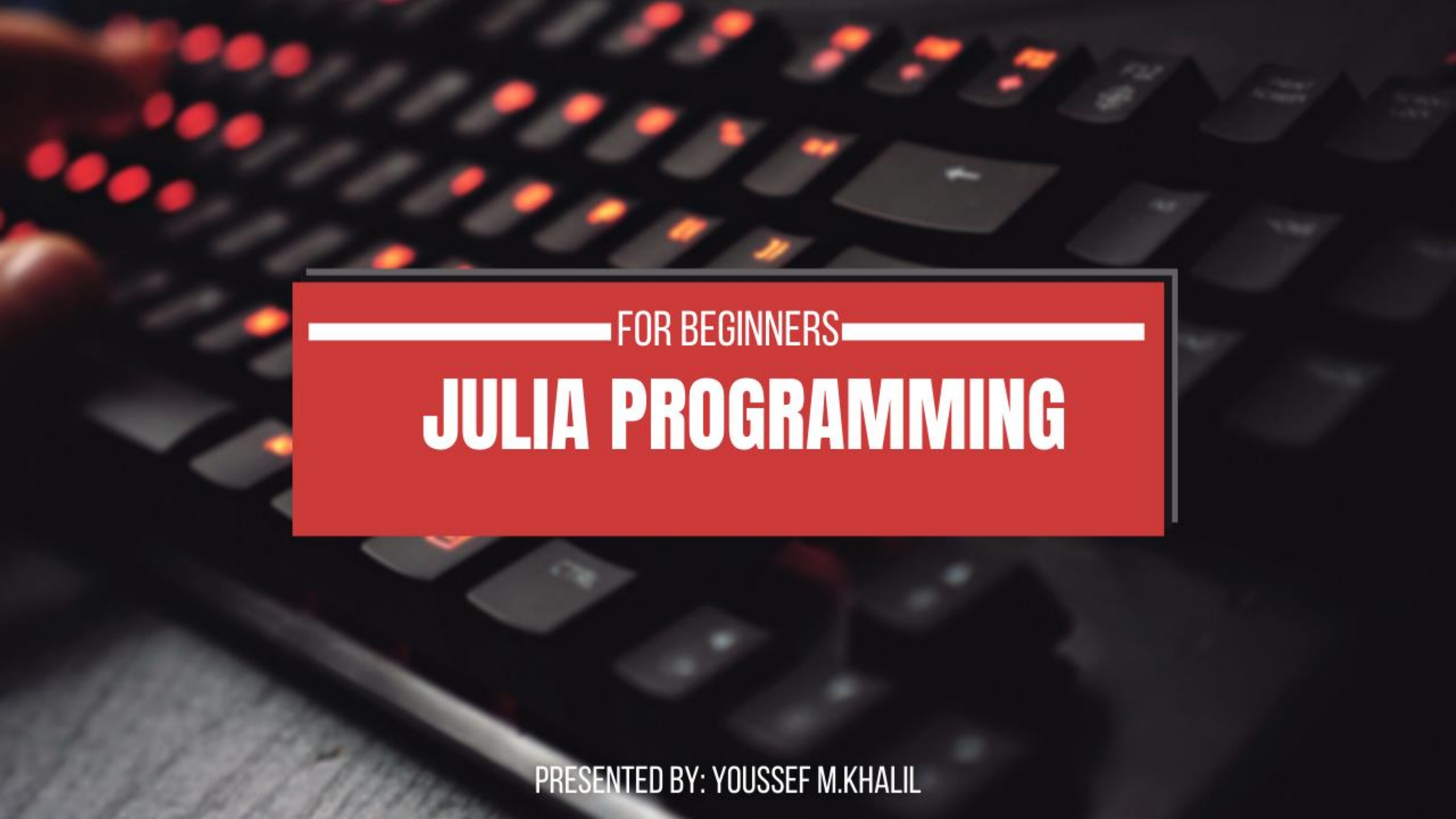

FOR BEGINNERS
JULIA PROGRAMMING
PRESENTED BY: YOUSSEF M.KHALIL

COURSE OUTLINES

- HISTORY OF JULIA AND WHY TO USE JULIA
- MATHEMATICAL OPERATIONS
- DECLARING VARIABLES
- DATA TYPES
- COMMENTS
- TYPE CASTING
- WHILE LOOP
- FOR LOOP
- DATA STRUCTURES (ARRAYS, TUPLES, DICTIONARIES, SETS)
- FUNCTIONS

DEFINITION OF THE LANGUAGE

Julia is an open-source, multi-platform, high-level, high-performance programming language widely used for statistical computing, data analytics, scientific research, data modeling, graphical representation and reporting. It is an open source programming language mainly designed for high performance scientific and numerical computation. Julia is dynamically typed programming language that support multiple dispatch, parallel and distributed computing. Julia has an LLVM-based JIT compiler which gives high performance without the typical need of separate compilation.

WHY JULIA

Comparison with other languages

One of the goals of data scientists is to achieve expressive capabilities and pure speed that avoids the need to go for 'C' programming language. Julia provides the programmers a new era of technical computing where they can develop libraries in a high-level programming language.

Following is the detailed comparison of Julia with the most used programming languages — Matlab, R, and Python:

MATLAB: The syntax of Julia is similar to MATLAB, however it is a much general purpose language when compared to MATLAB. Although most of the names of functions in Julia resemble OCTAVE (the open source version of MATLAB), the computations are extremely different. In the field of linear algebra, Julia has equally powerful capabilities as that of MATLAB, but it will not give its users the same license fee issues. In comparison to OCTAVE, Julia is much faster as well. **MATLAB.Jl** is the package with the help of which Julia provides an interface to MATLAB.

Python: Julia compiles the Python-like code into machine code that gives the programmer same performance as C programming language. If we compare the performance of Julia and Python, Julia is ahead with a factor of 10 to 30 times. With the help of ***PyCall*** package, we can call Python functions in Julia.

