5g. In-Place Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

This section focuses on **in-place operations**, where the contents of an existing object are directly modified. Unlike operations that generate new objects, in-place operations are characterized by the reuse of existing objects, giving rise to the expression *modifying values in place*.

Distinguishing between mutations and the creation of new copies is essential. If an operation mutates an object, any other variable that references the same object will also reflect the change. This can be intended if you seek to update data, but it can introduce subtle bugs if you expected the original to remain intact. In-place modifications are also relevant for performance, as they tend to reduce the memory overhead introduced when new objects are created. This aspect will be explored in Part II of the website.

At the heart of in-place operations is the concept of a slice introduced before. Before proceeding, I recommend reviewing that section before moving forward.

Essentially, a **slice** of a vector `x` is a subset of its elements, selected through the syntax `x[<indices>]`. Moreover, slices can behave in two distinct ways:

- As a **copy**, in which case a new object with its own memory address is created.
- As a **view**, in which case the slice references the original memory of `x`.

In what follows, we'll first examine how to mutate vectors by assigning new collections to their slices. From there, we'll cover the classic approach of using for-loops to modify elements one by one. Finally, we'll present the broadcasting assignment operator `.=`, which provides a concise tool for in-place updates.

MUTATIONS VIA COLLECTIONS

The most straightforward approach to mutating a vector is to replace an entire slice with another collection. This is achieved through statements of the form `x[<indices>] = <expression>`, where `<expression>` must match the length of `x[<indices>]`. Because slices on the left-hand side of `=` act as views, the assignment effectively modifies the original vector `x`, rather than creating a new one.

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

x[2:end]  = [20, 30]
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

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

x[x .≥ 2] = [2, 3] .* 10
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

A common use case is when `<expression>` depends on either elements of the original vector or on the slice being modified itself. This allows for self-referential updates, where new values are computed from old ones.

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

x[2:end]  = [x[i] * 10 for i in 2:length(x)]
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

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

x[x .≥ 2] = x[x .≥ 2] .* 10
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

Importantly, when the left-hand side is a single-element slice, the right-hand side of `=` accepts a scalar. This property will be particularly relevant when we present mutations via for-loops.

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

x[3]      = 30
```

```
julia> x
3-element Vector{Int64}:
  1
  2
 30
```

Warning! - Vectors can only be mutated by objects of the same type

When a vector is created, the type of its elements is implicitly defined. Consequently, attempting to replace elements with an incompatible type will result in an error. For instance, a vector of type `Int64` can only be mutated with other `Int64` values or `Float64` values that can be converted into it. This is shown below.

```
x         = [1, 2, 3]    # Vector{Int64}

x[2:3]    = [3.5, 4]     # 3.5 is Float64
```

```
ERROR: InexactError: Int64(3.5)
```

```
x         = [1, 2, 3]    # Vector{Int64}

x[2:3]      = [3.0, 4]        # 3.0 is Float64 but accepts
conversion
```

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

MUTATIONS VIA FOR-LOOPS

Previously, we indicated that single-element slices on the left-hand side of `=` permit seamless mutations with scalar values. Thus, statements like `x[i] = 0` directly updates the element at position `i`, without requiring a collection on the right-hand side. Extending this idea, multiple elements of a vector can be updated within a for-loop.

A common use case of this approach arises when populating a vector with values. Typically, this involves first initializing a vector, whose initial contents are irrelevant, and then iterating over its elements with a for-loop to assign the desired values. The strategy is especially prevalent when storing outputs generated during a computation.

```
x    = Vector{Int64}(undef, 3)  # `x` is initialized with 3 undefined elements

x[1] = 0
x[2] = 0
x[3] = 0
```

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

```
x    = Vector{Int64}(undef, 3)  # `x` is initialized with 3 undefined elements

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

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

The approach presented above relies on `x[i]` on the left-hand side of `=`, ensuring each element is treated as a view. However, an alternative strategy is to leverage the function `view`. This function enables the creation of a variable that contains all the elements to be modified. In doing so, the mutation can be performed directly on the entire object created, rather than repeatedly accessing elements from the original object.

In the following, we illustrate the technique by mutating a vector initialized with zeros. Note that the function `zeros` defaults to zeros with type `Float64`, explaining why `1` is automatically converted to `1.0`.

```
x     = zeros(3)


for i in 2:3
    x[i] = 1
end
```

```
julia> x
3-element Vector{Float64}:
 0.0
 1.0
 1.0
```

```
x     = zeros(3)
slice = view(x, 2:3)

for i in eachindex(slice)
    slice[i] = 1
end
```

```
julia> x
3-element Vector{Float64}:
 0.0
 1.0
 1.0
```

> **Warning!** - For-Loops Should Always be Wrapped in Functions
>
> In the example above, we left the for-loop outside a function to highlight the mutating strategy. In practice, however, placing for-loops in the global scope is highly discouraged: it not only severely hurts performance, but also introduces different variable-scoping rules. In fact, earlier versions of Julia completely disallowed mutations in the global scope through for-loops. To perform mutations via for-loops within functions, though, we first need to introduce the concept of mutating functions. This is done in the next section, where we'll return to this subject.

