6d. Useful Functions for Vectors

Martin Alfaro
PhD in Economics

INTRODUCTION

This section provides an overview of essential functions for manipulating vectors. We cover in particular common operations like sorting, finding unique elements, counting occurrences, and ranking data. Our ultimate goal is to illustrate the utility of these functions in a practical context, as we'll do in the next section.

SORTING VECTORS

The `sort` function allows the user to arrange elements in a specific order. Tt sorts elements in ascending order by default, with the possibility of a descending order by setting the keyword argument `rev = true`.

The function comes in two variants: `sort`, which returns a new sorted copy, and `sort!`, the in-place version that directly updates the vector.

SORT (ASCENDING)

```
x = [4, 5, 3, 2]

y = sort(x)
```

```
julia> y
4-element Vector{Int64}:
 2
 3
 4
 5
```

SORT (DESCENDING)

```
x = [4, 5, 3, 2]

y = sort(x, rev=true)
```

```
julia> y
4-element Vector{Int64}:
 5
 4
 3
 2
```

SORT!

```julia
x = [4, 5, 3, 2]

sort!(x)
```

```
julia> x
4-element Vector{Int64}:
 2
 3
 4
 5
```

Both `sort(x)` and `sort!(x)` have the option of defining the sorting order based on transformations of `x`. Specifically, given a function `foo`, the sorted order can be determined by the values of `foo(x)`. We demonstrate this below through the function `sort`, whose implementation requires the keyword argument `by`.

SORT - ABSOLUTE

```julia
x       = [4, -5, 3]


y       = sort(x, by = abs)       # 'abs' computes the absolute value
```

```
julia> abs.(x)
3-element Vector{Int64}:
 4
 5
 3

julia> y
3-element Vector{Int64}:
  3
  4
 -5
```

SORT - QUADRATIC

```julia
x       = [4, -5, 3]

foo(a) = a^2
y       = sort(x, by = foo)       # same as sort(x, by = x -> x^2)
```

```
julia> foo.(x)
3-element Vector{Int64}:
 16
 25
  9

julia> y
3-element Vector{Int64}:
  3
  4
 -5
```

SORT - NEGATIVE

```
x      = [4, -5, 3]

foo(a) = -a
y      = sort(x, by = foo)      # same as sort(x, by = x -> -x)
```

```
julia> foo.(x)
3-element Vector{Int64}:
 -4
  5
 -3
julia> y
3-element Vector{Int64}:
  4
  3
 -5
```

RETRIEVING INDICES OF SORTED ELEMENTS

While `sort` returns the ordered *values* of the vectors, you may also be interested in the *indices* of the sorted elements. This functionality is provided by the function `sortperm`, which returns the indices of `x` that would result in `sort(x)`. In other words, `x[sortperm(x)] == sort(x)` returns `true`. [1]

EXAMPLE 1

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

sort_index = sortperm(x)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 2
 3
 4
```

EXAMPLE 2

```
x          = [3, 4, 5, 6]

sort_index = sortperm(x)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 2
 3
 4
```

EXAMPLE 3

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

sort_index = sortperm(x)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 4
 2
 3
```

Note that the elements in the first two examples are already in ascending order. As a result, `sortperm` returns the trivial permutation `[1, 2, 3, 4]`. In contrast, the last example features an unordered vector `x = [1, 3, 4, 2]`. Thus, the resulting vector `[1, 4, 2, 3]` indicates that the smallest element is at index 1, the second smallest is at index 4, the third smallest is at index 2, and the largest at index 3.

Like `sort`, `sortperm` also supports retrieving indices in descending order. This requires including the keyword argument `rev = true`.

EXAMPLE 1

```
x          = [9, 3, 2, 1]

sort_index = sortperm(x, rev=true)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 2
 3
 4
```

EXAMPLE 2

```
x          = [9, 5, 3, 1]

sort_index = sortperm(x, rev=true)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 2
 3
 4
```

EXAMPLE 3

```
x          = [9, 3, 5, 1]

sort_index = sortperm(x, rev=true)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 3
 2
 4
```

Finally, `sortperm` also accepts the keyword argument `by` to define a custom transformation.

