julia-intro

February 27, 2017

1 Introduction to Julia

Pearl Li March 10th and 13th, 2017

1.0.1 Outline

1. Syntax Review
2. Types and Multiple Dispatch
3. Exercises

1.1 Syntax Review

1.1.1 Hello World

```
In [1]: println("Hello world!")
```

```
Hello world!
```

1.1.2 Variable Assignment

```
In [2]: # Assign the value 10 to the variable x
        x = 10
```

```
In [3]: # Variable names can have Unicode characters
        # To get ϵ in the REPL, type \epsilon<TAB>
        ϵ = 1e-4
```

```
Out[3]: 0.0001
```

In Julia, a variable name is just a reference to some data, not the piece of data itself. Multiple names can be associated with the same piece of data, unlike in MATLAB, where the name of a piece of data is bound to the data itself.

Variable names are case-sensitive. By convention, they are in snake_case.

1.1.3 Numbers

Basic operations:

```
In [4]: y = 2 + 2
```

```
Out[4]: 4
```

```
In [5]: -y
```

```
Out[5]: -4
```

```
In [6]: 0.2 * 10
```

```
Out[6]: 2.0
```

```
In [7]: 3 / 4
```

```
Out[7]: 0.75
```

```
In [8]: # Scalar multiplication doesn't require *
        3(4 - 2)
```

```
Out[8]: 6
```

Built-in constants and functions:

```
In [9]: e
```

```
Out[9]: e = 2.7182818284590...
```

```
In [10]: sqrt(9)
```

```
Out[10]: 3.0
```

```
In [11]: log(e^10)
```

```
Out[11]: 10.0
```

1.1.4 Booleans

Equality comparisons:

```
In [12]: 0 == 1
```

```
Out[12]: false
```

```
In [13]: 2 != 3
```

```
Out[13]: true
```

```
In [14]: 3 <= 4
```

```
Out[14]: true
```

Boolean operators:

```
In [15]: true && false
```

```
Out[15]: false
```

```
In [16]: true || false
```

```
Out[16]: true
```

```
In [17]: !true
```

```
Out[17]: false
```

1.1.5 Strings

```
In [18]: # Strings are written using double quotes
         str = "This is a string"
```

```
Out[18]: "This is a string"
```

```
In [19]: # Strings can also contain Unicode characters
         fancy_str = "α is a string"
```

```
Out[19]: "α is a string"
```

```
In [20]: # String interpolation using $
         # The expression in parentheses is evaluated and the result is
         # inserted into the string
         "2 + 2 = $(2+2)"
```

```
Out[20]: "2 + 2 = 4"
```

```
In [21]: # String concatenation using *
         "hello" * "world"
```

```
Out[21]: "helloworld"
```

```
In [22]: # Is "string" a substring of str?
         contains(str, "string")
```

```
Out[22]: true
```

1.1.6 Functions

In [23]:
```
# Regular function definition
function double(x)
    y = 2x
    return y
end
```

Out[23]: double (generic function with 1 method)

In [24]:
```
# Inline function definition
inline_double(x) = 2x
```

Out[24]: inline_double (generic function with 1 method)

In [25]:
```
# Functions can refer to variables that are in scope when the
# function is defined
a = 5
add_a(x) = x + a
add_a(1)
```

Out[25]: 6

In [26]:
```
# Functions can return multiple arguments
duple_of(x) = x, x + 1
a, b = duple_of(3)
```

Out[26]: (3,4)

In [27]:
```
# Optional arguments - no more varargin!
function add_5(x, n_times = 1)
    for i = 1:n_times
        x = x + 5
    end
    return x
end

# Call with one argument
add_5(0)
```

Out[27]: 5

In [28]:
```
# Call with two arguments
add_5(0, 3)
```

Out[28]: 15

In [29]:
```
# Keyword arguments allow arguments to be identified by name
# instead of only by position
function join_strings(string1, string2; separator = ",")
```

```
    return string1 * separator * string2
end

# Call without keyword argument
join_strings("ciao", "mondo")
```

```
Out[29]: "ciao,mondo"
```

```
In [30]: # Call with keyword argument
         join_strings("ciao", "mondo"; separator = " ")
```

```
Out[30]: "ciao mondo"
```

1.1.7 Arrays

Explicit array construction:

```
In [31]: A = [1, 2]
```

```
Out[31]: 2-element Array{Int64,1}:
          1
          2
```

```
In [32]: B = [1 2 3; 4 5 6]
```

```
Out[32]: 2×3 Array{Int64,2}:
          1  2  3
          4  5  6
```

One-dimensional arrays `Array{Int64,1}` are also called (type-aliased) `Vector{Int64}`s. Two-dimensional arrays are called `Matrix{Int64}`s.