MUTATIONS VIA .=

Broadcasting provides a streamlined alternative to for-loops. This principle extends to mutations as well. The implementation is based on the broadcasting of the assignment operator `=`, denoted as `.=`. Specifically, the syntax is `x[<indices>] .= <expression>`, where `<expression>` can be either a *vector* or a *scalar*.

When in particular `x[<indices>]` appears on the left-hand side of `.=` and `<expression>` is a vector, the `.=` operator produces the same outcome as using `=`. In fact, using `=` rather than `.=` tends to be more performant in those cases.

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

x[1:2]  = x[1:2] .* 10
```

```
julia> x
3-element Vector{Int64}:
 30
 40
  5
```

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

x[1:2] .= x[1:2] .* 10    # identical output (less performant)
```

```
julia> x
3-element Vector{Int64}:
 30
 40
  5
```

Considering this, the primary use cases of `.=` for mutating `x` is for expressions such as:

- `x[<indices>].= <scalar>`,
- `x .= <expression>`, and
- `y .= <expression>` where `y` is a view of `x`.

Next, we analyze each case separately.

SCALARS ON THE RIGHT-HAND SIDE OF .=

A common scenario with mutations is when multiple elements must be replaced with the *same* scalar value. Implementing this operation with `=` requires providing a collection on the right-hand side, whose length must match the number of elements on the left. This not only introduces unnecessary boilerplate, but also assumes prior knowledge of the elements being replaced.

The broadcasting assignment operator `.=` makes such operations much simpler, simply requiring the execution of `x[<indices>] .= <scalar>`. The following code snippet employs this strategy to replace every negative value in `x` with zero.

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

x[x .< 0] .= 0
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 1
```

OBJECT ITSELF ON THE LEFT-HAND SIDE OF .=

We've already shown that the inclusion of terms like `x[indices]` on the left-hand side of `=` results in mutations. Now, let's turn to cases where an entire object appears on the left-hand side. Here, the focus is on scenarios where the object is `x` itself. Instead, scenarios with slices constructed via `view` will be deferred until the next subsection.

When an object appears on the left-hand side, we need to carefully **distinguish between in-place operations and reassignments**. Whether one or the other operation is implemented depends on whether `.=` or `=` is employed. Specifically, it's only when `.=` is used that a mutation takes place.

Instead, `=` will perform a reassignment, creating a new object at a new memory address. While the distinction seems irrelevant since `x` will ultimately hold the new values in both cases, we'll see in Part II of the website that the distinction actually matters for performance.

To illustrate, suppose our goal is to modify *all* the elements of a vector `x`. All the following approaches determine that `x` ends up holding the new values, but only the last two achieve this by mutation of `x`.

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

x    = x .* 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

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

x   .= x .* 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

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

x[:] = x .* 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

This risk of mixing up `.=` and `=` becomes even greater when using the `@.` macro for broadcasting, rather than manually inserting dots into each operator. The placement of `@.` relative to `=` determines whether the operation is a reassignment or a mutation. Specifically:

- If `@.` appears *before* `=`, `x` is mutated since `.=` is being used.
- If `@.` instead appears after `=`, only the right-hand side is broadcasted and the assignment is performed with `=`. This results in a reassignment, rather than a mutation.

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

x   .= x .* 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

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

@. x = x  * 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

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

x    = @. x * 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

VIEW ALIASES ON THE LEFT-HAND SIDE OF .=

Let's continue with our analysis of entire objects on the left-hand side of `.=`. Our focus now shifts to **view aliases**: variables such as `slices` defined by `slices = view(x[<indices>])`. They allow us to work directly with `slice` rather than `x[<indices>]`.

The introduction of view aliases is especially convenient when performing multiple operations on the same slice. It avoids repeated references to `x[<indices>]`, which would be inefficient, error-prone, and tedious.

As before, it's crucial to distinguish between using `.=` and `=`. In particular, only `.=` will perform a mutation, while `=` will result in a reassignment. With view aliases, however, additional care is required. The intended workflow involves first defining a slice (an assignment over a view) and then mutating that slice. This structure determines that there are now two possible wrong uses:

- the initial assignment is performed over a copy of `x[<indices>]`, rather than a view of `x[indices]`.
- the second step performs a reassignment (`=`), rather than a mutation (`.=`).

Below, we illustrate the correct usage, followed by these two incorrect patterns. The aim in the exercise is to replace all negative values in `x` with zero.

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

slice  = view(x, x .< 0)
slice .= 0
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 1
```

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

slice  = x[x .< 0]          # 'slice' is a copy
slice .= 0                  # this does NOT modify `x`
```

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

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

slice  = view(x, x .< 0)
slice  = 0                  # this does NOT modify `x`
```

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

Note that mutations with view aliases also allow `slice` to be included on the right-hand side of `=`. Below, we provide again the correct implementation, along with the two incorrect ones.

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

slice  = view(x, x .≥ 2)
slice .= slice .* 10        # same as 'x[x .≥ 2] = x[x .≥ 2] .* 10'
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

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

slice  = x[x .≥ 2]          # 'slice' is a copy
slice  = slice .* 10        # this does NOT modify `x`
```

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

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

slice  = view(x, x .≥ 2)
slice  = slice .* 10        # this does NOT modify `x`
```

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