3e. Map and Broadcasting

Martin Alfaro
PhD in Economics

INTRODUCTION

This section introduces operations on **iterable collections**, a class of data structures whose elements can be accessed sequentially. Common examples include vectors, tuples, and ranges.

A general way to process iterable collections is by using for-loops, which we'll study later. For now, our focus is on techniques that allow us to apply operations element-wise, without explicitly writing for-loops.

The first approach covered is the `map` function. This applies a given function to each element of a collection, producing a new collection with the transformed values. After this, we'll shift our focus to a fundamental technique in Julia known as **broadcasting**. Its distinctive syntax, which involves appending a dot `.` to the function/operator, makes it easily identifiable throughout the code.

Broadcasting enables the application of functions and operators element-wise. When two collections of the same size are involved, it pairs corresponding elements and applies the operation to each pair. Broadcasting also supports combinations of scalars and same-size collections, where the scalar is expanded to match the collection's size.

> **Broadcasting vs Vectorization**
>
> The terms **broadcasting** and **vectorization** will be used interchangeably throughout the website. Strictly speaking, they're not equivalent: the term vectorization applies when arrays have the same size, while broadcasting is an extension that allows for scalars.

> **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, for-loops are often the key to achieving high 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, although with different names. It's designed to transform collections by applying a function to each of their elements.

Depending on the number of inputs required, `map` can be applied in two different ways. In its simplest form, it takes a single-argument function `foo` and a collection `x`. The syntax is then `map(foo,x)`, returning a new collection with `foo(x[i])` as *i*-th element. A common practice is to use anonymous functions in place of `foo`, as illustrated below.

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

output     = map(log, x)
equivalent = [log(x[1]), log(x[2]), log(x[3])]
```

```
julia> output
3-element Vector{Float64}:
 0.0
 0.693147
 1.09861
julia> equivalent
3-element Vector{Float64}:
 0.0
 0.693147
 1.09861
```

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

output     = map(a -> 2 * a, x)
equivalent = [2*x[1], 2*x[2], 2*x[3]]
```

```
julia> output
3-element Vector{Int64}:
 2
 4
 6
julia> equivalent
3-element Vector{Int64}:
 2
 4
 6
```

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

To demonstrate its use, let's consider the sum operation. Recall that `+` 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]

output     = map(+, x, y)        # `+` exists as both operator and function
equivalent = [+(x[1],y[1]), +(x[2],y[2]), +(x[3],y[3])]
```

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

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

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

output     = map((a,b) -> a+b, x, y)
equivalent = [x[1]+y[1], x[2]+y[2], x[3]+y[3]]
```

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

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

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

output     = map(+, x, y)        # `+` exists as both operator and function
equivalent = [+(x[1],y[1]), +(x[2],y[2])]
```

```
julia> output
2-element Vector{Int64}:
 0
 0

julia> equivalent
2-element Vector{Int64}:
 0
 0
```

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

output     = map(+, x, y)        # `+` exists as both operator and function
equivalent = [+(x[1],y[1])]
```

```
julia> output
1-element Vector{Int64}:
 0
julia> equivalent
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 combinations of scalars and collections, despite not supporting operations with collections of different sizes. In such cases, 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**. This means that broadcasting is 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)` applies to scalars `a`
x          = [1,2,3]

output     = log.(x)
equivalent = [log(x[1]), log(x[2]), log(x[3])]
```

```
julia> output
3-element Vector{Float64}:
 0.0
 0.693147
 1.09861
julia> equivalent
3-element Vector{Float64}:
 0.0
 0.693147
 1.09861
```

```
square(a)  = a^2     #user-defined function for a scalar 'a'
x          = [1,2,3]

output     = square.(x)
equivalent = [square(x[1]), square(x[2]), square(x[3])]
```

```
julia> output
3-element Vector{Int64}:
 1
 4
 9
julia> equivalent
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]

output     = max.(x,y)
equivalent = [max(x[1],y[1]), max(x[2],y[2]), max(x[3],y[3])]
```

```
julia> output
3-element Vector{Int64}:
 2
 4
 8
julia> equivalent
3-element Vector{Int64}:
 2
 4
 8
```

```
foo(a,b)   = a + b        # user-defined function for scalars 'a' and 'b'
x          = [-2, -4, -10]
y          = [ 2,  4,  10]

output     = foo.(x,y)
equivalent = [foo(x[1],y[1]), foo(x[2],y[2]), foo(x[3],y[3])]
```

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

Broadcasting Applies to Any Function

Broadcasting can be used not only with numeric functions, but functions that take other types as inputs. To illustrate, 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"]

output  = string.(country, is_in, region)
```

```
julia> output
2-element Vector{String}:
 "France is in Europe"
 "Canada is in North America"
```

BROADCASTING OPERATORS

It's also possible to **broadcast operators**, in which case they apply them element-wise across collections. Its use requires prepending a dot ***before*** the operator.

Classifying operators by the number of operands helps apply broadcasting, since it directly determines its syntax. Specifically, recall that *unary operators* are written as `<symbol>x`. Thus, for example, `.√x` broadcasts the operator `√`. Likewise, the syntax for *binary operators* is `x <symbol> y`. For instance, `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]

output = x .+ y
```

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

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


output = .√x
```

```
julia> output
3-element Vector{Float64}:
 1.0
 1.41421
 1.73205
```

BROADCASTING OPERATORS WITH SCALARS

Broadcasting thus far was applied with inputs of the same size. This is because collections of dissimilar size, such as `x = [1,2]` and `y=[3,4,5]`, aren't in general allowed.

One exception to this rule is when broadcasting applies to vectors of equal size combined with scalars. In such 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

output = x .+ y
```

```
julia> output
3-element Vector{Int64}:
  5
 15
 25
```

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

output = x .+ y
```

```
julia> output
3-element Vector{Int64}:
  5
 15
 25
```

Broadcasting Can be Applied with Strings

The example based on strings presented above can be rewritten as follows.

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

output  = string.(country, is_in, region)
```

```
julia> output
2-element Vector{String}:
 "France is in Europe"
 "Canada is in North America"
```

ITERABLE OBJECTS AND BROADCASTING COMBINATION

Broadcasting isn't exclusive to vectors. Indeed, it can be applied to any iterable collection, including tuples and ranges.