Note that `A` is a `Vector{Int64}` of length 2, which is distinct from a `Matrix{Int64}` of size 2×1 (like a MATLAB "column vector") or a `Matrix{Int64}` or size 1×2 ("row vector").

Built-in array constructors:

```
In [33]: zeros(2)
```

```
Out[33]: 2-element Array{Float64,1}:
          0.0
          0.0
```

```
In [34]: ones(2)
```

```
Out[34]: 2-element Array{Float64,1}:
          1.0
          1.0
```

```
In [35]: eye(2)
```

```
Out[35]: 2×2 Array{Float64,2}:
          1.0  0.0
          0.0  1.0
```

```
In [36]: fill(true, 2)

Out[36]: 2-element Array{Bool,1}:
          true
          true
```

Matrix operations:

```
In [37]: # Matrix transpose
         B'

Out[37]: 3×2 Array{Int64,2}:
          1  4
          2  5
          3  6

In [38]: # Matrix addition
         B + B

Out[38]: 2×3 Array{Int64,2}:
          2   4   6
          8  10  12

In [39]: # Add a matrix to a vector using broadcasting
         B .+ A

Out[39]: 2×3 Array{Int64,2}:
          2  3  4
          6  7  8

In [40]: # Matrix inverse
         C = 4*eye(2)
         inv(C)

Out[40]: 2×2 Array{Float64,2}:
          0.25  0.0
          0.0   0.25

In [41]: # Elementwise operations
         B .> 3

Out[41]: 2×3 BitArray{2}:
          false  false  false
           true   true   true
```

Access array elements using square brackets:

```
In [42]: # First row of B
         B[1, :]
```

```
Out[42]: 3-element Array{Int64,1}:
          1
          2
          3
```

```
In [43]: # Element in row 2, column 3 of B
         B[2, 3]
```

```
Out[43]: 6
```

1.1.8 Control Flow

If statements:

```
In [44]: x = -3
         if x < 0
             println("x is negative")
         elseif x > 0 # optional and unlimited
             println("x is positive")
         else         # optional
             println("x is zero")
         end
```

```
x is negative
```

While loops:

```
In [45]: i = 3
         while i > 0
             println(i)
             i = i - 1
         end
```

```
3
2
1
```

For loops:

```
In [46]: # Iterate through ranges of numbers
         for i = 1:3
             println(i)
         end
```

```
1
2
3
```

```
In [47]: # Iterate through arrays
         cities = ["Boston", "New York", "Philadelphia"]
         for city in cities
             println(city)
         end
```

```
Boston
New York
Philadelphia
```

```
In [48]: # Iterate through arrays of tuples using zip
         states = ["MA", "NY", "PA"]
         for (city, state) in zip(cities, states)
             println("$city, $state")
         end
```

```
Boston, MA
New York, NY
Philadelphia, PA
```

```
In [49]: # Iterate through arrays and their indices using enumerate
         for (i, city) in enumerate(cities)
             println("City $i is $city")
         end
```

```
City 1 is Boston
City 2 is New York
City 3 is Philadelphia
```

1.2 Types and Multiple Dispatch

A **data type** is a classification identifying the kind of data you have. An object's type determines the possible values it can take on, which operations and functions can be applied to it, and how the computer stores it.

Examples:

- Numeric types: `Int64`, `Float64`
- String types: `ASCIIString`, `UTF8String`
- `Bool`
- `Array`

Names of types are written in UpperCamelCase.

A **concrete instance** (also an object or a value) of a type `T` is a piece of data in memory that has type `T`.

Variables are not data, but are simply names that point/refer to a specific piece of data. The underlying data that a variable refers to has a specific type.

```
In [50]: # What is the type of 10?
         typeof(10)

Out[50]: Int64

In [51]: # Is 10 an Int64?
         isa(10, Int64)

Out[51]: true

In [52]: # What is the type of the elements of an array?
         X = [1.0, 2.0, 3.0]
         eltype(X)

Out[52]: Float64
```

1.2.1 Composite Types

A **composite type** is a collection of named fields that can be treated as a single value. They bear a passing resemblance to MATLAB structs.

All fields must be declared ahead of time. The double colon, `::`, constrains a field to contain values of a certain type. This is optional for any field.

```
In [53]: # Type definition
         type Parameter
             value::Float64
             transformation::Function # Function is a type!
             tex_label::String
             description::String
         end
```

