

2. *A short overview of basic syntax*

Now let's learn Julia. I will assume you have some previous knowledge in programming. In particular you are somewhat familiar with the concepts,

- variables
- arrays or vectors
- loops (for and while)
- conditionals (if/else)
- strings
- functions

If you are familiar to some but not all, it's fine for me, but you need to catch up as we go along. We will discuss these concepts, but definitely not in a way that is appropriate for someone who is learning these things for the first time.

Versions

I am using the Julia version 1.5.4. However, if you are using 1.5.3., I think it is fine for now. There is a way we can solve this issue, by creating an **environment**, but I decided to talk about this later. At this point I think it is not strictly necessary. However I must admit that creating an environment is the neat way to solve all kinds of package or version clashes. This also ensures that an environment for running code is reproducible, so that anyone can replicate the precise set of package and versions used in construction. So bottom line - this is important but for later.

Other References

- The definitive reference is [Julia's own documentation](#).
- Also look at the page [Quantitative Economics with Julia](#). This lecture is heavily borrowed from there and the 2020 JuliaCon talk from David Sanders. You will find the complete list of references below.

The Julia manual or documentation is really thorough piece of work, but also it could be overwhelming if you are not used to reading manuals (but you will get used to with it hopefully after this course!)

*1. Using **function** in Julia*

Some functions are built into the **Base** Julia, such as **rand**, which returns by default a single draw from a uniform distribution between 0 and 1. But it can be also be used for other things. So let's see:

```
## CodeBlock 1.
rand() # draw one random number
rand(10) # draw a vector of random numbers, notice the dimension

rand(3, 3) # draw a matrix of random numbers
rand(1:5) # draw a random number from 1 to 5 (range type, will see later!)
rand(1:5, 2, 4) # a 2 by 4 matrix of random integers between 1 to 5
rand("stat", "business", "econ", "data science", "physics") ## will throw an error
rand(("stat", "business", "econ", "data science", "physics")) ## fine
rand(["stat", "business", "econ", "data science", "physics"]) ## fine
```

As you can see the **rand** function looks pretty powerful, it can produce a single random number or a bunch of random numbers and they can be very different objects. Let's closely look at **rand** using the Julia help. The way to do this is, you put a **?** in the REPL (read eval print loop) of Julia, and type **rand**. Let's analyze the output (output omitted).

Sometimes we might have to use some external packages in Julia to do different things. For example, for plotting we need the plots package. So let's generate some numbers, save them in a variable and then plot them, here

```
## CodeBlock 2.
# ] add Plots ## add the package, this you have to do only once in your lifetime
using Plots # everytime you use any package, you have to use `using`
x = rand(10) # x is a vector
plot(x) # by default it plots a line plot
scatter(x) # this will plot a scatter plot
```

There is another function which is called **randn** function which will do exact same thing, however it will generate random numbers from a **standard normal distribution**. This means in this case we can plot the data coming from a so called **white noise process**,

```
## CodeBlock 3.
n = 100 # number of periods
ϵ = randn(n) # this is the random numbers, notice we used epsilon type \epsilon + <Tab>
plot(1:n, ϵ) # finally plot the line plot!

# however we can do this in one line, but in Julia there is no difference
plot(randn(100))
```

This is what we call the pre-defined or built-in functions in Julia. The functions are already defined, and we are just calling the functions. However, there is a very important thing you

should observe, that is, the function name is same, but the arguments we are giving are very different. In fact we can look at all these options at once, if we type another function called **methods**, so let's do this.

```
## CodeBlock 4.  
methods(rand)
```

As you can see, this shows all already defined functions of this name in Julia.

👉 *So how does Julia know which one we are calling for?*

🔗 As you can guess based on the arguments, right? This concept in Julia is known as **multiple dispatch**. This is more like a key-feature in Julia and we will come back to this again and again over the course. But for now we have to defer the details of this concept, roughly what Julia does is, it looks at **all arguments** and it looks at the **types of all arguments**, and then call the appropriate function that matches with the types. If you have the question now, *what do we mean by type of the argument?* This is a million dollar question, and we will see now a bit of *types* in Julia.

