Julia

Julia is a high-level, dynamic programming language which is created by many technical and scientific minds together. There main aim was to keep the language easy to learn and write like python or Matlab but at the same time it does not reduces the speed of execution and should be able to match up the performance of C and Fortran.

People used to think that you have to choose between the programming language that did your calculation quickly and the one that require less effort to write the code. But luckily the creators of modern programming languages are able to learn from all the programming languages of earlier decades. Julia was designed to give you everything – to be relatively easy and quick to write programs in, but also to run code and perform calculations very quickly.

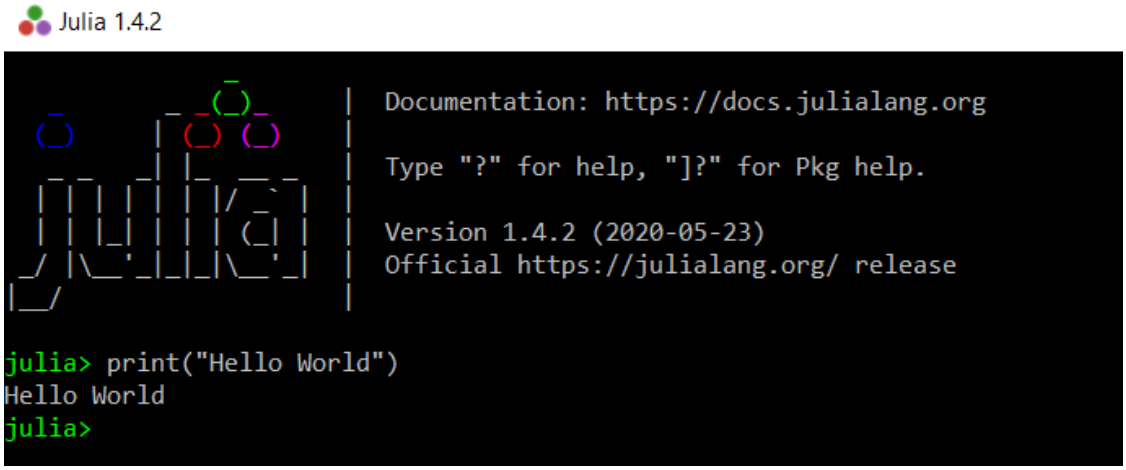
Installing Julia:

There are several ways of running Julia:

1. In the terminal using the built-in Julia command line by downloading the binary files. (https://julialang.org/downloads/)
2. Using Docker images from Docker Hub maintained by the Docker Community.
3. By using Juno IDE, it is a free environment for running Julia code.
4. Using JuliaBox, it runs on browser and you don't have to install it (https://www.juliabox.com/)

For running a basic code on Julia command line:

Print ('Hello World')



Basic Syntax:

Variables:

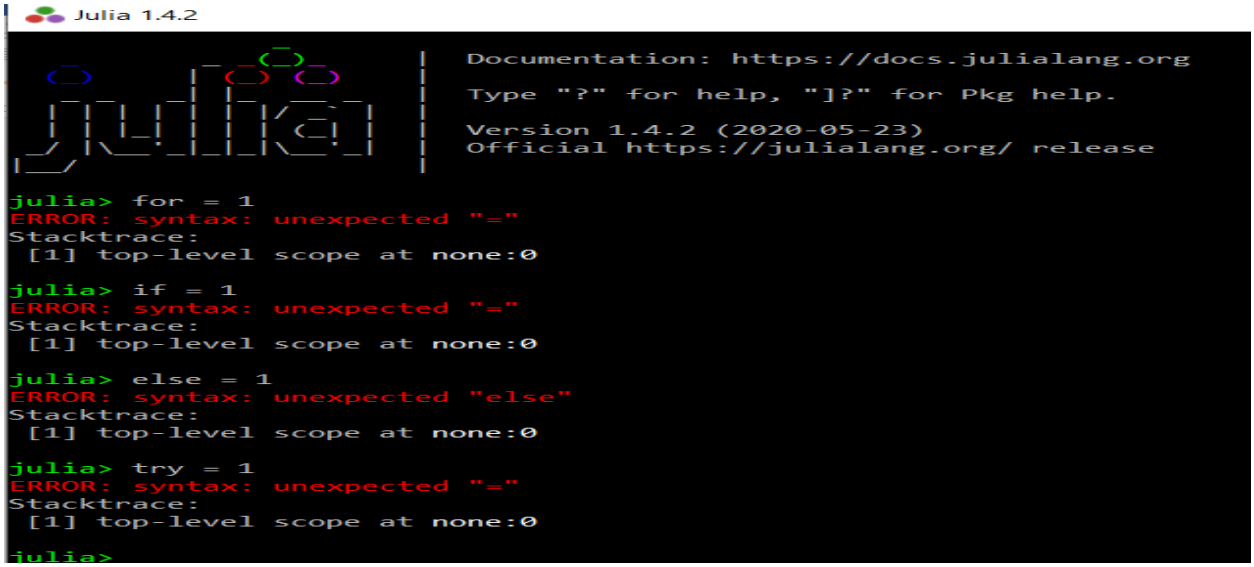
As Julia is a dynamic programming language, we do not have to provide data type explicitly.