R: As we know, in statistical domain, R is one of the best development languages, but with a performance increase of a factor of 10 to 1,000 times, Julia is as usable as R in statistical domain. MATLAB is not a fit for doing statistics and R is not a fit for doing linear algebra, but Julia is perfect for doing both statistics and linear algebra. On the other hand, if we compare Julia's type system with R, the former has much richer type system.

WHY JULIA?

Julia incorporates various key advantages of many other high level programming languages such as C, C++, Python, and Ruby etc. Julia provides the just-in-time compiler that gives it major speed boost over other compiled languages such as C++ and Fortran. It incorporates the dynamism of many other high-level languages, such as Python or Ruby. Julia incorporates mathematical notations similar to many scientific computing languages like Matlab, R, Mathematica, and Octave. Julia provides math-friendly syntax makes it easier for non-programmers to learn and understand. Furthermore, Julia provides the statistical ease of R and the general usage of Python. Combing all these features makes Julia an incredibly powerful language.

LANGUAGE FEATURES

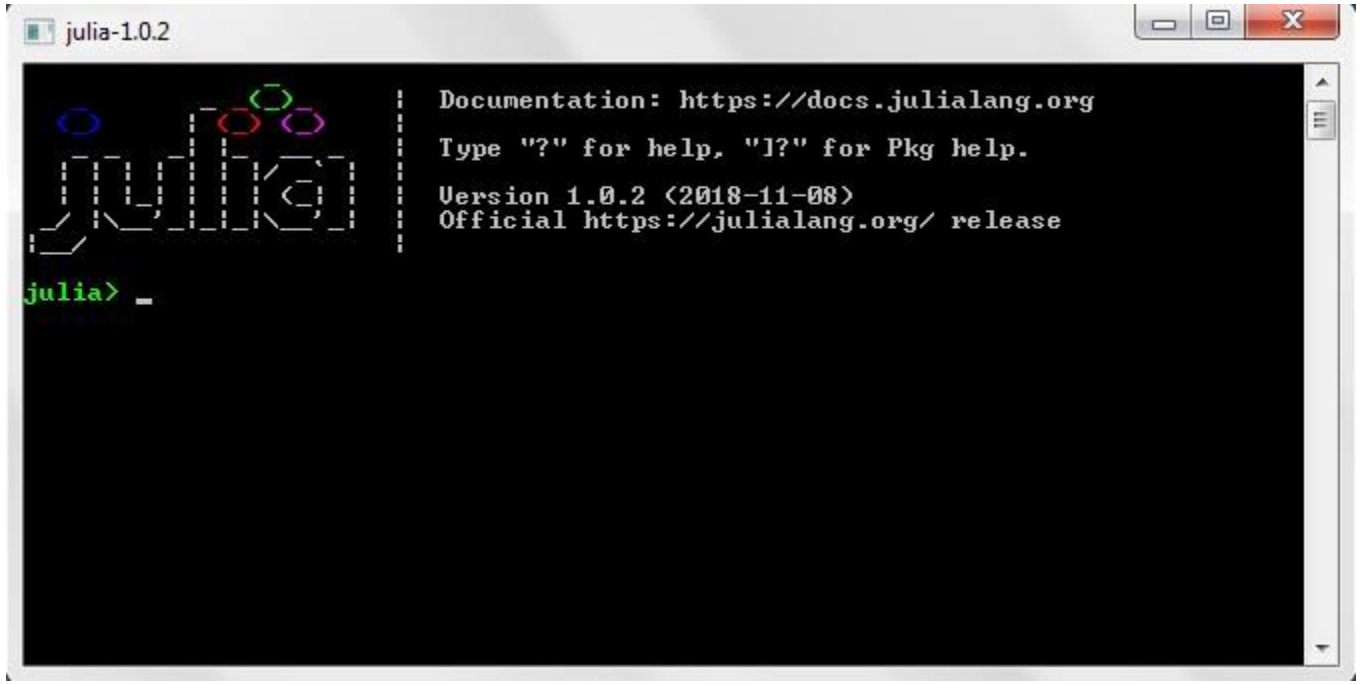
- Julia is Open source and free under MIT license
- Julia is Compiled, not interpreted, makes it faster
- Julia is inherently designed for distributed and parallel computing
- Julia provides Automatic memory management mechanism
- Julia is interoperable with other programming languages such as Python and C
- Julia is dynamic typed programming language
- It has easy to learn straightforward and math friendly syntax
- It has in-built package manager
- It comes with a powerful interactive command line tool
- It is highly extensible

JULIA INSTALLATION

Follow this link for Easy installation

Follow this link to add Julia extension to Vs code

HELLO, WORLD




```
julia> println("Hello World!")
Hello World!

julia>
```

CREATE JULIA SCRIPT

Julia Script File

Step 1:- Create a file called **"hello_world.jl"** using a text editor program of your choice. The **.jl** file extension is used to specify Julia file.

Step 2:- Let's open the **"hello_world.jl"** file that we created, and put the following line of code in it and save it.

```
1 println("Hello World!")
```

The **println()** is a function that tells the julia to display or output the content inside the parentheses.

To execute the code

Step 3:- Now, open terminal or command prompt and switch to the folder where our **"hello_world.jl"** file is saved. Run the program by typing the following command –

```
1 julia hello_world.jl
```

Once the program is executed it will print **"Hello World!"** as below –

Output:-

```
1 Hello world!
```

BASIC OPERATOR

Expression	Name	Description
+x	unary plus	It is the identity operation.
-x	unary minus	It maps values to their additive inverses.
x + y	binary plus	It performs addition.
x - y	binary minus	It performs subtraction.
x * y	times	It performs multiplication.
x / y	divide	It performs division.
x ÷ y	integer divide	Denoted as x / y and truncated to an integer.
x \ y	inverse divide	It is equivalent to y / x.
x ^ y	power	It raises x to the yth power.
x % y	remainder	It is equivalent to rem(x,y).
!x	negation	It is negation on bool types and changes true to false and vice versa.

