4d. For-Loops

Martin Alfaro
PhD in Economics

INTRODUCTION

A key feature of programming is its ability to automate repetitive tasks, making **for-loops** play a crucial role in coding. They let you execute the same block of code repeatedly, treating each element in a list as a different input.

While for-loops are fundamental in every programming language, their importance is especially pronounced in Julia: unlike other languages (e.g., Matlab, Python, and R), which often discourage for-loops in performance-critical code, Julia relies on them to achieve high performance.

The role of for-loops in optimizing performance will be explored in Part II. Here, we'll primarily introduce the tool itself, focusing on its syntax, constructions, and common iteration techniques.

SYNTAX

For-loops delimit their scope via the keywords `for` and `end`. To illustrate their syntax, consider the function `println(a)`, which evaluates `a` and displays its output in the REPL. In case `a` is a string, `println(a)` simply displays the word stored in `a`. The following script repeatedly applies `println` to display each word contained in a collection.

FOR-LOOPS SYNTAX

```
for x in ["hello","beautiful","world"]
    println(x)
end
```

```
"hello"
"beautiful"
"world"
```

> **Remark**
>
> The keyword `in` can be replaced by `∈` or `=`. [1] Consequently, the following constructions are all equivalent.
>
> **IN**
>
> ```
> for x in ["hello","beautiful","world"]
>     println(x)
> end
> ```

∈

```
for x ∈ ["hello","beautiful","world"]
    println(x)
end
```

=

```
for x = ["hello","beautiful","world"]
    println(x)
end
```

Furthermore, we can employ any character or term to describe the iteration variable. For instance, we iterate below using `word`.

ALTERNATIVE NAME FOR ITERATION VARIABLE

```
for word in ["hello","beautiful","world"]
    println(word)
end
```

```
"hello"
"beautiful"
"world"
```

Based on this example, we can identify three components that characterize a for-loop:

- **A code block to be executed**: represented in the example by `println(x)`.
- **A list of elements**: represented in the example by `["hello","beautiful","world"]`. This specifies the elements over which we'll apply the code block. The list can contain elements with any data type (e.g., strings, numbers, and even functions). The only requirement is that the list must be an **iterable object**, defined as a collection whose elements can be accessed individually. An example of iterable object is vectors, as in the example. However, we'll also introduce others most commonly used, such as ranges.
- **An iteration variable**: represented in the example by `x`. This serves as a label that takes on the value of each element in the list, one at a time, during each iteration. The iteration variable is a local variable, with no significance outside the for-loop. Its sole purpose is to provide a convenient way to access and manipulate the elements of the list within the loop.

In the following sections, we'll explore different objects that can serve as lists. Furthermore, we'll show that these lists can comprise elements not immediately obvious. A typical example is functions, making it possible to apply different functions to the same object.

Always Wrap For-Loops in Functions

At this stage of the website, we're still introducing fundamental concepts. Thus, we're presenting subjects in their simplest form for

> learning purposes. In particular, this explains why for-loops will be written in the global scope.
>
> However, **you should always wrap for-loops in functions**. Executing for-loops outside a function severely degrades performance, and is additionally subject to different rules regarding variable scoping. [2]

ITERATING OVER INDICES

So far, we've considered a simple list like `["hello", "beautiful", "world"]` to demonstrate how for-loops work. In real applications, however, manually specifying each element in a list is impractical. Fortunately, when a list follows a predictable pattern (e.g., a sequence of numbers), we can simply describe the pattern that generates those elements.

Building on this insight, we'll next explore how to define ranges. They let users define a sequence of numbers, which is particularly useful to access elements of a collection through their indices.

RANGES

Ranges in Julia are defined via the syntax `<begin>:<steps>:<end>`, where `<begin>` represents the starting index and `<end>` the ending index. Likewise, `<steps>` sets the increment between values, defaulting to one when the term is omitted. We can also reverse the order of the sequence, by providing a negative value for `<steps>`. All this is demonstrated below.

RANGE WITH STEPS GIVEN

```
for i in 1:2:5
    println(i)
end
```

```
1
3
5
```

RANGE WITH REVERSE ORDER

```
for i in 3:-1:1
    println(i)
end
```

```
3
2
1
```

> **Remark**
>
> The application of ranges isn't limited to for-loops. They can also define vectors when used in combination with the `collect` function.

CREATING A VECTOR FROM A RANGE

```
x = collect(4:6)
```

```
julia> x
3-element Vector{Int64}:
 4
 5
 6
```

ITERATING OVER INDICES OF AN ARRAY

Ranges can be employed to access elements of a collection. When combined with a for-loop, it makes it possible to apply the same code block to each element of a vector.

