6b. Named Tuples and Dictionaries

Martin Alfaro
PhD in Economics

INTRODUCTION

Our previous discussions on collections have centered around vectors and tuples. The current section expands on the subject 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 the manipulation of 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 of their elements, and have a corresponding **value** associated with each. [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—they encompass all the possible identifiers of a collection's elements (e.g., strings, numbers, or other objects). Instead, indices exclusively employ integers as identifiers.

To identify the keys and values of a collection, Julia offers the functions `keys` and `values`. The following code snippets demonstrate their usage based on vectors and tuples, whose keys are represented by indices. Note that neither `keys` nor `values` return a vector, requiring the `collect` function for this purpose.

```
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
```

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

```
julia> collect(keys(x))
"a" => 1
julia> collect(values(x))
1
```

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 its key and `1` its corresponding value. In addition, we can create pairs through the function `Pair(<key>, <value>)`, making `Pair("a",1)` equivalent to the previous example. Finally, given a pair `x`, its key can be accessed by either `x[1]` or `x.first`, while its value can be retrieved using `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 can vary depending on the collection. An important type used as a key is `Symbol`, which provides an efficient way to represent string-based keys. A symbol labeled `x` is denoted `:x`, and can be created from strings using the function `Symbol(<some string>)`. [2]

```
x = (a=4, b=5, c=6)
```

```
julia> some_symbol
```

```
julia> vector_symbols
3-element Vector{Symbol}:
 :a
 :b
 :c
```

NAMED TUPLES

> **Warning!**
>
> **Tuples and named tuples should only be used for small collections**. Using them for large collections can lead to slow operations or directly result in a fatal error (the so-called stack overflow). Arrays remain the preferred choice for large collections.
>
> Defining what constitutes *small* is challenging, and unfortunately there's no definitive answer. We can only indicate that collections with fewer than 10 elements are undoubtedly small, while those exceeding 100 elements violate the definition.

Named tuples share several similarities with regular tuples, including their **immutability**. However, they also exhibit some notable differences. One of them 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 named tuple with a single element, the notation 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, you can use a pair `<key with Symbol type> => <value>`, as in `:a => 10`. Once a named tuple `nt` is created, you can access its element `a` by using either `nt[:a]` or `nt.a`.

The following code snippets illustrate all this.

```
# 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 individual variables. For instance, given variables `x = 10` and `y = 20`, you can define `nt = (; x, y)`. This creates a named tuple with keys `x` and `y`, and corresponding values `10` and `20`.

The semicolon `;` is crucial 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:

- **The keys of dictionaries can be any object**: strings, numbers, and other objects are possible.
- **Dictionaries are mutable**: you can modify, add, and remove elements.
- **Dictionaries are unordered**: keys don't have any order attached.

Dictionaries are created using the function `Dict`, with each argument consisting of a key-value pair denoted by `<key> => <value>`.

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

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

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

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

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

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

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

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

Note that regular dictionaries are inherently unordered, meaning access to their elements doesn't follow any pattern. The following example illustrates this, where a vector collects the keys of a dictionary. [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, provided that the source collection possesses a key-value structure. The following examples demonstrate how various collections can be used to create **dictionaries** in particular.

```
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've demonstrated how to create a tuple and a named tuple 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 individual variables**. This process is known as **destructuring**, and allows users to "unpack" the values of a collection into separate variables.

Destructuring involves the assignment operator `=`, with a tuple or named tuple on the left-hand side. The key difference between using or the other lies in their compatibility with other collections: named tuples on the left-hand side require a matching named tuple on the right-hand side, whereas tuples can be paired with several forms of collection. We illustrate each case below.

DESTRUCTURING COLLECTIONS THROUGH TUPLES

Given a collection `list` with two elements, destructuring enables the creation of variables `x` and `y` with the values of `list`. This is implemented by the syntax `<tuple> = <collection>`, such as `x,y = list`. The following illustrates this 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 destructuring all elements in `list`, it's possible to destructure 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. If you still need to disregard a value, it's common to use the special variable name `_` as a placeholder. This is merely a convention to indicate that the value is unimportant, without any impact on execution.

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

```
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 also allows for named tuples on the left-hand side. This approach extracts values by directly referencing field names, rather than relying on their positional order. Its key advantage is that values can be assigned to variables in any order, provided their names correspond to some field name of 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 tuple assignments are strictly positional. This means that variable names don't influence the assignment operation, and therefore completely ignores whether the variable names match the keys of the named tuple.

```
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 single-variable assignments.

```
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 especially useful in scientific models, where numerous parameters may be used repeatedly. By storing all these parameters in a named tuple, you can pass a *single* argument to functions. Then, destructuring the named tuple at the beginning of the function body enables the extraction of 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
```

Another useful application of destructuring occurs when retrieving multiple outputs from a function. In this way, you can store each result in a separate variable. Below, we illustrate this application using a tuple and 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 application of destructuring is 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 output's field names.

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

```
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 keys.

[2.] `Symbol` also enables the creation of variables programmatically. For example, it can be employed for defining new columns in the package `DataFrames`, which provides a table representation of data.

[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.