juliastart

July 6, 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 [15]: ## 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[15]: 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 [16]:
```
x = 2
y = 3
z = 4

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

Out[16]: 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 () inclueded in the expression, we can control the order by ourselves like the following code

In [17]:
```
x*y+z^2
x*(y+z^2)
(x*y)+(z^2)
```

Out[17]: 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 [18]:
```
Sys.WORD_SIZE
```

Out[18]: 64

In [19]:
```
typemax(Int)
```

Out[19]: 9223372036854775807

In [20]:
```
typemin(Int)
```

Out[20]: -9223372036854775808

In [21]:
```
typemax(Int64)
```

Out[21]: 9223372036854775807

```
In [22]: typemax(Float32)

Out[22]: 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 [23]: literal_int = 1
         println("typeof(literal_int):",typeof(literal_int))

typeof(literal_int):Int64


In [24]: x = typemax(Int64)

Out[24]: 9223372036854775807

In [25]: x += 1

Out[25]: -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 [26]: 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 [27]: 9.2_4==9.24
         ## digit separation using an _

Out[27]: 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 [28]: eps()

Out[28]: 2.220446049250313e-16
```

```
In [29]: 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 [30]: s1 = "Hi"

Out[30]: "Hi"
```

```
In [31]: s2 = """I have a "quote" character"""
```

```
Out[31]: "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 [32]: ## some examples of strings
         str = "Data science is fun!"
         str[1]
```

```
Out[32]: 'D': ASCII/Unicode U+0044 (category Lu: Letter, uppercase)
```

```
In [33]: str[end]
```

```
Out[33]: '!': ASCII/Unicode U+0021 (category Po: Punctuation, other)
```

```
In [34]: str[4:7]
```

```
Out[34]: "a sc"
```

```
In [35]: str[end-3:end]
```

```
Out[35]: "fun!"
```

```
In [36]: string(str," Sure is :)")
```

```
Out[36]: "Data science is fun! Sure is :)"
```

```
In [37]: str * "Sure is :)"
```

```
Out[37]: "Data science is fun!Sure is :)"
```

```
In [38]: # Interpolation
         "1+2=$(1+2)"
```

```
Out[38]: "1+2=3"
```

```
In [39]: word1="Julia"
         word2="data"
         word3="science"
         "$word1 is great for $word2 $word3"
```

```
Out[39]: "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 [40]: # Lexicographical comparison
         s1 = "abcd"
         s2 = "abde"

         s1 == s2

Out[40]: false

In [41]: s1 < s2

Out[41]: true

In [42]: s1 > s2

Out[42]: false

In [43]: # String functions
         str = "Data science is fun!"

         findfirst("Data",str)

Out[43]: 1:4

In [44]: occursin("ata",str)

Out[44]: true

In [45]: replace(str,"fun"=>"great")

Out[45]: "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 [46]: # Regular expresions

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

Out[46]: true

In [47]: occursin(r"^[a-zA-z0-9]",str)

Out[47]: true

In [48]: occursin(r"[a-zA-z0-9]$",str)

Out[48]: false

In [49]: occursin(r"[^a-zA-Z0-9]",str)

Out[49]: true

In [50]: ## matches the first non-alpha-numeric character in the string
         match(r"[^a-zA-Z0-9]",str)

Out[50]: RegexMatch(" ")

In [51]: ## matches all the non-alpha-numeric characters in the string
         collect(eachmatch(r"[^a-zA-Z0-9]",str))

Out[51]: 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 [52]:
```
# A tuple comprising only floats
tup1 = (3.0,0.1,0.8,1.9)
typeof(tup1)
```

Out[52]: NTuple{4,Float64}

In [53]:
```
## A tuple comprising strings and floats
tup2 = ("Data",2.5,"Science", 8.8)
typeof(tup2)
```

Out[53]: Tuple{String,Float64,String,Float64}

In [54]:
```
## variable assignment
a,b,c = ("Fast", 1, 5.2)
a
```

Out[54]: "Fast"

