juliastart

June 25, 2019

1 Core Julia

This chapter consists of six sections:

1. Variable names;
2. Operators;
3. Types;
4. Data Structure;
5. Control flow;
6. Functions.

1.1 Variable names

Variable names are case sensitive and Unicode names (in UTF-8 encoding) may be used. And names must begin with a letter, no matter lowercase or uppercase, an underscore or a Unicode code point larger than 00A0, and other Unicode points, even a latex symbol. We can also redefine the constants here, such as π. But bulid-in statements are not allowed in this language. The examples are below.

```
In [1]: ## right variable names
        z = 100
        y = 1.0
        s = "my_variable"
        data_science = "true"
        datascience = true

        ## wrong variable names are here if run them, return an error
        ## if = 1.2
        ## else = true
        ##
```

```
Out[1]: true
```

1.2 Operators

The operators are similar to *R* and *Python*. Four main categories of operators in Julia:

1. Arithmetic;

2. Updating;

3. Numeric comparison;

4. Bitwise.

```
In [ ]: x = 2
        y = 3
        z = 4

        x + y
        x^y
        x += 2
        x
        y
```

When constructing expressing with multiple operators, the order in which these operators are applied to the expression is known as operator precedence.

With the parentheses () inclueded in the expression, we can control the order by ourselves like the following code

```
In [ ]: x*y+z^2
        x*(y+z^2)
        (x*y)+(z^2)
```

1.3 Types

1.3.1 Numeric

Julia offers full support for real and complex numbers. The internal variable `Sys.WORD_SIZE` displays the architecture type of the computer. Minimum and Maximum can be showed by `typemin()` and `typemax()`.

```
In [2]: Sys.WORD_SIZE
```

```
Out[2]: 64
```

```
In [3]: typemax(Int)
```

```
Out[3]: 9223372036854775807
```

```
In [4]: typemin(Int)
```

```
Out[4]: -9223372036854775808
```

```
In [5]: typemax(Int64)
```

```
Out[5]: 9223372036854775807
```

```
In [6]: typemax(Float32)
```

```
Out[6]: Inf32
```

2

Some types use leftmost bit to control the sign, such as `Int64`, but others use that bit as value and without sign like `UInt128`.

Boolean values are 8-bit integers, with false being 0 and true being 1.Overflow errors will happen if the result is larger or smaller than its allowable size.

```
In [7]: literal_int = 1
        println("typeof(literal_int):",typeof(literal_int))

typeof(literal_int):Int64
```

```
In [8]: x = typemax(Int64)

Out[8]: 9223372036854775807

In [9]: x += 1

Out[9]: -9223372036854775808
```

1.3.2 Floats

Floats are similar to scientific notation. They are made up of three components: asigned interger whose length determines the precision, the base used to represent te number and a signed integer htat changes the magnitude of floating point number (the exponent). `Float64` literals are distinguished by having an `e` before hte power, and can be defined in hexadecimal. `Float32` literals are distinguished by having an `f` in place of the `e`. THere are three `Float64` values that do not occur on the real line:

1. `Inf`, positive indinity: a value larger than all finite floating point numbers, equal to itself, and greater than every other floating point value but `NaN`;
2. `-Inf`, negative infinity: a value less than all finite floating point numbers, equal to itself, and less than every other floating point value but `NaN`;
3. `NaN`, not a number: a value not equal to any floating point value, and not `==`,`<` or `>` than any floating point value, including itself.

Some tips:

1. digit separation using an `_`;

```
In [10]: x1 = 1.0
         x64 = 15e-15
         x32 = 2.5f-4
         println("x1 is ",typeof(x1))
         println("\nx64 is ", typeof(x64))
         println("\nx32 is ", typeof(x32))

x1 is Float64

x64 is Float64

x32 is Float32
```

```
In [11]: 9.2_4==9.24
         ## digit separation using an _
```

```
Out[11]: true
```

In machine, it is defined that the smallest value is $1 + z \neq 1$. In Julia, the value of epsilon for a particular machine can be found via the `eps()` function.

The spacing between floating point numbers and the value of machine epsilon is important to understand because it can help avoid certain types of errors.