Specifically, the expression `1:length(x)`, where `length(x)` returns the number of elements in `x`, allows iteration over all indices of a vector `x`. The same functionality can be achieved with the function `eachindex(x)`. In fact, this is the recommended approach for iterating over all elements, as it returns an iterator optimized for each iterable object.

1:LENGTH(X)

```
x = [4, 6, 8]

for i in 1:length(x)
    println(x[i])
end
```

```
4
6
8
```

EACHINDEX

```
x = [4, 6, 8]

for i in eachindex(x)
    println(x[i])
end
```

```
4
6
8
```

Remark

There are other approaches to iterating over all indices of a vector `x`. For instance, you can use `LinearIndices(x)`, or `firstindex(x):lastindex(x)` to specify a range from the first to the last index of `x`.

This multiplicity of methods exists to handle non-standard indices, such as those provided by the `OffsetArrays.jl` package. This package sets the first index of arrays to 0, a common convention in many programming languages. Nevertheless, unless you're developing a package for other users, you don't need to worry about which approach to implement. Indeed, they can all be used interchangeably, as shown below.

EACHINDEX

```
x = [4, 6, 8]

for i in eachindex(x)
    println(x[i])
end
```

```
4
6
8
```

1:LENGTH(X)

```
x = [4, 6, 8]

for i in 1:length(x)
    println(x[i])
end
```

```
4
6
8
```

LINEARINDICES

```
x = [4, 6, 8]

for i in LinearIndices(x)
    println(x[i])
end
```

```
4
6
8
```

FIRSTINDEX(X):LASTINDEX(X)

```
x = [4, 6, 8]

for i in firstindex(x):lastindex(x)
    println(x[i])
end
```

```
4
6
8
```

> Among the available alternatives, `eachindex` is preferable because it automatically selects the most efficient method for each type of collection. Additionally, the syntax is consistent across all indexing conventions.

RULES FOR VARIABLE SCOPE IN FOR-LOOPS

Similar to functions, for-loops create a new variable scope. In fact, the scoping rules for both are similar, with one key difference: **for-loops can modify global variables, whereas functions cannot**.

> **Warning!**
>
> The general scoping rules presented here apply universally, except in rare edge cases that result from poor coding practices. Since this scenario is uncommon, we only outline it next.
>
> Basically, the issue occurs when *i)* the for-loop is *not* wrapped in a function, *ii)* a local variable shares the same name as a global variable, and *iii)* the script is run non-interactively (i.e., using the function `include` and a script file). [3]
>
> Unless the three conditions hold simultaneously, you don't have to worry about this scenario. And even if this occurs, Julia will display a warning in the REPL indicating that there's a problem with your code.

To formalize the variable scope of for-loops, we'll refer to a variable `x`. The rules governing its scope are:

- the variable of iteration `x` is always local, regardless of whether there's a variable `x` defined outside the for-loop.
- if there's no variable named `x` outside the for-loop, `x` is a new local variable. Moreover, this variable won't be accessible outside the for-loop.
- if there's a variable named `x` outside the for-loop, `x` refers to this variable.

The following code snippets illustrate the first two rules, which exclusively refer to local variables. The second example is particularly noteworthy, as it highlights **a common mistake made by beginners**: running a for-loop that defines a local variable, and then trying to access it outside the for-loop.

ITERATION VARIABLE IS LOCAL

```
x = 2

for x in ["hello"]          # this 'x' is local, not related to 'x = 2'
    println(x)
end
```

```
"hello"
```

DEFINING LOCAL VARIABLE

```
#no `x` outside the for-loop

for word in ["hello"]
    x = word                # `x` is local to the for-loop, not available outside it
end
```

```
julia> x
ERROR: UndefVarError: x not defined
```

Likewise, the following example demonstrates the consequences of the last rule we mentioned. This refers to the consequences of variable scope for global variables.

REFERRING TO THE GLOBAL X

```
x = [2, 4, 6]

for i in eachindex(x)
    x[i] * 10        # it refers to the `x` outside of the for-loop
end
```

```
julia> x
3-element Vector{Int64}:
 20
 40
 60
```

REASSIGNING THE GLOBAL X

```
x = [2, 4, 6]

for word in ["hello"]
    x = word                        # it reassigns the `x` defined outside the for-loop
end
```

```
julia> x
"hello"
```

ARRAY COMPREHENSIONS

To seamlessly create arrays via for-loops, you can use **array comprehensions**. Their syntax is `[<expression> for... if...]`, where `<expression>` denotes either an operation or a function.