In [55]:
```
b
```

Out[55]: 1

In [56]:
```
c
```

Out[56]: 5.2

In [57]:
```
(a,b,c)=("safa",2341,123.412)
typeof(a)
```

Out[57]: 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 [58]:
```
# A vector of length 5 containing integers
a1 = Array{Int64}(undef,5)
typeof(a1)
```

```
Out[58]: Array{Int64,1}
```

```
In [59]: # A 2\times2 matrix containting integers
         a2 = Array{Int64}(undef,(2,2))
```

```
Out[59]: 2×2 Array{Int64,2}:
          296955136  296955296
          296955280  112897568
```

```
In [60]: # A 2\times 2 matrix containing Any type
         a2 = Array{Any}(undef,(2,2))
```

```
Out[60]: 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 [61]: ## A three-element row "Vector"
         a4 = [1,2,3]
         typeof(a4)
```

```
Out[61]: 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 [62]: ## A 1\times 3 colum vector  -- a two-dimensional array
         a5 = [1 2 3]
         typeof(a5)
```

```
Out[62]: Array{Int64,2}
```

```
In [63]: ## A 2\times 3 matrix, where ; is used to separate rows
         a6 = [80 81 82 ; 90 91 92]
         typeof(a6)
```

```
Out[63]: Array{Int64,2}
```

```
In [64]: ## Arrays containing elements of a specific type can be constructed like:
         a7 = Float64[3.0 5.0 ; 1.1 3.5]
```

```
Out[64]: 2×2 Array{Float64,2}:
          3.0  5.0
          1.1  3.5
```

```
In [65]: ## Arrays can be explicitly created like this:
         Vector(undef,3)
```

```
Out[65]: 3-element Array{Any,1}:
          #undef
          #undef
          #undef
```

```
In [66]: Matrix(undef, 2,2)
```

```
Out[66]: 2×2 Array{Any,2}:
          #undef  #undef
          #undef  #undef
```

```
In [67]: # A 3-element Float array
         a3 = collect(Float64, 3:-1:1)
```

```
Out[67]: 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 [68]: ## Create a 2x2 identity matrix
         using LinearAlgebra
         imat = Matrix{Int8}(I, (2,2))
```

```
Out[68]: 2×2 Array{Int8,2}:
          1  0
          0  1
```

In [69]:
```
## return rando numbers between 0 and 1
rand(2)
```

Out[69]:
```
2-element Array{Float64,1}:
 0.9011806932440589
 0.7846081397345164
```

In [70]:
```
B = [80 81 82 ; 90 91 92]
```

Out[70]:
```
2Œ3 Array{Int64,2}:
 80  81  82
 90  91  92
```

In [71]:
```
rand(B,2)
```

Out[71]:
```
2-element Array{Int64,1}:
 90
 82
```

In [72]:
```
## The number of elements in B
length(B)
```

Out[72]:
```
6
```

In [73]:
```
## The dimensions of B
size(B)
```

Out[73]:
```
(2, 3)
```

In [74]:
```
## the number of dimensions of B
ndims(B)
```

Out[74]:
```
2
```

In [75]:
```
## A new array with the same elements (data) as B but different dimensions
reshape(B, (3,2))
```

Out[75]:
```
3Œ2 Array{Int64,2}:
 80  91
 90  82
 81  92
```

In [76]:
```
ndims(B)
```

Out[76]:
```
2
```

In [77]:
```
## A copy of B, where elements are recursively copied
B2 = deepcopy(B)
```

Out[77]:
```
2Œ3 Array{Int64,2}:
 80  81  82
 90  91  92
```

```
In [78]: ## When slicing, a slice is specified for each dimension
         ## The first two rows of the first column done two ways
         B[1:2, ]

Out[78]: 2-element Array{Int64,1}:
          80
          90

In [79]: B[1:2,1]

Out[79]: 2-element Array{Int64,1}:
          80
          90

In [80]: ## The first two rows of the second column
         B[1:2,2]

Out[80]: 2-element Array{Int64,1}:
          81
          91

In [81]: # The first row
         B[1,:]

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

In [82]: ## The third element
         B[3]

Out[82]: 81

In [83]: # Another way to build an array is using comprehensions
         A1 = [sqrt(i) for i in [16,25,64]]

Out[83]: 3-element Array{Float64,1}:
          4.0
          5.0
          8.0

In [84]: A2 = [i^2 for i in [1,2,3]]

Out[84]: 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 [85]: ## 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[85]: Dict{Integer,String}

In [86]: ## 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[86]: Dict{Int64,String} with 3 entries:
           2 => "maple syrup"
           3 => "touriere"
           1 => "salmon"

In [87]: ## 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[87]: 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 [88]: ## Values can be accessed similarly to an array, but by key:
         food_dict[1]
```