BASIC OPERATOR

```
julia> 2+20-5
17

julia> 3-8
-5

julia> 50*2/10
10.0

julia> 23%2
1

julia> 2^4
16
```

BITWISE OPERATORS

Expression	Name
`~x`	bitwise not
`x & y`	bitwise and
`x | y`	bitwise or
`x ^ y`	bitwise xor (exclusive or)
`x >>> y`	logical shift right
`x >> y`	arithmetic shift right

UNDERSTANDING BITWISE OPERATORS

Bitwise and

	Binary Digits							
37	0	0	1	0	0	1	0	1
& 23	0	0	0	1	0	1	1	1
= 5	0	0	0	0	0	1	0	1
Digits Were Both 1?	No	No	No	No	No	Yes	No	Yes

Bitwise or

	Binary Digits							
37	0	0	1	0	0	1	0	1
| 23	0	0	0	1	0	1	1	1
= 55	0	0	1	1	0	1	1	1
Either Digit is 1?	No	No	Yes	Yes	No	Yes	Yes	Yes

Bitwise not

Two's Complement of 37								
Starting Value (37)	0	0	1	0	0	1	0	1
Flip Bits	1	1	0	1	1	0	1	0
Add One	1	1	0	1	1	0	1	1

NUMERIC COMPARISONS OPERATORS

Operator	Name
==	Equality
!=, ≠	inequality
<	less than
<=, ≤	less than or equal to
>	greater than
>=, ≥	greater than or equal to

NUMERIC COMPARISONS OPERATORS

```
julia> 100 == 100
true

julia> 100 == 101
false

julia> 100 != 101
true

julia> 100 == 100.0
true

julia> 100 < 500
true
```

OPERATOR PRECEDENCE & ASSOCIATIVITY

Category	Operators	Associativity
Syntax	`.` followed by `::`	Left
Exponentiation	`^`	Right
Unary	`+ - √`	Right
Bitshifts	`<< >> >>>`	Left
Fractions	`//`	Left
Multiplication	`* / % & \ ÷`	Left
Addition	`+ - | ⊻`	Left
Syntax	`: ..`	Left
Syntax	`|>`	Left
Syntax	`<|`	Right
Comparisons	`> < >= <= == === != !== <:`	Non-associative
Control flow	`&&` followed by `||` followed by `?`	Right
Pair	`=>`	Right

VARIABLE DECLARATION

Let us see how we can assign data to a variable. It is quite simple, just type it. For example,

```
student_name = “Ram”
roll_no = 15
marks_math = 9.5
```

Here, the first variable i.e. **student_name** contains a **string,** the second variable i.e. **roll_no** contains a **number**, and the third variable i.e. **marks_math** contains a **floating-point number**. We see, unlike other programming languages such as C++, Python, etc.,

VARIABLE DECLARATION

Stylistic Conventions and Allowed Variable Names

Following are some conventions used for variables names:

- The names of the variables in Julia are case sensitive. So, the variables **student_name** and **Student_name** would not be same.
- The names of the variables in Julia should always start with a letter and after that we can use anything like digits, letters, underscores, etc.
- In Julia, generally lower-case letter is used with multiple words separated by an underscore.
- We should use clear, short, and to the point names for variables.
- Some of the valid Julia variable names are **student_name**, **roll_no**, **speed**, **current_time**.

COMMENTS

Single Line Comments

In Julia, the single line comments start with the symbol of **# (hashtag)** and it lasts till the end of that line. Suppose if your comment exceeds one line then you should put a # symbol on the next line also and can continue the comment. Given below is the code snippet showing single line comment:

Example

```
julia> #This is an example to demonstrate the single lined comments.

julia> #Print the given name
```

COMMENTS

Multi-line Comments

In Julia, the multi-line comment is a piece of text, like single line comment, but it is enclosed in a delimiter **#=** on the start of the comment and enclosed in a delimiter **=#** on the end of the comment. Given below is the code snippet showing multi-line comment:

Example

```
julia> #= This is an example to demonstrate the multi-line comments that tells
us about tutorialspoint.com. At this website you can browse the best resource
for Online Education.=#

julia> print(www.tutorialspoint.com)
```

BASIC MATHEMATICAL FUNCTIONS

Function	Description	Return type
round(x)	This function will round x to the nearest integer.	typeof(x)
round(T, x)	This function will round x to the nearest integer.	T
floor(x)	This function will round x towards -Inf returns the nearest integral value of the same type as x. This value will be less than or equal to x.	typeof(x)
floor(T, x)	This function will round x towards -Inf and converts the result to type T. It will throw an InexactError if the value is not representable.	T
ceil(x)	This function will round x towards +Inf and returns the nearest integral value of the same type as x. This value will be greater than or equal to x.	typeof(x)
ceil(T, x)	This function will round x towards +Inf and converts the result to type T. It will throw an InexactError if the value is not representable.	T
trunc(x)	This function will round x towards zero and returns the nearest integral value of the same type as x. The absolute value will be less than or equal to x.	typeof(x)
trunc(T, x)	This function will round x towards zero and converts the result to type T. It will throw an InexactError if the value is not representable.	T

BASIC MATHEMATICAL FUNCTIONS

Example

The example given below represent the rounding functions:

```
julia> round(3.8)
4.0
julia> round(Int, 3.8)
4
julia> floor(3.8)
3.0
julia> floor(Int, 3.8)
```

```
3
julia> ceil(3.8)
4.0
julia> ceil(Int, 3.8)
4
julia> trunc(3.8)
3.0
julia> trunc(Int, 3.8)
3
```

BASIC MATHEMATICAL FUNCTIONS

Function	Description
div(x,y), x÷y	It is the quotation from Euclidean division. Also called truncated division. It computes x/y and the quotient will be rounded towards zero.
fld(x,y)	It is the floored division. The quotient will be rounded towards -Inf i.e. largest integer less than or equal to x/y. It is shorthand for div(x, y, RoundDown).
cld(x,y)	It is ceiling division. The quotient will be rounded towards +Inf i.e. smallest integer less than or equal to x/y. It is shorthand for div(x, y, RoundUp).
rem(x,y)	remainder; satisfies x == div(x,y)*y + rem(x,y); sign matches x
mod(x,y)	It is modulus after flooring division. This function satisfies the equation x == fld(x,y)*y + mod(x,y). The sign matches y.
mod1(x,y)	This is same as mod with offset 1. It returns r∈(0,y] for y>0 or r∈[y,0) for y<0, where mod(r, y) == mod(x, y).
mod2pi(x)	It is modulus with respect to 2pi. It satisfies 0 <= mod2pi(x) < 2pi
divrem(x,y)	It is the quotient and remainder from Euclidean division. It equivalents to (div(x,y),rem(x,y)).

