4d. For-Loops

Martin Alfaro
PhD in Economics

INTRODUCTION

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

In Julia, for-loops play an even more prominent role than in many other high-level languages, due to its role in high performance. Environments such as Matlab, Python, and R often encourage programmers to avoid explicit loops in performance-critical code, favoring vectorized operations or specialized library calls instead. Julia takes a different approach: well-written loops aren't only idiomatic but also fast, often matching or surpassing the performance of vectorized alternatives. As a result, mastering for-loops isn't just a matter of convenience. Rather, it's essential for writing clear and efficient Julia programs.

Part II of this book will examine how for-loops contribute to high performance. For now, our focus is on understanding the construct itself: its syntax, its variations, and the most common patterns for iterating over data.

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-LOOP SYNTAX

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

```
hello
beautiful
world
```

> **Remark**

> The keyword `in` can be replaced by `∈` or `=`, where `∈` can be written through tab completion using the command `\in`. Considering this, 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)`, which shows the value of `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 that are 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 collections that are iterable and therefore 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. [1]

ITERATION 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 collection 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 be used to define vectors by combining them with the `collect` function.
>
> **CREATING A VECTOR FROM A RANGE**
>
> ```
> x = collect(4:6)
> ```
>
> ```
> 3-element Vector{Int64}:
> 4
> 5
> 6
> ```

ITERATING OVER INDICES OF AN ARRAY

Ranges provide a straightforward mechanism for traversing the elements of a collection. When used in combination with a for-loop, they let you access each element of a vector by its index. There are several ways to construct such ranges.

A straightforward method is to write `1:length(x)`, which generates all valid indices for the vector `x`. Since `length(x)` returns the number of elements, this range covers every position from the first to the last. While this approach works, it relies on the assumption of linear indexing starting at 1, which isn't always guaranteed. As a result, it can be fragile when applied to collections with different indexing schemes.

A more robust practice is to use `eachindex(x)`. This function produces an iterator that's optimized to the specific structure of the collection. Furthermore, it ensures that you're iterating over the correct set of indices, regardless of the underlying data type. This makes your code more general, more efficient, and less error-prone, especially when working with other iterable objects that may not use standard indexing.

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
```

Julia provides other methods to iterate over all indices of a collection. Two constructions worth mentioning are `LinearIndices(x)` and `firstindex(x):lastindex(x)`. Just like the previous methods defined, they specify a range from the first to the last index of `x`.

All these approaches can 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
```

The multiplicity of methods to implement the same functionality is necessary to handle non-standard indices. For instance, the `OffsetArrays.jl` package sets the first index of arrays to 0, a common convention in many programming languages. When this package is loaded, using `1:length(x)` would break portability, while `firstindex(x):lastindex(x)` adapts seamlessly to the modified indexing scheme.

Unless you're developing a package for other users, you don't need to worry about which approach to implement.

RULES FOR VARIABLE SCOPE IN FOR-LOOPS

Like functions, for-loops introduce a new variable scope. The rules governing these scopes are largely the same, with one important distinction: **for-loops are allowed to modify global variables, whereas functions aren't**.

> **Warning!**
>
> The scoping rules presented for for-loops apply except in some edge cases, which primarily result from discouraged coding practices. Since these scenarios aren't widespread, we briefly outline them next but won't focus on them further.
>
> The issue appears only when three conditions occur simultaneously: *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). [2]

> Unless all three conditions are met, you don't need to worry about this scenario. And even if it does occur, Julia will issue a warning in the REPL, alerting you that the code may not behave as intended.

To make the rules of variable scope in for-loops precise, let's consider a variable named `x`. Its behavior is governed by the following principles:

- if `x` is the variable of iteration, then `x` is always local to the for-loop. This holds regardless of whether there's a variable `x` defined outside the for-loop.
- if there's a variable `x` defined within the for-loop body, this variable is local and won't be accessible outside the for-loop.
- if there's a variable named `x` outside the for-loop and no `x` is defined within the for-loop body, `x` refers to the global variable. Moreover, the for-loop can reassign or mutate the value of `x`.

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 beginner mistake**: defining a variable inside a for loop and then attempting to use it outside the loop, only to discover that it's no longer accessible.

ITERATION VARIABLE IS LOCAL

```
x = 2

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

```
hello
```

NEW VARIABLES ARE LOCAL

```
#no `x` defined 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 third rule. This affects the treatment of global variables.

REFERENCE TO GLOBAL X

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

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

```
julia> x
3-element Vector{Int64}:
 2
 4
 6
```

REASSIGNING GLOBAL X

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

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

```
julia> x
3-element Vector{Int64}:
 2
 4
 6
```

ARRAY COMPREHENSIONS

Julia provides **array comprehensions** as a concise and expressive way to generate new arrays via for-loops. The general syntax is `[<expression> for... if...]`, where `<expression>` denotes either an operation or a function.

