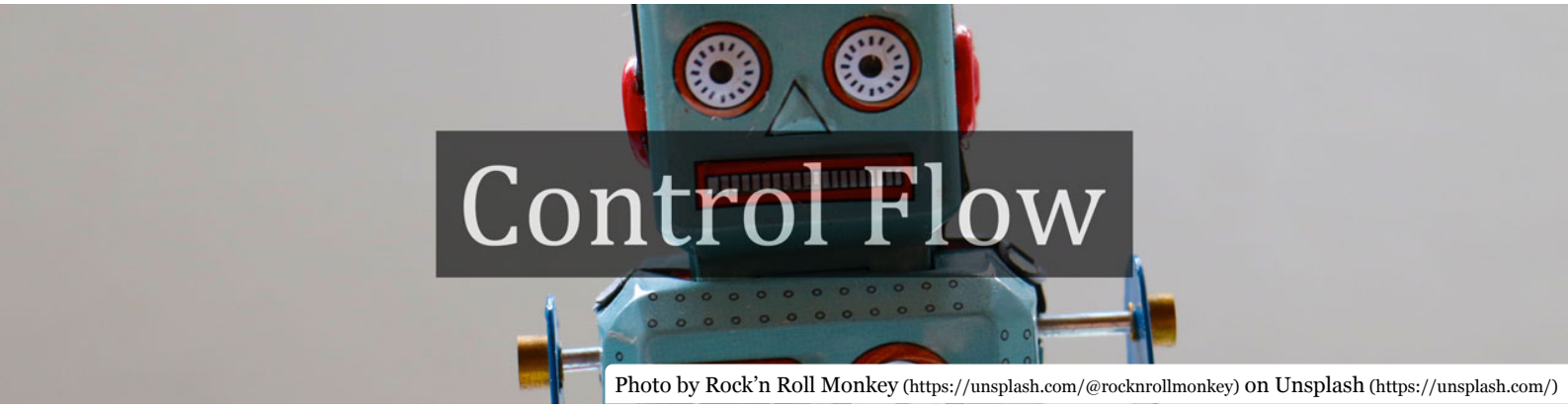

Photo by Rock'n Roll Monkey (https://unsplash.com/@rocknrollmonkey) on Unsplash (https://unsplash.com/)

Control Flow

5 minute read

In this lesson we will learn how to work with control statements. We will first learn how to use conditional blocks like `if ... else` blocks and then we will learn how to perform loops.

If … else

When a program needs to take decisions according to certain conditions, the `if ... else` block is the default choice.

Let's suppose that we want to write a simple implementation of the absolute value of a number. The absolute value of a number is defined as the number itself, if the number is positive, or the opposite if the value is negative. This is the typical case where the `if ... else` construct is useful! We can write a simple `absolute` function in this way:

```
function absolute(x)
    if x >= 0
        return x
    else
        return -x
    end
end
```

As you can see, an `if ... else` block is closed with the word `end`, like a function.

If we need to check more than one condition, we can bind together conditions using:

- "and" is written as `&` (if both statements are true return true, else return false)
- "or" is written as `||` (if at least one statement is true return true, else return false)

For example, if we want to check whether 3 is both minor than 4 and major than 1 we type:

```
if 1 < 3 & 3 < 4
    print("eureka!")
end
```

If we want to check if a value satisfies one of several different conditions, we can use the `elseif` statement: Julia will check if the first condition specified in the `if` is satisfied, if it is not met it moves on to the first `elseif` and so on.

```
x = 42
if x<1
    print("$x < 1")
elseif x < 3
    print("$x < 3")
elseif x < 100
    print("$x < 100")
else
    print("$x is really big!")
end

>>> 42 < 100
```

I take the occasion to introduce **string interpolation**. With the indication `$x` we tell Julia that it must substitute to `$x` the value of `x`, in this case 42. This is particularly useful when we want to print values, or we want to make custom messages:

```
name1 = "traveler"
name2 = "techytok"
print("Welcome $name1, this is $name2 :)")
```

Loops

A loop is the operation of repeating the same set of instructions several times. Loops are useful when we want to compute the value of a function over several points, we need to perform some operations on the elements of an array or we need to print the elements of a list.

For loops

Sometimes we want to iterate over a list of values and perform some operation on each element.

For example let's suppose we want to print all the squares of the numbers comprised between 1 and 10, we can do it using a `for` loop:

```
for i in 1:10
    println(i^2)
end
```

`i` is the variable which contains the data which gets updated at each new cycle (in this case `i` holds the numbers from 1 to 10 respectively), while `1:10` is a **range**. A range is an iterable object which does exactly what its name suggests: it specifies the range on which the loop has to be performed (in this case 1 to 10).

> It is also possible to use the alternative notation `for i = 1:10` which is completely equivalent.

Please notice that it is possible to loop not only over ranges (which can also be specified using the `range` (https://docs.julialang.org/en/v1/base/math/#Base.range) function) but also lists (i.e. arrays, tuples, etc).

For example, let's suppose we have a list of persons and we want to greet all of them, we can do it with the `for` statement:

```
persons = ["Alice", "Bob", "Carla", "Daniel"]

for person in persons
    println("Hello $person, welcome to techytok!")
end
```

Here instead of a range, we iterate over the elements of `persons` (i.e. the names of the persons that I want to greet) and in this case `person` will hold the name of a single person, which changes at each iteration step.