SORT - ABSOLUTE

```
x       = [4, -5, 3]


value   = sort(x, by = abs)       # 'abs' computes the absolute value
index   = sortperm(x, by = abs)
```

```
julia> abs.(x)
3-element Vector{Int64}:
 4
 5
 3
julia> value
3-element Vector{Int64}:
  3
  4
 -5
julia> index
3-element Vector{Int64}:
 3
 1
 2
```

SORT - QUADRATIC

```julia
x       = [4, -5, 3]

foo(a) = a^2
value  = sort(x, by = foo)       # same as sort(x, by = x -> x^2)
index  = sortperm(x, by = foo)
```

```
julia> foo.(x)
3-element Vector{Int64}:
 16
 25
  9

julia> value
3-element Vector{Int64}:
  3
  4
 -5

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

SORT - NEGATIVE

```julia
x       = [4, -5, 3]

foo(a) = -a
value  = sort(x, by = foo)       # same as sort(x, by = x -> -x)
index  = sortperm(x, by = foo)
```

```
julia> foo.(x)
3-element Vector{Int64}:
 -4
  5
 -3

julia> value
3-element Vector{Int64}:
  4
  3
 -5

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

AN EXAMPLE

One common application of `sortperm` is to reorder one variable based on the values of another. For example, suppose we want to assess the daily failures of a machine. Focusing on the first three days of the month, the following code snippet ranks these days by their corresponding failure counts.

DAYS SORTED BY LOWEST NUMBER OF FAILURES

```
days     = ["one", "two", "three"]
failures = [8, 2, 4]

index            = sortperm(failures)
days_by_failures = days[index]                    # days sorted by lowest failures
```

```
julia> index
3-element Vector{Int64}:
 2
 3
 1
julia> days_by_earnings
3-element Vector{String}:
 "two"
 "three"
 "one"
```

REMOVING DUPLICATES

The function `unique` removes duplicates from a vector, returning a vector that contains each element only once. The function comes in two variants `unique` provides a new copy, and `unique!`, the in-place version that directly updates the original vector.

UNIQUE

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

y = unique(x)        # returns a new vector
```

```
julia> x
4-element Vector{Int64}:
 2
 2
 3
 4
julia> y
3-element Vector{Int64}:
 2
 3
 4
```

UNIQUE!

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

unique!(x)            # mutates 'x'
```

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

The `StatsBase` package provides a related function called `countmap`, which counts the occurrences of each element in a vector. It returns a dictionary where the unique elements act as keys, and their corresponding values represent the number of times each element appears.

Note that the keys in the resulting dictionary are unsorted by default. If you prefer sorted keys, you must apply the `sort` function to the result. This will automatically transform an ordinary dictionary into an object with type `OrderedDict`.

UNSORTED COUNT

```
using StatsBase
x           = [6, 6, 0, 5]

y           = countmap(x)             # Dict with `element => occurrences`

elements    = collect(keys(y))
occurrences = collect(values(y))
```

```
julia> y
Dict{Int64, Int64} with 3 entries:
  0 => 1
  5 => 1
  6 => 2

julia> elements
3-element Vector{Int64}:
 0
 5
 6

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

SORTED COUNT

```julia
using StatsBase
x           = [6, 6, 0, 5]

y           = sort(countmap(x))        # OrderedDict with `element => occurrences`

elements    = collect(keys(y))
occurrences = collect(values(y))
```

```
julia> y
OrderedCollections.OrderedDict{Int64, Int64} with 3 entries:
  0 => 1
  5 => 1
  6 => 2

julia> elements
3-element Vector{Int64}:
 0
 5
 6

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

ROUNDING NUMBERS

Julia provides standard functions to approximate numerical values to a specific precision:

- `round` approximates the number to its nearest integer.
- `floor` approximates the number down to its nearest integer.
- `ceil` approximates the number up to its nearest integer.

