5d. Initializing Vectors

Martin Alfaro
PhD in Economics

INTRODUCTION

We continue introducing preliminary concepts for mutations. The previous section distinguished between the use of `=` for assignments and mutations. Now, we'll deal with approaches to creating vectors.

Our presentation starts by outlining the process of initializing vectors, where memory is reserved without assigning initial values. We'll then discuss how to create vectors filled with predefined values such as zeros or ones. Finally, we show how to concatenate multiple vectors into new ones.

INITIALIZING VECTORS

Creating an array involves two steps: reserving memory for holding its content and assigning initial values to its elements. However, if you don't intend to populate the array with values right away, it's more efficient to only initialize the array. This means reserving memory space, but without setting any initial values.

Technically, initializing an array entails creating an array filled with `undef` values. These values represent arbitrary content in memory at the moment of allocation. Importantly, while `undef` displays concrete numbers when you output the array's content, they're meaningless and vary every time you initialize a new array.

There are two methods for creating vectors with `undef` values. The first one requires you to explicit specify the type and length of the array, which is accomplished via `Vector{<elements' type>}(undef, <length>)`. The second approach is based on the function `similar(x)`, which creates a vector with the same type and dimensions as an existing vector `x`.

```
x_length = 3

x = Vector{Int64}(undef, x_length)   # `x` can hold `Int64` values, and is initialized with
3 undefined elements
```

```
julia> x
3-element Vector{Int64}:
 140724480121488
   2497084710592
   2497285012816
```

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

x = similar(y)                    # `x` has the same type as `y`, which is Vector{Int64}
(undef, 3)
```

```
julia> x
3-element Vector{Int64}:
 2497063587648
 2497063587664
 2497355825296
```

The example demonstrates that `undef` values don't follow any particular pattern. Moreover, these values vary in each execution, as they reflect any content held in RAM at the moment of allocation. In fact, a more descriptive way to call `undef` values would be **uninitialized values**.

CREATING VECTORS WITH GIVEN VALUES

In the following, we present several approaches to creating arrays filled with predefined values.

VECTORS WITH RANGE

If our goal is to generate a sequence of values, we can employ the function `collect(<range>)`. Recall that the syntax for defining ranges is `<start>: <steps>: <stop>`, where `<steps>` establishes the gap between elements.

```
some_range = 2:5

x          = collect(some_range)
```

```
julia> x
4-element Vector{Int64}:
 2
 3
 4
 5
```

Notice that when a range is created, `<steps>` implicitly dictates the number of elements to be generated. Alternatively, you could specify the number of elements to be stored, letting `<steps>` be implicitly determined. This is achieved by the function `range`, whose syntax is `range(<start>, <end>, <number elements>)`. [1]

The following code snippet demonstrates the use of `range`, by generating five evenly spaced elements between 0 and 1.

```
x = range(0, 1, 5)
```

```
julia> x
0.0:0.25:1.0
```

```julia
x = range(start=0, stop=1, length=5)
```

```
julia> x
0.0:0.25:1.0
```

```julia
x = range(start=0, length=5, stop=1)     # any order for keyword arguments
```

```
julia> x
0.0:0.25:1.0
```

VECTORS WITH SPECIFIC VALUES

We can also create vectors of some given length filled with the same repeated value. In particular, the functions `zeros` and `ones` respectively create vectors with zeros and ones. By default, these functions define `Float64` elements, although you can specify a different type in the first argument of the function.

```julia
length_vector = 3

x             = zeros(length_vector)
```

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

```julia
length_vector = 3

x             = zeros(Int, length_vector)
```

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

```julia
length_vector = 3

x             = ones(length_vector)
```

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

```julia
length_vector = 3

x             = ones(Int, length_vector)
```

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

For creating Boolean vectors, Julia provides two convenient functions called `trues` and `falses`.

```julia
length_vector = 3

x             = trues(length_vector)
```

```
julia> x
3-element BitVector:
 1
 1
 1
```

```julia
length_vector = 3

x             = falses(length_vector)
```

```
julia> x
3-element BitVector:
 0
 0
 0
```

VECTORS FILLED WITH A REPEATED OBJECT

To define vectors comprising elements different from zeros or ones, Julia provides the `fill` function. Unlike the previous functions, this accepts any arbitrary scalar to be repeated.

```julia
length_vector    = 3
filling_object   = 1

x                = fill(filling_object, length_vector)
```

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

```julia
length_vector    = 3
filling_object   = [1,2]

x                = fill(filling_object, length_vector)
```

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

```julia
length_vector    = 3
filling_object   = [1]

x                = fill(filling_object, length_vector)
```

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

CONCATENATING VECTORS

Finally, we can create a vector `z` that merges all the elements of two vectors `x` and `y`. One simple approach for doing this is through `z = [x ; y]`. While this method is suitable for concatenating a few vectors, it becomes impractical with a large number of vectors, and directly infeasible when the number of vectors to concatenate is unknown.

For these scenarios, we can instead employ the function `vcat`, which merges all its arguments into one vector. By use of the splat operator `...`., the function can also be applied with a single argument that comprises a list of vectors. [2]

```julia
x = [3,4,5]
y = [6,7,8]


z = vcat(x,y)
```

```
julia> x
6-element Vector{Int64}:
 3
 4
 ⋮
 7
 8
```

```julia
x = [3,4,5]
y = [6,7,8]

A = [x, y]
z = vcat(A...)
```

```
julia> x
6-element Vector{Int64}:
 3
 4
 ⋮
 7
 8
```

Closely related to vector concatenation is the `repeat` function, which defines a vector containing the same object multiple times. Importantly, unlike `fill`, `repeat` **requires an array as its input**, throwing an error if a scalar in particular is passed.

