REPL 101

- `]` enters the package manager
- `;` enters the terminal
- `?` gives you the documentation of a function

Multi-dimensional `Arrays`

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When exactly one index `i` is provided, that index no longer represents a location in a particular dimension of the array but the `i` th element using the **column-major order**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

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!

`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: Quantum Ising

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

A first taste of functional programming

Functional programming a programming paradigm where programs are constructed by applying and composing functions.

Function definitions are trees of expressions that map values to other values. Unlike imperative programming, in which a sequence of imperative statements which updates the running state of the program.

There are 2 main conceptes from FP which have clear benefits for scientific code

- Higher-order functions
- Pure functions

Higher order functions

In Julia the functions are already first-class citizens (can pass them as arguments, return them and assign them to variable names).

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.

One of the most glaring differences between functional and imperative programming is that functional avoids *side effects*, which are needed in imperative style to control the state of the program.

Here are examples of higher-order functions that take us closer to the functional heaven.

map

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

```
map(g, [a1,a2,...], [b1,b2,...]) = [g(a1,b1), g(a2,b2), ...]
```

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

```
# data
v1 = [1, 2, 3]
v2 = [10, 20, 30]

# imperative
out = zero(v1)
for i in eachindex(v1) # equivalent to 1:length(v1)
    out[i] = v1[i] + v2[i]
```

```
end

# functional
out = map(+, v1, v2)
```

`map` vs `broadcast`

- Broadcast only handles containers with th "shapes" `M×N×...` (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)

foreach

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

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

```
# data
v = [1, 2, 3]

# imperative
for i in eachindex(v)
    println(v[i])
end

# functional
foreach(println, v)
```

foldl and foldr

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

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

Folds are ubiquitous in functional and scientific programming. They are a class of functions that process some data structure and return a value. Basically a left- and right- associative reduce.

```
# data
v = [1, 2, 3, 4]

# binary function
```

```
op = *

# imperative
out = v[1]
for i in 2:length(v)
    out = op(out, v[i])
end

# functional
out = foldl(op, v)
```

Question: why is there a distinction between left and right folds?

For most cases these binary functions are not associative!

We implicitely have left- and right- associativeness hard coded in our brain. What's `0 - 1 - 2 - 3 - 4` ?

- left-associative `((((0 - 1) - 2) - 3) - 4) == -10`
- right-associative `(0 - (1 - (2 - (3 - 4)))) == 2`

An initial value cal also be passed as a *keyword*

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

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

mapreduce

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

Applies a function `f` to each element of the container and `reduces` the result using a binary function `op`.

```
# data
v = [1, 2, 3, 4]

# operations
f = sin
op = *

# imperative
out = 1.0
for i in eachindex(v)
    out = op(out, f(v[i]))
end

# functional
out = mapreduce(f, op, v; init=1.0)
```

Unlike a `fold`, a `reduce` operation **assumes associativity** with the binary operations. If this cannot be guaranteed, a `mapfoldl` or `mapfoldr` can be used instead.

Question: Where could a `reduce` operation be preferred (computationally!) over a fold?

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

Exercise: Wilson chains

- Consider a finite 1D Wilson chain with N sites $H_N = -\sum_i^N \alpha^{-i} \sigma_i^z \otimes \sigma_{i+1}^z$ for $\alpha \geq 1$
  - Build the Hamiltonian in a functional way. Suggestion:
    - Use `mapfold` or `fold` to take the Kronecker products
- Numerically renormalize the problem by iterating
  - Diagonalise $H_N =: U_N D_N U_N^\dagger$ `using LinearAlgebra's eigen` function
  - Truncate $H_N \to \tilde{H}_N$ by taking only the half lowest eigenvalues
  - Coupling $\tilde{H}_N$ to a next site in the chain $H_{N+1} = \tilde{D}_N \otimes \mathbb{1}_2 - \alpha^{-(N+1)} \tilde{\sigma}_N^z \otimes \sigma_{N+1}^z$ where $\tilde{\sigma}_N^z = \tilde{U}_N \sigma_N^z \tilde{U}_N^\dagger$

Tip: A diagonal matrix can be created out of the vector of eigenvalues with the `diagm` function and the adjoint of a matrix is achieved with the `adjoint` function or `'` operator after the matrix.

Pure functions

Pure thoughts

Another great tool of functional programming we should steal for scientifc programming is the concept of pure functions. These functions are very close to mathematical functions.

- The function return values are identical for identical arguments
- The application of the function has no side effects
    - No mutation of non-local variables or mutable reference arguments

Examples of *indecent thoughts*:

Mutation of non local variables 

```
f() = x[1] += 1

x = [1]
@show x

f()
@show x
```

Different return values with identical arguments 

```
f() = x

x = 1
@show f()

x = 2
@show f()
```

Why do I care?