BASIC MATHEMATICAL FUNCTIONS

```
julia> div(11, 4)
2

julia> div(7, 4)
1

julia> fld(11, 4)
2

julia> fld(-5,3)
-2

julia> fld(7.5,3.3)
2.0

julia> cld(7.5,3.3)
3.0

julia> mod(5, 0:2)
2

julia> mod(3, 0:2)
0

julia> mod(8.9,2)
0.9000000000000004
```

BASIC MATHEMATICAL FUNCTIONS

Function	Description
abs(x)	It the absolute value of x. It returns a positive value with the magnitude of x.
abs2(x)	It returns the squared absolute value of x.
sign(x)	This function indicates the sign of x. It will return -1, 0, or +1.
signbit(x)	This function indicates whether the sign bit is on (true) or off (false). In simple words, it will return true if the value of the sign of x is -ve, otherwise it will return false.
copysign(x,y)	It returns a value Z which has the magnitude of x and the same sign as y.
flipsign(x,y)	It returns a value with the magnitude of x and the sign of x*y. The sign will be flipped if y is negative. Example: abs(x) = flipsign(x,x).

BASIC MATHEMATICAL FUNCTIONS

Example

The example given below represent the sign and absolute value functions:

```
julia> abs(-7)
7


julia> abs(5+3im)
5.830951894845301


julia> abs2(-7)
49


julia> abs2(5+3im)
34


julia> copysign(5,-10)
-5


julia> copysign(-5,10)
```

BASIC MATHEMATICAL FUNCTIONS

Function	Description
sqrt(x), √x	It will return the square root of x. For negative real arguments, it will throw DomainError.
cbrt(x), ∛x	It will return the cube root of x. It also accepts the negative values.
hypot(x,y)	It will compute the hypotenuse $\sqrt{\|x\|^2 + \|y\|^2}$ of right-angled triangle with other sides of length x and y. It is an implementation of an improved algorithm for hypot(a,b) by Carlos and F.Borges.
exp(x)	It will compute the natural base exponential of x i.e. e^x
expm1(x)	It will accurately compute $e^x - 1$ for x near zero.

BASIC MATHEMATICAL FUNCTIONS

Function	Description
sqrt(x), √x	It will return the square root of x. For negative real arguments, it will throw DomainError.
cbrt(x), ∛x	It will return the cube root of x. It also accepts the negative values.
hypot(x,y)	It will compute the hypotenuse $\sqrt{\|x\|^2 + \|y\|^2}$ of right-angled triangle with other sides of length x and y. It is an implementation of an improved algorithm for hypot(a,b) by Carlos and F.Borges.
exp(x)	It will compute the natural base exponential of x i.e. e^x
expm1(x)	It will accurately compute $e^x - 1$ for x near zero.

BASIC MATHEMATICAL FUNCTIONS

ldexp(x,n)	It will compute $X * 2^n$ efficiently for integer values of n.
log(x)	It will compute the natural logarithm of x. For negative real arguments, it will throw DomainError.
log(b,x)	It will compute the base b logarithm of x. For negative real arguments, it will throw DomainError.
log2(x)	It will compute the base 2 logarithm of x. For negative real arguments, it will throw DomainError.
log10(x)	It will compute the base 10 logarithm of x. For negative real arguments, it will throw DomainError.
log1p(x)	It will accurately compute the log(1+x) for x near zero. For negative real arguments, it will throw DomainError.
exponent(x)	It will calculate the binary exponent of x.
significand(x)	It will extract the binary significand (a.k.a. mantissa) of a floating-point number x in binary representation. If x = non-zero finite number, it will return a number of the same type on the interval [1,2), else x will be returned.

TRIGONOMETRIC AND HYPERBOLIC FUNCTIONS

Following is the list of all the standard trigonometric and hyperbolic functions:

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc
```

Julia also provides two additional functions namely sinpi(x) and cospi(x) for accurately computing sin(pi*x) and cos(pi*x).

If you want to compute the trigonometric functions with degrees, then suffix the functions with d as follows:

```
sind   cosd   tand   cotd   secd   cscd
asind  acosd  atand  acotd  asecd  acscd
```

TRIGONOMETRIC AND HYPERBOLIC FUNCTIONS

Some of the example are given below:

```
julia> cos(56)
0.853220107722584


julia> cosd(56)
0.5591929034707468
```

STRINGS

A single character is represented with **Char** value. Char is a 32-bit primitive type which can be converted to a numeric value (which represents Unicode code point).

```
julia> 'a'
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> typeof(ans)
Char
```

We can convert a Char to its integer value as follows:

```
julia> Int('a')
97

julia> typeof(ans)
Int64
```

We can also convert an integer value back to a Char as follows:

STRINGS

```
julia> 'X' < 'x'
true

julia> 'X' <= 'x' <= 'Y'
false

julia> 'X' <= 'a' <= 'Y'
false

julia> 'a' <= 'x' <= 'Y'
false

julia> 'A' <= 'X' <= 'Y'
true

julia> 'x' - 'b'
22

julia> 'x' + 1
'y': ASCII/Unicode U+0079 (category Ll: Letter, lowercase)
```

STRINGS

Delimited by double quotes or triple double quotes

As we discussed, strings in Julia can be declared using double or triple double quotes. For example, if you need to add quotations to a part in a string, you can do so using double and triple double quotes as shown below:

```
julia> str = "This is Julia Programming Language.\n"
"This is Julia Programming Language.\n"


julia> """See the "quote" characters"""
"See the \"quote\" characters"
```

STRINGS

Extracting substring by using range indexing

We can extract substring from a string by using range indexing. Check the below given example:

```
julia> str[6:9]
"is J"
```

Using SubString

In the above method, the Range indexing makes a copy of selected part of the original string, but we can use SubString to create a view into a string as given in the below example:

```
julia> substr = SubString(str, 1, 4)
"This"


julia> typeof(substr)
SubString{String}
```

STRINGS

String Concatenation

Concatenation is one of the most useful string operations. Following is an example of concatenation:

```
julia> A = "Hello"
"Hello"
julia> B = "Julia Programming Language"
"Julia Programming Language"
julia> string(A, ", ", B, ".\n")
"Hello, Julia Programming Language.\n"
```

We can also concatenate strings in Julia with the help of *. Given below is the example for the same:

```
julia> A = "Hello"
"Hello"
julia> B = "Julia Programming Language"
"Julia Programming Language"
julia> A * ", " * B * ".\n"
"Hello, Julia Programming Language.\n"
```

Interpolation

It is bit cumbersome to concatenate strings using concatenation. Therefore, Julia allows interpolation into strings and reduce the need for these verbose calls to strings. This interpolation can be done by using dollar sign ($). For example:

```
julia> A = "Hello"
"Hello"
julia> B = "Julia Programming Language"
"Julia Programming Language"
julia> "$A, $B.\n"
"Hello, Julia Programming Language.\n"
```

Julia takes the expression after $ as the expression whose whole value is to be interpolated into the string. That's the reason we can interpolate any expression into a string using parentheses. For example:

```
julia> "100 + 10 = $(100 + 10)"
"100 + 10 = 110"
```

Triple-quoted strings

We know that we can create strings with triple-quotes as given in the below example:

```
julia> """See the "quote" characters"""
"See the \"quote\" characters"
```

This kind of creation has the following advantages:

Triple-quoted strings are dedented to the level of the least-intended line, hence this becomes very useful for defining code that is indented. Following is an example of the same:

```
julia> str = """
                 This is,
                 Julia Programming Language.
               """
"  This is,\n  Julia Programming Language.\n"
```

The longest common starting sequence of spaces or tabs in all lines is known as the

Common String Operations

Using string operators provided by Julia, we can compare two strings, search whether a particular string contains the given sub-string, and join/concatenate two strings.

Standard Comparison operators

By using the following standard comparison operators, we can lexicographically compare the strings:

```
julia> "abababab" < "Tutorialspoint"
false

julia> "abababab" > "Tutorialspoint"
true

julia> "abababab" == "Tutorialspoint"
false

julia> "abababab" != "Tutorialspoint"
true
```

Search operators

Julia provides us **findfirst** and **findlast** functions to search for the index of a particular character in string. You can check the below example of both these functions:

```
julia> findfirst(isequal('o'), "Tutorialspoint")
4
```

Julia also provides us **findnext** and **findprev** functions to start the search for a character at a given offset. Check the below example of both these functions:

```
julia> findnext(isequal('o'), "Tutorialspoint", 1)
4
julia> findnext(isequal('o'), "Tutorialspoint", 5)
11
julia> findprev(isequal('o'), "Tutorialspoint", 5)
4
```

STRINGS

It is also possible to check if a substring is found within a string or not. We can use **occursin** function for this. The example is given below:

```
julia> occursin("Julia", "This is, Julia Programming.")
true

julia> occursin("T", "Tutorialspoint")
true

julia> occursin("Z", "Tutorialspoint")
false
```

The repeat() and join() functions

In the perspective of Strings in Julia, repeat and join are two useful functions. Example below explains their use:

```
julia> repeat("Tutorialspoint.com ", 5)

"Tutorialspoint.com Tutorialspoint.com Tutorialspoint.com Tutorialspoint.com
Tutorialspoint.com "


julia> join(["TutorialsPoint","com"], " . ")

"TutorialsPoint . com"
```

IF STATEMENT

If, elseif and else

We can also use **if**, **elseif**, and **else** for conditions execution. The only condition is that all the conditional construction should finish with **end**.

Example

```
julia> fruit = "Apple"
"Apple"


julia> if fruit == "Apple"
          println("I like Apple")
       elseif fruit == "Banana"
          println("I like Banana.")
          println("But I prefer Apple.")
       else
          println("I don't know what I like")
       end
```

FOR LOOP

for loops

Some of the common example of iteration are:

- working through a list or
- set of values or
- from a start value to a finish value.

We can iterate through various types of objects like arrays, sets, dictionaries, and strings by using **"for"** loop (**for...end** construction). Let us understand the syntax with the following example:

```
julia> for i in 0:5:50
                    println(i)
                end
0
5
10
15
20
25
30
```

WHILE LOOP

We use while loops to repeat some expressions while a condition is true. The construction is like **while...end**.

Example

```
julia> n = 0
0


julia> while n < 10
                  println(n)
                  global n += 1
              end
0
1
2
```

Creating Simple 1D Arrays

Following is the example showing how we can create a simple 1D array:

```
julia> arr = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3
```

The above example shows that we have created a 1D array with 3 elements each of which is a 64-bit integer. This 1D array is bound to the variable **arr**.

ARRAYS

Creating 2D arrays & matrices

Leave out the comma between elements and you will be getting 2D arrays in Julia. Below is the example given for single row, multi-column array:

```
julia> [1 2 3 4 5 6 7 8 9 10]
1×10 Array{Int64,2}:
 1  2  3  4  5  6  7  8  9  10
```

Here, **1×10** is the first row of this array.

To add another row, just add a semicolon(;). Let us check the below example:

```
julia> [1 2 3 4 5 ; 6 7 8 9 10]
2×5 Array{Int64,2}:
 1  2  3  4   5
 6  7  8  9  10
```

Here, it becomes **2×5** array.

Creating arrays using range objects

We can create arrays using range objects in the following ways:

Collect() function

First useful function to create an array using range objects is collect(). With the help of colon(:) and collect() function, we can create an array using range objects as follows:

```
julia> collect(1:5)
5-element Array{Int64,1}:
  1
  2
  3
  4
  5
```

ARRAYS

```
start:step:stop.
```

Below is an example to build an array with elements that go from 0 to 50 in steps of 5:

```
julia> collect(0:5:50)
11-element Array{Int64,1}:
  0
  5
 10
 15
 20
 25
 30
 35
 40
 45
 50
```

ARRAYS

range() function

Range() is another useful function to create an array with range objects. It goes from **start** value to **end** value by taking a specific **step** value.

For example, let us see an example to go from 1 to 150 in exactly 15 steps:

```
julia> range(1, length=15, stop=150)
1.0:10.642857142857142:150.0


Or you can use range to take 10 steps from 1, stopping at or before 150:
julia> range(1, stop=150, step=10)
1:10:141
```

ARRAYS

```
julia> collect(range(1, length=15, stop=150))
15-element Array{Float64,1}:
   1.0
  11.642857142857142
  22.285714285714285
  32.92857142857143
  43.57142857142857
  54.214285714285715
  64.85714285714286
  75.5
  86.14285714285714
  96.78571428571429
 107.42857142857143
 118.07142857142857
 128.71428571428572
 139.35714285714286
 150.0
```

Creating arrays using comprehensions and generators

Another useful way to create an array is to use comprehensions. In this way, we can create array where each element can produce using a small computation. For example, we can create an array of 10 elements as follows:

```
julia> [n^2 for n in 1:10]
10-element Array{Int64,1}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

ARRAYS

We can easily create a 2-D array also as follows:

```
julia> [n*m for n in 1:10, m in 1:10]
10×10 Array{Int64,2}:
  1   2   3   4   5   6   7   8   9   10
  2   4   6   8  10  12  14  16  18   20
  3   6   9  12  15  18  21  24  27   30
  4   8  12  16  20  24  28  32  36   40
  5  10  15  20  25  30  35  40  45   50
  6  12  18  24  30  36  42  48  54   60
  7  14  21  28  35  42  49  56  63   70
  8  16  24  32  40  48  56  64  72   80
  9  18  27  36  45  54  63  72  81   90
 10  20  30  40  50  60  70  80  90  100
```

Arrays of arrays

Following example demonstrates creating arrays of arrays:

```
julia> ABC = Array[[3,4],[5,6]]
2-element Array{Array,1}:
 [3, 4]
 [5, 6]
```

It can also be created with the help of **Array** constructor as follows:

```
julia> Array[1:5,6:10]
2-element Array{Array,1}:
 [1, 2, 3, 4, 5]
 [6, 7, 8, 9, 10]
```

Copying arrays

Suppose you have an array and want to create another array with similar dimensions, then you can use **similar()** function as follows:

```
julia> A = collect(1:5);
```

Here we have hide the values with the help of semicolon(;)

```
julia> B = similar(A)
5-element Array{Int64,1}:
 164998448
 234899984
 383606096
 164557488
 396984416
```

Here the dimension of array A are copied but not values.

Copying arrays

Suppose you have an array and want to create another array with similar dimensions, then you can use **similar()** function as follows:

```
julia> A = collect(1:5);
```

Here we have hide the values with the help of semicolon(;)

```
julia> B = similar(A)
5-element Array{Int64,1}:
 164998448
 234899984
 383606096
 164557488
 396984416
```

Here the dimension of array A are copied but not values.

ARRAYS

Accessing the contents of arrays

In Julia, to access the contents/particular element of an array, you need to write the name of the array with the element number in square bracket.

Below is an example of 1-D array:

```
julia> arr = [5,10,15,20,25,30,35,40,45,50]
10-element Array{Int64,1}:
  5
 10
 15
 20
 25
 30
 35
 40
 45
 50
```

ARRAYS

```
julia> arr[4]
20
```

In some programming languages, the last element of an array is referred to as **-1**. However, in Julia, it is referred to as **end**. You can find the last element of an array as follows:

```
julia> arr[end]
50
```

And the second last element as follows:

```
julia> arr[end-1]
45
```

ARRAYS

To access row1, column2 element, we need to use the command below:

```
julia> arr2[1,2]
11
```

Similarly, for row1 and column3 element, we have to use the below command:

```
julia> arr2[1,3]
12
```

We can also use **getindex()** function to obtain elements from a 2-D array:

```
julia>  getindex(arr2,1,2)
11


julia>  getindex(arr2,2,3)
15
```

Adding Elements

We can add elements to an array in Julia at the end, at the front and at the given index using **push!()**, **pushfirst!() and splice!()** functions respectively.

At the end

We can use push!() function to add an element at the end of an array. For example,

```
julia> push!(arr,55)
11-element Array{Int64,1}:
  5
 10
 15
 20
 25
 30
 35
```

ARRAYS

At the front

We can use **pushfirst!()** function to add an element at the front of an array. For example,

```
julia> pushfirst!(arr,0)
12-element Array{Int64,1}:
  0
  5
 10
 15
 20
 25
 30
 35
 40
 45
 50
 55
```

Remove the last element

We can use **pop!()** function to remove the last element of an array. For example,

```
julia> pop!(arr)
55


julia> arr
12-element Array{Int64,1}:
  0
  2
  3
  4
  5
  6
 25
 30
 35
 40
```

ARRAYS

Removing the first element

We can use **popfirst!()** function to remove the first element of an array. For example,

```
julia> popfirst!(arr)
0


julia> arr
11-element Array{Int64,1}:
  2
  3
  4
  5
  6
 25
 30
 35
 40
 45
 50
```

TUPLES

Creating tuples

We can create tuples as arrays and most of the array's functions can be used on tuples also. Some of the example are given below:

```
julia> tupl=(5,10,15,20,25,30)
(5, 10, 15, 20, 25, 30)

julia> tupl
(5, 10, 15, 20, 25, 30)

julia> tupl[3:end]
(15, 20, 25, 30)

julia> tupl = ((1,2),(3,4))

((1, 2), (3, 4))

julia> tupl[1]
(1, 2)

julia> tupl[1][2]
2
We cannot change a tuple:
julia> tupl[2]=0
ERROR: MethodError: no method matching
setindex!(::Tuple{Tuple{Int64,Int64},Tuple{Int64,Int64}}, ::Int64, ::Int64)
Stacktrace:
 [1] top-level scope at REPL[7]:1
```

NAMED TUPLES

Named tuples

A named tuple is simply a combination of a tuple and a dictionary because:

- A named tuple is ordered and immutable like a tuple and
- Like a dictionary in named tuple, each element has a unique key which can be used to access it.

