

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

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

```
Out[2]: 3
```

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 () included in the expression, we can control the order by ourselves like the following code

```
In [3]: x*y+z^2
        x*(y+z^2)
        (x*y)+(z^2)
```

```
Out[3]: 28
```

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 [4]: Sys.WORD_SIZE
```

```
Out[4]: 64
```

```
In [5]: typemax{Int}
```

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

```
In [6]: typemin{Int}
```

```
Out[6]: -9223372036854775808
```

```
In [7]: typemax{Int64}
```

```
Out[7]: 9223372036854775807
```

```
In [8]: typemax(Float32)
```

```
Out[8]: Inf32
```

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

```
typeof(literal_int):Int64
```

```
In [10]: x = typemax(Int64)
```

```
Out[10]: 9223372036854775807
```

```
In [11]: x += 1
```

```
Out[11]: -9223372036854775808
```

1.3.2 Floats

Floats are similar to scientific notation. They are made up of three components: assigned integer whose length determines the precision, the base used to represent the number and a signed integer that changes the magnitude of floating point number (the exponent). Float64 literals are distinguished by having an e before the 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 infinity: 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 [12]: 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 [13]: 9.2_4==9.24
        ## digit separation using an _
```

```
Out[13]: 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 [14]: eps()
```

```
Out[14]: 2.220446049250313e-16
```

```
In [15]: 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 characters. 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 [16]: s1 = "Hi"
```

```
Out[16]: "Hi"
```

```
In [17]: s2 = ""I have a "quote" character""
```

```
Out[17]: "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 characters 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 languages use `+` to perform concatenation. String interpolation takes place when a string literal is defined with a variable inside its instantiation. The variable is prepended with `$`. By using variables inside the string's definition, complex strings can be built in a readable form, without multiple string multiplications.

```
In [18]: ## some examples of strings
        str = "Data science is fun!"
        str[1]
```

```
Out[18]: 'D': ASCII/Unicode U+0044 (category Lu: Letter, uppercase)
```

```
In [19]: str[end]
```

```
Out[19]: '!': ASCII/Unicode U+0021 (category Po: Punctuation, other)
```

```
In [20]: str[4:7]
```

```
Out[20]: "a sc"
```

```
In [21]: str[end-3:end]
```

```
Out[21]: "fun!"
```

```
In [22]: string(str, " Sure is :)"
```

```
Out[22]: "Data science is fun! Sure is :)"
```

```
In [23]: str * "Sure is :)"
```

```
Out[23]: "Data science is fun!Sure is :)"
```

```
In [24]: # Interpolation
        "1+2=$(1+2)"
```

```
Out[24]: "1+2=3"
```

```
In [25]: word1="Julia"
        word2="data"
        word3="science"
        "$word1 is great for $word2 $word3"
```

```
Out[25]: "Julia is great for data science"
```

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

- `findfirst(pat, 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 characters 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 [26]: # Lexicographical comparison
```

```
s1 = "abcd"
s2 = "abde"
```

```
s1 == s2
```

```
Out[26]: false
```

```
In [27]: s1 < s2
```

```
Out[27]: true
```

```
In [28]: s1 > s2
```

```
Out[28]: false
```

```
In [29]: # String functions
```

```
str = "Data science is fun!"
```

```
findfirst("Data",str)
```

```
Out[29]: 1:4
```

```
In [30]: occursin("ata",str)
```

```
Out[30]: true
```

```
In [31]: replace(str,"fun"=>"great")
```

```
Out[31]: "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 in the string `str`.
- `match(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` in the 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 interested readers are encouraged to consult [Friedl\(2006\)](#) for further details.

