3e. Map and Broadcasting

Martin Alfaro
PhD in Economics

INTRODUCTION

This section explores element-wise operations on **iterable collections**. A collection is characterized as iterable when its elements can be accessed sequentially, thus including examples like vectors, tuples, and ranges.

The first approach covered is the `map` function, which applies a given function to each element of a collection. This function is particularly convenient for avoiding for-loops when transforming collections.

After this, we'll shift our focus to a fundamental technique in Julia known as **broadcasting**. This enables the application of functions and operators element-wise, while maintaining concise and expressive code. Broadcasting is quite versatile, supporting operations on collections of equal size or combinations of scalars and same-size collections. Its distinctive syntax, which involves appending a dot `.` to the function/operator, makes it easily identifiable throughout the code.

> **Remark**
>
> The terms **broadcasting** and **vectorization** will be used interchangeably throughout the website, although strictly speaking they're not equivalent. [1] Furthermore, vectorization has multiple meanings, depending on the context in which the definition is applied.

> **Warning!**
>
> Later on the website, we'll explore **for-loops** as an alternative approach to transforming arrays. Several languages strongly recommend vectorizing operations to improve speed, instead highly discouraging for-loops. **Such advice does not apply to Julia**. In fact, when it comes to optimizing code in Julia, for-loops are often the key to achieving faster performance.
>
> Considering this, the main advantage of vectorization in Julia is to streamline code without sacrificing speed.

THE "MAP" FUNCTION

The `map` function is available in most programming languages, allowing you to take a collection and generate a new one with transformed elements. It can be applied in two ways, depending on the number of inputs passed.

In its simplest form, `map` takes a single-argument function `foo` and a collection `x`. Its syntax is `map(foo,x)`, returning a new collection with `foo(x[i])` as *i*-th element. `map` is commonly applied with an anonymous function playing the role of `foo`, as illustrated below.

```
x = [1, 2, 3]


z = map(log,x)
```

```
julia> z
3-element Vector{Float64}:
 0.0
 0.69315
 1.09861
julia> [log(x[1]), log(x[2]), log(x[3])]
3-element Vector{Float64}:
 0.0
 0.69315
 1.09861
```

```
x = [1, 2, 3]


z = map(a -> 2 * a, x)
```

```
julia> z
3-element Vector{Int64}:
 2
 4
 6
julia> [2*x[1], 2*x[2], 2*x[3]]
3-element Vector{Int64}:
 2
 4
 6
```

The second way to apply `map` arises when the function `foo` takes multiple arguments. In case `foo` is a two-argument function, the syntax is `map(foo, x, y)`, returning a new collection whose *i*-th element is `foo(x[i], y[i])`. When the collections `x` and `y` have different sizes, `foo` **is applied element-wise until the shortest collection is exhausted**. This rule applies even when either `x` or `y` is a scalar, in which case `map` would return a single element.

For demonstrating its use, let's consider the addition operation. As you may recall, `+` denotes both an operator (e.g., `2 + 3`) and a function (e.g., `+(2, 3)`). By using `+` in particular as a function, `map` can perform element-wise additions across multiple collections.

```
x = [ 1, 2, 3]
y = [-1,-2,-3]

z = map(+, x, y)        # recall that `+` exists as both operator and function
```

```
julia> z
3-element Vector{Int64}:
 0
 0
 0
julia> [+(x[1],y[1]), +(x[2],y[2]), +(x[3],y[3])]
3-element Vector{Int64}:
 0
 0
 0
```

```
x = [ 1, 2, 3]
y = [-1,-2,-3]

z = map((a,b) -> a+b, x, y)
```

```
julia> z
3-element Vector{Int64}:
 0
 0
 0
julia> [x[1]+y[1], x[2]+y[2], x[3]+y[3]]
3-element Vector{Int64}:
 0
 0
 0
```

```
x = [ 1, 2, 3]
y = [-1,-2]

z = map(+, x, y)        # recall that `+` is both an operator and a function
```

```
julia> z
2-element Vector{Int64}:
 0
 0
julia> [+(x[1],y[1]), +(x[2],y[2])]
2-element Vector{Int64}:
 0
 0
```

```
x = [ 1, 2, 3]
y =  -1

z = map(+, x, y)        # recall that `+` is both an operator and a function
```

```
julia> z
1-element Vector{Int64}:
 0
julia> [+(x[1],y[1])]
1-element Vector{Int64}:
 0
```

BROADCASTING

The function `map` can rapidly become unwieldy when dealing with complex functions or multiple arguments. This is where broadcasting comes into play, offering a more streamlined syntax.

