5d. Vector Creation and Initialization

Martin Alfaro
PhD in Economics

INTRODUCTION

We continue presenting preliminary concepts for introducing the concept of mutation. 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 through the `vcat` function.

INITIALIZING VECTORS

Creating an array involves two steps: reserving memory for holding its content and assigning initial values to its elements. When you don't intend to populate the array with values right away, it's more efficient to perform only the allocation step. 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 explicitly 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}:
  1
 46
 42
```

```
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}:
                1
               52
 128181526331504
```

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 COMPRISING A RANGE

To generate a sequence of values through ranges, we need to employ the expression `collect(<range>)`. Recall that the syntax for defining ranges is `<start>: <steps>: <stop>`, where `<steps>` specifies the increment between consecutive values.

The function `collect` is necessary, since ranges describe the values to be generated, without materializing them. This behavior is consistent with a broader concept known as lazy operations, which will be presented later on this book. [1]

```
some_range = 2:5

x          = collect(some_range)
```

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

When a range is created, `<steps>` dictates the number of elements to be generated. Considering this, Alternatively, we can specify the number of elements to be stored through the syntax `<start> : 1/<number of elements> : <end>`. A more direct way is via the `range` function, whose syntax is `range(<start>, <end>, <number of elements>)`.

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

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

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

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

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

```
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 REPEATED

We can also build vectors of a given length where every element is the same value. Two common examples are `zeros` and `ones`, which produce vectors filled with zeros and ones, respectively. By default, both functions create vectors whose elements have type `Float64`. You can override this behavior by providing the desired element type as the first argument.

```
length_vector = 3

x             = zeros(length_vector)
```

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

```
length_vector = 3

x             = zeros(Int, length_vector)
```

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

```
length_vector = 3

x             = ones(length_vector)
```

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

```
length_vector = 3

x             = ones(Int, length_vector)
```

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

To create vectors with Boolean values, Julia provides two convenient functions called `trues` and `falses`.

```
length_vector = 3

x             = trues(length_vector)
```

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

```
length_vector = 3

x             = falses(length_vector)
```

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

VECTORS WITH AN ARBITRARY VALUE REPEATED

More generally, we can define vectors whose elements share the same specified value. This is achieved by the `fill` function.

```
length_vector  = 3
filling_object = 1

x              = fill(filling_object, length_vector)
```

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

```
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]
```

```
length_vector  = 3
filling_object = [1]

x              = fill(filling_object, length_vector)
```

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

A VECTOR WITH CONCATENATED ELEMENTS OF VECTORS

There are several ways to construct a vector `z` that combines all elements of two vectors `x` and `y`. A straightforward method is using `z = [x; y]`. While this works well for concatenating a small number of vectors, it becomes cumbersome when many vectors must be merged. Moreover, it's not viable when the number of vectors is unknown in advance.

In such cases, we can employ the `vcat` function. This concatenates all of its arguments into a single vector. When paired with the splat operator `...`, it can also operate on a single argument containing multiple vectors.

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


z = vcat(x,y)
```

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

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

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

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

VECTORS WITH ELEMENTS OF AN OBJECT REPEATED

Closely related to `fill` is the `repeat` function. This uses a *collection* as an input, concatenating their elements repeatedly a given number of times. The `repeat` function **necessarily requires an array as its input**, throwing an error if a scalar is passed.

```
nr_repetitions     = 3
elements_to_repeat = [1,2]

x                  = repeat(elements_to_repeat, nr_repetitions)
```

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

```
nr_repetitions     = 3
elements_to_repeat = [1]

x                  = repeat(elements_to_repeat, nr_repetitions)
```

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

```
nr_repetitions   = 3
vector_to_repeat = 1

x                = repeat(vector_to_repeat, nr_repetitions)
```

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

Note that the behavior of `repeat` differs from `fill` function, since the latter repeats the same object without concatenating its elements. In fact, the output of `repeat` is the same as creating a vector with `fill` and then using `vcat` to concatenate its elements.

```
nr_repetitions     = 3
elements_to_repeat = [1,2]

x                  = repeat(filling_object, nr_repetitions)
```

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

```
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]
```

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

temp               = fill(filling_object, length_vector)
x                  = vcat(temp...)
```

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

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

> **Warning!**
>
> This subsection requires knowledge of a few **concepts we haven't discussed yet**. As such, it's marked as optional.
>
> One such concept is that of **in-place functions**, identified by the symbol `!` appended to the function's name. The symbol is simply a convention chosen by developers to indicate that the function modifies at least one

> of its arguments. A detailed discussion of in-place functions will be provided later.
>
> Another concept introduced is that of **pairs**, which will be thoroughly examined in a future section too. For the purposes of this subsection, it's sufficient to know that pairs are written in the form `a => b`. In our applications, `a` will represent a given value and `b` denotes its corresponding replacement value.

Next, we demonstrate how to add, remove, and replace elements within a vector. Below, we begin by presenting methods for adding a single element.

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


push!(x, element_to_insert)                # add 0 as last element - 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 as first element - slower
```

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

```
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
```

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


append!(x, vector_to_insert)                # add 6 and 7 as last elements
```

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

The function `push!` is particularly helpful to collect outputs in a vector. Since it doesn't require prior knowledge of how many elements will be stored, the vector can grow dynamically as new results are added. Notice that adding elements at the end with `push!` is generally faster than inserting them at the beginning with `pushfirst!`.

Analogous functions exist to remove elements, as shown below.

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


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

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

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


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

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

```
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
```

```
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
```

By analogy with the behavior of `deleteat!`, it's also possible to specify which elements should be retained.

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

keepat!(x, index_to_keep)
```

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

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

keepat!(x, index_to_keep)
```

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

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

Both functions make use of pairs `a => b`, where `a` denotes a given value and `b` specifies its replacement. Note that these functions perform substitutions based on values, not on indices.

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

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

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

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

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

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

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

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

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

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

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

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

FOOTNOTES

[1] Lazy operations specify how a computation should proceed, but postpone producing concrete results until they're explicitly required (either because you request them or because another computation depends on them.) They become especially valuable when combined with other operations, since this fusion may eliminate the need to store intermediate results altogether.