```
In [32]: # Regular expresions
```

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

```
Out[32]: true
```

```
In [33]: occursin(r"^[a-zA-z0-9]",str)
```

```
Out[33]: true
```

```
In [34]: occursin(r"[a-zA-z0-9]$",str)
```

```
Out[34]: false
```

```
In [35]: occursin(r"^[a-zA-Z0-9]",str)
```

```
Out[35]: true
```

```
In [36]: ## matches the first non-alpha-numeric character in the string
    match(r"^[a-zA-Z0-9]",str)
```

```
Out[36]: RegexMatch(" ")
```

```
In [37]: ## matches all the non-alpha-numeric characters in the string
    collect(eachmatch(r"^[a-zA-Z0-9]",str))
```

```
Out[37]: 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 tuples 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 a tuple where the value of each variable corresponds to a value in the tuple.

```

In [38]: # A tuple comprising only floats
          tup1 = (3.0,0.1,0.8,1.9)
          typeof(tup1)

Out[38]: NTuple{4,Float64}

In [39]: ## A tuple comprising strings and floats
          tup2 = ("Data",2.5,"Science", 8.8)
          typeof(tup2)

Out[39]: Tuple{String,Float64,String,Float64}

In [40]: ## variable assignment
          a,b,c = ("Fast", 1, 5.2)
          a

Out[40]: "Fast"

In [41]: b

Out[41]: 1

In [42]: c

Out[42]: 5.2

In [43]: (a,b,c)=("safa",2341,123.412)
          typeof(a)

Out[43]: String

```

1.4 Data Structures

1.4.1 Arrays

An array is a multidimensional grid that stores objects of any type. To improve performance, arrays should contain only one specific type, e.g., `Int`. Arrays do not require vectorizing for fast array computations. The array implementation used by Julia is written in Julia and relies on the compiler for performance. 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 initialized. 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 [44]: # A vector of length 5 containing integers
          a1 = Array{Int64}(undef,5)
          typeof(a1)

```



```
Out [44]: Array{Int64,1}
```

```
In [45]: # A 2\times 2 matrix containing integers  
a2 = Array{Int64}(undef,(2,2))
```

```
Out [45]: 2E2 Array{Int64,2}:  
 113449264 260573392  
 104915024 253234816
```

```
In [46]: # A 2\times 2 matrix containing Any type  
a2 = Array{Any}(undef,(2,2))
```

```
Out [46]: 2E2 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 [47]: ## A three-element row "Vector"  
a4 = [1,2,3]  
typeof(a4)
```

```
Out [47]: 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 [48]: ## A 1\times 3 column vector -- a two-dimensional array  
a5 = [1 2 3]  
typeof(a5)
```

```
Out [48]: Array{Int64,2}
```

```
In [49]: ## A 2\times 3 matrix, where ; is used to separate rows  
a6 = [80 81 82 ; 90 91 92]  
typeof(a6)
```

```
Out [49]: Array{Int64,2}
```

```
In [50]: ## Arrays containing elements of a specific type can be constructed like:  
a7 = Float64[3.0 5.0 ; 1.1 3.5]
```

```
Out [50]: 2E2 Array{Float64,2}:  
 3.0  5.0  
 1.1  3.5
```

```
In [51]: ## Arrays can be explicitly created like this:  
Vector(undef,3)
```

```

Out[51]: 3-element Array{Any,1}:
          #undef
          #undef
          #undef

In [52]: Matrix(undef, 2,2)

Out[52]: 2×2 Array{Any,2}:
          #undef #undef
          #undef #undef

In [53]: # A 3-element Float array
          a3 = collect(Float64, 3:-1:1)

Out[53]: 3-element Array{Float64,1}:
          3.0
          2.0
          1.0

```

Julia has many built-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 $n \times n$ 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 [54]: ## Create a 2x2 identity matrix
          using LinearAlgebra
          imat = Matrix{Int8}(I, (2,2))

Out[54]: 2×2 Array{Int8,2}:
          1  0
          0  1

