5f. Array Indexing

Martin Alfaro
PhD in Economics

INTRODUCTION

To mutate elements in a vector, you must first identify the ones you wish to modify. Such a selection process is known as **vector indexing**.

We've already covered common indexing methods, including index vectors (e.g., `x[[1,2,3]]`) and ranges (e.g., `x[1:3]`). While these approaches are effective for simple selections, they fall short when you need more expressive control. For example, they're limited when it comes to selections based on conditions.

This section broadens the set of tools available for indexing. In particular, we'll present techniques that build on broadcasting Boolean operations.

LOGICAL INDEXING

Logical indexing (also known as *Boolean indexing* or *masking*) lets you select elements based on conditions. The strategy is based on the creation of a Boolean vector `y` of the same length as `x`, which acts as a filter. This means that `x[y]` keeps 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 leveraging 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 either broadcast a comparison operator or broadcast an auxiliary 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, they must be combined using the **logical operators** `&&` and `||`. The following example illustrates the syntax for doing this. Note that *all* operators must be broadcast, since `&&` and `||` only work with scalar values. Since the repeated use of dots in the expression may result in verbose code, we also show an alternative based on the macro `@.`.

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)]
```

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
2-element Vector{Int64}:
 6
 7
```

LOGICAL INDEXING VIA `IN` AND `∈`

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

Another approach to selecting elements through logical indexing involves `in` and `∈`. Both are available as a function and an operator, although only `∈` supports broadcasting in its operating form. Considering this, the examples below use `in` when referring to the function form and `∈` for the operator form.

At its core, `in(a, list)` and `a ∈ list` check whether the scalar `a` matches any element in the vector `list`. For example, `in(2, [1, 2, 3])` and `2 ∈ [1, 2, 3]` both evaluate to `true`, because `2` is an element of `[1,2,3]`.

Leveraging `in` and `∈` for logical indexing requires replacing `a` by a collection `x`, while using its broadcast form. Note that a correct application necessitates that `list` is treated as a single object during broadcasting. Several techniques accomplish this, as discussed in a previous section. In particular, we'll consider the use of `Ref(list)` instead of `list`. [1]

The examples below demonstrate this approach by constructing a vector `y` that contains the minimum and maximum values of the vector `x`.

FUNCTIONS '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 '∈'

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

#logical indexing
bool_indices = 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 `∈` have negated counterparts `!in` and `∉`, which select elements *not* belonging to a set.

Below, we apply these to retain the elements of `x` that are neither its minimum nor its maximum value.

FUNCTIONS '!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))`
y            = x[bool_indices]
```

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

y            = x[bool_indices]
```

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

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

THE FUNCTIONS 'FINDALL' AND 'FILTER'

We close this section by presenting two additional methods for element selection: the functions `filter` and `findall`.

The function `filter` returns the *elements* of a vector `x` that satisfy a given condition. This condition must be expressed exclusively as a function that takes an element of `x` and returns a Boolean value. Note that, despite what the name might suggest, `filter` retains the elements that meet the condition, rather than discarding those that don't.

'FILTER'

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

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

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

The `findall` function behaves similarly to `filter`, but instead of returning the matching elements, it returns the *indices* within `x`. In addition, `findall` allows the condition to be specified in two forms: either as a Boolean-valued function (just like `filter`), or as a Boolean vector whose length matches that of `x`.

'FINDALL' - FUNCTION AS CONDITION

```
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' - BOOLEAN VECTOR AS CONDITION

```
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] Executing `in.(x, list)` or `x .∈ list` would either produce incorrect results or directly raise an error. The expression would simultaneously iterate over each pair of `x` and `list`, when our goal is actually to compare each element of `x` against the entire `list`.