There are constants and some functions already defined in Julia and we can override them.

For Example: pi, sqrt () etc.

But if those constants or functions are already in use then you can't override their value. It will give an error.

We also cannot declare a variable name same as the keywords, like: for, else, if, try etc.



Number:

There are two type of number in Julia:

1. Int (a = 5)
2. Float (a = 5.0)

We can do various operations over their numbers, like:

1. Addition (1+2)

2. Subtraction (1-2)
3. Multiplication (1*2)
4. Division (2/1 or div (2,1))
5. Bitwise AND (1 & 3)
6. Bitwise OR (1 | 3)
7. Power (3 ^ 2) etc.

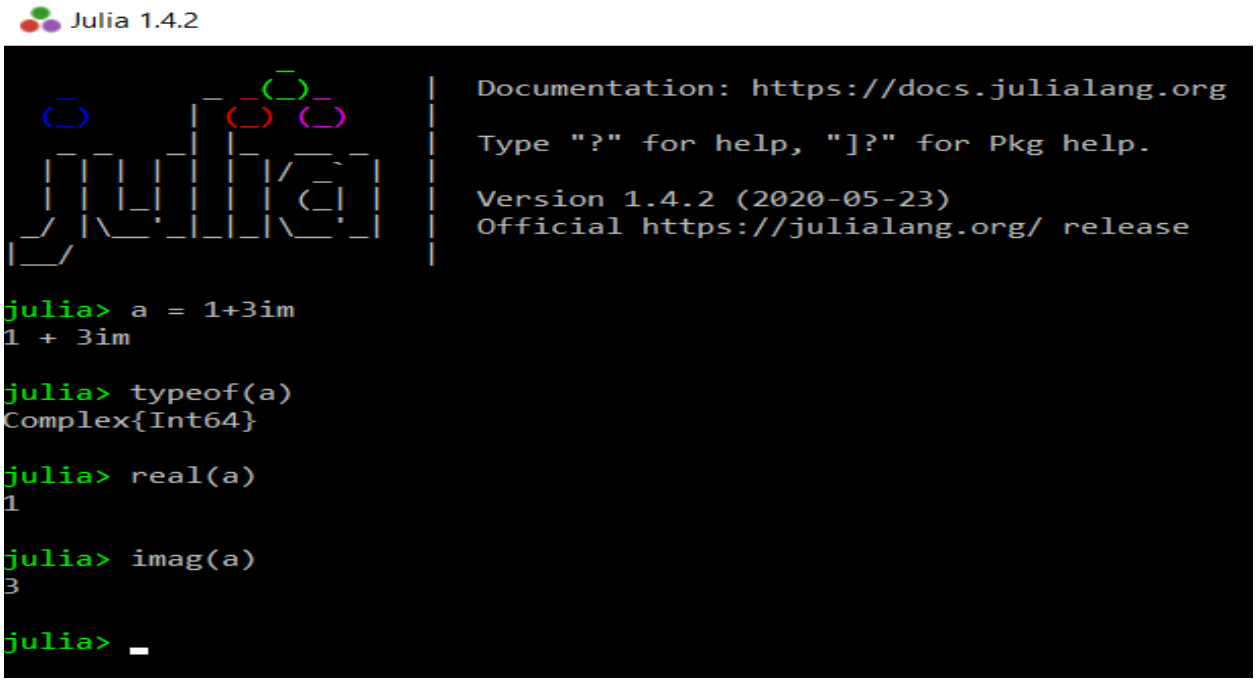
Complex and Rational Numbers:

Complex numbers are those numbers which can be expressed as a+bi, where a and b are real number and i is the imaginary number representing underoot of -1 or solution of $x^2 = -1$.

For this in Julia we assign variable as, a = 1+3im.

For getting real number out of this complex number, there is an inbuilt function (real(a)).

And for getting imaginary number out of this complex number, we can use imag function (imag(a)).



There are various methods which can be applied over these complex numbers:

1. abs
2. angle

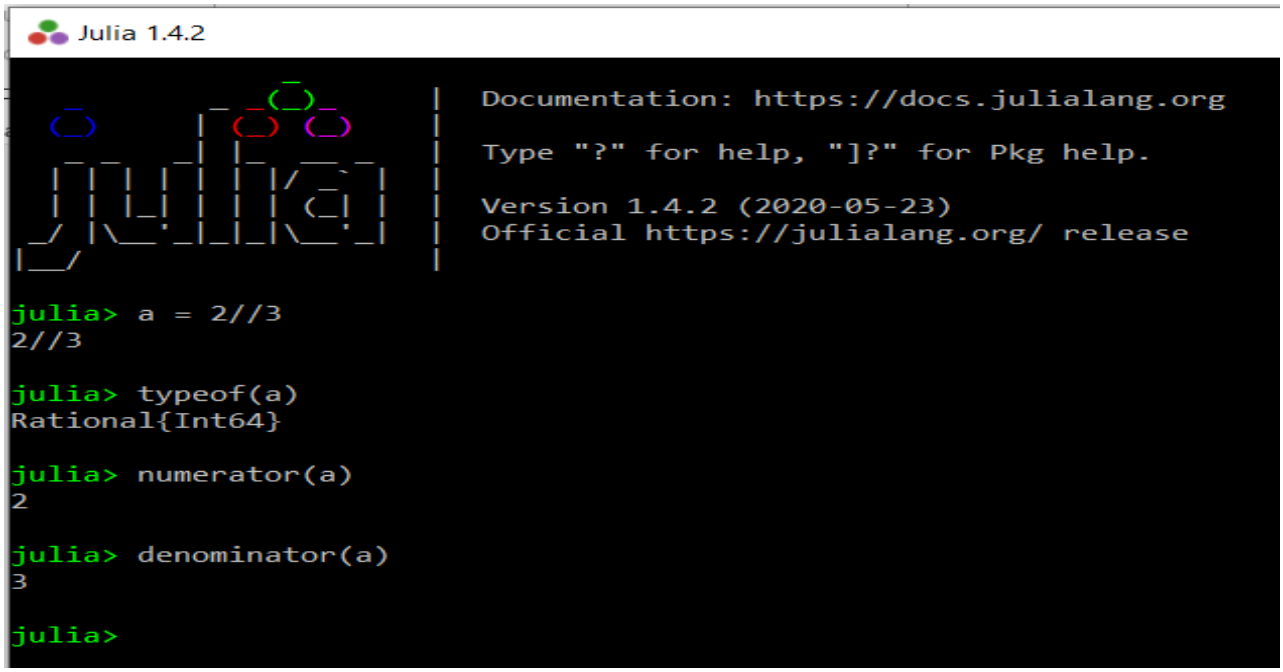
In Julia documentations you will be able to find all the functions available.

Rational Number:

A number that can be expressed as p/q where p and q are two integer number that number is known as rational number. (1/2, 3/5 etc.)

For declaring a rational number, we can write simply a = 2//3

For getting numerator and denominator from a rational number there are methods provided



String:

String is a sequence of character. (“Hello”, “Name”, etc.)

Declaring a String in Julia (test = “Hello World”) it is stored as an array in memory and the starting index is 1.

1	2	3	4	5	6	7	8	9	10
H	e	l	l	o		W	o	l	d

We can access a character at any index by typing the name of variable and then the index of that character. (test [1], test [2], test [5])

For concatenate two string we have to use string function we cannot concatenate them by + operator. (string (“First”,” Second”))

Julia 1.4.2

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.4.2 (2020-05-23)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> string("First ","Second")
"First Second"

julia> "First "+"Second"
ERROR: MethodError: no method matching +(::String, ::String)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:529
Stacktrace:
 [1] top-level scope at REPL[2]:1

