5f. Array Indexing

Martin Alfaro
PhD in Economics

INTRODUCTION

In order to mutate vectors, you first need to identify the elements you wish to modify. This process is known as **vector indexing**. We've already covered several basic methods for indexing, including vectors and ranges (e.g., `x[[1,2,3]]` or `x[1:3]`). While these approaches are effective for simple selections, they fall short for more complex scenarios, precluding for example selections based on conditions.

This section expands our toolkit by introducing some additional forms of indexing. The techniques presented primarily build on broadcasting Boolean operations.

LOGICAL INDEXING

Logical indexing (also known as *Boolean indexing* or *masking*) allows you to select elements based on conditions. Considering a vector `x`, this is achieved using a Boolean vector `y` of the same length as `x`, which acts as a filter: `x[y]` retains elements where `y` is `true` and excludes those where `y` is `false`.

LOGICAL INDEXING

```
x = [1,2,3]
y = [true, false, true]
```

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

OPERATORS AND FUNCTIONS FOR LOGICAL INDEXING

Logical indexing becomes a powerful tool when we leverage broadcasting operations, allowing you to easily specify conditions via Boolean vectors. For instance, to select all the elements of `x` lower than 10, you can broadcast a comparison operator or a custom function.

INDEXING VIA BROADCASTING OPERATOR

```
x            = [1, 2, 3, 100, 200]

y            = x[x .< 10]
```

```
julia> y
3-element Vector{Int64}:
 1
 2
 3
```

INDEXING VIA BROADCASTING FUNCTION

```
x            = [1, 2, 3, 100, 200]

condition(a) = (a < 10)             #function to eventually broadcast
y            = x[condition.(x)]
```

```
julia> y
3-element Vector{Int64}:
 1
 2
 3
```

When dealing with multiple conditions, the conditions must be combined using the logical operators `&&` and `||`. [1] The following example illustrates the syntax for doing this. Note that *all* operators must be broadcasted, since logical operators only work with scalar values.

INDEXING VIA BROADCASTING OPERATOR

```
x            = [3, 6, 8, 100]

# numbers greater than 5, lower than 10, but not including 8
y            = x[(x .> 5) .&& (x .< 10) .&& (x .≠ 8)]
```

```
julia> y
1-element Vector{Int64}:
 6
```

INDEXING VIA @.

```
x            = [3, 6, 8, 100]

# numbers greater than 5, lower than 10, but not including 8
y            = x[@. (x > 5) && (x < 10) && (x ≠ 8)]
```

```
julia> y
1-element Vector{Int64}:
 6
```

INDEXING VIA BROADCASTING FUNCTION

```
x            = [3, 6, 7, 8, 100]

# numbers greater than 5, lower than 10, but not including 8
condition(a) = (a > 5) && (a < 10) && (a ≠ 8)         #function to eventually broadcast
y            = x[condition.(x)]
```

```
julia> y
1-element Vector{Int64}:
 6
```

The example reveals that directly broadcasting *operators* may result in verbose code, due to the repeated use of dots in the expression. In contrast, approaches based on functions or the macro `@.` keep the syntax simple, reducing **boilerplate code**.

LOGICAL INDEXING VIA `IN` AND `∈`

> **Remark**
>
> The symbols `∈` and `∉` used in this section can be inserted via tab completion:
>
> - `∈` by `\in`
> - `∉` by `\notin`

Another approach to selecting elements through logical indexing involves `in` and `∈`. Each of these symbols is available as a function and an operator, and they check whether a *scalar* `a` belongs to a given collection `list`. For simplicity, next we'll refer to `in` as a function and `∈` as an operator.

The function `in(a, list)` evaluates whether the scalar `a` matches any element in the vector `list`, yielding the same result as `a ∈ list`. For example, both `in(2, [1, 2, 3])` and `2 ∈ [1, 2, 3]` return `true`, as `2` is an element of `[1,2,3]`.

By replacing the scalar `a` with a collection `x`, `in` and `∈` can define Boolean vectors via broadcasting. Recall, though, that broadcasting defaults to iterating over pairs of elements. This means that executing `in.(x, list)` or `x .∈ list` will result in a simultaneous iteration over each pair of `x` and `list`. However, this isn't the desired operation. Rather, our goal is to check whether each element of `x` belongs to `list`, which requires treating `list` as a single object. This can be accomplished in several ways, as it was shown **here**, including wrapping `list` in `Ref`.