There are also float underflow errors, which occur when the result of a calculation is smaller than machine epsilon or when numbers of similar precision are subtracted.

```
In [12]: eps()
```

```
Out[12]: 2.220446049250313e-16
```

```
In [13]: n1 = [1e-25,1e-5,1.,1e5,1e25]
         for i in n1
             println(*(i,eps()))
         end
```

```
2.2204460492503132e-41
2.2204460492503133e-21
2.220446049250313e-16
2.220446049250313e-11
2.2204460492503133e9
```

1.3.3 Strings

In Julia, a string is a sequence of Unicode code points, using UTF-8 encoding. Characters in strings have an index value within the string. It is worth noting that Julia indices start at **position 1**, similar to **R** but different to **Python**. The key word `end` can be used to represent the last index. Herein, we will deal with ASCII characters only. Note that `String` is the built-in type for string and string literals,and `Char` is the built-in type used to represent single chararcters. In fact, `Char` is a numeric value representing a Unicode code point. The value of a string cannot be changed, i.e., strings are immutable, and a new string must be built from another string. Strings are defined by double or triple quotes.

```
In [1]: s1 = "Hi"
```

```
Out[1]: "Hi"
```

```
In [2]: s2 = """I have a "quote" character"""
```

```
Out[2]: "I have a \"quote\" character"
```

String can be sliced using range indexing, e.g., `my_string[4:6]` would return a substring of `my_string` containing the 4th, 5th and 6th charaters of `my_string`. Concatenation can be done

in two ways: using the `string()` function or with the `*` operator. Note this is a somewhat unusual feature of Julia. many other language use + to perform concatenation. String interpolation takes place when a string literal is defined with a variable inside its instantiation. The cariable is prepended with $. By using variables insider the string's definition, complex strings can be built in a readbale form, without multiple string multiplications.

```
In [1]: ## some examples of strings
        str = "Data science is fun!"
        str[1]

Out[1]: 'D': ASCII/Unicode U+0044 (category Lu: Letter, uppercase)

In [2]: str[end]

Out[2]: '!': ASCII/Unicode U+0021 (category Po: Punctuation, other)

In [3]: str[4:7]

Out[3]: "a sc"

In [5]: str[end-3:end]

Out[5]: "fun!"

In [6]: string(str," Sure is :)")

Out[6]: "Data science is fun! Sure is :)"

In [7]: str * "Sure is :)"

Out[7]: "Data science is fun!Sure is :)"

In [8]: # Interpolation
        "1+2=$(1+2)"

Out[8]: "1+2=3"

In [9]: word1="Julia"
        word2="data"
        word3="science"
        "$word1 is great for $word2 $word3"

Out[9]: "Julia is great for data science"
```

Strings can be comared lexicographically using comarsion operators, e.g. ==, >, etc. Lexicograohical comparison involves sequentially comparing string elements with the same position, until one pair of elements falsiies the comparison, or the end of the string is reached. Some useful string functions are:

- `findfirst(par,str)` returns the indices of the characters in the string `str` matching the pattern pat.

- `occursin(substr, str)` returns `true`/`false` depending on the presence/absence of `substr` in `str`.
- 'repeat(str,n)' generates a new string that is the original string `str` repeated n times.
- 'length(str)' returns the number of charaters in the string `str`.
- 'replace(str,ptn => rep)' searches string `str` for the pattern `ptn` and, if it is present, replaces it with `rep`.

```
In [1]: # Lexicographical comparison
        s1 = "abcd"
        s2 = "abde"

        s1 == s2

Out[1]: false

In [2]: s1 < s2

Out[2]: true

In [3]: s1 > s2

Out[3]: false

In [6]: # String functions
        str = "Data science is fun!"

        findfirst("Data",str)

Out[6]: 1:4

In [7]: occursin("ata",str)

Out[7]: true

In [9]: replace(str,"fun"=>"great")

Out[9]: "Data science is great!"
```

Julia fully supports regular expressions (regexes). Regexes in Julia are fully Perl compatible and fully compatible and are used to hunt for patterns in string data. They are defined as string with a leading r outside the quotes. Regular expressions are commonly used with the following functions:

- `occursin(regex,str)` returns `true`/`false` if the regex has a match inthe string `str`.
- `mathc(regex,str)` returns the first match of `regex` in the string. If there is no match, it returns the special Julia value `nothing`.
- `eachmatch(regex, str)` returns all the matches of `regex` inthe string `str` as an array.

