# Higher order functions

*A first taste of functional programming*

A higher-order function is a function that

- takes other functions as arguments

and/or

- returns a function as result

## Function composition and piping

Many times in scientific code it's required that functions are chained together.

This can be quite of eye sore

```
sqrt(abs(sum([1,2,3])))
```

Function composition `\circ`

```
(sqrt ∘ abs ∘ sum)([1,2,3])
```

and piping

```
sum([1,2,3]) |> abs |> sqrt
```

alleviate this problem for *unary* (single-argument) functions.

### Exercise: Becoming a pastry chef

(Re)create some syntatic sugar such as

- a function composition operator `∘` for *unary* functions

- a reverse pipe `<|`

Note: infix operators such as `∘` need to be wrapped around `()` in method definitions for parsing reasons.

# map

`map(f, [a1,a2,...]) = [f(a1), f(a2), ...]`

Map a function over the elements of a container and collect the results

```julia
map(x -> x + 1, [1, 2, 3]) == [2, 3, 4]
map((x, y) -> x * y, [1,2,3], [1, 10, 100]) == [1, 20, 300]
```

# foreach

`foreach(f, [a1,a2,...]) = f(a1); f(a2); ...; nothing`

Map a function over the elements of a container but without collecting the results

```julia
foreach(println, [1, 2, 3]) # prints
```

# foldl and foldr

`foldl(f, [a1, a2, a3, ...]) = f(f(f(a1, a2), a3), ...)`

`foldr(f, [a1, a2, a3, ...]) = f(a1, f(a2, f(a3, f(...))))`

Basically a left- and right- associative reduce.

```julia
foldl(=>, 1:4) == ((1 => 2) => 3) => 4
foldl(=>, 1:4; init=0) == (((0 => 1) => 2) => 3) => 4

foldr(=>, 1:4) == 1 => (2 => (3 => 4))
```

It doesn't end here: check also `filter`, `reduce`, `mapreduce` and possible multiple chains with `Transducers.jl`.

## Exercise: Folding left and right

- Use a folding operator to find the `min` imum element in a container
  - Define your own `min(a,b)` function or use Julia's
- Consider a finite $1$D spin-chain Hamiltonian with $N$ sites $H_N = -\sum_i^N T_i^{(N)} = -\sum_i^N \sigma_i^z \otimes \sigma_{i+1}^z$ with $\sigma_i^z \otimes \sigma_{i+1}^z := \ldots \otimes \mathbb{1}_{i-1} \otimes \sigma_i^z \otimes \sigma_{i+1}^z \otimes \mathbb{1}_{i+1} \otimes \ldots$
  - Define the identity $\mathbb{1}_2$ and Pauli matrices.
  - Define `T(i,N)`. Suggestion:
    - Create a generator `T(i,N)` which returns the to-be-kronecked matrices. This is simple because you definitely didn't forget about `if` (or ternary) clauses in generator / list comprehensions.
    - Redefine `T(i,N)` `kron` ecking away the generator using a `fold`
  - Define the Hamiltonian `H(N)` using `T(i,N)`

# Multi-dimensional `Arrays`

(**read more**)

There are a bunch of basic functions one has to know to effectively work with arrays.

## Constructing arrays

```
A = zeros(T, sizes...)
B = ones(T, sizes...)
C = rand(T, sizes...)
D = copy(A)
E = reshape(A, new_sizes...)
```

## Working with arrays

```
eltype(A)
length(A)
size(A)
eachindex(A) # iterator for visiting each position in A
```

### Concatenating arrays

Arrays can be concatenated with the `;` syntax

```
A = rand(3)
B = zeros(3)
C = [A; B]
```

or with the `cat` functions

```
C = cat(A,B,dims=1) # What would happens if 'dims=2'
```

# Linear indexing

When exactly one index `i` is provided, that index no longer represents a location in a particular dimension of the array but the `i` th element using the column-major order

```
A = [2 6; 4 7; 3 1]
```

```
A[5] == vec(A)[5] == 7
```

- Question: How to understand the matrix-creation notation `A` from the concatenation operator `;` ?

# Array `@views`

**Slicing operations like x[1:2] create a copy by default in Julia**

A "view" is a data structure that acts like an array, but the underlying data is actually part of another array (reference!).

```
A = [1 2; 3 4]
```

```
b = view(A, :, 1)
b = @views A[:,1]
```

```
b[2] = 99
A
```

Note: Sometimes it's not faster to use `view` s!