2. About Types

If you are coming from C, C++ background, then you surely know what do we mean by type. However, if you do not know anything about types at all, and hearing this word for the first time, then you can think about it is like a characteristic of any data that we give the computer to process. For example, when we see

- 2
- 2.0
- "2"

To us humans intuitively it probably means the same thing, however if we want the program to execute systematically, then we should separate these three objects, right? Because computer do not work intuitively, it works with logic and programs it is equipped with. Now recall, computer can only process 0 and 1? So at the end it is going to be a bunch of 0s and 1s. This means if we want to differentiate between 2.0 and 2, then representation in the memory of a computer should be different for 2.0 and 2. I am skipping the details of data representation, but you can look into these beautiful [notes](#) by Professor Chua Hock Chuan at Nanyang Technological University. He gave a very nice and brief summary of data representation of Integers, Floating-point Numbers, and Characters. Particularly you should see the binary representations, that should give you feeling of how can we represent differently. Anyways, back to our topic, so for computers these would be very different objects, and heuristically this is what the idea of type is. When we think about a type in a programming language we think about a group of objects defined by the language to program systematically. Julia has a very rich type-

system, meaning the whole system of types. We will get to the details of types in coming lectures, but for now let's see the types in action in Julia. The single most useful function for knowing type of any object (by literal meaning object not in OOP sense) is **typeof**. So let's use **typeof** and see some types

```
julia> typeof(6)
# Int64

julia> typeof(6.0)
# Float64

julia> typeof("foo baz")
# String

julia> typeof(1:3)
# UnitRange{Int64}

julia> typeof(1:2:10)
# StepRange{Int64,Int64}

julia> typeof('a')
# Char

julia> typeof([1, 2, 3])
# Array{Int64,1}

julia> typeof(rand(2, 3))
# Array{Float64,2}

julia> x = 10 # x is a variable, 10 is the value
# 10

julia> typeof(x) # the type is for the value NOT FOR THE VARIABLE
# Int64
```

There are different types in Julia, as you can see from the outputs. But all these examples are what we call *concrete types*. Concrete types are not the only kinds of types in Julia. There are also something **abstract**, which is defined in the Julia language but you cannot instantiate (meaning you cannot have a value of the type). For concrete types you can think them as taking specific bits in the memory. For example for **Int64** type, it will be an integer to us, however in the memory it will take 64 bits.

```
## CodeBlock 7.
julia> sizeof(3)
# 8
```

Here it shows 8 bytes (1 byte = 8 bits), you can check for others. We can quickly see also some examples of abstract types. To understand abstract types you can think about that there are certain types which we cannot instantiate however they define certain hierarchical orderings and certain relationships between the types in general. Let's see this hierarchical thing in action. Following commands are really useful,

```
## CodeBlock 8.
julia> supertype(Int64) # concrete type
# Signed

julia> supertype(Signed) # abstract supertype
# Integer

julia> supertype(Integer) # abstract supertype
# Real

julia> supertype(Real) # abstract supertype
# Number

julia> supertype(Number) # abstract supertype
# Any

julia> supertype(Any) ## So Any is the father of every one in the hierarchy!
# Any

julia> import Base.show_supertypes

julia> show_supertypes(Int64)
# Int64 <: Signed <: Integer <: Real <: Number <: Any
```

As we can see once we reach to **Any**, then there is no supertype. So **Any** is like the great great-grandfather of the family of types, which is on the top of the type system. And then gradually we have other types. Also note **a <: b** means **a** is a subtype of **b**. We can also see subtypes using following functions,

```
## CodeBlock 9.
julia> subtypes(Integer)
# 3-element Array{Any,1}:
# Bool
# Signed
# Unsigned

julia> subtypes(Real)
# 4-element Array{Any,1}:
# AbstractFloat
# AbstractIrrational
# Integer
# Rational
```