julia>
```

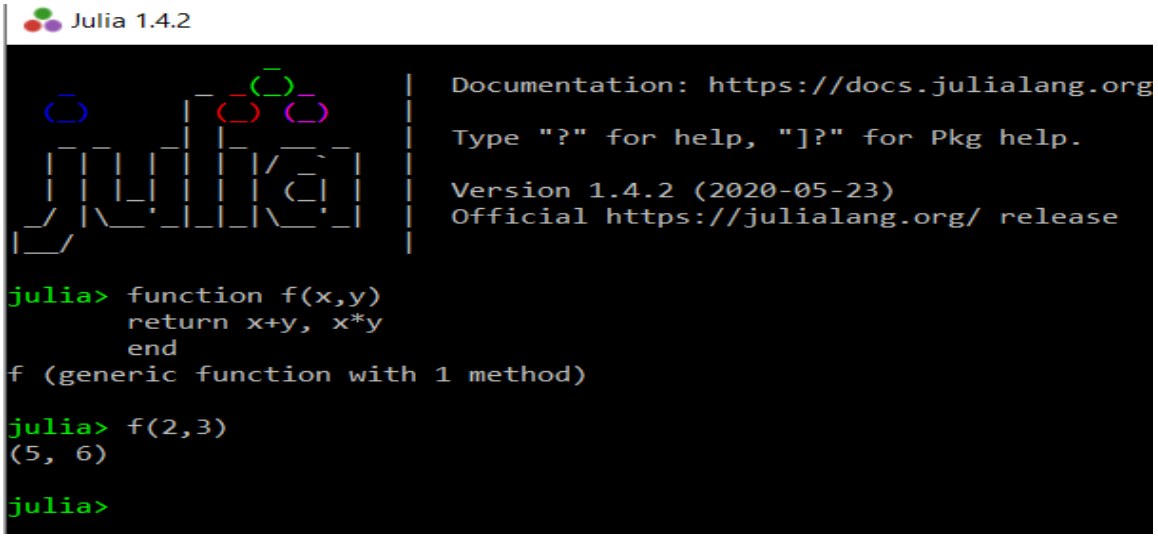
Functions:

In Julia the function always returns the last line of execution in that function if don’t return a value explicitly. But if we define a return statement that only that statement is returned.

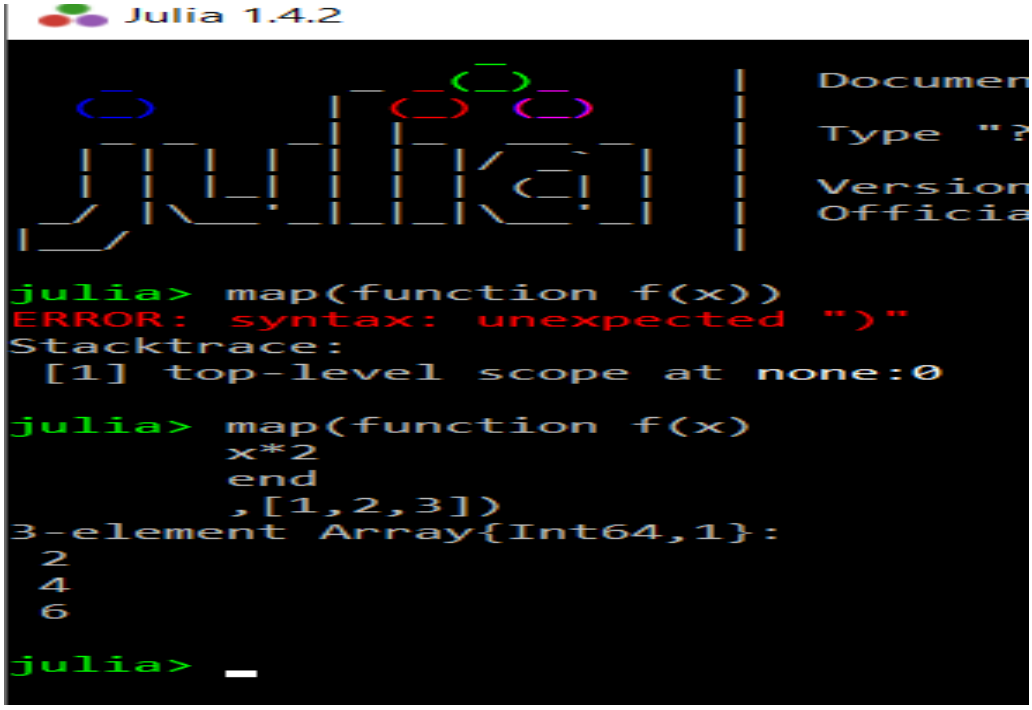
Julia 1.4.2

```
 _/ |\__'_|_|_|\__'_|   |  Official http
|__/                    |

julia> function f(x,y)
         return x+y
         end
f (generic function with 1 method)

julia> function f1(x,y)
         x+y
         x*y
         end
f1 (generic function with 1 method)

julia> function f2(x,y)
         return x+y
         x*y
         end
f2 (generic function with 1 method)