Out[88]: "salmon"

```
In [89]: ## The get() function can alos be used; note that "unknown" is
         ## the value returned here if the key is not in the dictionary
         get(food_dict,1,"unknown")
```

Out[89]: "salmon"

```
In [90]: get(food_dict,7,"unknown")
```

Out[90]: "unknown"

```
In [91]: ## we can also check directly for the presence of a particular key
         haskey(food_dict,2)
```

Out[91]: true

```
In [92]: haskey(food_dict,9)
```

Out[92]: false

```
In [93]: ## 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[93]: 1

```
In [94]: ## A new value can be associated with an existing key
         food_dict[1]
         food_dict[1] = "lobster"
```

Out[94]: "lobster"

```
In [95]: ## 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[95]:4
```

```
In [96]: ## 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[96]: "bannock"

In [97]: ## Just deleting entries by key is straightforward
         delete!(food_dict,4)

Out[97]: Dict{Int64,String} with 3 entries:
           2 => "maple syrup"
           3 => "touriere"
           1 => "lobster"

In [98]: ## 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[98]: Dict{Int64,String} with 2 entries:
           2 => "maple syrup"
           1 => "lobster"

In [99]: ## Keys can be coerced into arrays
         collect(keys(food_dict))

Out[99]: 2-element Array{Int64,1}:
          2
          1

In [100]: collect(values(food_dict))

Out[100]: 2-element Array{String,1}:
           "maple syrup"
           "lobster"

In [101]: ## 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 [102]: ## 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 [103]: ## 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 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 [104]: ## A begin block
          b1 = begin
              c = 20
              d = 5
              c * d
          end
          println("b1: ", b1)
```

```
b1: 100
```

```
In [105]: ## 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 [106]: # An if-else construct
          k = 1
          if k == 0
              "zero"
          else
              "not zero"
          end

Out[106]: "not zero"

In [107]: ## An if-elseif-else construct
          k = 11
          if k % 3 == 0
              0
          elseif k % 3 == 1
              1
          else
              2
          end

Out[107]: 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 [108]: # 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 [109]: # 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 [110]: e
```

```
UndefVarError: e not defined



Stacktrace:

 [1] top-level scope at In[110]:1
```

```
In [111]: using Base.MathConstants
          e
```

```
Out[111]:  = 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 [112]: str = "Julia"

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

```
Julia
```

```
In [113]: ## A for loop for a string, iterating by container element
          for s in str
              print(s)
          end
```

```
Julia
```

```
In [114]: ## 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 [115]: 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 [116]: ## 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 [117]: ## A while loop to estimate the median using an MM algorithm
          using Distributions, Random
          Random.seed!(1234)

          iter = 0
          N = 100
          x = rand(Normal(2,1),N)
          psi = fill!(Vector{Float64}(undef,2),1e9)
```

```
while (true)
    global iter, x, psi
    iter += 1
    if iter == 25
        println("Max iteration reached at iter = $iter")
        break
    end
    num, den = (0,0)
    ## elementwise operations in wgt
    wgt = (abs.(x .- psi[2])).^-1
    num = sum(wgt .* x)
    den = sum(wgt)
    psi = circshift(psi, 1)
    psi[2] = num/den

    dif = abs(psi[2]-psi[1])
    if dif < 0.001
        print("Converged at iteration $iter")
        break
    end
end
```

```
Converged at iteration 2
```

Loop termination When writing loops, it often advantageous to allow a loop to terminate early, before it has completed. In the case of a `while` loop, the loop would be broken before the test condition is falsified. when iterating over an iterable object with a `for` loop, it is stopped before the end of the object is reached. The `break` keyword can accomplish both tasks. The following code block has two loops, a `while` loop that calculates the square of the index variable and stops when the square is graeter than 16. Note that without the `break` keyword, this is an infinite loop. The second loop dose the same thing, but uses a `for` loop to do it. The `for` loop terminates before the end of the iterable range object is reached.

