5g. In-Place Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

This section focuses on **in-place operations**, a term that encompasses any method that mutates collections. These operations are characterized by the reuse of existing objects, rather than generating new ones, giving rise to the expression " *modifying values in place*."

Understanding whether an operation mutates an object or generates a new one is crucial, as the outcomes may differ depending on the application. Furthermore, even if results were identical, in-place modifications commonly entail performance benefits relative to the creation of new objects. This aspect will be explored in Part II of the website, when we discuss high performance.

Remark

Before proceeding, I recommend reviewing the definitions of slices introduced in the previous section. Recall that, given a vector `x`, a **slice** refers to a subset of `x`'s elements selected via `x[<indices>]`.

Moreover, a slice can act as a **copy**, in which case we're creating a new object with its own memory address. Alternatively, the slice can behave as a **view**, thereby referencing the original memory address of `x`. The distinction is crucial, as it'll determine whether modifying the slice will affect the original data or not.

MUTATIONS VIA COLLECTIONS

A simple way to mutate a vector is to replace an entire slice with another collection. This is achieved using statements of the form `x[<indices>] = <expression>`, where `<expression>` must match the length of `x[<indices>]`. The approach effectively mutates `x` because a slice on the left-hand side of `=` behaves as a view, thus referencing the original object. Below, we provide a few examples of this approach.

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

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

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

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

x[x .≥ 2] = [2, 3] .* 10
```

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

A common application of this method involves defining `<expression>` through elements from either the original vector or the slice being modified. This allows for self-referencing updates of the variable.

```julia
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
```

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

x[x .≥ 2] = x[x .≥ 2] .* 10
```

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

Importantly for the mutations via for-loops, a scalar can be used on the right-hand side of `=` for single-element slices.

```
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 defined, the type of elements that the vector can hold is implicitly defined. Consequently, attempting to replace elements with a different type will result in an error. For instance, the following examples only admit mutations with values of type `Int64`.

```
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

Replacing a single-element slice with a scalar value enables mutations through for-loops. This is implemented by substituting the value of a single element during each iteration.

To illustrate the procedure, let's consider a typical application of this approach: populating vectors with values. The process involves initializing a vector, and then iterating over its elements to assign desired values.

```julia
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
```

```julia
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 relies on `x[i]` on the left-hand side of `=` acting as a view. Alternatively, we could leverage the function `view` to create a variable that contains all the elements to be modified. This allows us to work with for-loops that mutate entire objects, rather than a subset of the original object.

The following example demonstrates this by mutating a vector of zeros. Note that the function `zeros` defaults to zeros with type `Float64`, explaining why `1` is automatically converted to `1.0`.

```julia
x     = zeros(3)


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

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

```julia
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 be Wrapped in Functions

Recall that for-loops should always be wrapped in functions, as failing to do so severely affects performance. In the next section, where we'll cover mutating functions, mutations via for-loops will be revisited.

MUTATIONS VIA .=

In terms of syntax, broadcasting serves as a streamlined alternative to for-loops. This principle even extends to mutations, which can be implemented by broadcasting the assignment operator `=`. The syntax for this is `x[<indices>] .= <expression>`, where `<expression>` can be either a *vector* or a *scalar*.

In the specific case of `x[<indices>]` on the left-hand side and a vector for `<expression>`, the `.=` operator produces the same outcome as using `=` with a corresponding collection. In fact, using `=` rather than `.=` in these cases tends to be more performant.

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

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

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

```julia
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` involve expressions like:

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

Next, we analyze each case separately.

SCALARS ON THE RIGHT-HAND SIDE OF =

Applying `=` to replace multiple elements with the *same* scalar value requires a collection matching the number of elements being substituted. However, with the `.=` operator, you can streamline the process by simply writing `x[<indices>] .= <scalar>`.

For instance, the following code snippet uses this approach to replace every negative value in `x` with zero.

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

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

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

VARIABLES ON THE LEFT-HAND SIDE OF =

Since both mutations and assignments rely on `=`, it's essential to **distinguish between in-place operations and reassignments**. In particular, the latter doesn't modify the original object, but actually creates a new one.

We've already shown that placing slices on the left-hand side of `=` results in mutations. Now, let's consider cases where an entire object like `x` appears on the left-hand side. In these cases, we need to be careful, as only `.=` will result in a mutation, whereas `=` will perform a reassignment.

For example, suppose our goal is to modify *all* the elements of a vector `x`. All the following approaches determine that `x` holds the same values, but only the last two achieve this by mutating `x`.

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

x    = x .* 10
```

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

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

x   .= x .* 10
```

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

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

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

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

Notice that the difference between approaches could go unnoticed, as they all yield the same outcome. However, we'll see in Part II of the website that these approaches can entail big differences in performance. In particular, reusing the original memory address of `x` tends to be more performant than creating a new memory address for `x`.

This difference will also manifest when using the macro `@.` for a seamless broadcasting. Depending on where `@.` is placed relative to `=`, we could end up with an assignment or a mutation.

```julia
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 =

The previous case exemplified a scenario where the ultimate outcome is the same, regardless of whether we employ `=` or `.=`. Instead, the results will differ when view aliases are on the left-hand side. View aliases allow us to work with `slice = view(x, <indices>)`, rather than `x[<indices>]`. Defining view aliases is convenient when we need to perform various operations over the same slice. This avoids repeatedly referencing the slice via `x[<indices>]`, which would be inefficient, error-prone, and tedious.

In these cases, it's only when we use `.=` that we'll perform a mutation.

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

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

There are a few additional incorrect uses that can emerge with view aliases. To demonstrate them, let's consider replacing all negative values in `x` with zero. Below, only the first approach achieves the desired outcome.

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

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

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

```julia
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
```

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

slice  = view(x, x .< 0)
slice  = 0                  # this creates a new object, it does NOT modify `x`
```

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