julia> f(2,3)
5

julia> f1(2,3)
6

julia> f2(2,3)
5

julia>
```

Even we can return multiple values in Julia



Map function in Julia, it takes two arguments as input 1st is a function and 2nd are an array. It then passes all the elements of that array into the function one by one and store the result into an array again.



Int this as we can see, each element in the list has been multiplies by 2.

There is another way we can do the same thing, which is more compact.

Optional Argument:

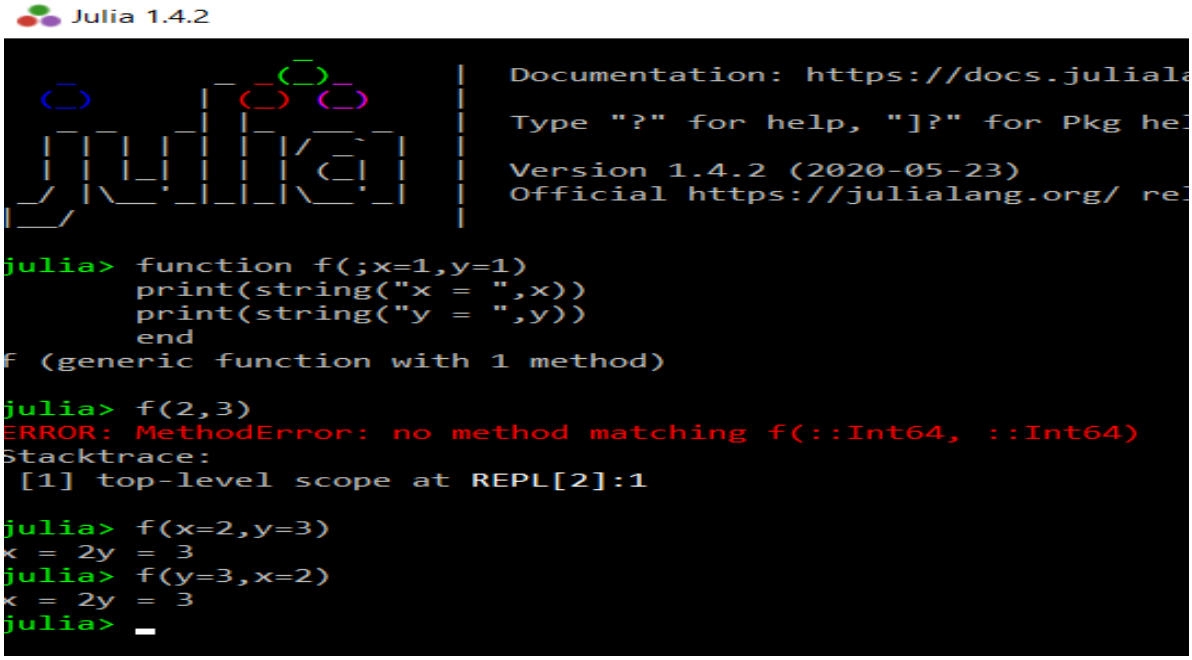
In this we can assign a default value to the variable in the function and if value for that variable is not provided than that value is taken.

Keyword argument:

In this we can assign a value to each input argument initially that the important condition and we have to provide a semicolon at the very beginning of the input arguments in keyword argument.

And by using this we can provide input arguments in any order we want to provide but name of argument must match with the one that we have provided the function.

But the very important thing to note here is that we can't call a function by just providing the values now. We always have to provide the name of the variable for which we are providing the value.



If statements:

For dealing with the conditions and running a piece of code only if that condition is meet, we use if statements.

There are two other things related to if statements, we can use else and elseif statements also with if statements. But the thing that we have to care off is that we can use else and elseif only with if statements we can't use them without If.