- Becase you're not a sociopath
- If the result of a pure expression is not used, it can be removed without affecting other expressions.
- Pure functions have no side-effects and you can intelectually refeer to them as being *referentially transparent*, just like mathematical functions
- Since they only depend on their arguments, different function calls can't interfer with each other (great for parallel programming!)
- No side effects means your compiler can, theoretically, safely apply
- Unit tests are valid and can be injected anywhere

Meta-discussion: mutable vs immutable/pure algorithms

Immutability doesn't really exist: immutability implies time-independence... and there's nothing really stopping time (at least until the heat-death of the universe).

The very process of storing information (that is ordering bits) requires mutation. But we can achieve immutability at least syntatically.

Tips to scientific code right, denying mutation and and promoting good hygiene

- Use `let` blocks to reduce global scope pollution
  - Global variables are **very** prone to be mutated since they don't have to be passed as an argument explicitely
- Pure thoughts: decompose programs into (pure) functions:
  - Break software into chunks to fit into the most limited memory: human memory.
- Give functions and variables meaningful names
  - Use `Pluto` notebooks to prototype
- Use tuples / structs to avoid repetition
  - `a1 = 1, a2 = 2` becomes `as = (1, 2)`
- Be defensive
  - Add `@assert`s to ensure validity of your inputs / results
  - Generate unit tests for your functions: these are as important as the problem you are ultimately solving
- Be smart by not oversmarting yourself:
  - avoid *premature optimisation*: write clear and concise code and only think about optimisations after unit testing
  - avoid *premature pessimisation*: take a chill pill and sketch on paper the data structures / algorithm design before writing any code
  - require of your code the same standards you require others' calculations / experiments / general care in life

Read more on good Scientific Practises

- **1**

- **2**
- **3**

Performance

We will step out of the beautiful pure world of functional programming and dive deep into the messy world of algorithm optimisation.

While some topics will be related to avoiding common pitfalls when writing in Julia, other universal topics on algorithm optimisation will also be covered.

Profiling

In order to know where and what to optimise we need tools to diagnose time spent, memory allocations and possibly how machine code is being generated.

- For quick-and-dirty diagnostics (time and memory tracking) we can use the `@time` macro behind function calls
- For an accurate measure of the latter preferer `using BenchmarkTools` and the `@btime` macro (this will give you the lower bound of `@benchmark`, which is what you want to measure!)
- For more serious profiling tools, consider reading the **manual section**

The code can be inspected at several stages with the macros

- The AST after parsing: `@macroexpand`
- The AST after type inference and some optimizations: `@code_warntype` and `@code_typed`
- The LLVM and assembly: `@code_llvm`, `@code_native`

Read more on **introspection**

Exercise: Not all "created" equal

Benchmark the different ways of mapping a function to a container and realise that computationally not all operations are equivalent

- `map`
- `broadcast`
- list comprehension
- explicit `for`-loop

Global (`isa Any`) variables: Electric Boogaloo

It's not over yet

1st performance tip on the Julia's official documentation: **Avoid global variables**

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables.

- Variables should be local
- Or passed as arguments to functions (this way the code will be specialized for the input types)

```
x = rand(100_000);

function sum_global()
    s = 0.0
    for i in x
       s += i
    end
    return s
end

function sum_arg(x)
    s = 0.0
    for i in x
       s += i
    end
    return s
end

@time sum_global()
# 0.016230 seconds (399.49 k allocations: 7.622 MiB)

@time sum_arg(x)
# 0.000123 seconds (1 allocation: 16 bytes)
```

This unfortunate dichotomy between the functions is due to one thing: Julia can't **specialize** the function `sum_global` since it cannot guarantee the type of `x`. While for `sum_arg`, it will specialize it for the type of `x` that is encountered (a complex `x` would trigger compilation).

Type inference did not require the type to be annotated!

`@code_warntype` can detect these `isa Any` variables or the also related type instabilities!

Exercise: Purging type instabilities / "untyped" containers

- Remember that an array with abstract type (e.g. `Any` or `Real`) will end up being an array of pointers and can't be operated on efficiently. `abstract` types can also prevent Julia from triggering the appropriate function specialization. Compare the `@code_native` outputs from a simple function operating on the fields of

```
mutable struct MyAmbiguousType
   a::AbstractFloat # less ambiguous than `Any` but still ambiguous
end

mutable struct MyType{T<:AbstractFloat}
   a::T
end
```

- Can you spot the type instability? Fix it and consequently `@btime` and `@code_native` the new function vs the old one.

```
function f()
    x = 1
    for i = 1:10
        x = x/2
    end
    return x
end
```

Understanding timecales

There's a rather famous table comparing computer with human timecales

Action (3GHz)	Average latency	Human time
1 clock-cyle	0.3 ns	1 s
L1 cache	0.9 ns	3 s
L2 cache	2.8 ns	9 s

L3 cache	12.9 ns	43 s
RAM	70-100 ns	3.5 to 5.5 min
SSD/IO	7-150 µs	2h to 2 days
Reboot	30-60s	1000-2000 years

In a single cycle a photon can only travel <10 cms to the RAM better not be too far away (imagine an electron).