In next section, let us see how we can create named tuples:

Creating named tuples

You can create named tuples in Julia by:

- Providing keys and values in separate tuples
- Providing keys and values in a single tuple
- Combining two existing named tuples

NAMED TUPLES

Named tuples

A named tuple is simply a combination of a tuple and a dictionary because:

- A named tuple is ordered and immutable like a tuple and
- Like a dictionary in named tuple, each element has a unique key which can be used to access it.

In next section, let us see how we can create named tuples:

Creating named tuples

You can create named tuples in Julia by:

- Providing keys and values in separate tuples
- Providing keys and values in a single tuple
- Combining two existing named tuples

NAMED TUPLES

Keys and values in separate tuples

One way to create named tuples is by providing keys and values in separate tuples.

Example

```
julia> names_shape = (:corner1, :corner2)
(:corner1, :corner2)

julia> values_shape = ((100, 100), (200, 200))
((100, 100), (200, 200))

julia> shape_item2 = NamedTuple{names_shape}(values_shape)
(corner1 = (100, 100), corner2 = (200, 200))
```

We can access the elements by using dot(.) syntax:

```
julia> shape_item2.corner1
(100, 100)

julia> shape_item2.corner2
(200, 200)
```

NAMED TUPLES

Keys and values in a single tuple

We can also create named tuples by providing keys and values in a single tuple.

Example

```
julia> shape_item = (corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0))
(corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0))
```

NAMED TUPLES

We can access the elements by using dot(.) syntax:

```
julia> shape_item.corner1
(1, 1)

julia> shape_item.corner2
(-1, -1)

julia> shape_item.center
(0, 0)

julia> (shape_item.center,shape_item.corner2)
((0, 0), (-1, -1))
```

We can also access all the values as with ordinary tuples as follows:

```
julia> c1, c2, center = shape_item
(corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0))

julia> c1
(1, 1)
```

NAMED TUPLES

Combining two named tuples

Julia provides us a way to make new named tuples by combining two named tuples together as follows:

Example

```
julia> colors_shape = (top = "red", bottom = "green")
(top = "red", bottom = "green")


julia> shape_item = (corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0))
(corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0))


julia> merge(shape_item, colors_shape)
(corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0), top = "red", bottom =
"green")
```

NAMED TUPLES

Combining two named tuples

Julia provides us a way to make new named tuples by combining two named tuples together as follows:

Example

```
julia> colors_shape = (top = "red", bottom = "green")
(top = "red", bottom = "green")


julia> shape_item = (corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0))
(corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0))


julia> merge(shape_item, colors_shape)
(corner1 = (1, 1), corner2 = (-1, -1), center = (0, 0), top = "red", bottom =
"green")
```

DICTIONARIES

Creating Dictionaries

The syntax for creating a simple dictionary is as follows:

```
Dict(“key1” => value1, “key2” => value2,,…, “keyn” => valuen)
```

In the above syntax, key1, key2…keyn are the keys and value1, value2,…valuen are the corresponding values. The operator => is the Pair() function. We can not have two keys with the same name because keys are always unique in dictionaries.

Example

```
julia> first_dict = Dict("X" => 100, "Y" => 110, "Z" => 220)
Dict{String,Int64} with 3 entries:
  "Y" => 110
  "Z" => 220
  "X" => 100
```

DICTIONARIES

Creating Dictionaries

The syntax for creating a simple dictionary is as follows:

```
Dict(“key1” => value1, “key2” => value2,,…, “keyn” => valuen)
```

In the above syntax, key1, key2…keyn are the keys and value1, value2,…valuen are the corresponding values. The operator => is the Pair() function. We can not have two keys with the same name because keys are always unique in dictionaries.

Example

```
julia> first_dict = Dict("X" => 100, "Y" => 110, "Z" => 220)
Dict{String,Int64} with 3 entries:
  "Y" => 110
  "Z" => 220
  "X" => 100
```

DICTIONARIES

We can also create dictionaries with the help of comprehension syntax. The example is given below:

Example

```
julia>  first_dict = Dict(string(x) => sind(x) for x = 0:5:360)
Dict{String,Float64} with 73 entries:
  "320" => -0.642788
  "65"  => 0.906308
  "155" => 0.422618
  "335" => -0.422618
  "75"  => 0.965926
  "50"  => 0.766044
    ⋮     => ⋮
```

DICTIONARIES

Searching for a key

We can use **haskey()** function to check whether the dictionary contains a key or not:

```
julia> first_dict = Dict("X" => 100, "Y" => 110, "Z" => 220)
Dict{String,Int64} with 3 entries:
  "Y" => 110
  "Z" => 220
  "X" => 100

julia> haskey(first_dict, "Z")
true

julia> haskey(first_dict, "A")
false
```

DICTIONARIES

Searching for a key/value pair

We can use **in()** function to check whether the dictionary contains a key/value pair or not:

```
julia> in(("X" => 100), first_dict)
true


julia> in(("X" => 220), first_dict)
false
```

DICTIONARIES

Add a new key-value

We can add a new key-value in the existing dictionary as follows:

```
julia> first_dict["R"] = 400
400


julia> first_dict
Dict{String,Int64} with 4 entries:
  "Y" => 110
```

DICTIONARIES

Add a new key-value

We can add a new key-value in the existing dictionary as follows:

```
julia> first_dict["R"] = 400
400


julia> first_dict
Dict{String,Int64} with 4 entries:
  "Y" => 110
  "Z" => 220
  "X" => 100
  "R" => 400
```

