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:

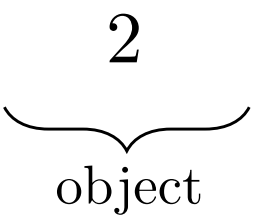
- the distinction between assignment and mutation
- methods for initializing arrays to eventually mutate them
- techniques for vector indexing to select elements

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

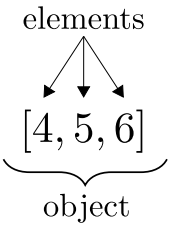
SOME BACKGROUND

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.

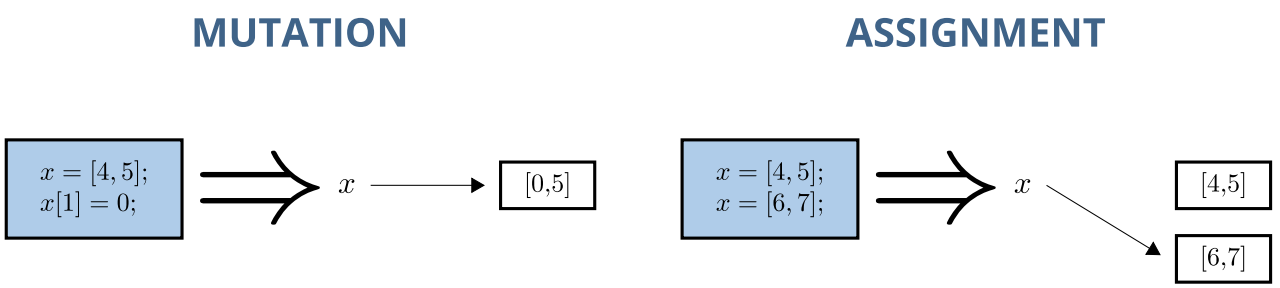
ASSIGNMENTS VS MUTATIONS

Assignments establish a binding between a variable and an object through the `=` operator. For instance, `x = 3` and `x = [1,2,3]` are examples of assignments, where the objects `3` and `[1,2,3]` are to `x`.

Mutations, by contrast, **modify the elements of an object, without creating a new one**. These operations also rely on the `=` operator. An example of mutation is given by `x[1] = 0`, which modifies the first element of `x`.

Despite sharing the same operator `=`, assignments and mutations are conceptually distinct. The difference becomes clearer when we **views objects as residing at specific memory addresses**. Thus, an assignment for `x` involves two steps: *i*) finding a memory location to store the object, and *ii*) labeling the memory address as `x` for easy access. A mutation, in contrast, modifies the data stored at an existing memory address, but leaving the address itself unchanged.

To illustrate, if `x = [6,7]` is run, `x` becomes associated with an object containing `[6,7]`. Thus, this constitutes an *assignment*. However, if `x[1] = 0` is executed afterwards, the operation modifies the original object `[6,7]`, which now becomes `[6,0]`. This operation constitutes a *mutation*: `x` continues to reference the same memory address, even though its content has changed.



Mutating All Elements vs Assignment

Mutating all the elements of `x` doesn't imply a new assignment. For example, this occurs by modifying `x` using `x[:]`.

```
x    = [4,5]

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

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

> The distinction is important for performance, as mutating existing objects is generally faster than creating new ones. This point will be particularly relevant in Part II of the website, where we discuss optimization strategies.

ALIAS VS COPY

So far, we've introduced the fundamental distinction between assignments and mutations. Because both operations employ the `=` operator, the natural question that arises is when `=` results in an assignment or a mutation.

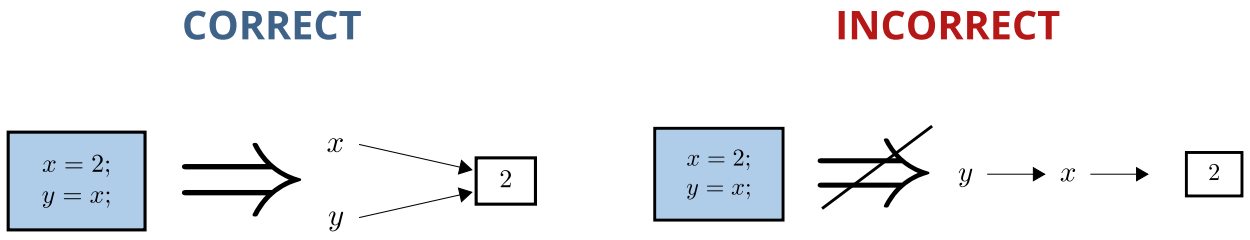
In this section, we focus on cases where entire objects appear on both sides of `=`, as in `y = x`. Other scenarios are left for upcoming sections, after we introduce the concept of slices (i.e, subsets of elements from a vector).

When `y = x` is executed in Julia, `y` becomes another name for the object referenced by `x`. In other words, `x` and `y` become different labels for the same underlying object. Formally, we say that `y` is an **alias** of `x`.

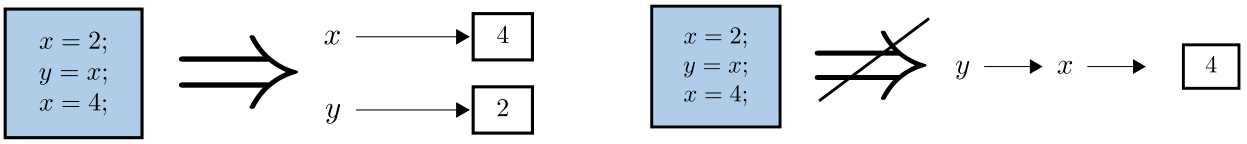
It's important to stress that `y = x` doesn't bind `y` to `x` itself. Rather, `y` becomes another label for the object that `x` references. This subtle distinction carries a significant practical implication: reassigning `x` to a new object won't affect `y`'s reference.

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 holding 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 visually illustrated below.



CONSEQUENCE **NOT THE CONSEQUENCE**



The conclusion can also be drawn from examining code execution.

```
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
```

Two variables may contain identical elements and yet refer to different objects

The claim can be demonstrated using the operators `==` and `===`, which capture two different notions of equality. The expression `x == y` checks whether `x` and `y` have the same *values*, regardless of whether they reference the same object. In contrast, `x === y` checks whether both `x` and `y` point to the same *object*, thus verifying if they point to 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
```

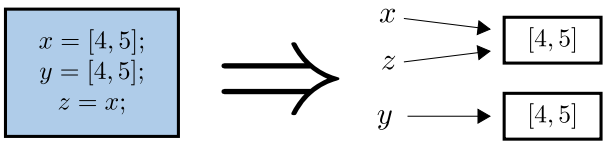
```
julia> x == y
true                    #`x` and `y` have identical elements
julia> x === y
true                         #`x` and `y` DO point to the same
object
```

```
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 a new alias for `x`, turning `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 referenced by both**. The following diagram illustrates this.

GRAPHICAL REPRESENTATION

The corresponding code snippet captures this case.

```
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 aim to treat `x` and `y` as pointing to separate objects, you must use the `copy` function. This creates a *new object* with the same 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
```