```

```

In [55]: ## return random numbers between 0 and 1
         rand(2)

Out[55]: 2-element Array{Float64,1}:
         0.3542359887124156
         0.8424354785965347

In [56]: B = [80 81 82 ; 90 91 92]

Out[56]: 2x3 Array{Int64,2}:
         80  81  82
         90  91  92

In [57]: rand(B,2)

Out[57]: 2-element Array{Int64,1}:
         91
         82

In [58]: ## The number of elements in B
         length(B)

Out[58]: 6

In [59]: ## The dimensions of B
         size(B)

Out[59]: (2, 3)

In [60]: ## the number of dimensions of B
         ndims(B)

Out[60]: 2

In [61]: ## A new array with the same elements (data) as B but different dimensions
         reshape(B, (3,2))

Out[61]: 3x2 Array{Int64,2}:
         80  91
         90  82
         81  92

In [62]: ndims(B)

Out[62]: 2

In [63]: ## A copy of B, where elements are recursively copied
         B2 = deepcopy(B)

Out[63]: 2x3 Array{Int64,2}:
         80  81  82
         90  91  92

```

```

In [64]: ## When slicing, a slice is specified for each dimension
         ## The first two rows of the first column done two ways
         B[1:2, ]

Out[64]: 2-element Array{Int64,1}:
          80
          90

In [65]: B[1:2,1]

Out[65]: 2-element Array{Int64,1}:
          80
          90

In [66]: ## The first two rows of the second column
         B[1:2,2]

Out[66]: 2-element Array{Int64,1}:
          81
          91

In [67]: # The first row
         B[1,:]

Out[67]: 3-element Array{Int64,1}:
          80
          81
          82

In [68]: ## The third element
         B[3]

Out[68]: 81

In [69]: # Another way to build an array is using comprehensions
         A1 = [sqrt(i) for i in [16,25,64]]

Out[69]: 3-element Array{Float64,1}:
          4.0
          5.0
          8.0

In [70]: A2 = [i^2 for i in [1,2,3]]

Out[70]: 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 k th element of the $n \times m$ matrix X is the k th element of the nm -vector $\text{vec}(X)$. Array comprehensions, illustrated above, are another more sophisticated way of building arrays. They 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, `Pair{typeof(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 [71]: ## Three dictionaries, D0 is empty, D1 and D1 are the same
        D0 = Dict{<int>,<string>}()
        D1 = Dict{<int>,<string>}(1 => "red", 2 => "white")
        D2 = Dict{Integer, String}(1 => "red", 2 => "white")
        typeof(D2)
```

```
Out[71]: Dict{Integer,String}
```

```
In [72]: ## 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[72]: Dict{Int64,String} with 3 entries:
        2 => "maple syrup"
        3 => "touriere"
        1 => "salmon"
```

```
In [73]: ## 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[73]: 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 [74]: ## Values can be accessed similarly to an array, but by key:
        food_dict[1]
```

```
Out [74]: "salmon"
```

```
In [75]: ## 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 [75]: "salmon"
```

```
In [76]: get(food_dict,7,"unknown")
```

```
Out [76]: "unknown"
```

```
In [77]: ## we can also check directly for the presence of a particular key  
haskey(food_dict,2)
```

```
Out [77]: true
```

```
In [78]: haskey(food_dict,9)
```

```
Out [78]: false
```

```
In [79]: ## 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 [79]: 1
```

```
In [80]: ## A new value can be associated with an existing key  
food_dict[1]  
food_dict[1] = "lobster"
```

```
Out [80]: "lobster"
```

```
In [81]: ## Two common ways to add new entries:  
food_dict[4] = "bannock"  
get!(food_dict,4,"miss")  
food_dict[5]
```

```
KeyError: key 5 not found
```

```
Stacktrace:
```

```
[1] getindex(::Dict{Int64,String}, ::Int64) at .\dict.jl:478
```

```
[2] top-level scope at In[81]:4
```

```

In [82]: ## 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[82]: "bannock"

In [83]: ## Just deleting entries by key is straightforward
         delete!(food_dict,4)

Out[83]: Dict{Int64,String} with 3 entries:
          2 => "maple syrup"
          3 => "touriere"
          1 => "lobster"

In [84]: ## But we can also delete by key and return the value associated with the key;
         ## note that 999 is returned here if the key is not present
         deleted_fd_value = pop!(food_dict,3,999)
         food_dict

Out[84]: Dict{Int64,String} with 2 entries:
          2 => "maple syrup"
          1 => "lobster"

In [85]: ## Keys can be coerced into arrays
         collect(keys(food_dict))

Out[85]: 2-element Array{Int64,1}:
          2
          1

In [86]: collect(values(food_dict))

Out[86]: 2-element Array{String,1}:
          "maple syrup"
          "lobster"

In [87]: ## 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: 1value: lobster

In [88]: ## 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: 1
```

```
In [89]: ## 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:lobster
```