```julia
nr_repetitions   = 3
vector_to_repeat = [1,2]

x                = repeat(vector_to_repeat, nr_repetitions)
```

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

```julia
nr_repetitions   = 3
vector_to_repeat = [1]

x                = repeat(vector_to_repeat, nr_repetitions)
```

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

```julia
nr_repetitions   = 3
vector_to_repeat = 1

x                = repeat(vector_to_repeat, nr_repetitions)
```

```
ERROR: MethodError: no method matching repeat(::Int64, ::Int64)
```

ADDING, REMOVING, AND REPLACING ELEMENTS (*OPTIONAL*)

Warning!

This subsection requires knowledge of a few **concepts that we haven't discussed yet**. As such, it's marked as optional.

One such concept is **in-place functions**, identified by the symbol `!` appended to the function's name. The symbol is simply notation added by developers, hinting that the function modifies the value of at least one of its arguments. In-place functions will be explored thoroughly later).

Another concept introduced is **pairs**, which will also be examined comprehensively in a future section. For the purposes of this subsection, it's sufficient to know that pairs are denoted by `a => b`, where `a` in our application will refer to some value and `b` represent its corresponding replacement value.

Next, we show how to add, remove, and replace elements of a vector. To add a single element in particular, the methods are as follows.

```
x                 = [3,4,5]
element_to_insert = 0


push!(x, element_to_insert)                    # add 0 at the end - faster
```

```
julia> x
4-element Vector{Int64}:
 3
 4
 5
 0
```

```
x                 = [3,4,5]
element_to_insert = 0


pushfirst!(x, element_to_insert)               # add 0 at the beginning - slower
```

```
julia> x
4-element Vector{Int64}:
 0
 3
 4
 5
```

```julia
x                 = [3,4,5]
element_to_insert = 0
at_index          = 2

insert!(x, at_index, element_to_insert)      # add 0 at index 2
```

```
julia> x
4-element Vector{Int64}:
 3
 0
 4
 5
```

```julia
x                 = [3,4,5]
vector_to_insert  = [6,7]


append!(x, vector_to_insert)                 # add 6 and 7 at the end
```

```
julia> x
5-element Vector{Int64}:
 3
 4
 5
 6
 7
```

The function `push!` is particularly helpful to collect results in a vector. This is because, as it doesn't require any prior knowledge about the number of elements to be stored, we can dynamically grow the vector by adding more results. Notice that adding elements at the end via `push!` is faster than doing it at the beginning via `pushfirst!`.

Analogous functions exist to remove elements, as shown below.

```julia
x                  = [5,6,7]


pop!(x)                            # delete last element
```

```
julia> x
2-element Vector{Int64}:
 5
 6
```

```julia
x                  = [5,6,7]


popfirst!(x)                       # delete first element
```

```
julia> x
2-element Vector{Int64}:
 6
 7
```

```julia
x                    = [5,6,7]
index_of_removal    = 2

deleteat!(x, index_of_removal)       # delete element at index 2
```

```
julia> x
2-element Vector{Int64}:
 5
 7
```

```julia
x                    = [5,6,7]
indices_of_removal = [1,3]

deleteat!(x, indices_of_removal)    # delete elements at indices 1 and 3
```

```
julia> x
1-element Vector{Int64}:
 6
```

Emulating the behavior of `deleteat!`, we can also indicate which elements should be retained.

```julia
x                 = [5,6,7]
index_to_keep    = 2

keepat!(x, index_to_keep)
```

```
julia> x
1-element Vector{Int64}:
 6
```

```julia
x                 = [5,6,7]
indices_to_keep = [2,3]

keepat!(x, index_to_keep)
```

```
julia> x
1-element Vector{Int64}:
 6
```

Finally, we can replace specific values with new ones. This can be done by creating a new copy via `replace` or by updating the original vector with `replace!`.

Both functions make use of pairs `a => b`, where `a` is some value and `b` its corresponding replacement value. Note that these functions perform substitutions based on values, rather than indices.

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

replace!(x, 3 => 0)             # in-place (it updates x)
```

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

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

replace!(x, 3 => 0, 5 => 1)     # in-place (it updates x)
```

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

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

y = replace(x, 3 => 0)          # new copy
```

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

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

y = replace(x, 3 => 0, 5 => 1)  # new copy
```

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

FOOTNOTES

[1.] Note that `range` represents a convenient syntax for `<start> : 1 / <number elements> : <end>`.

[2.] Recall that the operator `...` splits a collection into multiple arguments. This enables the use of a vector or tuple to denote multiple function arguments. For further details, see here under the subsection "Splatting".