The slowness of RAM can be mitigated by transfering data into the caches. **BUT** when the CPU requests data from the RAM, it is checked if it's in the cache. If not, there will be a *cache miss* and the program will stall until the data is fetched from the RAM.

Naturally, to reduce these cache misses, consider

- *Temporal locality*: If you need to access a piece of memory multiple times, do it close in time
- *Spatial locality*: Access memory which is close to each other (since the CPU fetches chunks of data at a time)

As a corollary

- Use little memory
- Access data sequentially (since the CPU can prefetch data that you may need)

It doesn't end here: read more on **alignment issues** **(show-in-class)**

Allocations

Memory allocation can be a significant bottleneck in critical operations. Dynamic languages such as Julia usually employ a *garbage collector* to allocate and deallocate objects in the RAM for us

```
a = [1,2,3] # allocation
a = nothing # the previous value of `a` is now garbage (since in this case no other
variables are pointing at it) and shall be collected automatically
```

- Allocation and deallocation create *overhead* which can be very costly
- More allocations results in more memory usage which results in more cache misses

The 3 most encountered problems where this can be fixed are

- When updating some value *inplace*

```
N = 1000
```

```
a = rand(10,10);

function f1(a, N)
    x = zero(a)
    for i in 1:N
        x += i * a # remember that updating operators such as '+=' reassign 'x'
    end
    x
end

@btime f1($a, $N)
# 436.185 μs (2001 allocations: 1.71 MiB)
```

In this case use the **inplace (broadcasted) assignment**

```
function f2(a, N)
    x = zero(a)
    for i in 1:N
        x .+= i * a # "add element-wise to 'x'
    end
    x
end

@btime f2($a, $N)
# 354.372 μs (1001 allocations: 875.88 KiB)
```

- When running a compution whose output memory can be recycled (see the exercise!)
- When taking slices (see the exercise!)

Single instruction, multiple data: `@simd`

CPUs operate on data present in registers inside the CPU, which are meant to hold small fixed size slots, like floats (see the `r`s in `@code_native`). Since this is a major bottleneck, modern CPUs have bigger registries (instead of 64-bit, 256+), which allow a **S**ingle **I**nstruction operate on **M**ultiple **D**ata.

Show the difference of

```
using StaticArrays
a = @SVector Int32[1,2,3,4,5,6,7,8]
code_native(+, (typeof(a), typeof(a)), debuginfo=:none)

a = @SVector Int64[1,2,3,4,5,6,7,8]
```

```
code_native(+, (typeof(a), typeof(a)), debuginfo=:none)
```

```
a = @SVector Int64[1,2,3,4]
code_native(+, (typeof(a), typeof(a)), debuginfo=:none)
```

- SIMD needs uninterrupted iteration of fixed length
- Bound-checking causes branching so deactivate it with `@inbounds for i in ...` (This is also may be desirable for critical non-`@simd` loops)
  - Avoid branching at all costs. Even with *branch prediction*, a misprediction (common for random braches) will cost several CPU cycles.
- SIMD needs associative operations (since the loop will be reordered)

Since float addition is **not** associative, automatic `@simd` is not "automatically" on for, e.g., float addition

```
@show 0.1 + (0.2 + 0.3)
@show (0.1 + 0.2) + 0.3
```

Actually, IEEE 754 float arithmetic is tricky! Consider a number that is undefined or unrepresentable, a `NaN`

```
@show NaN == NaN # not even reflexive!
```

Reference

Exercise: The 3 ecology Rs: RECYCLE, REUSE and REDUCE

- Avoid extra allocations by *recycling*
  - Rewrite `loopinc` using an in-place version of `xinc -> xinc!`
  - Compare the performance of the new `loop` function
  - Note: The point is not to optimise away the `ret[2]` part so keep it!

```
xinc(x) = [x, x+1, x+2]
function loop()
    y = 0
    for i = 1:10^7
        ret = xinc(i)
        y += ret[2]
    end
```

```
    return y
end
```

- Avoid extra allocations by *reusing*
  - Create a `(50, 100_000)` `rand`om matrix `A` and a vector `x` with size `(100_000,)`
  - Compare the performance of $\sum_i \sum_{j=1}^{80000} A_{ij} x_j$ using `slice`s and `@view`s
  - Note that performance from `@view` is just a **rule of thumb**.
- Optimise away by *reducing* (and everything else)

```
function work()
    A = zeros(N,N)
    for i in 1:N
        for j in 1:N
            val = mod(v[i],256);
            A[i,j] = B[i,j]*(sin(val)*sin(val)-cos(val)*cos(val))
        end
    end
    return A
end
```

given the parameters

```
N = 4_000
B = [float(i-j) for i in 1:N, j in 1:N]
v = [i for i in 1:N]
```

Pro-tip: Compare different implementations of `work!` using the `≈` (`\approx`) operator, since the `==` may be too strict given the shenanigans we encountered with float arithmetics.

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
```