As an illustration, below we create a vector `y` that contains the minimum and maximum of the vector `x`.

FUNCTION 'IN' AND '∈'

```
x            = [-100, 2, 4, 100]
list = [minimum(x), maximum(x)]

# logical indexing (both versions are equivalent)
bool_indices = in.(x, Ref(list))     #`Ref(list)` can be replaced by `(list,)`
bool_indices = (∈).(x,Ref(list))

y             = x[bool_indices]
```

```
julia> bool_indices
4-element BitVector:
 1
 0
 0
 1
julia> y
2-element Vector{Int64}:
 -100
  100
```

OPERATOR 'IN' AND '∈'

```
x            = [-100, 2, 4, 100]
list = [minimum(x), maximum(x)]

# logical indexing
bool_indices = x .∈ Ref(list)        #only option, not possible to broadcast `in`


y             = x[bool_indices]
```

```
julia> bool_indices
4-element BitVector:
 1
 0
 0
 1
julia> y
2-element Vector{Int64}:
 -100
  100
```

Remark

The `in` function has an alternative curried version, allowing the user to directly broadcast `in` while treating `list` as a single element. The syntax for doing this is `in(list).(x)`, as shown in the example below.

CURRIED 'IN'

```
x            = [2, 4, 100]
list = [minimum(x), maximum(x)]

#logical indexing
bool_indices = x[in(list).(x)]    #no need to use `Ref(list)`
y            = x[bool_indices]
```

```
julia> bool_indices
4-element BitVector:
 1
 0
 0
 1

julia> y
2-element Vector{Int64}:
 -100
  100
```

Remark

The functions and operators `in` and `∈` allow for negated versions `!in` and `∉` (equivalent to `!∈`), which select elements *not* belonging to a set.

Below, we apply them to retain the elements of `x` that are not its minimum or its maximum.

FUNCTION '!IN' AND '∉'

```
x            = [-100, 2, 4, 100]
list = [minimum(x), maximum(x)]

#identical vectors for logical indexing
bool_indices = (!in).(x, Ref(list))
bool_indices  =  (∉).(x,  Ref(list))          #or  `(!∈).(x,
Ref(list))`
```

```
julia> bool_indices
4-element BitVector:
 0
 1
 1
 0

julia> x[bool_indices]
2-element Vector{Int64}:
 2
 4
```

OPERATORS '!IN' AND '∉'

```
x           = [-100, 2, 4, 100]
list = [minimum(x), maximum(x)]

#vector for logical indexing
bool_indices = x .∉ Ref(list)
```

```
julia> bool_indices
4-element BitVector:
 0
 1
 1
 0
julia> x[bool_indices]
2-element Vector{Int64}:
 2
 4
```

THE FUNCTIONS 'FINDALL' AND 'FILTER'

We close this section by presenting two additional methods for element selection. They're provided by the functions `filter` and `findall`.

The function `filter` returns the *elements* of a vector `x` satisfying a given condition. Despite what the name may suggest, `filter` retains elements rather than discard them. The condition is specified by a function that returns a Boolean scalar.

'FILTER'

```
x = [5, 6, 7, 8, 9]

y = filter(a -> a < 7, x)
```

```
julia> y
2-element Vector{Int64}:
 5
 6
```

The function `findall` does the same as `filter`, but returns the *indices* of `x`. With `findall`, the condition can be stated in two ways: either via a Boolean scalar function or a Boolean vector.

'FINDALL' - VIA FUNCTION

```
x = [5, 6, 7, 8, 9]

y = findall(a -> a < 7, x)
z = x[findall(a -> a < 7, x)]
```

```
julia> y
2-element Vector{Int64}:
 1
 2

julia> z
2-element Vector{Int64}:
 5
 6
```

'FINDALL' - VIA BOOLEAN VECTOR

```
x = [5, 6, 7, 8, 9]

y = findall(x .< 7)
z = x[findall(x .< 7)]
```

```
julia> y
2-element Vector{Int64}:
 1
 2

julia> z
2-element Vector{Int64}:
 5
 6
```

FOOTNOTES

[1] The logical operators `&&` and `||` were introduced in the section about conditional statements.