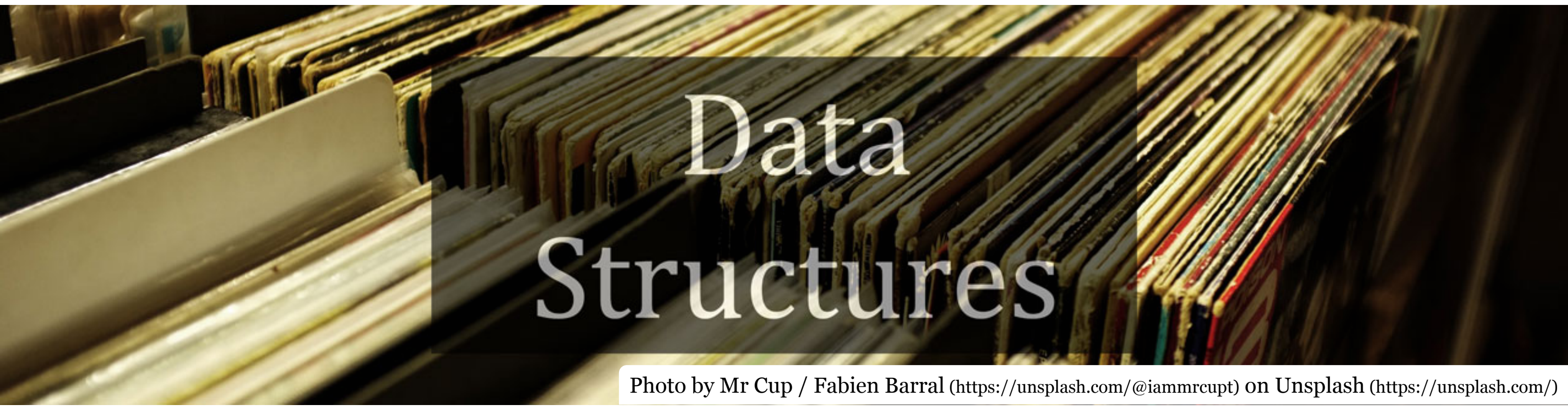

Photo by Mr Cup / Fabien Barral (https://unsplash.com/@iammrcupt) on Unsplash (https://unsplash.com/)

Data structures

7 minute read

Photo by Mr Cup / Fabien Barral (https://unsplash.com/@iammrcupt) on Unsplash (https://unsplash.com/)

In this lesson we will study how data can be collected and stored in memory. In particular we will deal with vectors, matrices, n-dimensional arrays, tuples and dictionaries.

Arrays

Vectors

A vector is a list of ordered data which share a common type (be it `Int` , `Float` or `Any`). Furthermore a vector is a one-dimensional array, and often “vector” and “array” are used a synonyms.

Contrarily to the mathematical definition of a vector, in programming a vector is simply a list of values and has no a priori geometrical meaning.

In Julia, to create a vector we use the following syntax:

```
a = [1,2,3,4,5]
b = [1.2, 3,4,5]
c = ["Hello", "it's me", "TechyTok"]
```

We can access the members of an array using the indexing syntax: `array_name[element_number]` , for example, if we want to retrieve the third element of `c` we type `c[3]` . Contrarily to other programming languages, in Julia **vectors start at 1**, there is not much to say, it is just like that.

If you want to append an element to an array, we can do it by using the `append!` function. Notice the `!` , this is a Julia convention to say that the function will modify the first argument given to the function, in fact `append!(a, 6)` results in `a = [1, 2, 3, 4, 5, 6]` .

Previously I have said that vector elements must share the same type, if we try to append (or add) a value to an array with a different type it will result into an error:

```
>>>append!(a, 3.14)
ERROR: InexactError: Int64(2.3)
Stacktrace:
 [1] Type at .\float.jl:703 [inlined]
 [2] convert at .\number.jl:7 [inlined]
 [3] setindex! at .\array.jl:766 [inlined]
 [4] _append!(::Array{Int64,1}, ::Base.HasShape{0}, ::Float64) at .\array.jl:907
 [5] append!(::Array{Int64,1}, ::Float64) at .\array.jl:899
 [6] top-level scope at none:0
```

In this case, we cannot convert a `Float64` to an `Int64` without losing precision, thus the error.

To check the type of an array we can type:

```
>>>typeof(a)
Array{Int64, 1}
```

As we can see, `a` can only store `Int64` values or values that can be safely converted to `Int64` (such as `Int32` for example).

An alternative way to define a Vector, if we want to specify the type of the elements included, is by using a type name followed by a square bracket and the desired elements:

```
d = Int[1,2,3,4,5]
```

Matrices

Matrices are two dimensional arrays. We can define a matrix with the following syntax:

```
mat1 = [1 2 3; 4 5 6]
```

Rows are separated by semi colons `;` and columns are separated by spaces.

When we want to access the elements of a matrix, we use the following notation `mat1[row_index, column_index]`. If we want to access the 2nd element of the first row we shall write `mat1[1, 2]`.

N-dimensional Arrays

Sometimes we need to create tables with more than 2 dimensions. In this case usually the tables tend to be big, so there is no explicit way to create an n-dimensional array. The suggested practice is to create an empty array first, using a new function called `zeros`, and then fill it either manually of using a loop.