1.5 Control Flow

1.5.1 Compound Expressions

In Julia, a compound expression is one expression that is used to sequentially evaluate a group of subexpressions. The value of the last subexpression is returned as the value of the expression. There are two ways to achieve this: `begin` blocks and `chain`.

```
In [90]: ## A begin block
        b1 = begin
            c = 20
            d = 5
            c * d
        end
        println("b1: ", b1)
```

```
b1: 100
```

```
In [91]: ## 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 be 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 a given Boolean expression evaluates to true, the associated block of code is executed. No other code blocks or conditional expressions within the `if-elseif-else` construct return 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 [92]: # An if-else construct
k = 1
if k == 0
    "zero"
else
    "not zero"
end
```

```
Out[92]: "not zero"
```

```
In [93]: ## An if-elseif-else construct
k = 11
if k % 3 == 0
    0
elseif k % 3 == 1
    1
else
    2
end
```

```
Out[93]: 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 associates from right to left, and it can be useful for short conditional statements. Ternary operators can be chained together to accommodate situations analogous to an if-elseif-else construct with one or more ifelse blocks.

```
In [94]: # A short-circuit evaluation
b = 10; c = 20;
println("SCE: b < c: ", b < c ? "less than" : "not less than")
```

```
SCE: b < c: less than
```

```
In [95]: # A short-circuit evaluation with nesting
d = 10; f = 10;
println("SCE: chained d vs e: ",
        d < f ? "less than " :
        d > f ? "greater than" : "equal")
```

```
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 [96]: e
```

```
UndefVarError: e not defined
```

```
Stacktrace:
```

```
[1] top-level scope at In[96]:1
```

```
In [97]: using Base.MathConstants
         e
```

```
Out[97]: = 2.7182818284590...
```

1.5.3 Loops

Basics Two looping constructs exist in Julia: for loops and while loops. These 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 loop, 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 [98]: str = "Julia"
```

```
## A for loop for a string, iterating by index
for i = 1:length(str)
    print(str[i])
end
```

```
Julia
```

```
In [100]: ## A for loop for a string, iterating by container element
          for s in str
              print(s)
          end
```

```
Julia
```

```
In [101]: ## A nested for loop
          for i in str, j = length(str)
              println((i,j))
          end
```

```
('J', 5)
('u', 5)
('l', 5)
('i', 5)
('a', 5)
```

```
In [103]: odd = [1,3,5]
          even = [2,4,6]
          for i in odd, j in even
            println("i*j: $(i*j)")
          end
```

```
i*j: 2
i*j: 4
i*j: 6
i*j: 6
i*j: 12
i*j: 18
i*j: 10
i*j: 20
i*j: 30
```

A while loop evaluates a conditional expression and, as long as it is true, the loop evaluates the code in the body of the loop. To ensure that the loop will end at some stage, an operation inside the loop has to falsify the conditional expression. Programmers must ensure that a while loop will falsify the conditional expression, otherwise the loop will become “infinity” and never finish executing.

```
In [1]: ## Example of an infinity while loop (nothing inside the loop can falsify
        ## the condition x<10)
        n=0
        x=1
        while x<10
            x=x+1
            print(10)
        end
```

```
101010101010101010
```

```
In [ ]:
```