Now the interesting thing, we can see the whole hierarchy of types,

```
## CodeBlock 10.  
# go to package manager and add AbstractTrees  
using AbstractTrees # a package to show tree structures  
AbstractTrees.children(x::Type) = subtypes(x) ## you will understand this later  
  
print_tree(Real)
```

this is what we get

```
julia> print_tree(Real)  
# Real  
# └─ AbstractFloat  
#   └─ BigFloat  
#   └─ Float16  
#   └─ Float32  
#   └─ Float64  
# └─ AbstractIrrational  
#   └─ Irrational  
# └─ Integer  
#   └─ Bool  
#   └─ Signed  
#     └─ BigInt  
#     └─ Int128  
#     └─ Int16  
#     └─ Int32  
#     └─ Int64  
#     └─ Int8  
#   └─ Unsigned  
#     └─ UInt128  
#     └─ UInt16  
#     └─ UInt32  
#     └─ UInt64  
#     └─ UInt8  
# └─ Rational
```

How cool is that? So it shows the entire type hierarchy for reals. Anyways you get some idea of the type system in Julia. Some very last remarks,

- The concrete types are always in the leaf nodes of the tree, and there is no subtype of concrete types
- Everything else that is on the top of the concrete types are abstract types
- We can only instantiate (attach value) for concrete types!

There are also kinds of types called **composite types** and **parametric types**. In fact the examples of the concrete types that we have seen are called **primitive types**. But

let's look at the details regarding these issues in coming sections. For now this is good enough!

3. Again Functions (but user defined)

Suppose there is a **vaccine for Covid-19** and a person just received the vaccination, but we have doubts whether vaccination was any good for us. So maybe we could ask, whether the person again get infected or not. If we know the probability of getting infected again is p , then the random variable which returns *whether the person get infected or not* is what we call a **Bernoulli random** variable, or a random variable which is distributed as **Bernoulli distribution**. Now suppose we know p , and we would like to generate some data which follows this distribution. How can we do this?

In **R** for example, there is a function called **rbinom** right? But that is already defined for us. Similarly here for example we already used a functions **rand** and **randn** which are also predefined and generated some data according to Uniform and Normal distributions. But let's define our own function which will generate some data points from a Bernoulli distribution (or we can also say Bernoulli trials)

Here is one possible way to do this! We write following *function*,

```
## CodeBlock 11.  
function infect_again(p)  
    r = rand()  
    return r < p  
end
```

Let's run the definition and then call the function

```
## CodeBlock 12.  
julia> infect_again(0.3)  
# true  
  
julia> x = 0.2  
# 0.2  
  
julia> infect_again(x)  
# false
```

Note the function returns true or false, which is actually another type in Julia, which is called **Boolean** type,

```
## CodeBlock 13.
julia> typeof(infect_again(0.3))
# Bool
```

If you want this to be an integer type then we can call the function which is the same as the name of the type, so for example we can call **Int64()** or **Int32()**

```
## CodeBlock 14.
julia> x = 0.4
# 0.4

julia> data = infect_again(x)
# false

julia> data_integer = Int64(data)
# 0

julia> typeof(data_integer)
# Int64
```

If you have noticed, we did a very interesting thing, that is we used the name of the types (in this case Int64) as a function! and that **converts** the value to an Integer type. We will come back to this again, when we will talk about **constructors** of types, that is we can use the name of the types to instantiate (i.e., create) an object of a particular type.