When a type with n fields is defined, a constructor (function that creates an instance of that type) that takes n ordered arguments is automatically created. Additional constructors can be defined for convenience.

```
In [54]: # Creating an instance of the Parameter type using the default
         # constructor
         β = Parameter(0.9, identity, "\beta", "Discount rate")

Out[54]: Parameter(0.9,identity,"\beta","Discount rate")

In [55]: # Alternative constructors end with an appeal to the default
         # constructor
         function Parameter(value::Float64, tex_label::String)
             transformation = identity
             description = "No description available"
             return Parameter(value, transformation, tex_label, description)
         end

         α = Parameter(0.5, "\alpha")
```

```
Out[55]: Parameter(0.5,identity,"\alpha","No description available")
```

```
In [56]: # Find the fields of an instance of a composite type
         fieldnames(α)
```

```
Out[56]: 4-element Array{Symbol,1}:
          :value
          :transformation
          :tex_label
          :description
```

```
In [57]: # Access a particular field using .
         α.value
```

```
Out[57]: 0.5
```

```
In [58]: # Fields are modifiable and can be assigned to, like
         # ordinary variables
         α.value = 0.75
```

```
Out[58]: 0.75
```

1.2.2 Subtyping

Types are hierarchically related to each other. All are subtypes of the `Any` type.
There are two main kinds of types in Julia:

1. Concrete types: familiar types that you can create instances of, like `Int64` or `Float64`.
2. Abstract types: nodes in a type graph that serve to group similar kinds of objects. Abstract types cannot be instantiated and do not have explicitly declared fields. For example, `Integer` or `Number`.

```
In [59]: # Define an abstract type
         abstract Model
```

```
In [60]: # Define concrete subtypes of that abstract type
         type VAR <: Model
             n_lags::Int64
             variables::Vector{Symbol}
             coefficients::Matrix{Float64}
         end
```

```
In [61]: # Check subtyping relation
         VAR <: Model
```

```
Out[61]: true
```

```
In [62]: # Instances of the VAR type are also instances of the Model type
         model = VAR(1, [:gdp, :inflation], eye(2))
         isa(model, Model)
```

```
Out[62]: true
```

```
In [63]: # Why does this throw an error?
         3 <: Number
```

```
TypeError: subtype: expected Type{T}, got Int64
```

1.2.3 Parameterized Types

Parameterized types are data types that are defined to handle values identically regardless of the type of those values.

Arrays are a familiar example. An `Array{T,1}` is a one-dimensional array filled with objects of any type `T` (e.g. `Float64`, `String`).

```
In [64]: # Defining a parametric point
         type Duple{T} # T is a parameter to the type Duple
             x::T
             y::T
         end
```

This single declaration defines an unlimited number of new types: `Duple{String}`, `Duple{Float64}`, etc. are all immediately usable.

```
In [65]: Duple(3, -15)
```

```
Out[65]: Duple{Int64}(3,-15)
```

```
In [66]: Duple("Broadway", "42nd St")
```

```
Out[66]: Duple{String}("Broadway","42nd St")
```

```
In [67]: # What happens here?
         Duple(1.5, 3)
```

```
MethodError: no method matching Duple{T}(::Float64, ::Int64)
Closest candidates are:
  Duple{T}{T}(::T, ::T) at In[64]:3
  Duple{T}{T}(::Any) at sysimg.jl:53
```

We can also restrict the type parameter `T`:

```
In [68]: # T can be any subtype of Number, but nothing else
         type PlanarCoordinate{T<:Number}
             x::T
             y::T
         end

In [69]: PlanarCoordinate("4th Ave", "14th St")
```

```
        MethodError: no method matching PlanarCoordinate{T<:Number}(::String, ::Str
    Closest candidates are:
      PlanarCoordinate{T<:Number}{T}(::Any) at sysimg.jl:53
```

1.2.4 Why Use Types?

You can write all your code without thinking about types at all. If you do this, however, you'll be missing out on some of the biggest benefits of using Julia.

If you understand types, you can:

- Write faster code
- Write expressive, clear, and well-structured programs (keep this in mind when we talk about functions)
- Reason more clearly about how your code works

Even if you only use built-in functions and types, your code still takes advantage of Julia's type system. That's why it's important to understand what types are and how to use them.

```
In [70]: # Example: writing type-stable functions
         function f_unstable()
             sum = 0
             for i = 1:100_000 # start:step:stop
                 sum = sum + i/2
             end
         end

         function f_stable()
             sum = 0.0
             for i = 1:100_000
                 sum = sum + i/2
             end
         end

         # Compile and run
         f_unstable()
         f_stable()
```

```
In [71]: @time f_unstable()

  0.003024 seconds (300.13 k allocations: 4.585 MB)


In [72]: @time f_stable()

  0.000002 seconds (4 allocations: 160 bytes)
```

In `f_stable`, the compiler is guaranteed that `sum` is of type `Float64` throughout; therefore, it saves time and memory. Because `f_stable` starts with `sum` as a `Float64`, it's much faster than `f_unstable` (which starts with sum as an `Int64`.

