5c. Assignments vs Mutations

Martin Alfaro
PhD in Economics

INTRODUCTION

The upcoming sections will be entirely devoted to **vector mutation**. However, to properly cover this subject, we first need to introduce some preliminary concepts, including:

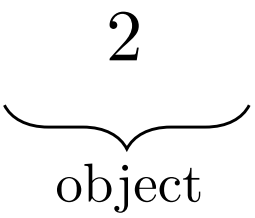
- the distinction between assignments and mutations
- methods for initializing arrays to eventually mutate them
- techniques for vector indexing to select elements

The current section in particular focuses on **distinguishing between assignments and mutations of variables**. The difference between both operations can easily go unnoticed by new users, as both operations use the operator `=`, despite being fundamentally different. Clearly delineating these operations is important not only for Julia, but also other programming languages.

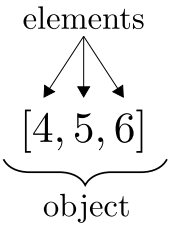
SOME BACKGROUND

Recall that **variables** serve as labels for **objects**, with objects in turn holding **values**. Moreover, objects can be classified according to the number of **elements** contained, ranging from single-element objects (e.g. integers and floating-point numbers) to collections (e.g. vectors).

NUMBER



VECTOR



The distinction between objects and their elements is crucial for the remainder of the section. This is because *assignments apply to objects, whereas mutations apply to elements*. More specifically, assignments rebind variables to new objects, while mutations simply modify existing elements of an existing object.

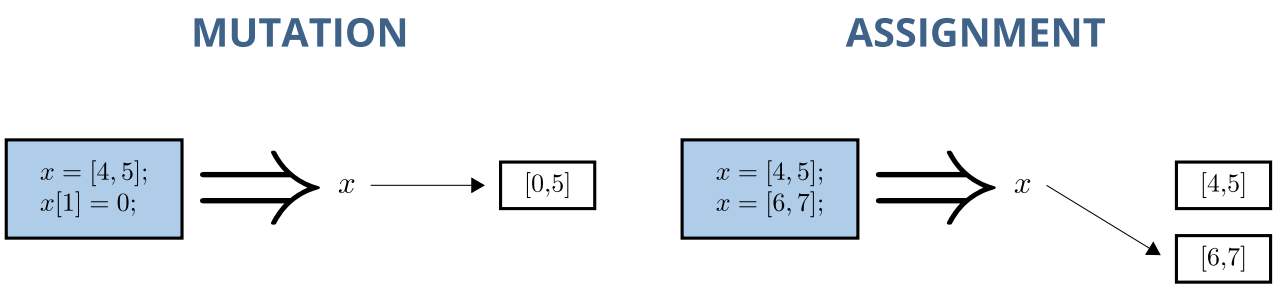
ASSIGNMENTS VS MUTATIONS

Assignments bind variables to objects, a process implemented via the `=` operator. For instance, `x = 3` and `x = [1,2,3]` are examples of assignments, where `3` and `[1,2,3]` represent the objects being assigned to `x`.

Mutations, by contrast, **modify the elements of an object, without creating a new one**. These operations also rely on the `=` operator, with `x[1] = 0` being an example of mutation.

Despite sharing the same operator `=`, assignments and mutations are conceptually distinct. This difference can be better appreciated by **visualizing objects as specific memory addresses**. This implies that assignments like `x = [4,5]` involve two steps: *i*) finding a memory location to store the object with value `[4,5]`, and *ii*) labeling the memory address as `x` for easy access. In contrast, a mutation modifies the contents of the object, but without changing its memory address.

To illustrate this, consider the example `x = [4,5]`, where the object `[4,5]` is stored in a particular memory location. If you then run `x = [6,7]`, `x` becomes associated with a new object containing `[6,7]`, thus constituting an *assignment*. However, if you execute `x[1] = 0` afterwards, the operation modifies the original object `[6,7]` to `[6,0]`. This operation constitutes a *mutation* because `x` continues to reference the same memory address, even though its content has changed.



Remark

You can mutate all the elements of `x`, without this necessarily entailing a new assignment. For example, this occurs when we modify the values of `x` by mutating `x[:]`.

```
x    = [4,5]

x[:] = [0,0]
```

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

> The distinction is particularly important since mutations tend to be faster than creating new objects. This will become relevant in Part II, where we explore strategies for speeding up operations.