Next, we'll explore the concept of broadcasting in a step-by-step manner. First, we'll show how it applies to collections of equal size, covering both functions and operators. After this, we'll demonstrate that broadcasting accepts combinations of scalars and collections, even though it typically doesn't support operations with collections of different sizes. In such instances, the scalar is treated as a vector that matches the size of the corresponding collections.

Unlike other programming languages, **broadcasting is an intrinsic feature of Julia** and thereby applicable to *any* function or operator, including user-defined ones.

BROADCASTING FUNCTIONS

Broadcasting expands the versatility of functions, allowing them to be applied element-wise to a collection. This feature is implemented by appending a dot ***after*** the name of the function, as in `foo.(x)`.

Remarkably, **any function `foo` has a broadcasting counterpart `foo.`**. This entails that broadcasting is automatically available for user-defined functions. Furthermore, it determines that broadcasting isn't restricted to numeric collections, but to any type of collection.

Similarly to `map`, broadcasting can be applied to both single- and multiple-argument functions. Each case warrants separate consideration.

As for single-argument functions, broadcasting `foo` over a collection `x` returns a new collection with `foo(x[i])` as its *i*-th element. The following examples demonstrate this.

```
# `log(a)` is a function appying to scalars `a`

x         = [1,2,3]
```

```
julia> log.(x)
3-element Vector{Float64}:
 0.0
 0.69315
 1.09861
julia> [log(x[1]), log(x[2]), log(x[3])] # identical to log.(x)
3-element Vector{Float64}:
 0.0
 0.69315
 1.09861
```

```
square(a) = a^2     #user-defined function for a single element 'a'

x         = [1,2,3]
```

```
julia> square.(x)
3-element Vector{Int64}:
 1
 4
 9
julia> [square(x[1]), square(x[2]), square(x[3])] # identical to square.(x)
3-element Vector{Int64}:
 1
 4
 9
```

As for multiple-argument functions, suppose a function `foo` and collections `x` and `y`. Then, `foo.(x,y)` returns a new collection with `foo(x[i],y[i])` as its *i*-th element.

Importantly, **collections with different sizes aren't allowed**, establishing a clear contrast between broadcasting and `map`. The sole exception to this rule is when one of the objects is a scalar, as we'll see later.

Below, we provide several examples. The first example in particular makes use of the built-in function `max`, which provides the maximum value among its scalar arguments.

```
# 'max(a,b)' returns 'a' if 'a>b', and 'b' otherwise

x        = [0, 4, 0]
y        = [2, 0, 8]
```

```
julia> max.(x,y)
3-element Vector{Float64}:
 2
 4
 8
julia> [max(x[1],y[1]), max(x[2]),y[2]), max(x[3]),y[3])] # identical to max.(x,y)
3-element Vector{Float64}:
 2
 4
 8
```

```
foo(a,b) = a + b        # user-defined function for single elements 'a' and 'b'

x        = [-2, -4, -10]
y        = [ 2,  4,  10]
```

```
julia> foo.(x)
3-element Vector{Int64}:
 0
 0
 0
julia> [foo(x[1],y[1]), foo(x[2]),y[2]), foo(x[3]),y[3])] # identical to foo.(x,y)
3-element Vector{Float64}:
 0
 0
 0
```

Remark

Broadcasting applies not only to numeric functions, but to any function. For instance, consider the built-in function `string`, which concatenates its arguments to form a sentence (e.g., `string("hello ","world")` returns `"hello world"`).

```
country = ["France", "Canada"]
is_in   = [" is in "  , " is in "]
region  = ["Europe", "North America"]
```

```
julia> string.(country, is_in, region)
2-element Vector{String}:
 "France is in Europe"
 "Canada is in North America"
```

BROADCASTING OPERATORS

It's also possible to **broadcast operators**, making them apply element-wise. Its use requires prepending a dot ***before*** the operator.