> For more informations on arrays and collection types, please refer to this lesson.

Break

In the case we want to forcefully interrupt a for loop we can use the `break` statement, for example:

```
for i in 1:100
    if i>10
        break
    else
            println(i^2)
    end
end
```

Here we check if `i>10` , in that case we break the loop and finish the iteration, else we print `i^2` .

Continue

This is the opposite of `break` , `continue` will forcefully skip the current iteration and move to the next cycle:

```
for i in 1:30
    if i % 3 == 0
        continue
    else
        println(i)
    end
end
```

This loop prints all the numbers from 1 to 30 except the multiples of 3.

While loop

When a loop needs to continue until a certain condition is met, a `while` loop is the preferable choice:

```
function while_test()
    i=0
    while(i<30)
        println(i)
        i += 1
    end
end
```

While blocks can access and change the values of variables in the scope of the block in which they are defined. For more info on variable scope, see this (https://techytok.com/lesson-variable-scope/) lesson.

Enumerate

`enumerate` is a function which comes in handy when we need to iterate on an array (or similar) and we need to keep track of the number of iterations we have already performed.

`enumerate` will return an iterator (which is something like an array which can be iterated in for loops). It will produce couples of the form `(i, x[i])`.

For example:

```
x = ["a","b","c"]
for couple in enumerate(x)
    println(couple)
end

(1, "a")
(2, "b")
(3, "c")
```

The same result could have been obtained “manually”:

```
x = ["a","b","c"]
enum_array = [(1,"a"), (2,"b"), (3,"c")]
for i in 1:length(x)
    println(enum_array[i])
end

(1, "a")
(2, "b")
(3, "c")
```

Let's say we want to read the elements from an array, square them and store them in another array, we can do it in this way:

```
my_array1 = collect(1:10)
my_array2 = zeros(10)
for (i, element) in enumerate(my_array1)
    my_array2[i] = element^2
end

>>>print(my_array2)
[1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
```

For comparison, we could have written the same loop in the following way:

```
my_array1 = collect(1:10)
my_array2 = zeros(10)
for i in 1:length(my_array1)
    my_array2[i] = my_array1[i]^2
end

>>>print(my_array2)
[1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
```

For more information on iterators and the enumerate function, please refer to this documentation page (https://docs.julialang.org/en/v1/base/iterators/index.html).

Conclusion

In this lesson we have learned how to let a program take "decisions" using `if ... elseif ... else` blocks, how to perform loops using `for` and `while` and how to have control on such loops using `break` and `continue`. We have then given an example of how `enumerate` can be used to help the process of filling an array.

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

Tags: Julia

Categories: Lessons Tutorial

Updated: November 19, 2019

LEAVE A COMMENT

ALSO ON TECHYTOK

Variable Scope

7 months ago • 4 comments

From zero to Julia Lesson 6. Variable scope

Multiprocessing in Julia: writing a module

9 months ago • 3 comments

A tutorial on how to write a module in Julia which uses multiprocessing

Parallel computin

7 months ago • 4 comm

From zero to Julia Le 15. Parallel computin

5 Comments techytok **Privacy Policy** 1 **Login**

Recommend Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH **OR SIGN UP WITH DISQUS** ?

Name

Joe Miller • a month ago • edited

Thank you for your tutorial, I really enjoy it doing it!
One thing which probably could be improved it something that confused me.

When you introduced **enumerate** you instantly jump to show how to implement it manually, but at this point I was rather interested how to access the index, which seems the whole purpose of enumerate.
I also don't see that you introduced the tuples at this point which you are using, maybe I missed something.

Also when the example comes after that, you use the functions **collect** and **zero**, which I don't understand yet, cause I'm still trying to figure out how to use enumerate. I probably would not change the point that you are trying to make. Just a few ideas and my impressions.
Thank you again!

Thank you again!

• Reply • Share ›

aureamerio Mod ➚ Joe Miller • a month ago • edited

Hello, I'll try to better elaborate on the subject, as soon as possible I'll rethink that section of the lesson and better explain it.
Enumerate takes an array or sequence (like a range) and produces an iterator which can be used in loops (sorry if this confuses you more, but you may want to look up what an iterator is). It yields subsequently a tuple containing the index of the element and the element itself. I didn't want to deal with what iterators are in this lesson, but apparently it would be useful to first explain what they are and then introduce enumerate as an example. Thank you for your feedback!

On the other hand, `zeros` is a function used to create an array full of zeros and `collect` is used to transform a range - for example 1:10 - into an array - [1,2,3,4,5,6,7,8,9,10]).

• Reply • Share ›

joe • 6 months ago

Could you explain a little bit more how this work, please?
for (i, element) in enumerate(my_array1)

• Reply • Share ›

aureamerio Mod ➚ joe • 6 months ago

Hello, I have updated the guide with an example to better understand how enumerate works, please tell me if it is now more clear and if you have other doubts!

2 • Reply • Share ›

joe ➚ aureamerio • 6 months ago

My problem wasn't with enumerate but with the "for" syntax: "for (i, element)".
Though I guess it means the first element of each couple returned by enumerate is assigned to "i" and the second

/