4. Good Documentation

The manual has a whole chapter about [documentation](#). So you should read this to write codes in a better way. Often we ignore this issue, but this is really an important part, if in the future you would like to share the code and develop a package in Julia. You do not want that someone will look into your codes and say *what the heck this guy did?*. In general, we should help others to understand our code, and in the Julia community this is an overall theme. Many packages and functions are used by other packages. Here is a simple documentation for functions, the strings on the top is what we call **docstring**.

```
## CodeBlock 15.

"""
    infect_again(p)

    Return boolean type values for a given probability  $0 \leq p \leq 1$ 

# Examples

julia> infect_again(0.2)
false
```

```
"""  
function infect_again(p)  
  r = rand()  
  return r < p  
end
```

There are better ways to do this in the documentation, but this is fine for now. Also how to see the help just type **?** and then type **infect_again** you will see the details of the function that we just defined.

5. Back to functions again!

👉 Some related questions?

1. Okay maybe not related to functions, but since we used strings, let's ask how to do operations with string? (concatenation and so on)
2. Can you set the **p** first and then call it?
3. Can you see the details of the function that you wrote?
4. If you close this session and open a new REPL, then does **infect_again** remain? does **rand** remain?
5. What is this **return** word doing here? (keyword!)
6. Can we write this function without **return**, if yes then what is the point of **return**?
7. Is there a simpler way to define a function, maybe in one line?
8. Do you have the value of **r** outside the function (i.e., global scope)

Let us think about these questions and answer them gradually!

```
## CodeBlock 16.  
  
"a"*"a" # will concatenate  
"a"+"a" # will throw an error.  
  
x = "abncjakkkayl"  
  
x[1] # it will return a character  
x[begin]  
x[end]  
  
x[end-1]  
  
x[begin:2:end]  
x[end-1:end]
```

Read the [Strings](#) in the documentation to know more about strings. For now, we will skip this and move on.

The second question is obvious, yes we can. For the third we already saw the details. Let's check 4 by ourselves. **return** is a keyword that is specifically used if you want your function to return something. However, Julia by default returns the last line of the function. So if your return is already the last line, it is not necessary to use this **keyword**. But if you do some operations and in your function there is a line which you want to return and that is not the last line then you can use return. There is definitely a simpler way to write the above function,

```
## CodeBlock 17.  
infect_again(p) = rand() < p # same function, does exactly same thing, but more compact way
```

Note, we defined the same function twice. Did we replace the **old** function? YES indeed, later we will see the concept known as **multiple dispatch**, which has the idea of defining the same function for different type of objects (recall we already saw an example **rand**). But here we did not do multiple dispatch, rather we defined the same object again, so Julia automatically replaced the old function.

Cool thing! we can pretend to be mathematicians also in Julia, and write super fancy looking functions 😊

```
## CodeBlock 18.  
ϕ(x) = e^(-x^2)/2 # type \euler + <TAB>
```

Can you interpret the function we defined? So not with programming language, but intuitively what is going on?

4. Loops

Here is a simple for loop,

```
# poor style  
n = 100  
ϵ = zeros(n)  
for i in 1:n  
    ϵ[i] = randn()  
end
```

Here we first declared ϵ to be a vector of n numbers, initialized by the floating point 0.0 . The **for** loop then populates this array by successive calls to `randn()`. Like all code blocks in Julia, the end of the **for** loop code block (which is just one line here) is indicated by the keyword **end**. The word **in** from the **for** loop can be replaced by either \in or `=`. The index variable is looped over for all integers from $1:n$ – but **this does not actually create a vector of those indices**. Instead, it creates an **iterator** that is looped over in this case the **range** of integers from 1 to n . While this example successfully fills in ϵ with the correct values, it is very indirect as the connection between the index i and the ϵ vector is unclear. To fix this, we can use **eachindex**

```
# better style
n = 100
ϵ = zeros(n) # pre-allocation
for i in eachindex(ϵ)
    ϵ[i] = randn()
end
```

Here, **eachindex(ϵ)** returns an iterator of indices which can be used to access ϵ . While iterators are **memory efficient** because the elements are generated on the fly rather than stored in memory, the main benefit is

- (1) it can lead to code which is clearer and less prone to typos; and
- (2) it allows the compiler flexibility to creatively generate fast code.