DICTIONARIES

Delete a key

We can use **delete!()** function to delete a key from an existing dictionary:

```
julia> delete!(first_dict, "R")
Dict{String,Int64} with 3 entries:
  "Y" => 110
  "Z" => 220
  "X" => 100
```

Getting all the keys

We can use **keys()** function to get all the keys from an existing dictionary:

```
julia> keys(first_dict)
Base.KeySet for a Dict{String,Int64} with 3 entries. Keys:
  "Y"
  "Z"
  "X"
```

DICTIONARIES

Sorting a dictionary

Dictionaries do not store the keys in any particular order hence the output of the dictionary would not be a sorted array. To obtain items in order, we can sort the dictionary:

Example

```
julia> first_dict = Dict("R" => 100, "S" => 220, "T" => 350, "U" => 400, "V" =>
575, "W" => 670)
Dict{String,Int64} with 6 entries:
  "S" => 220
  "U" => 400
  "T" => 350
  "W" => 670
  "V" => 575
  "R" => 100


julia> for key in sort(collect(keys(first_dict)))
           println("$key => $(first_dict[key])")
           end
R => 100
S => 220
T => 350
U => 400
V => 575
W => 670
```

SETS

Like an array or dictionary, a set may be defined as a collection of unique elements. Following are the differences between sets and other kind of collections:

- In a set, we can have only one of each element.
- The order of element is not important in a set.

Creating a Set

With the help of **Set** constructor function, we can create a set as follows:

```
julia> var_color = Set()
Set{Any}()
```

We can also specify the types of set as follows:

```
julia> num_primes = Set{Int64}()
Set{Int64}()
```

SETS

We can also create and fill the set as follows:

```
julia> var_color = Set{String}(["red","green","blue"])
Set{String} with 3 elements:
  "blue"
  "green"
  "red"
```

Alternatively we can also use **push!()** function, as arrays, to add elements in sets as follows:

```
julia> push!(var_color, "black")
Set{String} with 4 elements:
  "blue"
  "green"
  "black"
  "red"
```

SETS

Standard operations

Union, intersection, and difference are some standard operations we can do with sets. The corresponding functions for these operations are **union()**, **intersect()**, and, **setdiff()**.

Union

In general, the union (set) operation returns the combined results of the two statements.

SETS

```
julia> color_rainbow =
Set(["red","orange","yellow","green","blue","indigo","violet"])

Set{String} with 7 elements:
  "indigo"
  "yellow"
  "orange"
  "blue"
  "violet"
  "green"
  "red"


julia> union(var_color, color_rainbow)

Set{String} with 8 elements:
  "indigo"
  "yellow"
  "orange"
  "blue"
  "violet"
  "green"
  "black"
  "red"
```

SETS

Intersection

In general, an intersection operation takes two or more variables as inputs and returns the intersection between them.

Example

```
julia> intersect(var_color, color_rainbow)
Set{String} with 3 elements:
  "blue"
  "green"
  "red"
```

SETS

Difference

In general, the difference operation takes two or more variables as an input. Then, it returns the value of the first set excluding the value overlapped by the second set.

Example

```
julia> setdiff(var_color, color_rainbow)
Set{String} with 1 element:
  "black"
```

FUNCTIONS

Defining Functions

There are following three ways in which we can define functions:

When there is a **single expression** in a function, you can define it by writing the name of the function and any arguments in parentheses on the left side and write an expression on the right side of an equal sign.

Example

```
julia> f(a) = a * a
f (generic function with 1 method)


julia> f(5)
25


julia> func(x, y) = sqrt(x^2 + y^2)
func (generic function with 1 method)


julia> func(5, 4)
6.4031242374328485
```

FUNCTIONS

If there are **multiple expressions** in a function, you can define it as shown below:

```
function functionname(args)
    expression
    expression
    expression
    ...
    expression
end
```

FUNCTIONS

Example

```
julia> function bills(money)
           if money < 0
              return false
           else
              return true
           end
       end
bills (generic function with 1 method)

julia> bills(50)
true

julia> bills(-50)
false
```

FUNCTIONS

Example

```
julia> function bills(money)
           if money < 0
              return false
           else
              return true
           end
       end
bills (generic function with 1 method)

julia> bills(50)
true

julia> bills(-50)
false
```

Optional Arguments

It is often possible to define functions with optional arguments i.e. default sensible values for functions arguments so that the function can use that value if specific values are not provided. For example:

```
julia> function pos(ax, by, cz=0)
           println("$ax, $by, $cz")
       end
pos (generic function with 2 methods)

julia> pos(10, 30)
10, 30, 0
```

```
julia> pos(10, 30, 50)
10, 30, 50
```

You can check in the above output that when we call this function without supplying third value, the variable **cz** defaults to 0.

Keyword Arguments

Some functions which we define need a large number of arguments but calling such functions can be difficult because we may forget the order in which we have to supply the arguments. For example, check the below function:

```
function foo(a, b, c, d, e, f)
...
end
```

Now, we may forget the order of arguments and the following may happen:

```
foo("25", -5.6987, "hello", 56, good, 'ABC')
or
foo("hello", 56, "25", -5.6987, 'ABC', good)
```

FUNCTIONS

Julia provides us a way to avoid this problem. We can use keywords to label arguments. We need to use a semicolon after the function's unlabelled arguments and follow it with one or more **keyword-value** pair as follows:

```
julia> function foo(a, b ; c = 10, d = "hi")
         println("a is $a")
         println("b is $b")
         return "c => $c, d => $d"
       end
foo (generic function with 1 method)

julia> foo(100,20)
a is 100
b is 20
"c => 10, d => hi"

julia> foo("Hello", "Tutorialspoint", c=pi, d=22//7)
a is Hello
b is Tutorialspoint
"c => π, d => 22//7"
```

APPLICATIONS

Check this Git repo for Julia projects

SOURCES

- https://www.tutorialspoint.com/julia/index.htm
- https://www.w3adda.com/julia-tutorial/julia-hello-world-program
- https://www.mygreatlearning.com/blog/julia-tutorial/
- https://docs.julialang.org/en/v1/