In [118]:
```
## break keyword
i = 0
while true
    global i
    sq = i^2
    println("i: $i --- sq: = $sq")
    if sq > 16
        break
    end
    i += 1
end
```

```
i: 0 --- sq: = 0
i: 1 --- sq: = 1
```

```
i: 2 --- sq: = 4
i: 3 --- sq: = 9
i: 4 --- sq: = 16
i: 5 --- sq: = 25
```

```
In [119]: for i = 1:10
              sq = i^2
              println("i: $i --- sq: $sq")
              if sq > 16
                  break
              end
          end
```

```
i: 1 --- sq: 1
i: 2 --- sq: 4
i: 3 --- sq: 9
i: 4 --- sq: 16
i: 5 --- sq: 25
```

In some situations, it might be the case that a programmer wants to move from the current iteration of a loop immediately into the next iteration before the current one is finished. This can be accomplished using the `continue` keyword.

```
In [120]: ## continue keyword
          for i in 1:5
              if i % 2 == 0
                  continue
              end
              sq = i^2
              println("i: $i --- sq: $sq")
          end
```

```
i: 1 --- sq: 1
i: 3 --- sq: 9
i: 5 --- sq: 25
```

In real world scenarios, continue could be used multiple times in a loop and there could be more complex code after the `continue` keyword.

Exception handling Exceptions are unexpected contions that can occur in a program while it is carrying out its computations. The program may not be able to carry out the required computations or return a sensible value to its caller. Usually, exceptions teminate the function or program that generates it and prints some sort of diagnostic message to standard output. An example of this is given in the following code block, where we try and take the logarithm of a negative number and the `log()` function throws an exception.

```
In [121]: ## generate an exception
          log(-1)
```

```
        DomainError with -1.0:
    log will only return a complex result if called with a complex argument. Try log(Complex(x

        Stacktrace:

         [1] throw_complex_domainerror(::Symbol, ::Float64) at .\math.jl:31

         [2] log(::Float64) at .\special\log.jl:285

         [3] log(::Int64) at .\special\log.jl:395

         [4] top-level scope at In[121]:1
```

In the above code block the `log()` function threw a `DomainError` exception. Julia has a number of built-in exceptions that can be thrown captured by Julia program. Any exception can be explicitly thrown using `throw()` function.

```
In [122]: ## throw()
          for i in [1,2,-1,3]
              if i < 0
                  throw(DomainError())
              else
                  println("i:$(log(i))")
              end
          end
```

```
i:0.0
i:0.6931471805599453
```

```
        MethodError: no method matching DomainError()
    Closest candidates are:
      DomainError(!Matched::Any) at boot.jl:256
      DomainError(!Matched::Any, !Matched::Any) at boot.jl:257

        Stacktrace:

         [1] top-level scope at .\In[122]:4
```

```
In [123]: ## error
          for i in [1,2,-1,3]
              if i<0
                  error("i is a negative number")
              else
                  println("i:$(log(i))")
              end
          end
```

```
i:0.0
i:0.6931471805599453
```

```
        i is a negative number



        Stacktrace:

         [1] error(::String) at .\error.jl:33

         [2] top-level scope at .\In[123]:4
```

In the previous code block, we throw the `DomainError()` exception when the input to `log()` is negative. Note that `DomainError()` requires the bracket `()` to return an exception object of type `ErrorException` that will immediately stop all execution of the Julia program.

If we want to test for an excepiton and handle it gracefully, we can use a `try-each` statment to do this. These statements allow us to catch an exception, store it in a variable if required, and try an alternative way of processing the input that generated the exception.

```
In [124]: ## try/catch
          for i in [1,2,-1,"A"]
              try log(i)
              catch ex
                  if isa(ex, DomainError)
                      println("i: $i --- Domain Error")
                      log(abs(i))
                  else
                      println("i: $i")
                      println(ex)
                      error("Not a DomainError")
                  end
              end
          end
```

```
i: -1 --- Domain Error
i: A
```

```
MethodError(log, ("A",), 0x000000000000641c)


        Not a DomainError



        Stacktrace:

         [1] error(::String) at .\error.jl:33

         [2] top-level scope at .\In[124]:11
```

In the previous code block, the exception is stored in the `ex` variable and when the error is not a `DomainError()`, its value is returned along with the `ErrorException` defined by the call to `error()`. Note that `try-catch` blocks can degrade the preformance code, it is better to use standard condtitional evaluation to handle known exceptions.

1.6 Functions

A function is an object that takes argument values as a tuple and maps them to return value. Functions are first-class objects in Julia. They can be:

- assigned to variables;
- called from these variables;
- passed as arguments to other functions;
- returned a svalues from a function.