1.2.5 Multiple Dispatch

So far we have defined functions over argument lists of any type. Methods allow us to define functions "piecewise". For any set of input arguments, we can define a **method**, a definition of one possible behavior for a function.

```
In [73]: # Define one method of the function print_type
         function print_type(x::Number)
             println("$x is a number")
         end

Out[73]: print_type (generic function with 1 method)

In [74]: # Define another method
         function print_type(x::String)
             println("$x is a string")
         end

Out[74]: print_type (generic function with 2 methods)

In [75]: # Define yet another method
         function print_type(x::Number, y::Number)
             println("$x and $y are both numbers")
         end

Out[75]: print_type (generic function with 3 methods)

In [76]: # See all methods for a given function
         methods(print_type)

Out[76]: # 3 methods for generic function "print_type":
         print_type(x::String) at In[74]:3
         print_type(x::Number) at In[73]:3
         print_type(x::Number, y::Number) at In[75]:3
```

Julia uses **multiple dispatch** to decide which method of a function to execute when a function is applied. In particular, Julia compares the types of *all* arguments to the signatures of the function's methods in order to choose the applicable one, not just the first (hence "multiple").

```
In [77]: print_type(5)
```

```
5 is a number
```

```
In [78]: print_type("foo")
```

```
foo is a string
```

```
In [79]: # This throws an error because no method of print_type has been
         # defined for this set of arguments
         print_type([1, 2, 3])
```

```
        MethodError: no method matching print_type(::Array{Int64,1})
    Closest candidates are:
      print_type(::String) at In[74]:3
      print_type(::Number) at In[73]:3
      print_type(::Number, ::Number) at In[75]:3
```

How is multiple dispatch useful for economic research? Recall that we defined the type `VAR` earlier, and made it a subtype of our abstract type `Model`. Let's define another subtype of `Model`:

```
In [80]: # Define a general linear model
         type GLM <: Model
             independent_variables::Vector{Symbol}
             dependent_variables::Vector{Symbol}
             coefficients::Matrix{Float64}
         end
```

Now we can use the same function name, `estimate`, to define different estimation behaviors for the different subtypes of `Model`:

```
In [81]: using Distributions

         function estimate(model::GLM)
             # Estimate a general linear model using OLS
         end

         function estimate(model::VAR)
             # Estimate a VAR using maximum likelihood
         end

         function estimate(model::VAR, prior::Distribution)
             # Estimate a Bayesian VAR
         end
```

```
Out[81]: estimate (generic function with 3 methods)

In [82]: methods(estimate)

Out[82]: # 3 methods for generic function "estimate":
         estimate(model::VAR) at In[81]:9
         estimate(model::GLM) at In[81]:5
         estimate(model::VAR, prior::Distributions.Distribution) at In[81]:13
```

1.2.6 Writing Julian Code

As we've seen, you can use Julia just like you use MATLAB and get faster code. However, to write faster and *better* code, attempt to write in a "Julian" manner:

- Define composite types as logically needed
- Write type-stable functions for best performance
- Take advantage of multiple dispatch to write code that looks like math
- Add methods to existing functions

1.2.7 Just-in-Time Compilation

How is Julia so fast? Julia is just-in-time (JIT) compiled, which means (according to this StackExchange answer, with emphasis mine):

> A JIT compiler runs after the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set. *A JIT has access to dynamic runtime information whereas a standard compiler doesn't and can make better optimizations like inlining functions that are used frequently.*
>
> This is in contrast to a traditional compiler that compiles all the code to machine language before the program is first run.

In particular, Julia uses type information at runtime to optimize how your code is compiled. This is why writing type-stable code makes such a difference in speed!

1.3 Exercises

Taken from QuantEcon's Julia Essentials and Vectors, Arrays, and Matrices lectures.

1. Given two vectors `x` and `y`, both of type `Vector{Float64}`, compute their inner product using `zip`.

2. Consider the polynomial

$$p(x) = \sum_{i=0}^{n} a_0 x^0$$

   Using `enumerate`, write a function `p` such that `p(x, coeff)` computes the value of the polynomial with coefficients `coeff` evaluated at `x`.

3. Write a function `linapprox` that takes as arguments:

- A function `f` mapping some interval $[a, b]$ into $\mathbb{R}$
- Two scalars `a` and `b` providing the limits of this interval
- An integer `n` determining the number of grid points
- A number `x` satisfying $a \leq x \leq b$

and returns the piecewise linear interpolation of `f` at `x`, based on `n` evenly spaced grid points `a = point[1] < point[2] < ... < point[n] = b`. Aim for clarity, not efficiency.

4. Write a function `solve_discrete_lyapunov` that solves the discrete Lyapunov equation

$$S = ASA' + \Sigma\Sigma'$$

using the iterative procedure

$$S_0 = \Sigma\Sigma'$$

$$S_{t+1} = AS_tA' + \Sigma\Sigma'$$

taking in as arguments the $n \times n$ matrix A, the $n \times k$ matrix Σ, and a number of iterations.