To illustrate, suppose we want to define a new vector `y`, whose elements are the squares of the corresponding entries in `x`. Array comprehensions make this task straightforward. The following code snippets demonstrate the approach by using a direct expression and by calling a function.

ARRAY COMPREHENSION WITH OPERATION

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


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

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

ARRAY COMPREHENSION WITH FUNCTION CALL

```
x      = [1,2,3]
foo(a) = a^2

y      = [foo(a) for a in x]
z      = [foo(x[i]) for i in eachindex(x)]
```

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

Array comprehensions can also incorporate conditions, allowing you to filter elements as they're generated. In such cases, the condition must be placed at the end of the comprehension.

ARRAY COMPREHENSION WITH CONDITION

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

y = [a for a in x if a ≤ 2]
z = [x[i] for i in eachindex(x) if x[i] ≤ 2]
```

```
julia> y
2-element Vector{Int64}:
 1
 2
julia> z
2-element Vector{Int64}:
 1
 2
```

> **Array Comprehensions for Generating Matrices**
>
> Array comprehensions can also be used to construct matrices. In this case, the syntax requires a comma to separate the iteration variables for each dimension.
>
> **ARRAY COMPREHENSION FOR MATRICES**
>
> ```
> y = [i * j for i in 1:2, j in 1:2]
> ```
>
> ```
> julia> x
> 2×2 Matrix{Int64}:
>  1  2
>  2  4
> ```

ITERATING OVER MULTIPLE OBJECTS

For-loops can handle more than just single-value iterations. They also support **simultaneous iteration over multiple values**.

We'll focus on two common scenarios: iterating over two iterable collections at the same time, and iterating over both the indices and values of a single collection. In each case, we'll explore how to implement the iteration using both plain for-loops and array comprehensions, highlighting the differences in syntax and use cases.

ITERATING OVER TWO COLLECTIONS

Consider two iterable objects `x` and `y`. There are two main ways to combine their elements during iteration, depending on the desired outcome.

First, the function `Iterators.product(x,y)` generates all the possible combinations of elements from `x` and `y`. Thus, it produces a pair `(x[i], y[j])`, where `i` and `j` aren't necessarily equal. The `Iterators.product` function is part of the package `Iterators`, imported by default in each Julia session.

Alternatively, it's possible to 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. Thus, compared to `Iterators.product`, it restricts the iterations `(x[i], y[i])`.

MULTIPLE ITERATORS (ALL COMBINATIONS)

```
list1 = ["A","B"]
list2 = [ 1 , 2 ]

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

```
("A", 1)
("B", 1)
("A", 2)
("B", 2)
```

MULTIPLE ITERATORS (PAIRS)

```
list1 = ["A","B"]
list2 = [ 1 , 2 ]

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

```
("A", 1)
("B", 2)
```

We can also iterate over multiple values to generate vectors via array comprehension. While `zip` can be employed to create a new vector by pairing elements, `Iterators.product` would return a matrix. Instead, to construct a vector that exhausts all combinations between two iterable collections, Julia offers a special syntax: simply repeat the `for` keyword to specify the iteration range of each variable.

MULTIPLE ITERATORS (ALL COMBINATIONS)

```
list1 = ["A","B"]
list2 = [ 1 , 2 ]

x     = [(i,j) for i in list1 for j in list2]
```

```
julia> x
4-element Vector{Tuple{String, Int64}}:
 ("A", 1)
 ("A", 2)
 ("B", 1)
 ("B", 2)
```

MULTIPLE ITERATORS (PAIRS)

```
list1 = ["A","B"]
list2 = [ 1 , 2 ]

x     = [(i,j) for (i,j) in zip(list1,list2)]
```

```
julia> x
2-element Vector{Tuple{String, Int64}}:
 ("A", 1)
 ("B", 2)
```

Note that the `for` clauses must be written sequentially, without commas. A comma changes the comprehension's behavior, instructing Julia to build a matrix instead of a vector, as demonstrated before.

SIMULTANEOUSLY ITERATING OVER INDICES AND VALUES

When iterating over a collection, there are scenarios where you need access to both the value and index of an element, rather than handling each separately. For example, when analyzing data, you may want to flag outliers not only by their magnitude, but by their position in the dataset.

Julia provides a direct way to achieve this through the `enumerate` function. This transforms a collection into an iterator that yields pairs of indices and values, thus providing access to both pieces of information during each iteration.

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)]
```

```
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> x
2-element Vector{Int64}:
 100
  10
```

FOOTNOTES

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

[2] There are two methods to execute a script. The first method is the one used thus far, where we 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 containing scripts through the function `include`.