garbage collection and requires no explicit memory management

■ Julia Type System

- Julia's type system is dynamic, nominative and parametric
- It only has abstract, primitive and composite types (and the latter ones are all final)
- Every object has a type; type declaration can be made

■ Julia Methods

- A method is the concrete definition of a function's behavior
- Every function can have many methods; Julia run-time picks the right method based on the types of the arguments (*method dispatch*)

- We only scratched the surface; keep using the language to get confident!

See you next week!