Regexes are a very powerful programming tool for working with text data. However, an in-depth discussion of them is beyond the scope of this book, and intersted readers are encouraged to consult Friedl(2006) for further details.

```
In [10]: # Regular expresions

         # match alpha-numeric charaters at the start of the str
         occursin(r"[a-zA-z0-9]",str)

Out[10]: true

In [11]: occursin(r"^[a-zA-z0-9]",str)

Out[11]: true

In [12]: occursin(r"[a-zA-z0-9]$",str)

Out[12]: false

In [13]: occursin(r"[^a-zA-Z0-9]",str)

Out[13]: true

In [14]: ## matches the first non-alpha-numeric character in the string
         match(r"[^a-zA-Z0-9]",str)

Out[14]: RegexMatch(" ")

In [15]: ## matches all the non-alpha-numeric characters in the string
         collect(eachmatch(r"[^a-zA-Z0-9]",str))

Out[15]: 4-element Array{RegexMatch,1}:
          RegexMatch(" ")
          RegexMatch(" ")
          RegexMatch(" ")
          RegexMatch("!")
```

1.3.4 Tuples

Tuples are a Julia type. They are an abstraction of function arguments without the function. The arguments are defined in a specific order and have well-defined types. Tuples can have any number of parameters, and they do not have field names. Fields are accessed by their index, and turples are defined using brackets () and commas. A very usefual feature of tuples in Julia is that each element of a tuples in Julia is that each element of a tuple can have its own type. Variable values can be assigned directly from atuple where the value of each variable corresponds to a value in the tuple.

```
In [16]: # A tuple comprising only floats
         tup1 = (3.0,0.1,0.8,1.9)
         typeof(tup1)

Out[16]: NTuple{4,Float64}

In [17]: ## A tuple comprising strings and floats
         tup2 = ("Data",2.5,"Science", 8.8)
         typeof(tup2)
```

```
Out[17]: Tuple{String,Float64,String,Float64}

In [18]: ## variable assignment
         a,b,c = ("Fast", 1, 5.2)
         a

Out[18]: "Fast"

In [19]: b

Out[19]: 1

In [20]: c

Out[20]: 5.2

In [22]: (a,b,c)=("safa",2341,123.412)
         typeof(a)

Out[22]: String
```

1.4 DATA STRUCTURES

1.4.1 Arrays

An array is a multidimensional grid that stores objects of any type. To improve preformance, arrays should contain only one specific type, e.g., `Int`. Arrays do not require vectroizing for fast array computations. The array implementation used by Julia is written in Julia and relies on the comiler for performace. The compiler uses type inference to make optimized code for array indexing, which makes programs more readable and easier to maintain. Arrays are a subtype of the `AbstractArray` type. As such, they are laid out as a contiguous block of memory. this is not true of other members of the `AbstractArray` type, such as `SparseMatrixCSC` and `SubArray`.

The type and dimensions of an array can be specified using `Array{T}(D)`, where `T` is any valid Julia type and `D` is the dimension of the array. The first entry in the tuple `D` is a singleton that specifies how the array values are initialiezed. Users can specify `undef` to create an uninitialized array, `nothing` to create arrays with no value, or `missing` to create arrays of missing values. Arrays with different types can be created with type `Any`.

```
In [23]: # A vector of length 5 containing integers
         a1 = Array{Int64}(undef,5)
         typeof(a1)

Out[23]: Array{Int64,1}

In [25]: # A 2\times2 matrix containting integers
         a2 = Array{Int64}(undef,(2,2))

Out[25]: 2Œ2 Array{Int64,2}:
           7926335345396809728  872415232
           3891110079994265600  295721712
```

8

```
In [26]: # A 2\times 2 matrix containing Any type
         a2 = Array{Any}(undef,(2,2))
```

```
Out[26]: 2×2 Array{Any,2}:
          #undef  #undef
          #undef  #undef
```

In Julia, [] can also be used to generate arrays. In fact, the `Vector()`, `Matrix()` and `collect()` functions can also be used

```
In [27]: ## A three-element row "Vector"
         a4 = [1,2,3]
         typeof(a4)
```

```
Out[27]: Array{Int64,1}
```