# Broadcasting and loop fusions

It is common to have "vectorized" versions of functions, which simply `map` a function `f(x)` to each element of an array `a`. Unlike some languages, in `Julia` this is **not required** for performance: it's just a convenient for-loop.

This is achieved through the `dot` syntax

```
a = [0 π/2; -π/2 π/6]
sin.(a)
```

Due to historical (and parsing) reasons, the `dot` syntax for infix operators in on the left

```
a = [1.0, 2.0, 3.0]
a .^ 3 # NOT the power of a vector
```

On complicated expression with several `dot` calls, the operation is fused together (there will be a single loop)

```
b = 2 .* a.^2 .+ sin.(a)
```

The `@.` macro can be used to convert all function / operator calls in an expression to a `dot`-call

```
b = @. 2 * a^2 + sin(a)
```

and a `$` inserted to bypass the dot

```
@. sqrt(abs($sort(x))) # equivalent to sqrt.(abs.(sort(x)))
```

Naturally calls like

```
sin.(sort(cos.(X)))
```

can't be completely fused together.

**Singleton (size=1) and missing dimensions are expanded to match the extents of the other arguments by virtually repeating the value.**

```
a = rand(2,2)
b = zeros(2,2,3)
a .+ b
```

Dot calls are just syntatic sugar for `broadcast(f, As...)` so you can extend *broadcasting* for custom types.

### Map vs Broadcasting

No winner, each has things they can do that the other cannot

- Broadcast only handles containers with th "shapes" $M\times N\times\cdots$ (i.e., a size and dimensionality) while map is more general (unknown length iterator)
- Map requires all arguments to have the same length (and hence cannot combine arrays and scalars)

### Exercise:

- Convince yourself that the loop is indeed fused by `@time` ing a complex `dot` ed expression vs the expression terms seperately computed. Run the code once beforehand to avoid timing the compilation time!
- Compute the spin-chain Hamiltonian for `N = [2, 4, 6, 7, 8]`

# `using LinearAlgebra`

"High-level" mathematical functions to operate on (multi-)dimensional `Array` s.

Assume that when you call methods from **`LinearAlgebra`** that libraries like OpenBLAS (default) will be called for standard types such as `Float64` .

If your life depends on OpenBLAS-like operations its worth to check other implementations such as Intel's MKL or even pure Julia's like `Octavian.jl` (especially if your matrices are more interesting than dense ones).

## `LinearAlgebra` vs `GenericLinearAlgebra`

Generic programming allied with multiple dispatch allows one to share types with generic algorithms. An example of this is `GenericLinearAlgebra`, which implements some of `LinearAlgebra` functions in pure Julia.

### Exercise: Ising & Wilson chains

- (Re) Consider a finite $1$D spin-chain Hamiltonian with $N$ sites coupled to a magnetic field $H_N = -\sum_i^N \sigma_i^z \otimes \sigma_{i+1}^z - h\sum_i^N \sigma_i^x$
  - Construct this Hamiltonian
  - Diagonalise the Hamiltonian `using LinearAlgebra`'s `eigen` function
    - Which states (columns of the eigenvectors matrix) have the lowest energy for $h = 0$ and $h \gg 1$?

- Consider a finite $1$D spin-chain Hamiltonian with $N$ sites $H_N = -\sum_i^N \alpha^{-i}\sigma_i^z \otimes \sigma_{i+1}^z$ for $\alpha \geq 1$
  - Diagonalise $H_N =: U_N D_N U_N^\dagger$ `using LinearAlgebra`'s `eigen` function
  - Consider the $\frac{2^N}{2}$ lowest-energy eigenvalue matrix $D_N \to \tilde{D}_N$ and couple it to a next chain site $H_{N+1} = \tilde{D}_N \otimes \mathbb{1}_2 - \alpha^{-(N+1)}\tilde{\sigma}_N \otimes \sigma_{N+1}^z$ where $\tilde{\sigma}_N = \tilde{U}\sigma_N\tilde{U}^\dagger$
  - Iterate the last step

# Performance

## Profiling

## Hardware

## Threading

# Iteration Utilities

For more examples see **here**

## zip

Run multiple iterators at the same time, until any of them is exhausted

```
a = [1,2,3]
b = (10,20,30)

for z in zip(a,b)
    println(z) # (1,10) ... (2, 20) ... (3, 30)
end
```

Question: What is the mathematical operation equivalent of zipping?

## enumerate

An iterator that yields `(i, x)` where `i` is a counter starting at `1`, and `x` is the `i`-th value from the given iterator

```
a = [10, 20, 30]

for (i, aᵢ) in enumerate(a)
    println("The $i-th entry of a is $aᵢ)
end
```