For example, let's suppose we want to create a 2x3x4 table, we would do it with `zeros(2,3,4)`. Let's suppose we want to fill it with the product of the indexes, we can do it in the following way:

```
table = zeros(2,3,4)
for k in 1:4
    for j in 1:3
        for i in 1:2
            table[i,j,k] = i*j*k
        end
    end
end

>>>table
2×3×4 Array{Float64,3}:
 [:, :, 1] =
  1.0  2.0  3.0
  2.0  4.0  6.0

 [:, :, 2] =
  2.0  4.0   6.0
  4.0  8.0  12.0

 [:, :, 3] =
  3.0   6.0   9.0
  6.0  12.0  18.0

 [:, :, 4] =
  4.0   8.0  12.0
  8.0  16.0  24.0

```

Please not that Julia stores values in memory differently from Python: in Julia to obtain fast loops we need to iter first over columns (which means that the first index must vary first and so on). For this reason if we plan to store, for example, 42 2x2 matrices, we need to create an array of size 2x2x42 (while in Python we would have created a 42x2x2 table).

Slices

There is a convenient notation (called **slicing**) to access a subset of elements from an array. Let's suppose we want to access the 2nd to 5th elements of an array of length 6, we can do it in the following way:

```
a = [1,2,3,4,5,6]
b = a[2:5]
```

We can also use this notation to access a subset of a matrix, for example:

```
 1  mat1 = reshape([i for i in 1:16],4,4)
 2  mat2 = mat1[2:3, 2:3]
 3
 4  >>>mat1
 5  4×4 Array{Int64,2}:
 6   1  5   9  13
 7   2  6  10  14
 8   3  7  11  15
 9   4  8  12  16
10
11  >>>mat2
12  2×2 Array{Int64,2}:
13   6  10
14   7  11
```

On line 1 we have used a handy notation called **list comprehension**. `[i for i in 1:16]` means “create an array containing each `i` comprised from 1 to 16”. We then reshape it to have a size of 4x4 and store the result in `mat1`.

It is also possible to have nested list comprehensions like this one:

```
[i+j for i in 1:10 for j in 1:5]
```

This will create an array of length 50 containing all the sums of `i+j`.

Views

As in other programming languages, arrays are pointers to location in memory, thus we need to pay attention when we handle them.

If we create an array `a` and we assign `a` to `b` with `b=a`, the elements of `a` be modified by accessing `b`:

```
a=[1,2,3]
b=a
b[2] = 42
print(a)
```

This is particularly useful because it lets us save memory, but may have undesirable effects. If we want to make a copy of an array we need to use the function `copy`

```
a=[1,2,3]
b=copy(a)
b[2] = 42

>>>print(a)
[1,v2v,3]

>>>print(b)
[1, 42, 3]
```

In some cases, when there are arrays containing other arrays, if we want to make a full copy of all the contents we need to use `deepcopy` instead.

Tuples

A tuple is a fixed size group of variables which may share a common type but don't need to.

Unlike arrays, you cannot increase the size of a tuple once it has been created. Tuples are created using the following syntax:

```
a = (1,2,3)
b = 1, 2, 3
```

So tuples can be created by using regular brackets or no brackets at all! Tuples are really handy, as it is possible to "unpack" a tuple over many values:

```
tuple1 = (1, 2, 3)
a, b, c = tuple1

>>>print("$a $b $c")
1 2 3
```

It is also possible to use tuples to emulate multiple return values from functions:

```
function return_multiple()
    return 42, 43, 44
end

a, b, c = return_multiple()

>>>print("$a $b $c")
42 43 44
```

Splatting

It is possible to "unpack" a tuple and pass its arguments to a function with the following syntax:

```
function splat_me(a, b, c)
    return a*b*c
end

tuple1 = (1,2,3)

>>>splat_me(tuple1...)
6
```

So the `...` after a tuple will unpack it! This is useful but addictive, use it only if needed as it is better for clarity (and to avoid multiple dispatch errors) to call a function with its single parameters.

Named tuples

Named tuples are like tuples but with a name identifier for a single value, for example:

```
>>>namedTuple1 = (a = 1, b = "hello")
(a = 1, b = "hello")

>>>namedTuple1[:a]
1
```

or in alternative:

```
>>>namedTuple2 = NamedTuple{(:a, :b)}((2,"hello2"))
(a = 2, b = "hello2")

>>>namedTuple2[:b]
hello2
```

Dictionaries

A dictionary is a collection of keys and values. They are unordered (which means that the order of the keys is random) and are really useful when you need to organise, for example, a dataset.

Let's suppose we want to create an address book. A single entry should be able to store all the fundamental characteristics needed to identify a friend: the name of the contact, the phone number and the shoe size!

```
person1 = Dict("Name" => "Aurelio", "Phone" => 123456789, "Shoe-size" => 40)
person2 = Dict("Name" => "Elena", "Phone" => 123456789, "Shoe-size" => 36)
```

I can even make a dictionary containing other dictionaries:

```
addressBook = Dict("Aurelio" => person1, "Elena" => person2)
```

We can add another friend to the `addressBook` , once it has been created, with the following syntax:

```julia
person3 = Dict("Name" => "Vittorio", "Phone" => 123456789, "Shoe-size" => 42)
addressBook["Vittorio"] = person3

print(addressBook)
```

Conclusion

In this lesson we have learned how to operate on arrays, tuples and dictionaries. Those structures are the basics of in-memory storage for any data. Arrays are lightweight and useful solutions to pass blocks of data, so use them whenever needed! Contrarily to C++, Julia has a built in garbage collector, so you don't have to care about freeing memory and deleting pointers, as Julia will take care of it!

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

Tags: Julia

Categories: Lessons Tutorial

Updated: November 19, 2019

LEAVE A COMMENT