In Julia you can also loop directly over arrays themselves, like so

```
## CodeBlock 22.
ϵ
ϵ_sum = 0.0 # careful to use 0.0 here, instead of 0
m = 5
for ϵ_val in ϵ[1:m]
    ϵ_sum = ϵ_sum + ϵ_val # we can also use ϵ_sum += ϵ_val
end
ϵ_mean = ϵ_sum / m
```

where **$\epsilon[1:m]$** returns the elements of the vector at indices 1 to m . Of course, in Julia there are built in functions to perform this calculation which we

```
## CodeBlock 23.
ϵ_mean ≈ sum(ϵ[1:m]) / m # type \approx and hit <tab>
```

In these examples, note the use of \approx to test equality, rather than `==`, which is appropriate for integers and other types. Approximately equal, is the appropriate way to compare any floating point numbers due to the standard issues of [floating point math](#).

```
# check this out!  
julia> 0.1 + 0.2  
# 0.30000000000000004
```

Couple of remarks about loops before we go to the next section

1. Every command in Julia returns something, however for loop returns **nothing**, **nothing** is a value in Julia, which has the type **Nothing**. Let's take the last command

```
## CodeBlock 24.  
julia> x = for i in eachindex(ϵ)  
           ϵ[i] = randn()  
         end  
  
julia> x  
  
julia> x  
  
julia> typeof(x)  
# Nothing
```

If you now type **x**, it will print absolutely nothing.

2. You can check the syntax for while loop in the manual. Can you think about an example where we absolutely need while loop? If you can loop using for loop you can definitely use while loop. However not the other way. There are cases where you do not know beforehand how long your loop might run, so you need while loop then.

Type issues

You should ask why did we set `ϵ_sum = 0.0`? But before this let us ask if we try to add apples with oranges, e.g., two apples + three oranges, what would be the result? Well! Mathematically this might be fine, if we define x apples + y oranges = combined juice, but the idea is we need to at least think about this before doing these kinds of operations. Maybe the example is not a good one. But I think this will get the ideas across what I want to say, now if you ask computer to perform `1+2.0` it will do this for you and return `3.0` however this is not same as asking to do `1 + 2`. The latter is much easy operation because both types are same, for the former it first need to convert it and then do the operation, which is definitely more costly than straightforward addition of same types. Let's understand this a bit more. But first

let me point this out for you **operations are also function calls in Julia**. Meaning when you are asking Julia to perform **3 + 3** under the hood it is calling a function like **+(3, 3)**

```
## CodeBlock 25.  
3 + 3  
+(3, 3) ## operators are functions  
# Also try ? +
```

Now we will use a very fancy but very useful thing in Julia called **macro**. Think about macro as a super function, which takes some codes and returns some other codes. This is a really cool thing in Julia. However we need to use it with caution. Hopefully we will see this as the last topic.

```
## CodeBlock 26.  
  
@which +(3, 3)  
@which +(3, 3.0)  
@which +(3.0, 3.0)  
@which +(Int64(3), Int64(3))  
methods(+)
```

The case for **+(3, 3.0)** hints to something called **promotion.jl**. So let's see what that is, Ah ha! this calls a functions **promote**. So what is this promote? So in Julia you have these two operations, **conversion** and **promotion**. Conversion literally converts from one type to the other, however not all case we can do this!

```
## CodeBlock 27.  
julia> convert(Int64, 2.0)  
# 2  
  
julia> convert(Int64, 2.2)  
# ERROR: InexactError: Int64(2.2)  
# Stacktrace:  
# [1] Int64 at ./float.jl:710 [inlined]  
# [2] convert(::Type{Int64}, ::Float64) at ./number.jl:7  
# [3] top-level scope at REPL[33]:1  
  
julia> convert(AbstractFloat, "foo")  
# ERROR: MethodError: Cannot `convert` an object of type String to an object of type AbstractFloat  
# Closest candidates are:  
#  convert(::Type{T}, ::T) where T<:Number at number.jl:6  
#  convert(::Type{T}, ::Number) where T<:Number at number.jl:7  
#  convert(::Type{T}, ::Base.TwicePrecision) where T<:Number at twiceprecision.jl:250  
# ...  
# Stacktrace:  
# [1] top-level scope at REPL[34]:1
```

