5h. In-Place Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

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

Many built-in functions in Julia have a corresponding in-place counterpart. These versions can be easily identified by the symbol `!` appended to their names. In-place functions enable users to store the output in one of the function's arguments, thereby avoiding the creation of a new object. They can also be used to update the values of a variable directly. For example, given a vector `x`, `sort(x)` returns a new vector with ordered elements, but without altering the original `x`. In contrast, the in-place version `sort!(x)` directly stores the result within `x` itself.

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 variables globally, 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 take vectors as arguments are available in two forms: a "standard" version and an in-place version. To distinguish between them, Julia's developers follow a convention that **any function ending with `!` corresponds to an in-place function**.

Appending ! To A Function Has No Impact at All

> **Appending `!` to a function doesn't change the function's behavior**. It's simply a convention adopted by Julia's developers to emphasize the mutable nature of the operation. Its purpose is to alert users about the potential side effects of the function, thus preventing unintended modifications of objects.

To illustrate these forms, let's start considering single-argument functions. In particular, we'll focus on `sort`. This arranges 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 the sorted elements, leaving the original vector `x` unchanged. In contrast, the in-place version `sort!(x)` updates the original vector `x` directly, overwriting its values with the sorted result.

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

y = sort(x)
```

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

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

sort!(x)
```

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

Regarding multiple-argument functions, it's common to include an argument whose sole purpose is to store outputs. For instance, given a function `foo` and a vector `x`, the built-in function `map(foo, x)` has an in-place version `map!(foo, <output vector>, x)`.

```
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
```

MUTATIONS VIA FOR-LOOPS

Recall that for-loops in Julia should always 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 this context, the ability of functions to mutate their arguments is crucial. It determines that we can initialize vectors with `undef` values, pass them to a function, and fill them through a function via for-loops. The examples below illustrate this application.

```
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{Float64}(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.