For illustration purposes, consider a vector `x`. Suppose that the goal is to create a vector `y` with elements equal to the square of the corresponding element in `x`. The following code snippets show two approaches to creating `y` via array comprehensions.

COMPREHENSION USING AN OPERATION

```
x      = [1,2,3]


y      = [a^2 for a in x]              # or y = [x[i]^2 for i in eachindex(x)]
```

```
julia> y
3-element Vector{Int64}:
  1
  4
  9
```

COMPREHENSION USING A FUNCTION

```
x      = [1,2,3]

foo(a) = a^2
y      = [foo(a) for a in x]           # or y = [foo(x[i]) for i in eachindex(x)]
```

```
julia> y
3-element Vector{Int64}:
  1
  4
  9
```

Array comprehensions also allow for creating vectors based on conditions. In such instances, the condition must be placed at the end of the expression.

COMPREHENSION WITH CONDITION

```
x = [i for i in 1:4 if i ≤ 3]
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

> **Remark**
>
> Array comprehensions can also create matrices. Its syntax demands a comma to separate the description of each dimension.

COMPREHENSION FOR MATRICES

```
y = [i * j for i in 1:2, j in 1:2]
```

```
julia> y
2×2 Matrix{Int64}:
 1  2
 2  4
```

ITERATING OVER MULTIPLE OBJECTS

Thus far, we've considered for-loops that iterate over single values. We now extend their application to **simultaneous iterations over multiple values**. Specifically, we'll examine two scenarios: simultaneous iterations over two lists and over both the indices and values of a vector.

ITERATING OVER TWO LISTS

Depending on how elements should be combined, we can define two approaches to simultaneously iterating over two lists `x` and `y`. First, the function `Iterators.product(x,y)` makes it possible to iterate over all the possible combinations of elements. This function is part of the package `Iterators`, imported by default in each Julia session.

Alternatively, you can iterate over all the ordered pairs of `x` and `y`. This is implemented through the function `zip(x,y)`, which provides the pair of *i*-th elements from `x` and `y` in the *i*-th iteration.

MULTIPLE ITERATORS (ALL COMBINATIONS)

```
list1 = [1, 2]
list2 = [3, 4]

for (a,b) in Iterators.product(list1,list2) #it takes all possible combinations
    println([a,b])
end
```

```
[1,3]
[2,3]
[1,4]
[2,4]
```

MULTIPLE ITERATORS (PAIRS)

```
list1 = [1, 2]
list2 = [3, 4]

for (a,b) in zip(list1,list2)              #it takes pairs of elements with the same index
    println([a,b])
end
```

```
[1,3]
[2,4]
```

Using `zip`, we can also iterate over multiple values via array comprehensions.

MULTIPLE ITERATORS (ALL COMBINATIONS)

```
x = [i * j for i in 1:2 for j in 1:2]
```

```
julia> x
4-element Vector{Int64}:
 1
 2
 2
 4
```

MULTIPLE ITERATORS (PAIRS)

```
x = [i * j for (i,j) in zip(1:2, 1:2)]
```

```
julia> x
2-element Vector{Int64}:
 1
 4
```

SIMULTANEOUSLY ITERATING OVER INDICES AND VALUES

To iterate over each pair of index-value of a vector, we can employ the `enumerate` function.

FOR-LOOPS

```
x = ["hello", "world"]

for (index,value) in enumerate(x)
    println("$index $value")
end
```

```
"1 hello"
"2 world"
```

ARRAY COMPREHENSION

```
x = [10, 20]


y = [index * value for (index,value) in enumerate(x)]
```

```
julia> y
2-element Vector{Int64}:
 10
 40
```

ITERATING OVER FUNCTIONS

Functions in Julia are **first-class objects**, also referred to as **first-class citizens**. This means that functions can be manipulated just like any other data type, such as strings and numbers. In particular, this property makes it possible to store functions in a vector and apply them sequentially to an object. The following example illustrates this by computing descriptive statistics of a vector `x`.

ITERATION OVER FUNCTIONS

```
x                       = [10, 50, 100]
list_functions          = [maximum, minimum]

descriptive(vector,list) = [foo(vector) for foo in list]
```

```
julia> descriptive(x, list_functions)
4-element Vector{Real}:
 100
  10
```

FOOTNOTES

[1] Recall that `∈` can be written through tab completion using the command `\in`.

[2] In fact, older versions of Julia were restricting the use of for-loops in the global scope.

[3] There are two methods to execute a script. The first method is what we've been using so far, where you work interactively with Julia. This includes running commands in the REPL's prompt `julia>` and the execution of a script through a code editor. The second method consists of executing files that store scripts through the function `include`.