However **promotion** works differently than conversion. Promotion means we go to the common type of all arguments. We can also check the types after the promotion takes place

```
## CodeBlock 28.
promote(1, 3.0)
promote_type(Int64, Float64)
promote_type(BigInt, Float64)
```

So let's have a small glimpse of what happens under the hood, the macro we will use is called **@code_llvm**, loosely this prints what the LLVM compiler could compile

```
julia> @code_llvm 3 + 1

# ; @ int.jl:86 within `+'
# define i64 @"julia+_1118"(i64, i64) {
# top:
#   %2 = add i64 %1, %0
#   ret i64 %2
# }

julia> @code_llvm 3.0 + 1

# ; @ promotion.jl:311 within `+'
# define double @"julia+_1120"(double, i64) {
# top:
# ;  ⌈ @ promotion.jl:282 within `promote'
# ;  |⌈ @ promotion.jl:259 within `_promote'
# ;  ||⌈ @ number.jl:7 within `convert'
# ;  |||⌈ @ float.jl:60 within `Float64'
#       %2 = sitofp i64 %1 to double
# ; LLLL
# ; @ promotion.jl:311 within `+' @ float.jl:401
#   %3 = fadd double %2, %0
# ; @ promotion.jl:311 within `+'
#   ret double %3
# }

julia> @code_llvm 3.0 + 1.0

# ; @ float.jl:401 within `+'
# define double @"julia+_1121"(double, double) {
# top:
#   %2 = fadd double %0, %1
#   ret double %2
# }
```

Arrays and Tuples (brief overview)

here is a simple example of a random array

```
a = rand(4) ## where a is a variable, look at the type
a = rand(1, 2)
a = rand(3, 4)
a = rand(3, 4, 3) # multi dimensional
a = rand(3, 4, 3, 2)
typeof(a)
```

In general we can also construct an array in different ways, and Julia gives us some more than one options

```
b = [1, 2, 3]
b = [1 2 3]
b = [1; 2; 3] # same as a the first one
b = [2 3; 4 5]
typeof(b)
```

```
c = rand(4, 5)
c[:, 3] ## this is already indexing the array
c[3, :] ## again indexing
c[1, 1]
c[1:2:end, :] # every second row, all columns
c[end-1:end, :] ## last two rows
```

Checking the size

```
size(c)
size(c, 1)
size(c, 2)
```

Arrays are mutable

```
c
c[1, 2] = 100
c
```

Copying an array

```
a = c
a[1] = 99
```

```
c ## Note C has changed, so it is not really copying everything again,
```

also we could check whether a also changes

```
c[2, 3] = 109  
c  
a
```

However if we do want to have a different copy

```
z = deepcopy(c)  
c[1, 4] = 10000  
c  
z ## z has not changed
```

In general when we can change an object, we call this behavior **mutable**. So array is a mutable object, however for tuple it is not the case. A tuple is a fixed-length container that can hold any values, but cannot be modified (it is immutable). Tuples are constructed with commas and parentheses, and can be accessed via indexing. The components of tuples can optionally be named, in which case a named tuple is constructed.

```
(1, 1+1)  
(1, )  
x = (0.0, "hello", 6*7)  
x[2] ## indexing works similar to array
```

Tuple can be very powerful, and using this we can improve the performance a lot, again this is for later!

map and . (aka broadcasting)