5e. Slices: Copies vs Views

Martin Alfaro
PhD in Economics

INTRODUCTION

This section concludes our preparation for the study of mutations. The attention is now shifted to the concept of vector **slices**: subsets of elements drawn from a vector, written as `x[<indices>]`. In particular, the focus will be on the critical distinction between whether slices act as:

- **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.

Which of these two forms arises is subtle, since it depends on where the slice appears within an expression.

The difference is central to implementing mutations correctly, as they're only possible when the slice is a view. If a slice is instead a copy, the parent object and the slice become entirely independent entities, so that modifications to the slice have no impact on the original object.

In Part II of this website, we'll also see that the distinction between copies and views has implications for performance. Essentially, creating a copy requires allocating new memory, which introduces computational overhead. Views, by contrast, avoid this cost by reusing the memory of the original vector.

Because of its broad significance, we'll examine the topic of copies and views independently of mutations. In particular, we'll identify how slices behave in different cases. Moreover, we'll explain how to instruct Julia to treat slices as views or copies.

SLICES AND THE ASSIGNMENT OPERATOR

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 role, slices directly reference the original elements, enabling mutation of the parent object. In contrast, **slices on the right-hand side (RHS) of `=` create a copy**. Since copies point to a new object in memory, any modification to the slice won't affect the original object.

The following code snippet demonstrates these contrasting 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 that case, `y` and `x` will reference the same object, so that 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'

Beyond assignments, we must also distinguish whether slices represent copies or views within other expressions. As a rule of thumb, **slices typically default to creating copies**. Such behavior arises when, for instance, a slice is passed as a function argument or incorporated into a computation. Several of the 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, if you instead want to work with views, you must indicate this explicitly. Several methods exist for achieving this, with the most straightforward being the function `view`. Its syntax is `view(x, <indices>)`, where `<indices>` specify the indices that define the slice. To demonstrate its usage, we revisit the previous examples.

```
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
```

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

The `@view` macro is equivalent to `view`, allowing you to write `@view x[1:2]` instead of `view(x, 1:2)`. Its benefits, however, are somewhat limited: it saves only a few characters and still requires parentheses when multiple slices appear in the same expression (e.g., `@view(x[1:2]) .+`

`@view(x[2:3])`). By contrast, the `@views` macro significantly streamlines notation, 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` arises in functions. When placed at the beginning of a function definition, `@views` ensures that every slice appearing in the function body or its arguments is treated as 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
```