The array a4 above does not have a second dimension, i.e., it is neither a 1×3 vector nor a 3×1 vector. In other words, Julia makes a distinction between `Array{T,1}` and `Array{T,2}`.

```
In [28]: ## A 1\times 3 colum vector  -- a two-dimensional array
         a5 = [1 2 3]
         typeof(a5)
```

```
Out[28]: Array{Int64,2}
```

```
In [33]: ## A 2\times 3 matrix, where ; is used to separate rows
         a6 = [80 81 82 ; 90 91 92]
         typeof(a6)
```

```
Out[33]: Array{Int64,2}
```

```
In [34]: ## Arrays containing elements of a specific type can be constructed like:
         a7 = Float64[3.0 5.0 ; 1.1 3.5]
```

```
Out[34]: 2×2 Array{Float64,2}:
          3.0  5.0
          1.1  3.5
```

```
In [35]: ## Arrays can be explicitly created like this:
         Vector(undef,3)
```

```
Out[35]: 3-element Array{Any,1}:
          #undef
          #undef
          #undef
```

```
In [36]: Matrix(undef, 2,2)
```

```
Out[36]: 2×2 Array{Any,2}:
          #undef  #undef
          #undef  #undef
```

```
In [37]: # A 3-element Float array
         a3 = collect(Float64, 3:-1:1)
```

```
Out[37]: 3-element Array{Float64,1}:
          3.0
          2.0
          1.0
```

Julia has many bulit-in functions that generate specific kinds of arrays. Here are some useful ones:

- `zeros(T, d1, ..)` is a d1-dimensional array of all zeros.
- `ones(T, d1, ..)` is a d1-dimensional array of all ones.
- `rand(T, d1, ..)`: if T is Float a d1-dimensional array of random numbers between 0 and 1 is returned; if an array is specified as the first argument, d1 random elements from the array are returned.
- `randn(T, d1, ..)` is a d1-dimensional array of random numbers from the standard normal distribution with mean zero and standard deviation 1.
- `MatrixT(I, (n,n))` is the n×n identity matrix. The identity operator `I` is available in the 'LinearAlgebra.jl' package.
- `fill!(A, x)` is the array A filled with value x.

Note that, in the above, `d1` can be a tuple specifying multiple dimensions.

Arrays can easily be concatenated in Julia. There are two functions commonly used to concatenate arrays:

- `vcat(A1, A2, ..)` concatenates arrays vertically, i.e., stacks `A1` on top of `A2`.
- `hcat(A1, A2, ..)` concatenates arrays horizontally, i.e., adds `A2` to the right of `A1`.

Of course, concatenations requires that the relevant dimensions match.

The following code blocks illustrates some useful array functions as we as slicing. Slicing for arrays works similarly to slicing for strings.

```
In [1]: ## Create a 2x2 identity matrix
        using LinearAlgebra
        imat = Matrix{Int8}(I, (2,2))
```

```
Out[1]: 2×2 Array{Int8,2}:
         1  0
         0  1
```

```
In [2]: ## return rando numbers between 0 and 1
        rand(2)
```

```
Out[2]: 2-element Array{Float64,1}:
         0.8953492020837617
         0.8218388470887807
```

```
In [3]: B = [80 81 82 ; 90 91 92]
```

```
Out[3]: 2⊖3 Array{Int64,2}:
         80  81  82
         90  91  92

In [4]: rand(B,2)

Out[4]: 2-element Array{Int64,1}:
          90
          81

In [5]: ## The number of elements in B
        length(B)

Out[5]: 6

In [6]: ## The dimensions of B
        size(B)

Out[6]: (2, 3)

In [7]: ## the number of dimensions of B
        ndims(B)

Out[7]: 2

In [8]: ## A new array with the same elements (data) as B but different dimensions
        reshape(B, (3,2))

Out[8]: 3⊖2 Array{Int64,2}:
         80  91
         90  82
         81  92

In [9]: ndims(B)

Out[9]: 2

In [10]: ## A copy of B, where elements are recursively copied
         B2 = deepcopy(B)

Out[10]: 2⊖3 Array{Int64,2}:
          80  81  82
          90  91  92

In [11]: ## When slicing, a slice is specified for each dimension
         ## The first two rows of the first column done two ways
         B[1:2, ]

Out[11]: 2-element Array{Int64,1}:
           80
           90
```

```
In [12]: B[1:2,1]

Out[12]: 2-element Array{Int64,1}:
          80
          90

In [13]: ## The first two rows of the second column
         B[1:2,2]

Out[13]: 2-element Array{Int64,1}:
          81
          91

In [15]: # The first row
         B[1,:]

Out[15]: 3-element Array{Int64,1}:
          80
          81
          82

In [16]: ## The third element
         B[3]

Out[16]: 81

In [17]: # Another way to build an array is using comprehensions
         A1 = [sqrt(i) for i in [16,25,64]]

Out[17]: 3-element Array{Float64,1}:
          4.0
          5.0
          8.0

In [18]: A2 = [i^2 for i in [1,2,3]]

Out[18]: 3-element Array{Int64,1}:
          1
          4
          9
```

From a couple of examples in the above code block, we can see that Julia counts array elements by column, i.e., the kth element of the $n \times m$ matrix X is the kth element of the nm-vector vec(X). Array comprehensions, illistrated above, are another more sophisticated way of building arrays. The generate the items in the array with a function and a loop. These items are then collected into an array by the brackets [] that surround the loop and function.