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

```
julia> log.(x)
(0.0, 0.693147, 1.09861)
julia> x .+ x
(2, 4, 6)
```

```
x = 1:3
```

```
julia> log.(x)
3-element Vector{Float64}:
 0.0
 0.6931471805599453
 1.0986122886681098
julia> x .+ x
2:2:6
```

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

```
julia> x .+ y
3-element Vector{Int64}:
 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
output    = temp .^ 2
```

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

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

square(x) = x^2
output    = square.(x .+ y)
```

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

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

square(x) = x^2
output    = @. square(x + y)
```

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

BROADCASTING FUNCTIONS VS BROADCASTING OPERATORS

We've demonstrated that both functions and operators can be broadcast. 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 broadcast operation.

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

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

number_squared(a) = a ^ 2                # function for scalar 'a'
output           = number_squared.(x)
```

```
julia> output
3-element Vector{Int64}:
 1
 4
 9
```

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

vector_squared(x) = x .^ 2               # function for a vector 'x'
output           = vector_squared(x)     # '.' not needed (it'd be redundant)
```

```
julia> output
3-element Vector{Int64}:
 1
 4
 9
```

While both approaches yield the same output, **defining a function that operates on a scalar is generally the better choice**. There are two main reasons for this claim.

First, a scalar function such as `number_squared(a)` can be flexibly applied to both individual values and collections: we can simply call the function directly for scalars or rely on its broadcasted form for collections. This means the function itself remains general-purpose, without being tied to a specific type of input.

Second, the broadcasting syntax makes the programmer's intent explicit. Continuing with the example, writing `number_squared.(x)` clearly indicates that the operation is applied element-wise across the collection. By contrast, a function like `vector_squared(x)` hides this detail, leaving the reader to infer that the computation is performed element-wise.

BROADCASTING OVER ONLY ONE ARGUMENT

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

A typical scenario arises when checking whether elements from `x` match any values in a predefined list `y`. To illustrate this, let's first introduce the function `in(a, list)`, which determines whether the scalar `a` equals some element in the vector `list`. For instance, `in(2, [1,2,3])` evaluates to `true`, since `2` is contained in `[1,2,3]`.

Suppose now that, instead of a scalar `a`, we have a vector `x` and the goal is to verify whether *each* of the elements in `x` is present in `list = [1,2,3]`. Specifically, our aim is to verify if `1` belongs to `[1,2,3]`, if `2` belongs to `[1,2,3]`, and if `3` belongs to `[1,2,3]`. Below, we show that this operation can't be directly implemented via a simple broadcast version of `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 length or one of them be a scalar. In the second example, although the expression yields an output, it's not the intended result: it performs pairwise comparisons, thus checking whether `1==1`, `2==2`, and `4==3`.

Intuitively, we need a mechanism to inform Julia that `list` should be treated as a single fixed argument while iterating over `x`. This can be accomplished in two different ways. First, we can enclose `list` in a collection (e.g., a vector or tuple).

While it's possible to use any collection to wrap `list`, we'll see in Part II of the book that there's some performance penalty involved when creating vectors. Consequently, **you should prefer tuples when implementing this approach**. This requires inserting `(list,)` as the function argument, which defines a tuple whose only element is `list`. [1] Alternatively, we can rely on the function `Ref`. This requires expressing the function argument as `Ref(list)`, which makes `list` be treated as a single element.

Below, we demonstrate each approach. For completeness, we also show the case where `list` is wrapped in a vector.

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

output = in.(x, [list])
```

```
julia> output
3-element BitVector:
 1
 0
 0
```

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

output = in.(x, (list,))
```

```
julia> output
3-element BitVector:
 1
 0
 0
```

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

output = in.(x, Ref(list))
```

```
julia> output
3-element BitVector:
 1
 0
 0
```

The vector returned in each case has a type `BitVector`, where `1` corresponds to `true` and `0` to `false`. Thus, the result could equivalently be expressed as `[true, false, false]`. This reflects that `x[1]` equals `2` and `2` belongs to `list`, whereas `x[2]` and `x[3]` don't equal any element in `list`.

While the previous example focused on functions, the same principle extends to broadcast operators. This can be illustrated through the `∈` operator, which serves a similar purpose to the `in` function: it determines whether a particular element exists within a collection. [2]

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

output = x .∈ (list,)       # only 'x[1]' equals an element in 'list'
```

```
julia> output
3-element BitVector:
 1
 0
 0
```

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

output = x .∈ Ref(list)     # only 'x[1]' equals an element in 'list'
```

```
julia> output
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 a sequence of functions, each evaluated with a single argument. [3] For instance, the curried form of `f(x,y)` would be `f(x)(y)`, providing 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
curried1(x)    = (y -> addition(x,y))
curried2       = x -> (y -> addition(x,y))
```

```
julia> curried1(2)(1)
5
julia> curried2(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 broadcast 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> n.(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.] Recall that tuples with a single element must be written with a trailing comma, as in `(list,)`. Instead, `(list)` is interpreted as the variable `list`, and hence treated as a vector.

[2.] `∈` 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)`.

[3.] The term honors the mathematician Haskell Curry, not the spice!