Below, we show that these functions are quite flexible, allowing users to specify the output's type (e.g., `Int64` or `Float64`), the number of decimals places via the keyword argument `digits`, and the significant digits.

ROUND

```julia
x = 456.175

round(x)                    # 456.0

round(x, digits=1)          # 456.2
round(x, digits=2)          # 456.18

round(Int, x)               # 456

round(x, sigdigits=1)       # 500.0
round(x, sigdigits=2)       # 460.0
```

FLOOR

```
x = 456.175

floor(x)                        # 456.0

floor(x, digits=1)              # 456.1
floor(x, digits=2)              # 456.17

floor(Int, x)                   # 456

floor(x, sigdigits=1)           # 400.0
floor(x, sigdigits=2)           # 450.0
```

CEIL

```
x = 456.175

ceil(x)                         # 457.0

ceil(x, digits=1)               # 456.2
ceil(x, digits=2)               # 456.18

ceil(Int, x)                    # 457

ceil(x, sigdigits=1)            # 500.0
ceil(x, sigdigits=2)            # 460.0
```

RANKINGS

Instead of sorting a vector, you may be interested in determining the rank position of each element. The `StatsBase` package offers two functions for this purpose: `competerank` and `ordinalrank`. The main difference between them lies in how they handle tied elements: `competerank` assigns the same rank to tied elements, while `ordinalrank` assigns consecutive ranks. Both functions return ranks where 1 corresponds to the lowest value. The keyword argument `rev = true` allows you to invert the ranking, so that the highest value corresponds to a rank of 1.

RANK (SAME RANK FOR TIES)

```
using StatsBase
x = [6, 6, 0, 5]

y = competerank(x)
```

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

DESCENDING RANK (SAME RANK FOR TIES)

```
using StatsBase
x = [6, 6, 0, 5]

y = competerank(x, rev=true)
```

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

RANK (UNIQUE POSITIONS)

```
using StatsBase
x = [6, 6, 0, 5]

y = ordinalrank(x)
```

```
julia> y
4-element Vector{Int64}:
 3
 4
 1
 2
```

DESCENDING RANK (UNIQUE POSITIONS)

```
using StatsBase
x = [6, 6, 0, 5]

y = ordinalrank(x, rev=true)
```

```
julia> y
4-element Vector{Int64}:
 1
 2
 4
 3
```

> **Do not confuse `ordinalrank` and `sortperm`**
>
> The function `ordinalrank` indicates the position of each value in the *sorted* vector, while `sortperm` indicates the position of each value in the *unsorted* vector.

'ORDINALRANK'

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

y = ordinalrank(x)
```

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

'SORTPERM'

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

y = sortperm(x)
```

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

EXTREMA (MAXIMUM AND MINIMUM)

We conclude by presenting a method for finding the indices and values of extrema in collections. The following examples are based on the maximum, with similar functions available for the minimum.

VALUE

```
x = [6, 6, 0, 5]

y = maximum(x)
```

```
julia> y
6
```

INDEX

```
x = [6, 6, 0, 5]

y = argmax(x)
```

```
julia> y
1
```

VALUE AND INDEX

```
x = [6, 6, 0, 5]

y = findmax(x)
```

```
julia> y
(6, 1)
```

Julia additionally provides the function `max` and `min`, which respectively return the maximum and minimum of their *arguments*. These functions will become particularly useful for procedures based on binary operations (e.g., the so-called reductions).

'MAX' FUNCTION

```
x = 3
y = 4

z = max(x,y)
```

```
julia> z
4
```

FOOTNOTES

[1]. The name `sortperm` originates from "sorting permutation". Although the name might seem somewhat opaque, it arises because the operation returns the permutation of indices that would sort the original vector.