5h. In-Place Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

This section continues exploring **approaches to mutating vectors**. The emphasis is now on **in-place functions**, defined as functions that mutate at least one of their arguments.

Many built-in functions in Julia have an in-place counterpart, which can easily recognized by the `!` suffix in their names. These versions store the output in one of the function arguments, thereby avoiding the creation of a new object. In practice, it means that variables can be updated immediately after executing the function. For example, given a vector `x`, the call `sort(x)` produces a new vector with ordered elements, but without altering the original `x`. In contrast, the in-place version `sort!(x)` overwrites the content of `x`.

The benefits of in-place functions will become evident in Part II, when discussing high-performance computing. Essentially, by reusing existing objects, in-place functions eliminate the overhead associated with creating new objects.

IN-PLACE FUNCTIONS

In-place functions, also known as **mutating functions**, are characterized by their ability to modify at least one of their arguments. For example, given a vector `x`, the following function `foo(x)` constitutes an example of in-place function, as it modifies the content of `x`.

```
y = [0,0]

function foo(x)
    x[1] = 1
end
```

```
julia> y
2-element Vector{Int64}:
 0
 0
julia> foo(y) #it mutates 'y'
julia> y
2-element Vector{Int64}:
 1
 0
```

> **Functions Can't Reassign Variables**
>
> While functions are capable of mutating values, they **can't reassign variables defined outside their scope**. Any attempt to redefine such variable will be interpreted as the creation of a new local variable. [1]
>
> The following code illustrates this behavior by redefining a function argument and a global variable. The output reflects that `foo` in each example treats the redefined `x` as a new local variable, only existing within `foo`'s scope.

```
x = 2

function foo(x)
    x = 3
end
```

```
julia> x
2
julia> foo(x)
julia> x #functions can't redefine global variables, only mutate them
2
```

```
x = [1,2]

function foo()
    x = [0,0]
end
```

```
julia> x
2-element Vector{Int64}:
 1
 2
julia> foo()
julia> x #functions can't redefine variables globally, only mutate them
2-element Vector{Int64}:
 1
 2
```

BUILT-IN IN-PLACE FUNCTIONS

In Julia, many built-in functions that operate on vectors are available in two forms: a standard version that returns a new object and an in-place version that mutates its argument. To distinguish them, Julia's developers follow the naming convention that **any function ending with `!` corresponds to an in-place function**.

> **Appending ! To A Function Has No Impact on the Code**
>
> **Appending `!` to a function doesn't change the function's behavior**. It simply signals to users that the function performs a mutating operation. The goal is to make side effects explicit, helping programmers avoid unintended modifications of objects.

An example is given by the functions `sort` and `sort!`. Both arrange the elements of a vector in ascending order, with the option `rev=true` implementing a descending order. In its standard form, `sort(x)` creates a new vector containing `x`'s elements sorted, but leaving the original vector `x` unchanged. In contrast, the in-place version `sort!(x)` directly updates the original vector `x`, overwriting its contents with the sorted values. Both functions perform the same conceptual task, but they differ in whether they allocate new memory or reuse existing storage.

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

output = sort(x)
```

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

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

sort!(x)
```

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

It's also common to have in-place functions accepting an argument that isn't used as input to the computation, but instead serves purely as a destination for the output. Such design makes it possible to provide preallocated storage, avoiding the need to allocate a new array each time the function runs. The approach becomes especially valuable when an intermediate operation must be performed repeatedly and its output doesn't be to be preserved.

For instance, `map(foo, x)` applies the function `foo` to each element of `x` and returns a freshly allocated vector. Instead, `map!(foo, output, x)` directly writes the results into the preexisting vector `output`.

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


output = map(a -> a^2, x)
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
julia> output
3-element Vector{Int64}:
 1
 4
 9
```

```
x      = [1, 2, 3]
output = similar(x)             # we initialize `output`

map!(a -> a^2, output, x)       # we update `output`
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
julia> output
3-element Vector{Int64}:
 1
 4
 9
```

FOR-LOOP MUTATION VIA IN-PLACE FUNCTION

Any performance-critical code in Julia must be wrapped in functions. This not only prevents issues with variable scope, but is also key for performance as we'll discuss in Part II. In addition, for-loops often provide the most direct path to high performance in Julia. In this context, the ability of functions to mutate their arguments becomes crucial: it enables the application of for-loops while reusing existing storage, rather than allocating new arrays repeatedly.

A typical strategy to implement these operations is to initialize vectors with `undef` values, pass them to a function, and fill them via a for-loop. The examples below illustrate this approach.

```
x = [3,4,5]

function foo!(x)
    for i in 1:2
        x[i] = 0
    end
end
```

```
julia> foo!(x)

julia> x
3-element Vector{Int64}:
 0
 0
 5
```

```
x = Vector{Int64}(undef, 3)          # initialize a vector with 3 elements

function foo!(x)
    for i in eachindex(x)
        x[i] = 0
    end
end
```

```
julia> foo!(x)

julia> x
3-element Vector{Int64}:
 0
 0
 0
```

FOOTNOTES

[1] Strictly speaking, it's possible to reassign a variable by using the `global` keyword. However, its use is typically discouraged, explaining why we won't cover it.