1.4.2 Dictionaries

In Julia, dictionaries are defined as associative collections consisting of a key value pair, i.e., the key is associated with a specific value. These key-value pairs have their own type in Julia, `Pairtypeof(Key), typeof(value)` which creates a `Pair` object. Alternatively, the `=>` symbol can be used to separate the key and value to create the same `Pair` object. One use of `Pair` objects is in the instantiation of dictionaries. Dictionaries in Julia can be used analogously to lists in *R*. Dictionaries are created using the keyword `Dict` and types can be specified for both the key and the value. The keys are hashed and are always unique.

```
In [21]: ## Three dictionaries, D0 is empty, D1 and D1 are the same
         D0 = Dict()
         D1 = Dict(1 => "red", 2 => "white")
         D2 = Dict{Integer, String}(1 => "red", 2 => "white")
         typeof(D2)

Out[21]: Dict{Integer,String}

In [23]: ## Dictionaries can be created using a loop
         food = ["salmon", "maple syrup", "touriere"]

         food_dict = Dict{Int, String}()

         ## keys are the foods index in the array
         for (n, fd) in enumerate(food)
             food_dict[n] = fd
         end
         food_dict

Out[23]: Dict{Int64,String} with 3 entries:
           2 => "maple syrup"
           3 => "touriere"
           1 => "salmon"

In [26]: ## Dictionaries can also be created using the generator syntax
         wine = ["red", "white", "rose"]
         wine_dict = Dict{Int, String}(i => wine[i] for i in 1:length(wine))

Out[26]: Dict{Int64,String} with 3 entries:
           2 => "white"
           3 => "rose"
           1 => "red"
```

Values can be accessed using `[]` with a value of dictionary key inserted between them or `get()`. The presence of a key can be checked using `haskey()` and a particular key can be accessed using `getkey()`. Keys can also be modified, as illustrated in the below code block. Here, we also demonstrate adding and deleting entries from a dictionary as well as various ways of manipulating keys and values. Note that the following code block builds on the previous.

```
In [27]: ## Values can be accessed similarly to an array, but by key:
         food_dict[1]
```

```
Out[27]: "salmon"

In [28]: ## The get() function can also be used;
         ## note that "unknown" is the value returned here if the key is not in the dictionary
         get(food_dict,1,"unknown")

Out[28]: "salmon"

In [29]: get(food_dict,7,"unknown")

Out[29]: "unknown"

In [30]: ## we can also check directly for the presence of a particular key
         haskey(food_dict,2)

Out[30]: true

In [31]: haskey(food_dict,9)

Out[31]: false

In [33]: ## The getkey() function can also be used; note that 999 is the
         ## value returned here if the key is not in the dictionary
         getkey(food_dict,1,999)

Out[33]: 1

In [36]: ## A new value can be associated with an existing key
         food_dict[1]
         food_dict[1] = "lobster"

Out[36]: Dict{Int64,String} with 3 entries:
            2 => "maple syrup"
            3 => "touriere"
            1 => "lobster"

In [40]: ## Two common ways to add new entries:
         food_dict[4] = "bannock"
         get!(food_dict,4,"miss")
         food_dict[5]

Out[40]: "miss"

In [41]: ## The advantage of get!() is that will not add the new entry if
         ## a value is already associated with the key
         get!(food_dict,4,"fake")

Out[41]: "bannock"

In [42]: ## Just deleting entries by key is straightforward
         delete!(food_dict,4)
```

```
Out[42]: Dict{Int64,String} with 4 entries:
           2 => "maple syrup"
           3 => "touriere"
           5 => "miss"
           1 => "lobster"
```

In [44]: *## But we can also delete by key and return the value assocated with the key;*
         ## note that 999 is reurned here if the key is not present
         deleted_fd_value = pop!(food_dict,3,999)
         food_dict

```
Out[44]: Dict{Int64,String} with 3 entries:
           2 => "maple syrup"
           5 => "miss"
           1 => "lobster"
```

In [45]: *## Keys can be coerced into arrays*
         collect(keys(food_dict))

```
Out[45]: 3-element Array{Int64,1}:
          2
          5
          1
```