Having one condition we use if.

Having two conditions we use if and else.

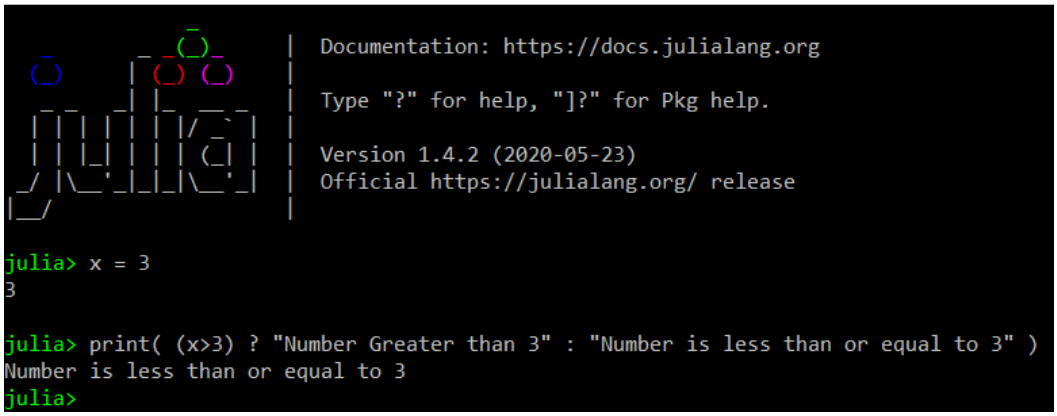
And when we have more that two condition we use if, elseif and else statements.

Julia 1.4.2

Ternary Operators:

These are used when we have one condition and we have to return only a single statement or to print a single statement.

Julia 1.4.2



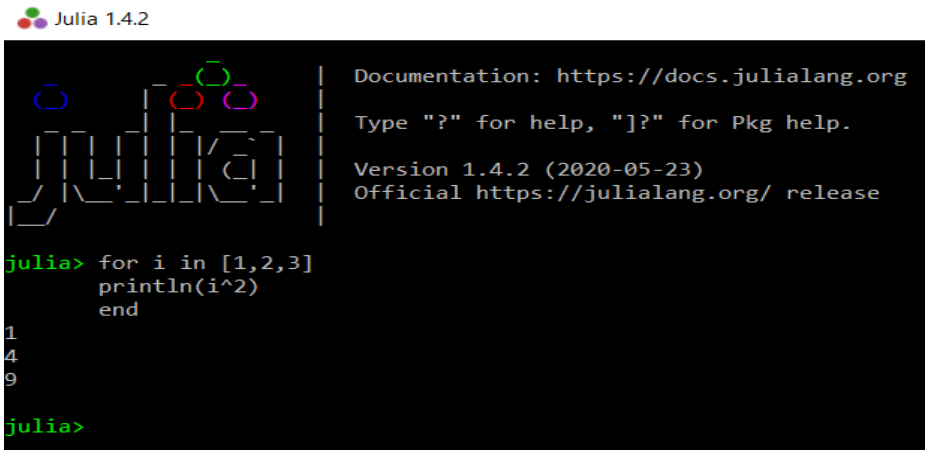
Loops in Julia:

1. for
2. while

We use for loop when we want to perform a same task again and again for a same value or for different values.

In for loop we can iterate through an array or a range.

Iterating through an array:



Iterating through a range:

While loop:

We use while loop when we do not know the ending condition in advance.

For example, if we are playing a game and game end's when one player is not having more cards to play the game. So, in this we do not know at what time one player will have no cards with him, that's why we use while condition in this case and continue to run that loop till one player is out of cards.

Program:

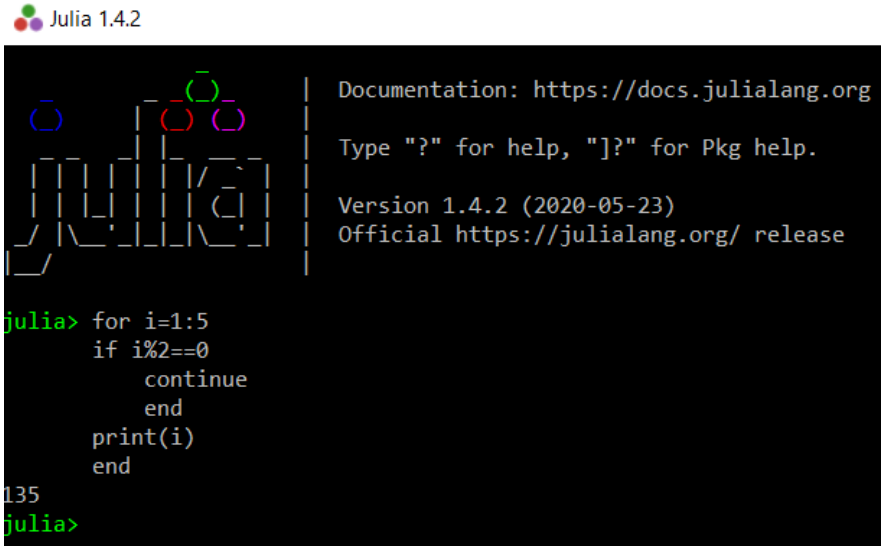
```
i = 1
while i<5
print(i)
i+=1
end
```

output:

1234

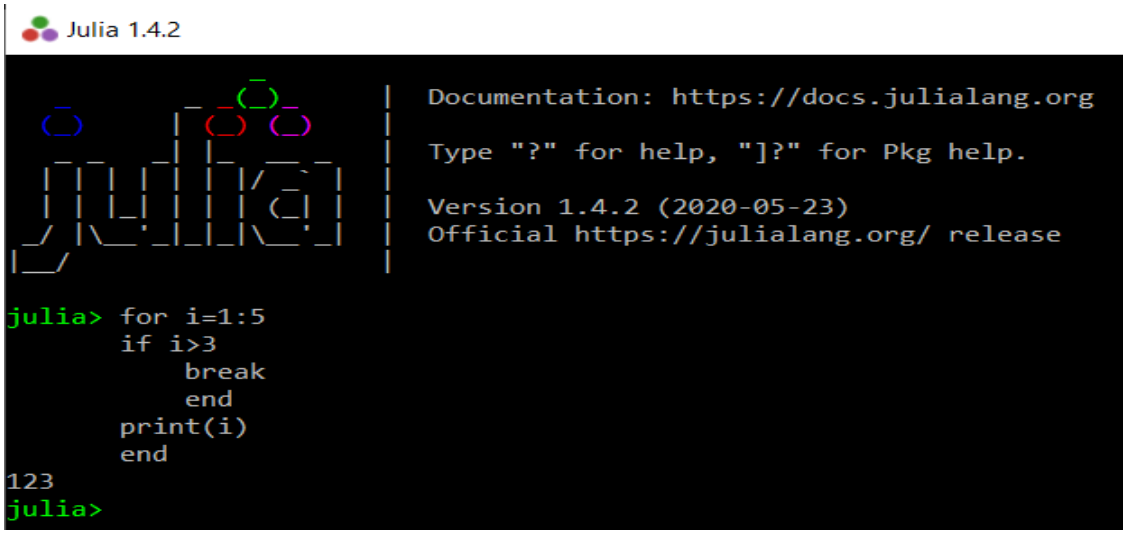
Continue:

It is a keyword used when we want to skip that iteration over condition.



Break:

We use break keyword, if we want to get out of a loop when a certain condition is satisfied.

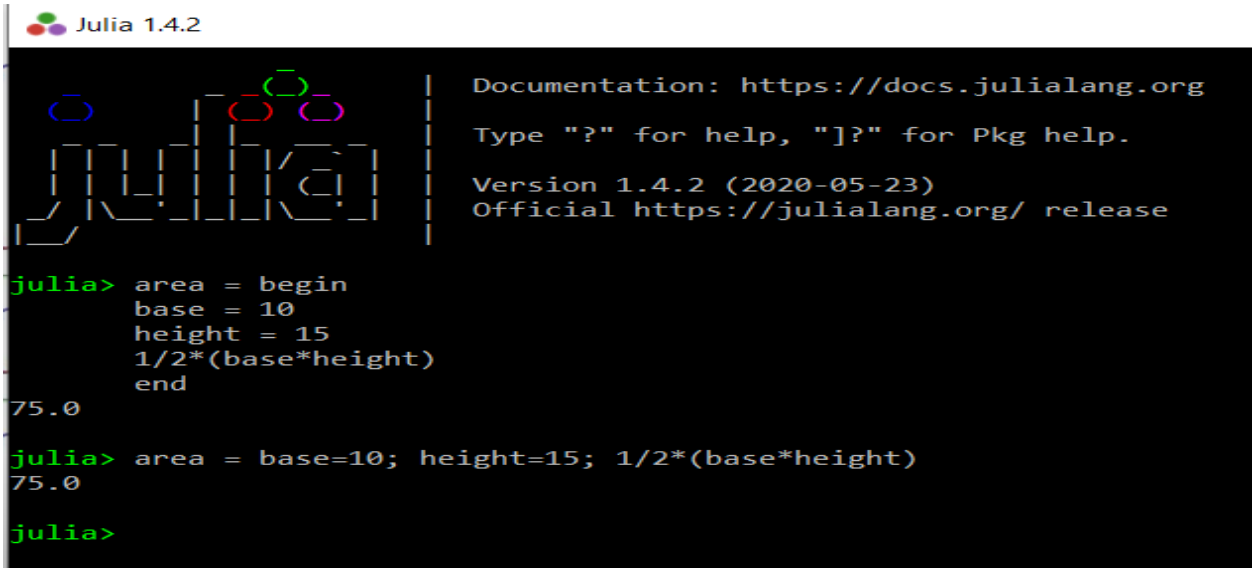