For its application, it's helpful to recall the classification of operators by the number of operands, as this determines their syntax. Specifically, the syntax of *unary operators* is `<symbol>x`, so that `.√x` broadcasts `√`. Likewise, the syntax for *binary operators* is `x <symbol> y`, such that `x .+ y` computes the element-wise sum of vectors `x` and `y`, resulting in `[x[1]+y[1], x[2]+y[2], ...]`.

```
x = [ 1,  2,  3]
y = [-1, -2, -3]
```

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

```
x = [1, 2, 3]
```

```
julia> .√x
3-element Vector{Float64}:
 1.0
 1.41421
 1.73205
```

BROADCASTING OPERATORS WITH SINGLE-ELEMENT OBJECTS

In all the cases covered so far, broadcasting was applied with inputs of the same size. In general, collections of dissimilar size, such as `x = [1,2]` and `y=[3,4,5]`, aren't allowed.

One exception to this rule occurs when broadcasting applies to vectors of equal size combined with scalars. In these cases, scalars are treated as objects having the same size as the vectors, with all entries equal to the scalar. For example, given `x = [1,2,3]` and `y = 2`, the expression `x .+ y` produces the same result as defining `y = [2,2,2]` and then executing `x .+ y`. This is demonstrated below.

```
x = [0,10,20]
y = 5
```

```
julia> x .+ y
3-element Vector{Int64}:
 5
 15
 25
```

```
x = [0,10,20]
y = [5, 5, 5]
```

```
julia> x .+ y
3-element Vector{Int64}:
 5
 15
 25
```

Remark

We emphasize that broadcasting can be applied to any iterable collection. Thus, the example based on strings presented above can be rewritten as follows.

```
country = ["France", "Canada"]
is_in   = " is in "
region  = ["Europe", "North America"]
```

```
julia> string.(country, is_in, region)
2-element Vector{String}:
 "France is in Europe"
 "Canada is in North America"
```

ITERABLE OBJECTS

So far, our examples have focused on broadcasting using vectors as collections. Furthermore, we've explored the technique by treating functions and operators separately, which sheds light on the underlying mechanics of broadcasting. Next, we'll take a more comprehensive perspective, applying broadcasting to other types of collections and to expressions combining functions and operators.

We first show that broadcasting can be applied to any iterable object, including tuples and ranges.

```
x = (1, 2, 3)    # or simply x = 1, 2, 3
```

```
julia> log.(x)
(0.0, 0.69315, 1.09861)
julia> x .+ x
(2, 4, 6)
```

```
x = 1:3
```

```
julia> log.(x)
(0.0, 0.69315, 1.09861)
julia> x .+ x
(2, 4, 6)
```

```
x = (1, 2, 3)    # or simply x = 1, 2, 3
y = 1:3
```

```
julia> x .+ y
(2, 4, 6)
```

Furthermore, it's possible to simultaneously broadcast operators and functions. Given the pervasiveness of such operations, Julia provides the macro `@.` for an effortless application. The macro should be added at the beginning of the statement, and has the effect of automatically adding a "dot" to each operator and function found.

To demonstrate its use, consider adding two vectors element-wise, which we then transform by squaring the elements of the resulting vector.

```
x = [1, 0, 2]
y = [1, 2, 0]

temp = x .+ y
z    = temp .^ 2
```

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

```
x = [1, 0, 2]
y = [1, 2, 0]

square(x) = x^2
```

```
julia> square.(x .+ y)
3-element Vector{Int64}:
 4
 4
 4
```

```
x = [1, 0, 2]
y = [1, 2, 0]

square(x) = x^2
```

```
julia> @. square(x + y)
3-element Vector{Int64}:
 4
 4
 4
```

BROADCASTING FUNCTIONS VS BROADCASTING OPERATORS

We've demonstrated that both functions and operators can be broadcasted. This lets us implement operations in two distinct ways: either broadcast a function that operates on a single element or define a function that directly performs the broadcasted operation.