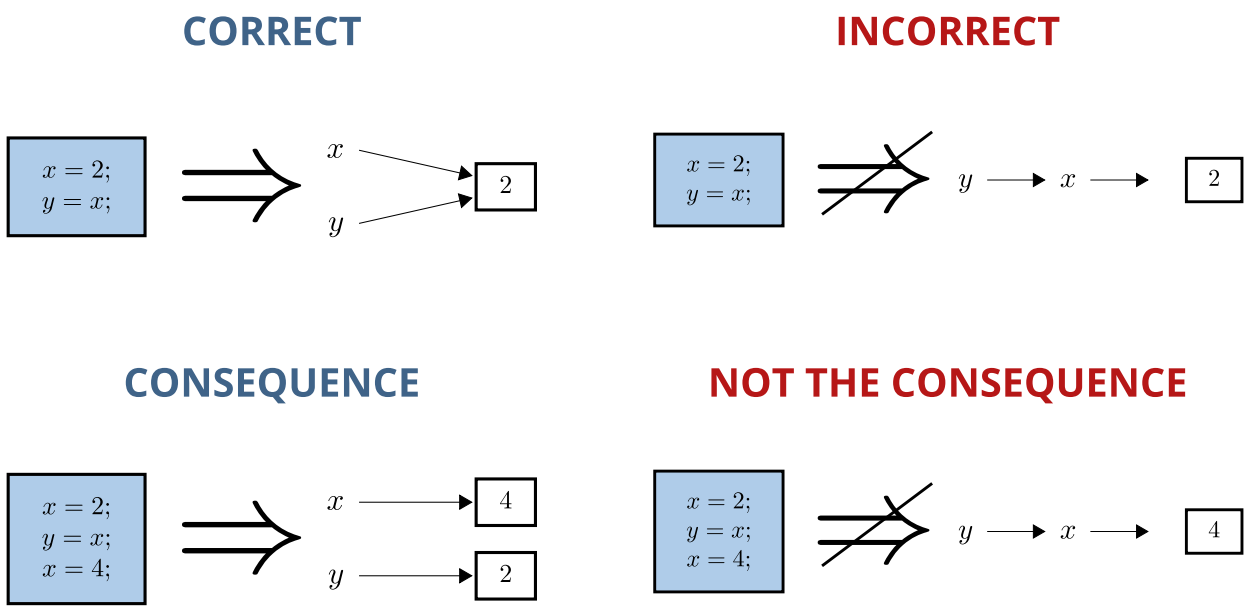
ALIAS VS COPY

So far, we've emphasized the critical distinction between assignments and mutations. Since both operations rely on the `=` operator, we must then inquire when `=` will entail one or the other operation. Next, we explore in particular cases like `y = x`, characterized by entire objects on each side of `=`. Other cases are left for the upcoming sections, after we introduce the concept of slices (i.e, subsets of elements from a vector).

In Julia, executing `y = x` makes `y` another name for the object referenced by `x`. This means that `x` and `y` become different labels for the same underlying object. Formally, it's said that `y` constitutes an **alias** of `x`.

Note that `y = x` shouldn't be understood as binding `y` to `x` itself. Rather, it means that `y` becomes another label for the object that `x` references. This subtle distinction carries a significant practical implication: reassigning `x` to a new object won't affect `y`'s reference.

To clarify this further, let's consider an example where we first execute `x = 2` and then `y = x`. At this point, both `x` and `y` reference the same object, which holds the value `2`. If we eventually execute `x = 4`, the variable `x` will start pointing to a new object that holds the value `4`. However, this won't affect the original object that `x` was referencing. As a result, `y` will still point to the original object with value `2`. This behavior is illustrated below.



```
x = 2   #'x' points to an object with value 2
y = x   #'y' points to the same object as 'x' (do not interpret it as 'y' pointing to 'x')

x = 4   #'x' now points to another object (but 'y' still points to the object holding 2)
```

```
julia> x
4
julia> y
2
```

Remark

Two variables could comprise identical elements and yet refer to different objects.

This can be demonstrated using the operators `==` and `===`, which assess two different types of equality. Specifically, `x == y` checks whether `x` and `y` have equal values, regardless of whether they refer to the same object. In contrast, `x === y` checks whether both `x` and `y` point to the same object, thus verifying if they share the same memory address. By applying these operators, the following example illustrates that objects with identical elements aren't necessarily referencing the same object.

```
x = [4,5]

y = x
```

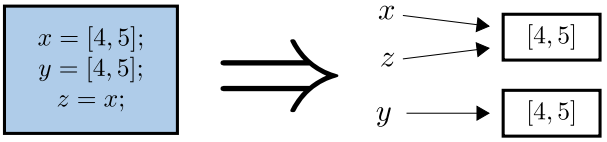
```
julia> x == y
true                    #`x` and `y` have identical elements
julia> x === y
true                        #`x` and `y` DO point to the same
object
```

```
x = [4,5]

y = [4,5]
```

```
julia> x == y
true                    #`x` and `y` have identical elements
julia> x === y
false                    #`x` and `y` DO NOT point to the same
object
```

GRAPHICAL REPRESENTATION



We've indicated that the operation `y = x` creates an alias of `x`, making `y` and `x` two different labels for the same object. This implies that **modifying the elements of either `x` or `y` will necessarily change the elements held by both**. The following diagram and code snippet illustrate this.

GRAPHICAL REPRESENTATION

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

x[1] = 0
```

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

If you instead want to treat `x` and `y` as separate objects, you must apply the function `copy`. This creates a *new object* with identical elements as the original. In this way, any modification to the new object won't affect the original one, allowing you to work with `x` and `y` independently.

```
x    = [4,5]
y    = copy(x)

x[1] = 0
```

```
julia> x
2-element Vector{Int64}:
 0
 5
julia> y
2-element Vector{Int64}:
 4
 5
```