A first-class object is one that accomodates all operations other objects support. Operations typically supported by first-class bojects in all programming languages are listed above. The basic syntax of a function is illustrated in the following code block.

```
In [125]: function add(x,y)
              return (x+y)
          end

Out[125]: add (generic function with 1 method)
```

In Julia, function names are all lowercase, without underscores, but can include Unicode charaters. It is best practice to avoid abbreviations, e.g., `fibonacci()` is preferable to `fib()`. The body of the function is the part contained on the line between the `function` and `end` keywords. Parenthesis syntax is used to call a function, e.g. `add(3,5)` returns `8`. Because functions are objects, they can be passed around like any value and, when passed, the parentheses are omitted.

```
In [126]: addnew = add
          addnew(3,5)

Out[126]: 8
```

Functions may also be written in assignment form, in which case the body of the function must be a single expression. This can be a very useful approach for simple functions because it makes code mych easier to read.

```
In [127]: add2(x,y) = x+y

Out[127]: add2 (generic function with 1 method)
```

Argument passing is done by refernece. Modeifications to the input data structure (e.g., array) inside the function will be visible outside it. If function inputs are not to be modified by a function, a copy of the input(s) should be made inside the functino before doing any modification. *Python* and other dynamic languages handle their functino arguments in a similar way.

```
In [128]: ## Argument passing
          function f1!(x)
              x[1] = 9999
              return(x)
          end

          ia = Int64[0,1,2]
          println("Array ia: ", ia)

Array ia: [0, 1, 2]


In [129]: f1!(ia)
          println("Argument passing be reference: ", ia)

Argument passing be reference: [9999, 1, 2]
```

By default, the last expression that is evaluated in the body of a function is its reutrn value. However, when the function body contains one or more `return` keywords, it returns immdiately when a `return` keyword is evaluated. The `return` keywird usually wraps an expression that provides a value when returned. When used with the control flow statements, the return keyword can be especially useful.

```
In [130]: ## A function with multiple option for return
          function gt(g1, g2)
              if (g1 > g2)
                  return("$g1 is largest")
              elseif (g1 < g2)
                  return ("$g2 is largest")
              else
                  return("$g1 and $g2 are equal")
              end
          end

          gt(2,5)
```

```
Out[130]: "5 is largest"
```

The majority of Julia operators are actually functions and can be called with parenthesized argument lists, just like other functinos.

```
In [131]: ## these are equivalent
          2*3

Out[131]: 6

In [132]: *(2,3)

Out[132]: 6
```

Functions can also be created without a name, and such functions are called anonymous functions. Anonymous functions can be used as arguments for functinos that take other functions as arguments.

```
In [133]: ## map() applies a function to each element of an array and returns a new
          ## array containing the resulting values
          a = [1,2,3,1,2,1]
           = mean(a)
          sd = std(a)

Out[133]: 0.816496580927726

In [134]: ## centers and scales a
          b = map(x -> (x-)/sd, a)

Out[134]: 6-element Array{Float64,1}:
           -0.8164965809277261
            0.4082482904638629
            1.632993161855452
           -0.8164965809277261
            0.4082482904638629
           -0.8164965809277261
```

Julia accommodates optional arguments by allowing funtion arguments to have default values, similar to *R* and many other languages. The value of an optional arguments dose not need to be specified in a function call.

```
In [135]: ## A function with an optional argument. This is a recursive function,
          ## i.e., a function that calls itself, for computing the sum of the first n
          ## elements of the Fibonacci sequence
          function fibonacci(n=20)
              if (n<=1)
                  return 1
              else
                  return fibonacci(n-1)+fibonacci(n-2)
              end
```

```
         end

         ## sum the first 12 elements of the Fibonacci sequence
         fibonacci(12)
```

```
Out[135]: 233
```

```
In [136]: fibonacci()
```

```
Out[136]: 10946
```

```
In [137]: fibonacci(20)
```

```
Out[137]: 10946
```

Function arguments determine its behaviour. In general, the more arguments a funtino has, the more varied its behaviour will be. Keyword arguments are useful because they help manage function behaviour; specifically, they allow arguments to be specified by name and not just position in the function call. In the below code block, an MM algorithm is demonstrated. Note that we have already used MM algorithm, but now we construct an MM algorithm as a functino. MM algorithms are blueprints for algorithms that eiger iteratively minimize a majorizing function or iteratively maximize a minorizing functino – see Hunter and Lange (2000,2004) for further details.