Compound Statements:

We use these when we want to perform a series of statements for finding something but we don't want to create a separate function for that. Then we use Compound statements for that.

We can write compound statements in two ways:

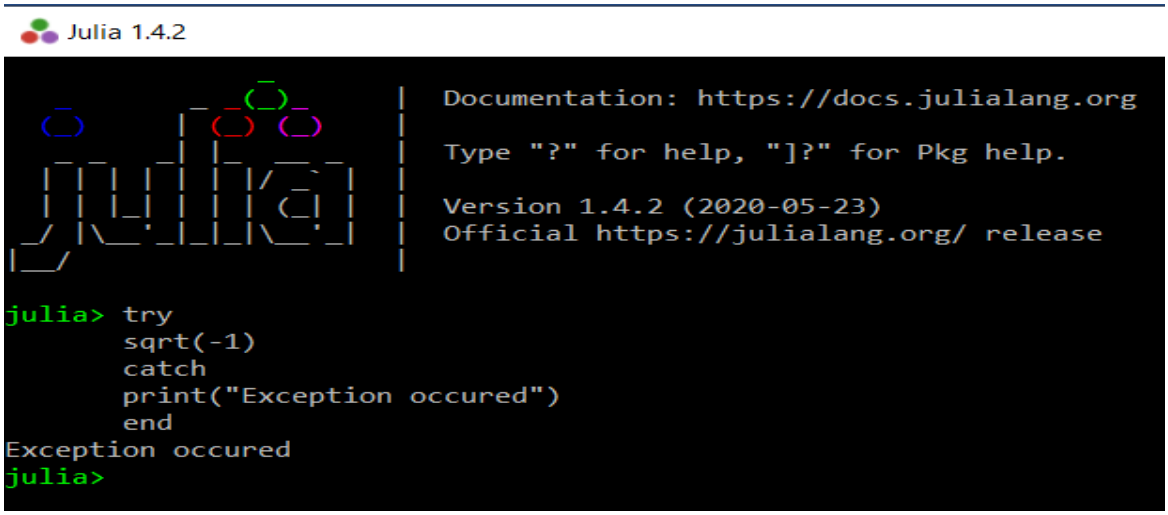
1. with begin
2. with semicolon



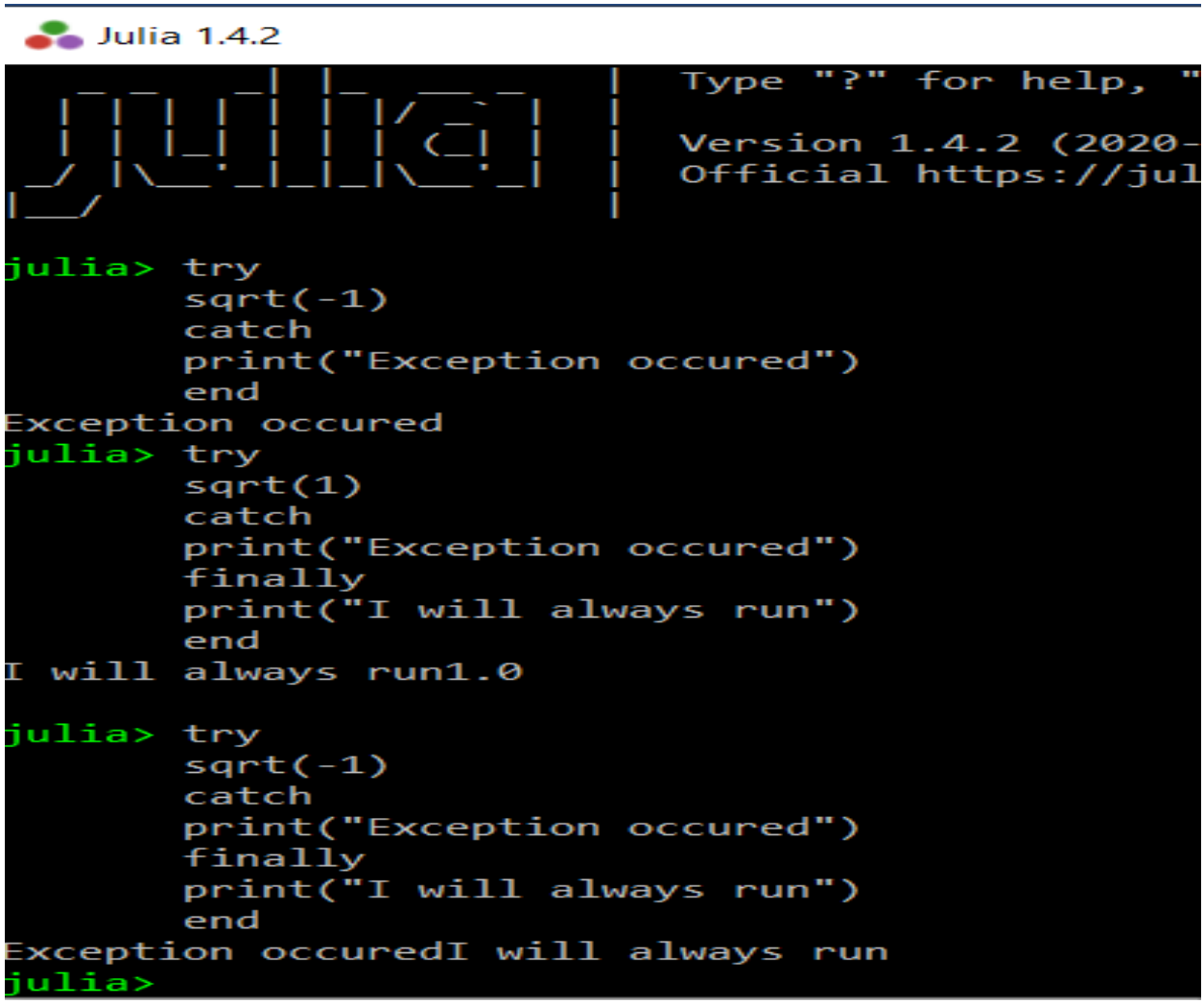
Exception Handling:

An **exception** is an event that occurs during the execution of a **program** that disrupts the normal flow of instructions.

To handle exceptions, we have try and catch blocks. In try block we write our risky code which can cause an exception. And if an exception is occurred during the execution of code than the control goes to the catch block.



We have one more keyword related to this and that is finally. Finally, is block of code that will always run whether an exception is occurred or not.


Julia 1.4.2
Type "?" for help, "
Version 1.4.2 (2020-
Official https://jul
julia> try
sqrt(-1)
catch
print("Exception occured")
end
Exception occured
julia> try
sqrt(1)
catch
print("Exception occured")
finally
print("I will always run")
end
I will always run1.0
julia> try
sqrt(-1)
catch
print("Exception occured")
finally
print("I will always run")
end
Exception occuredI will always run
julia>

Advantages of Julia:

1. It's blazingly fast right out of the box

Julia was designed from the beginning with high performance in mind without having to sacrifice ease of use like garbage collection, which is a common trade off in languages such as C++.

2. It solves the two-language problem

The industry is currently plagued by what's called the "two-language problem". Typically, developers first prototype their application using a slow dynamic language, and then rewrite it using a fast-static language for production.

3. It excels at technical computing

Designed with data science in mind, Julia excels at numerical computing with a syntax that is great for math, with support for many numeric data types, and providing parallelism out of the box, but more on that last bit later. Julia's multiple dispatch is a natural fit for defining number and array-like data types.

There are multiple symbols that can be written in Julia like:

- \Gamma Γ
- \mercury ☿
- \degree °
- \cdot ·
- \in ∈

4. It does parallel really well

Data has gotten so huge that it has become unpractical to run applications on a single computer. Nowadays everyone is computing big data on multiple nodes in a cluster in order to decrease the execution time by running tasks in parallel, but unfortunately many languages were never designed for this.

Disadvantages:

1. **New Language:**

The most apparent disadvantage of Julia compared to Python is that it is a relatively new language, so the developer community is small. A smaller developer community means that there are far fewer debugging tools for Julia than Python.

2. **Less Resources available:**

And the number of libraries and packages available for Julia is significantly lower than Python