In [46]: collect(values(food_dict))

```
Out[46]: 3-element Array{String,1}:
           "maple syrup"
           "miss"
           "lobster"
```

In [47]: *## We can also iterate over both keys and values*
         for (k, v) **in** food_dict
             println("food_dict: key: ", k, "value: ", v)
         end

```
food_dict: key: 2value: maple syrup
food_dict: key: 5value: miss
food_dict: key: 1value: lobster
```

In [48]: *## We can also just loop over keys*
         for k **in** keys(food_dict)
             println("food_dict: keys: ",k)
         end

```
food_dict: keys: 2
food_dict: keys: 5
food_dict: keys: 1
```

```
In [50]: ## Or could also just loop over values
         for v in values(food_dict)
             println("food_dict: value:", v)
         end

food_dict: value:maple syrup
food_dict: value:miss
food_dict: value:lobster
```

1.5 CONTROL FLOW

1.5.1 Compound Expressions

In Julia, a compound expression is one expression that is uesd to sequentially evalute a group of subexpressions. The value of the last subexpression is returned as the value of the expression. THere area two to achieve this: `Begin` blocks and chains.

```
In [51]: ## A begin block
         b1 = begin
             c = 20
             d = 5
             c * d
         end
         println("b1: ", b1)

b1: 100
```

```
In [52]: ## A chain
         b2 = (c= 20 ; d = 5 ; c * d)
         println("b2: ", b2)

b2: 100
```

1.5.2 Conditional Evaluation

Conditional evaluation allows parts of a program to evaluated, or not, based on the value of a Boolean expression, i.e., an expression that produces a true/false value. In Julia, conditional evaluation takes the form of an `if-elseif-else` construct, which is evaluated until the first BOolean expression evaluates to true or the else statement is reached. When is given Boolean expression evaluates to true, the associated block of code executed. No other code blocks or condition expressions within the `if-elseif-else` construct returns the value of the last executed statement. Programmers can use as many `elseif` blocks as they wish, including none, i.e., an `if-else` construct. In Julia, `if`, `elseif` and `else` statements do not require parentheses; in fact, their use is discouraged.

```
In [53]: # An if-else construct
         k = 1
```

```julia
        if k == 0
            "zero"
        else
            "not zero"
        end
```

Out[53]: "not zero"

In [54]: ## An if-elseif-else construct
```julia
        k = 11
        if k % 3 == 0
            0
        elseif k % 3 == 1
            1
        else
            2
        end
```

Out[54]: 2

An alternative approach to conditional evaluation is via shortcircuit evaluation. This construct has the form a ? b : c, where a is a Boolean expression, b is evaluated if a is true, and c is evaluated if a is false. Note that ? : is called the "ternary operator", it assocaites from right to left, and it can be useful for short conditional statements. Ternary operators cna be chained together to accommodate situations analogous to an if-elseif-else construct with one or more ifelse blocks.

In [55]: # A short-circuit evaluation
```julia
        b = 10; c = 20;
        println("SCE: b < c: ", b < c ? "less than" : "not less than")
```

SCE: b < c: less than

In [58]: # A short-circuit evaluation with nesting
```julia
        d = 10; f = 10;
        println("SCE: chained d vs e: ", d < f ? "less than " : d > f ? "greater than" : "equa
```

SCE: chained d vs e: equal

Note that we do not use e in the above example because it is a literal in Julia (the exponential function); while it can be overwritten, it is best practice to avoid doing so.

In [59]: e

UndefVarError: e not defined

```
Stacktrace:

  [1] top-level scope at In[59]:1
```

In [60]: **using** Base.MathConstants
         e

Out[60]: = 2.7182818284590...

1.5.3 Loops

Basics Two looping constructs exist in Julia: `for` loops and `while` loops. Theses loops can iterate over any container, such as a string or an array. The body of loop ends with the `end` keyword. Variables reference inside variables defined outside the body of the lop, pre-append them with the global keyword inside the body of the loop. A `for` loop can operate over a range object representing a sequence of number, e.g., `1:5`, which it uses to get each index to loop through the range of values in the range, assigning each one to an indexing variable. THe indexing variable only exists inside the loop. when looping over a container, `for` loops can access the elements of the container directly using the `in` operator. Rather than using simple nesting, nested `for` loops can be written as a single outer loop with multiple indexing variables forming a Cartesian product, e.g., if there are two indexing variables then, for each value of the first index, each value of the second index is evaluated.

In [ ]: