6b. Named Tuples and Dictionaries

Martin Alfaro
PhD in Economics

INTRODUCTION

Our previous discussions on collections have centered around vectors. Although we introduced the concept of tuples, haven't analyzed them depth. The current section fills that gap by offering a more comprehensive analysis of tuples. Additionally, we introduce two new types of collections: **named tuples** and **dictionaries**.

We'll also cover how to characterize collections through keys and values, methods for manipulating collections, and approaches to transforming one collection into another.

KEYS AND VALUES

Most collections in Julia are characterized by **keys**. They serve as unique identifiers for their elements and are paired with a corresponding **value**. [1] For instance, the vector `x = [2, 4, 6]` has the indices `1, 2, 3` as its keys, and `2, 4, 6` as their respective values.

Keys are more general than indices: while indices are limited to integer identifiers, keys can be any valid Julia object (e.g., strings, numbers, or other objects).

Julia provides the functions `keys` and `values` to extract the keys and values of a collection. The following code snippets demonstrate their usage with vectors and tuples, whose keys are represented by indices. Note that neither `keys` nor `values` return a vector, requiring the `collect` function to obtain a vector representation of the keys or values.

```
x = [4, 5, 6]
```

```
julia> collect(keys(x))
3-element Vector{Int64}:
 1
 2
 3
julia> collect(values(x))
3-element Vector{Int64}:
 4
 5
 6
```

```
x = (4, 5, 6)
```

```
julia> collect(keys(x))
3-element Vector{Int64}:
 1
 2
 3
julia> collect(values(x))
3-element Vector{Int64}:
 4
 5
 6
```

THE TYPE `PAIR`

Collections of key-value pairs in Julia are represented by the type `Pair{<key type>, <value type>}`. Although we won't directly work with objects of this type, they form the basis for constructing other collections such as dictionaries and named tuples.

A key-value pair can be created by using the operator `=>`, as in `<key> => <value>`. For instance, `"a" => 1` represents a pair, where `a` is the key and `1` the corresponding value. Alternatively, pairs can be created using the function `Pair(<key>, <value>)`, where `Pair("a",1)` is equivalent to the previous example.

Given a pair `x`, its key can be accessed via either `x[1]` or `x.first`. Likewise, its value is retrieved using either `x[2]` or `x.second`. All this is demonstrated below.

```
some_pair = ("a" => 1)      # or simply 'some_pair = "a" => 1'

some_pair = Pair("a", 1)    # equivalent
```

```
julia> some_pair
"a" => 1
julia> some_pair[1]
"a"
julia> some_pair.first
"a"
```


```
some_pair = ("a" => 1)      # or simply 'some_pair = "a" => 1'

some_pair = Pair("a", 1)    # equivalent
```


```
julia> some_pair
"a" => 1
julia> some_pair[2]
1
julia> some_pair.second
1
```

THE TYPE `SYMBOL`

The type used to represent keys may vary across collections. A commonly one used for keys is `Symbol`, which offers an efficient way to represent string-based identifiers. A symbol named `x` is written as `:x` and can be constructed from a string using the function `Symbol(<string>)`. [2]

```
vector_symbols = [:x, :y]
```

```
julia> vector_symbols
2-element Vector{Symbol}:
 :x
 :y
```

```
vector_symbols = [Symbol("x"), Symbol("y")]
```

```
julia> vector_symbols
2-element Vector{Symbol}:
 :x
 :y
```

NAMED TUPLES

> **Warning!**
>
> **Tuples and named tuples are only suitable for small collections**. Using them with large collections can result in poor performance or even fatal errors (such as stack overflows). For large collections, arrays remain the preferred choice.
>
> Defining what qualifies as *small* is challenging, and unfortunately there's no definitive answer. We can only indicate that collections with fewer than 10 elements are certainly small, while those exceeding 100 elements clearly exceed the intended use.

Named tuples share several properties with regular tuples, including their **immutability**. However, they also exhibit some notable differences. One important distinction is that **the keys of named tuples are objects of type** `Symbol`, in contrast to the numerical indices used for regular tuples.

Named tuples also differ syntactically, requiring being enclosed in parentheses `()`. Omitting them is not possible, unlike with regular tuples. Furthermore, when creating a single-element named tuple, the syntax requires either a trailing comma `,` after the element (similar to regular tuples) or a leading semicolon `;` before the element. [3]

To construct a named tuple, each element must be specified in the format `<key> = <value>`, such as `a = 10`. Alternatively, a pair `<key with Symbol type> => <value>` can be used, as in `:a => 10`. Once a named tuple `nt` is created, the element `a` can be accessed either by key lookup `nt[:a]` or by dot syntax `nt.a`.

The following code snippets illustrate these concepts.

```
# all 'nt' are equivalent
nt = (  a=10, b=20)
nt = (; a=10, b=20)

nt = (  :a => 10, :b => 10)
nt = (; :a => 10, :b => 10)
```

```
julia> nt
(a = 10, b = 20)
julia> nt.a
10
julia> nt[:a] #alternative way to access 'a'
10
```

```
# all 'nt' are equivalent
nt = (  a=10,)
nt = (; a=10 )

nt = (  :a => 10,)
nt = (; :a => 10 )

#not 'nt =  (a = 10)'  -> this is interpreted as 'nt = a = 10'
#not 'nt = (:a => 10)' -> this is interpreted as a pair
```

```
julia> nt
(a = 10, )
julia> nt.a
10
julia> nt[:a] #alternative way to access 'a'
10
```

> **Remark**
>
> To see the list of keys and values, we can employ the functions `keys` and `values`.

```
nt = (a=10, b=20)
```

```
julia> collect(keys(nt))
2-element Vector{Symbol}:
 :a
 :b
julia> values(nt)
(10, 20)
```

DISTINCTION BETWEEN THE CREATION OF TUPLES AND NAMED TUPLES

It's possible to create named tuples from existing variables. For instance, given variables `x = 10` and `y = 20`, one can define `nt = (; x, y)`. This creates a named tuple with keys `x` and `y`, and corresponding values `10` and `20`.

The semicolon `;` plays a crucial role in this construction, as it distinguishes named tuples from regular tuples. Omitting it, as in `nt = (x, y)`, would result in a regular tuple instead.

```
x   = 10
y   = 20

nt  = (; x, y)
tup = (x, y)
```

```
julia> nt
(x = 10, y = 20)
julia> tup
(10, 20)
```

```
x   = 10


nt  = (; x)
tup = (x, )
```

```
julia> nt
(x = 10,)
julia> tup
(10,)
```

DICTIONARIES

Dictionaries are collections of key-value pairs, exhibiting three distinctive features:

- **Dictionary keys can be any object**: strings, numbers, and other objects are possible.
- **Dictionaries are mutable**: elements can be modified, added, and removed after creation.
- **Dictionaries are unordered**: keys have no inherent order.

The function `Dict` can be used to create dictionaries, where each argument is a key-value pair written in the form `<key> => <value>`.

```
some_dict = Dict(3 => 10, 4 => 20)
```

```
julia> some_dict
Dict{Int64, Int64} with 2 entries:
  4 => 20
  3 => 10
julia> some_dict[1]
10
```

```
some_dict = Dict("a" => 10, "b" => 20)
```

```
julia> some_dict
Dict{String, Int64} with 2 entries:
  "b" => 20
  "a" => 10
julia> some_dict["a"]
10
```

```
some_dict = Dict(:a => 10, :b => 20)
```

```
julia> some_dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
julia> some_dict[:a]
10
```

```
some_dict = Dict((1,1) => 10, (1,2) => 20)
```

```
julia> some_dict
Dict{Tuple{Int64, Int64}, Int64} with 2 entries:
  (1, 2) => 20
  (1, 1) => 10
julia> some_dict[(1,1)]
10
```

Note that regular dictionaries are inherently unordered, meaning that the access to their elements doesn't follow any pattern. The following example illustrates this, by collecting the dictionary keys into a vector. [4]

```
some_dict      = Dict(3 => 10, 4 => 20)

keys_from_dict = collect(keys(some_dict))
```

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

```
some_dict      = Dict("a" => 10, "b" => 20)

keys_from_dict = collect(keys(some_dict))
```

```
julia> keys_from_dict
2-element Vector{String}:
 "b"
 "a"
```

```
some_dict      = Dict(:a => 10, :b => 20)

keys_from_dict = collect(keys(some_dict))
```

```
julia> keys_from_dict
2-element Vector{Symbol}:
 :a
 :b
```

```
some_dict      = Dict((1,1) => 10, (1,2) => 20)

keys_from_dict = collect(keys(some_dict))
```

```
julia> keys_from_dict
2-element Vector{Tuple{Int64, Int64}}:
 (1, 2)
 (1, 1)
```

CREATING TUPLES, NAMED TUPLES, AND DICTIONARIES

Tuples, named tuples, and dictionaries can be constructed from other collections. The only requirement is that the source collection possesses a key-value structure.

To demonstrate this possibility, we begin by creating **dictionaries** from a variety of collections.

```
vector = [10, 20] # or tupl = (10,20)



dict = Dict(pairs(vector))
```

```
julia> dict
Dict{Int64, Int64} with 2 entries:
  2 => 20
  1 => 10
```

```
keys_for_dict   = [:a, :b]
values_for_dict = [10, 20]


dict = Dict(zip(keys_for_dict, values_for_dict))
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
```

```
keys_for_dict   = (:a, :b)
values_for_dict = (10, 20)


dict = Dict(zip(keys_for_dict, values_for_dict))
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
```

```
nt_for_dict = (a = 10, b = 20)



dict = Dict(pairs(nt_for_dict))
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
```

```
keys_for_dict      = (:a, :b)
values_for_dict    = (10, 20)
vector_keys_values     =     [(keys_for_dict[i],     values_for_dict[i])     for     i     in
eachindex(keys_for_dict)]

dict = Dict(vector_keys_values)
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
```

Likewise, we can define a **tuple** from other collections, as shown below.

```
a               = 10
b               = 20

tup             = (a, b)
```

```
julia> tup
(10, 20)
```

```
values_for_tup = [10, 20]

tup             = (values_for_tup... ,)
```

```
julia> tup
(10, 20)
```

```
values_for_tup = [10, 20]

tup             = Tuple(values_for_tup)
```

```
julia> tup
(10, 20)
```

Finally, **named tuples** can also be constructed from other collections.

```
a = 10
b = 20


nt = (; a, b)
```

```
julia> nt
(a = 10, b = 20)
```

```
keys_for_nt   = [:a, :b]
values_for_nt = [10, 20]


nt = (; zip(keys_for_nt, values_for_nt)...)
```

```
julia> nt
(a = 10, b = 20)
```

```
keys_for_nt   = [:a, :b]
values_for_nt = [10, 20]


nt = NamedTuple(zip(keys_for_nt, values_for_nt))
```

```
julia> nt
(a = 10, b = 20)
```

```
keys_for_nt   = (:a, :b)
values_for_nt = (10, 20)


nt = NamedTuple(zip(keys_for_nt, values_for_nt))
```

```
julia> nt
(a = 10, b = 20)
```

```
keys_for_nt        = [:a, :b]
values_for_nt      = [10, 20]
vector_keys_values = [(keys_for_nt[i], values_for_nt[i]) for i in eachindex(keys_for_nt)]

nt = NamedTuple(vector_keys_values)
```

```
julia> nt
(a = 10, b = 20)
```

```
dict = Dict(:a => 10, :b => 20)



nt = NamedTuple(vector_keys_values)
```

```
julia> nt
(a = 10, b = 20)
```

DESTRUCTURING TUPLES AND NAMED TUPLES

Previously, we demonstrated how to create tuples and named tuples from variables. Next, we show that the reverse operation is also possible, where **values are extracted from a tuple or named tuple and assigned to separate variables**. This process is known as **destructuring**, enabbling users to "unpack" the values of a collection into distinct variables.

Destructuring involves the assignment operator `=` with either a tuple or named tuple on the left-hand side. The choice between one or the other determines what objects can be used on the right-hand side. Tuples on the left-hand side are quite flexible, allowing values to be unpacked from a variety of collections. Named tuples on the left-hand side, instead, necessarily require a named tuple on the right-hand side. Next, we develop each case separately.

DESTRUCTURING COLLECTIONS THROUGH TUPLES

Given a collection `list` with two elements, destructuring via tuples allows us to unpack its values into the variables `x` and `y`. The syntax for this is `<tuple> = <collection>`, as in `x,y = list`. In the following, we illustrate the process by considering different objects as `list`.

```
list = [3,4]

x,y  = list
```

```
julia> x
3
julia> y
4
```

```
list = 3:4

x,y  = list
```

```
julia> x
3
julia> y
4
```

```
list = (3,4)

x,y  = list
```

```
julia> x
3
julia> y
4
```

```
list = (a = 3, b = 4)

x,y  = list
```

```
julia> x
3
julia> y
4
```

In addition to unpacking all elements, destructuring can also be applied to only a subset of elements. The assignment is then performed in sequential order, following the collection's inherent order.

Importantly, this method excludes the possibility of skipping any specific value. When a value must be disregarded, the conventional approach is to bind this value to the special variable name `_`. This symbol serves purely as a placeholder to indicate that the value is unimportant and has no impact on execution.

For illustration, we'll use a vector as an example of `list`. Nonetheless, the same principle applies to any collection.

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

(x,)  = list
```

```
julia> x
3
```

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

x,y   = list
```

```
julia> x
3
julia> y
4
```

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

_,_,z = list                  # _ skips the assignment of that value
```

```
julia> z
5
```

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

x,_,z = list                  # _ skips the assignment of that value
```

```
julia> x
3
julia> z
5
```

DESTRUCTURING WITH NAMED TUPLES ON BOTH SIDES

Destructuring can also be applied with named tuples on the left-hand side. In this case, values are extracted by directly referencing field names, rather than relying on their positional order. The main advantage of this approach is that variables can be assigned in any order, provided their names correspond to some field in the named tuple.

```
nt             = (; key1 = 10, key2 = 20, key3 = 30)

(; key2, key1) = nt            # keys in any order
```

```
julia> key1
10
julia> key3
30
```

```
nt             = (; key1 = 10, key2 = 20, key3 = 30)

(; key3)       = nt            # only one key
```

```
julia> key3
30
```

Remark

When destructuring with a tuple on the left-hand side and a named tuple on the right-hand side, keep in mind that the assignment is carried out strictly by position. This means that variable names on the left don't influence the assignment operation. In other words, the keys of the named tuple are completely ignored during the process.

```
nt = (; key1 = 10, key2 = 20, key3 = 30)

 key2, key1    = nt             # variables defined according to
POSITION
(key2, key1)   = nt            # alternative notation
```

```
julia> key2
10
julia> key1
20
```

```
nt = (; key1 = 10, key2 = 20, key3 = 30)

(; key2, key1) = nt             # variables defined according to
KEY
 ; key2, key1  = nt            # alternative notation
```

```
julia> key1
10
julia> key2
20
```

The same caveat applies to assignments of single variables.

```
nt       = (; key1 = 10, key2 = 20)

(key2,)   = nt                 # variable defined according to
POSITION
```

```
julia> key2
10
```

```
nt       = (; key1 = 10, key2 = 20)

(; key2) = nt            # variable defined according to KEY
```

```
julia> key2
20
```

APPLICATIONS OF DESTRUCTURING

Destructuring named tuples is particularly valuable in scientific modelling, where numerous parameters are referenced repeatedly. By grouping all these parameters into a single named tuple, they can be passed to a function as *one* consolidated argument. When functions are defined following this procedure, the named tuple is then destructed at the beginning of the function body to extract the needed parameters.

```
β = 3
δ = 4
ϵ = 5

# function 'foo' only uses 'β' and 'δ'
function foo(x, δ, β)
    x * δ + exp(β) / β
end
```

```
julia> foo(2, δ, β)
14.695
```

```
parameters_list = (; β = 3, δ = 4, ϵ = 5)



# function 'foo' only uses 'β' and 'δ'
function foo(x, parameters_list)
    x * parameters_list.δ + exp(parameters_list.β) / parameters_list.β
end
```

```
julia> foo(2, parameters_list.β, parameters_list.δ)
14.695
```

```
parameters_list = (; β = 3, δ = 4, ϵ = 5)



# function 'foo' only uses 'β' and 'δ'
function foo(x, parameters_list)
    (; β, δ) = parameters_list

    x * δ + exp(β) / β
end
```

```
julia> foo(2, parameters_list)
14.695
```

Destructuring also provides an elegant solution for retrieving multiple outputs from a function. This makes it possible to unpack the returned outputs into separate variables. In the example below, the function `foo` returns tuple, which is then unpacked into variables `x`, `y`, and `z`.

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    out1, out2, out3
end

x, y, z = foo()
```

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    [out1, out2, out3]
end

x, y, z = foo()
```

Another common use of destructuring arises when only a subset of a function's outputs is needed. While both tuples and named tuples can be applied for this purpose, tuples offer greater flexibility as they can be combined with various types of collections. In contrast, named tuples are restricted to returning another named tuple as the function's output, thus requiring prior knowledge of the field names.

The following example illusates this functionality by extracting only the first and third output of `foo`.

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    out1, out2, out3
end

x, _, z = foo()
```

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    [out1, out2, out3]
end

x, _, z = foo()
```

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    (; out1, out2, out3)
end

(; out1, out3) = foo()
```

FOOTNOTES

[1.] Not all collections map keys to values. For example, the type `Set`, which represents a group of unique unordered elements, doesn't have a key-value structure.

[2.] `Symbol` also enables the programmatic creation of variables. A typical use case arises in data analysis, where symbols are employed to generate new columns in tabular data structures.

[3.] The semicolon notation `;` may seem odd, but it actually comes from the syntax for keyword arguments in functions.

[4.] The package `OrderedCollections` addresses this, by offering a special dictionary called `OrderedDict`. It behaves similarly to regular dictionaries, including their syntax, but endows the dictionary with an order.