The examples below demonstrate that the same output is obtained using either approach. For the illustration, we suppose that the goal is to square each element of `x`.

```
x                = [1, 2, 3]

number_squared(a) = a ^ 2          # function for a single element 'a'
```

```
julia> number_squared.(x)
3-element Vector{Int64}:
 1
 4
 9
```

```
x                = [1, 2, 3]

vector_squared(x) = x .^ 2          # function for a vector 'x'
```

```
julia> vector_squared(x) # '.' not needed (it'd be redundant)
3-element Vector{Int64}:
 1
 4
 9
```

While both approaches yield the same output, **defining a function that operates on a scalar is the more advisable choice**. This is due to a couple of reasons. Firstly, a function like `number_squared(a)` enables users to seamlessly perform computations on both scalars and collections. This is achieved by simply choosing between executing the function or its broadcasted version. A corollary of this is that scalar functions avoid committing to a specific application. Secondly, the notation `number_square.(x)` explicitly conveys that the operation is element-wise, an aspect that would remain hidden in `vector_squared(x)`.

BROADCASTING OVER ONLY ONE ARGUMENT

When we broadcast a function or operator over some vectors `x` and `y`, both objects are simultaneously iterated. However, there are instances where we only want to iterate over one argument, keeping the other argument fixed.

A typical scenario is when we need to check whether elements from `x` match any values in a predefined list `y`. To illustrate this, we first introduce the function `in(a, list)`. This assesses whether the scalar `a` equals some element in the vector `list`. For instance, executing `in(2, [1,2,3])` returns `true`, because `2` belongs to `[1,2,3]`.

Suppose now that, instead of a scalar `a`, we have a vector `x`. The goal then is to verify whether *each* of the elements in `x` is present in `list = [1,2,3]`. Below, we show that this operation can't be directly implemented by broadcasting `in`.

```
x    = [1, 2]
list = [1, 2, 3]
```

```
julia> in.(x, list)
ERROR: DimensionMismatch: arrays could not be broadcast to a common size; got a dimension
with lengths 2 and 3
```

```
x    = [1, 2, 4]
list = [1, 2, 3]
```

```
julia> in.(x, list)
3-element BitVector:
 1
 1
 0
```

In the first example, `in.(x, list)` errors because `x` and `list` should either have the same size or one of them be a scalar. The second example does produce an output, but not the one we're looking for: it checks whether `1==1`, `2==2`, and `4==3`. Instead, our goal is to determine if `1` is in `[1,2,3]`, if `2` is in `[1,2,3]`, and if `3` is in `[1,2,3]`.

Intuitively, we need a mechanism to inform Julia that `list` should be treated as a single element while iterating over `x`. This can be accomplished in two different ways: either by enclosing `list` in a collection (e.g., a vector or tuple) or by using the `Ref` function.

As for the first approach, let's consider a tuple as the wrapping collection. Then, the implementation would require `(list,)`, which converts the variable into a tuple whose only element is the tuple itself. [2] Explaining the specifics of `Ref` is beyond our current scope. What matters for practical purposes is that `Ref(list)` makes `list` be treated as a singe element. Below, we demonstrate each approach.

```
x    = [2, 4, 6]
list = [1, 2, 3]        # 'x[1]' equals the element 2 in 'list'
```

```
julia> in.(x, [list])
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]        # 'x[1]' equals the element 2 in 'list'
```

```
julia> in.(x, (list,))
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]        # 'x[1]' equals the element 2 in 'list'
```

```
julia> in.(x, Ref(list))
3-element BitVector:
 1
 0
 0
```

The output vector we obtain in each case is what's known as a `BitVector`, where `1` corresponds to `true` and `0` to `false`. Therefore, the result is `[true, false, false]`, reflecting that `x[1]` is `2` and `2` belongs to `list`, whereas `x[2]` and `x[3]` don't equal any element in `list`.

> **Warning!**
>
> It's possible to use any collection to wrap `list`. However, we'll see in Part II of the book that there's some performance penalty involved when vectors are created. Consequently, **you should stick to `(list,)` rather than `[list]` when implementing this approach**.

