5e. Slices: Copies vs Views

Martin Alfaro
PhD in Economics

INTRODUCTION

This section concludes the coverage of preliminary concepts for studying mutations by focusing on the behavior of a vector's **slice**. This is defined as a subset of a vector's elements, formally represented as `x[<indices>]`.

Importantly, **slices act differently depending on how they're included in a statement**, functioning as either:

- **copies** of the original vector, thus creating a new object at a new memory address.
- **views** of the original vector, where the original object and the slice share the same memory address.

In the following, we explain the distinction between copies and views in detail. Understanding it is crucial for mutating slices, as mutations can only occur when the slice references the original object. In contrast, if a slice acts as a copy, the parent object and the slice are unrelated, with changes to the slice having no impact on the original object.

SLICES AND THE ASSIGNMENT OPERATOR

Vector mutation involves modifying slices through the operator `=`. For this to be possible, a prerequisite is that the slice references the original object. Nonetheless, the behavior of slices in assignments varies depending on their position within the expression.

Specifically, **slices on the left-hand (LHS) side of `=` act as views**. In this case, slices reference the original elements, thus allowing for the mutation of its parent object. In contrast, **slices on the right-hand side (RHS) of `=` create a copy**. Since copies point to a new object, any modification to the slice won't affect the original object.

The following code snippet demonstrates both behaviors.

```
x    = [4,5]


x[1] = 0        # 'x[1]' is a view and mutates 'x'
```

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

```
x    = [4,5]
y    = x[1]     # 'y' is unrelated to 'x' because 'x[1]' is a copy

x[1] = 0        # it mutates 'x' but does NOT modify 'y'
```

```
julia> y
4
```

Aliasing vs Copy

Objects on the RHS of `=` are only treated as copies when it comes to **slices**, such as in statements `y = x[<indices>]`. Instead, if we insert the whole object `x` on the RHS of `=`, as in `y = x`, the operation creates an alias. In this case, `y` and `x` will reference the same object, and so any modification made to `y` will also be reflected in `x`.

```
x    = [4,5]
y    = x        # the whole object (a view)

x[1] = 0        # it DOES modify 'y'
```

```
julia> y
2-element Vector{Int64}:
 4
 5
```

```
x    = [4,5]
y    = x[:]     # a slice of the whole object (a copy)

x[1] = 0        # it does NOT modify 'y'
```

```
julia> y
2-element Vector{Int64}:
 0
 5
```

THE FUNCTION 'VIEW'

Identifying when slices act as copies or views is relevant for high performance, as views eliminate the overhead associated with memory allocations. Although this topic will be explored in Part II of the website, such considerations underscore the importance of distinguishing between copies and slices, beyond their use in assignments.

As a rule of thumb, **slices typically default to creating copies**. This is the case when, for instance, a slice is passed as a function argument or when used within an expression not involving an assignment. These scenarios are illustrated below.

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

#the following slices are all copies
log.(x[1:2])

x[1:2] .+ 2

[sum(x[:]) * a for a in 1:3]

(sum(x[1:2]) > 0) && true
```

In all these cases, transforming slices into views requires an explicit indication. To achieve this, you need to employ the function `view`. Its syntax is `view(x, <indices>)`, where `<indices>` represent the subset of indices defining the slice. To demonstrate its usage, we revisit and compare the previous code snippet.

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

#we make explicit that we want views
log.(view(x,1:2))

view(x,1:2) .+ 2

[sum(view(x,:)) * a for a in 1:3]

(sum(view(x,:)) > 0) && true
```

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

#the following slices are all copies
log.(x[1:2])

x[1:2] .+ 2

[sum(x[:]) * a for a in 1:3]

(sum(x[1:2]) > 0) && true
```

These examples reveal the potential verbosity involved when `view` isn't used sparingly. To address this issue, Julia provides the macros `@view` and `@views`.

The `@view` macro is equivalent to `view`, allowing you to write `@view x[1:2]` instead of `view(x, 1:2)`. However, its advantages are somewhat limited: it saves only a few characters, and additionally necessarily requires parentheses when multiple slices are used (e.g., `@view(x[1:2]) .+ @view(x[2:3])`). In contrast, the `@views` macro significantly streamlines notation, by automatically converting *every* slice within an expression into a view.

```
x = [4,5,6]

# the following are all equivalent
       y = view(x, 1:2)  .+ view(x, 2:3)
       y = @view(x[1:2]) .+ @view(x[2:3])
@views y = x[1:2] .+ x[2:3]
```

One of the most notable applications of `@views` is in functions. By placing `@views` at the beginning of a function, you automatically convert every slice within the function body and its arguments into a view.

```
@views function foo(x)
  y = x[1:2] .+ x[2:3]
  z = sum(x[:]) .+ sum(y)

  return z
end
```

```
function foo(x)
  y = @view(x[1:2]) .+ @view(x[2:3])
  z = sum(@view x[:]) .+ sum(y)

  return z
end
```