While the previous example focused on the broadcasting of functions, the same principle applies to operators. This can be illustrated through the `∈` operator, which serves a similar purpose to the `in` function. Just like `in`, the `∈` operator determines whether a particular element exists within a collection. [3]

```
x    = [2, 4, 6]
list = [1, 2, 3]
```

```
julia> x .∈ (list,) # only 'x[1]' equals an element in 'list'
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]
```

```
julia> x .∈ Ref(list) # only 'x[1]' equals an element in 'list'
3-element BitVector:
 1
 0
 0
```

CURRYING AND FIXING ARGUMENTS (*OPTIONAL*)

Currying is a technique that transforms the evaluation of a function with multiple arguments into evaluating a sequence of functions, each with a single argument. [4] For instance, the curried version of `f(x,y)` would be written `f(x)(y)` and provide an identical output.

Our interest in currying lies in its ability to simplify broadcasting: it enables the treatment of an argument as a single object, without the need to use `Ref` or encapsulate objects as vectors/tuples. The technique could seem confusing for new users. In particular, it requires a good understanding of functions as first-class objects, entailing that functions can be treated as variables themselves. My primary goal is that you can at least recognize the syntax of currying, and thus be able to read code that applies the technique.

We start by illustrating how currying can be applied in general.

```
addition(x,y) = 2 * x + y
```

```
julia> addition(2,1)
5
```

```
addition(x,y) = 2 * x + y

# the following are equivalent
curried(x)    = (y -> addition(x,y))
curried       = x -> (y -> addition(x,y))
```

```
julia> curried(2)(1)
5
```

```
addition(x,y) = 2 * x + y
curried(x)    = (y -> addition(x,y))

# the following are equivalent
f             = curried(2)          # function of 'y', with 'x' fixed to 2
g(y)          = addition(2,y)
```

```
julia> f(1)
5
julia> g(1)
5
```

The key to understanding the syntax is that `curried(x)` is a function itself, with `y` as its argument. The second tab illustrates this clearly through the equivalence between `f = curried(2)` and `addition(2,y)`. These functions help us understand the logic behind curry, but are only useful for the specific case of `x=2`. Instead, `curried(x)` allows the user to call the function through `curried(x)(y)`, and so be used for any `x`.

As for broadcasting, any function `foo` in Julia can be broadcasted through `f.`. And we've determined that `curried(x)` is a function just like any other. Therefore, `curried(x)` plays the same role as `foo`, and so we can broadcast over `y` for a fixed `x` through `curried(x).(y)`.

```
a             = 2
b             = [1,2,3]

addition(x,y) = 2 * x + y
curried(x)    = (y -> addition(x,y))   # 'curried(x)' is a function, and 'y' its argument
```

```
julia> curried(a).(b)
3-element Vector{Int64}:
 5
 6
 7
```

```
a             = 2
b             = [1,2,3]

addition(x,y) = 2 * x + y
curried(x)    = (y -> addition(x,y))

#the following are equivalent
f    = curried(a)             # 'foo1' is a function, and 'y' its argument
g(y) = addition(2,y)
```

```
julia> f.(b)
3-element Vector{Int64}:
 5
 6
 7
julia> g.(b)
3-element Vector{Int64}:
 5
 6
 7
```

Let's now explore how the currying technique can help treat a vector as a single element in broadcasting. To illustrate this, consider the function `in` used previously. This function has a built-in curried version, which can be applied through `in(list).(x)` for vectors `list` and `x`. To better demonstrate its usage, the following example compares an implementation with `Ref`, the built-in curried `in`, and our own curry implementation.

```
x    = [2, 4, 6]
list = [1, 2, 3]
```

```
julia> in.(x,Ref(list))
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]

our_in(list_elements) = (x -> in(x,list_elements))     # 'our_in(list_elements)' is a
function
```

```
julia> our_in(list).(x) # it broadcasts only over 'x'
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]
```

```
julia> in(list).(x) # similar to 'our_in'
3-element BitVector:
 1
 0
 0
```

FOOTNOTES

[1.] Vectorization refers to applications restricted to arrays of the same size, with broadcasting being an extension of it that allows for scalars.

[2.] Recall that tuples with a single element must be written with a trailing comma, as in `(list,)`. Instead, `(list)` is interpreted as `list`, and hence treated as a vector.

[3.] `∈` can also be applied as a function, with its syntax mirroring that of `in`. Thus, `∈(a, list)` for a scalar `a` yields the same results as `in(a, list)`.

[4.] The name comes from the mathematician Haskell Curry, not the spice!