```julia
# TensorMap & Tensor:
# general tensor implementation with arbitrary symmetries
#==========================================================#
"""
    struct TensorMap{S<:IndexSpace, N₁, N₂, ...} <: AbstractTensorMap{S, N₁, N₂}

Specific subtype of [`AbstractTensorMap`](@ref) for representing tensor maps
(morphisms in
a tensor category) whose data is stored in blocks of some subtype of `DenseMatrix`.
"""
struct TensorMap{S<:IndexSpace, N₁, N₂, G<:Sector,
A<:Union{<:DenseMatrix,SectorDict{G,<:DenseMatrix}}, F₁, F₂} <:
AbstractTensorMap{S, N₁, N₂}
    data::A
    codom::ProductSpace{S,N₁}
    dom::ProductSpace{S,N₂}
    rowr::SectorDict{G,FusionTreeDict{F₁,UnitRange{Int}}}
    colr::SectorDict{G,FusionTreeDict{F₂,UnitRange{Int}}}
    function TensorMap{S, N₁, N₂, G, A, F₁, F₂}(data::A,
                codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂},
                rowr::SectorDict{G,FusionTreeDict{F₁,UnitRange{Int}}},
                colr::SectorDict{G,FusionTreeDict{F₂,UnitRange{Int}}}) where
                    {S<:IndexSpace, N₁, N₂, G<:Sector,
A<:SectorDict{G,<:DenseMatrix},
                     F₁<:FusionTree{G,N₁}, F₂<:FusionTree{G,N₂}}
        eltype(valtype(data)) ⊆ field(S) ||
            @warn("eltype(data) = $(eltype(data)) ⊈ $(field(S)))", maxlog=1)
        new{S, N₁, N₂, G, A, F₁, F₂}(data, codom, dom, rowr, colr)
    end
    function TensorMap{S, N₁, N₂, Trivial, A, Nothing, Nothing}(data::A,
                codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂}) where
                    {S<:IndexSpace, N₁, N₂, A<:DenseMatrix}
        eltype(data) ⊆ field(S) ||
            @warn("eltype(data) = $(eltype(data)) ⊈ $(field(S)))", maxlog=1)
        new{S, N₁, N₂, Trivial, A, Nothing, Nothing}(data, codom, dom)
    end
end

const Tensor{S<:IndexSpace, N, G<:Sector, A, F₁, F₂} = TensorMap{S, N, 0, G, A,
F₁, F₂}
const TrivialTensorMap{S<:IndexSpace, N₁, N₂, A<:DenseMatrix} = TensorMap{S, N₁,
N₂, Trivial, A, Nothing, Nothing}

# Basic methods for characterising a tensor:
#--------------------------------------------
codomain(t::TensorMap) = t.codom
domain(t::TensorMap) = t.dom

blocksectors(t::TrivialTensorMap) = TrivialOrEmptyIterator(dim(t) == 0)
blocksectors(t::TensorMap) = keys(t.data)

Base.@pure storagetype(::Type{<:TensorMap{<:IndexSpace,N₁,N₂,Trivial,A}}) where
    {N₁,N₂,A<:DenseMatrix} = A
Base.@pure
```

```julia
storagetype(::Type{<:TensorMap{<:IndexSpace,N₁,N₂,G,<:SectorDict{G,A}}}) where
    {N₁,N₂,G<:Sector,A<:DenseMatrix} = A

dim(t::TensorMap) = mapreduce(x->length(x[2]), +, blocks(t); init = 0)

# General TensorMap constructors
#--------------------------------
# with data
function TensorMap(data::DenseArray, codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂}) where {S<:IndexSpace, N₁, N₂}
    if sectortype(S) === Trivial # For now, we can only accept array data for Trivial sectortype
        (d1, d2) = (dim(codom), dim(dom))
        if !(length(data) == d1*d2 || size(data) == (d1, d2) ||
            size(data) == (dims(codom)..., dims(dom)...))
            throw(DimensionMismatch())
        end
        data2 = reshape(data, (d1, d2))
        A = typeof(data2)
        return TensorMap{S, N₁, N₂, Trivial, A, Nothing, Nothing}(data2, codom, dom)
    else
        # TODO: allow to start from full data (a single DenseArray) and create the dictionary, in the first place for Abelian sectors, or for e.g. SU₂ using Wigner 3j symbols
        throw(SectorMismatch())
    end
end

function TensorMap(data::AbstractDict{<:Sector,<:DenseMatrix}, codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂}) where {S<:IndexSpace, N₁, N₂}
    G = sectortype(S)
    G == keytype(data) || throw(SectorMismatch())
    if G == Trivial
        if dim(dom) != 0 && dim(codom) != 0
            return TensorMap(data[Trivial()], codom, dom)
        else
            return TensorMap(valtype(data)(undef, dim(codom), dim(dom)), codom, dom)
        end
    end
    F₁ = fusiontreetype(G, StaticLength(N₁))
    F₂ = fusiontreetype(G, StaticLength(N₂))
    rowr = SectorDict{G, FusionTreeDict{F₁, UnitRange{Int}}}()
    colr = SectorDict{G, FusionTreeDict{F₂, UnitRange{Int}}}()
    blockiterator = blocksectors(codom ← dom)
    for c in blockiterator
        rowrc = FusionTreeDict{F₁, UnitRange{Int}}()
        colrc = FusionTreeDict{F₂, UnitRange{Int}}()
        offset1 = 0
        for s1 in sectors(codom)
            for f1 in fusiontrees(s1, c, map(isdual, codom.spaces))
                r = (offset1 + 1):(offset1 + dim(codom, s1))
                push!(rowrc, f1 => r)
```

```julia
                offset1 = last(r)
            end
        end
        offset2 = 0
        for s2 in sectors(dom)
            for f2 in fusiontrees(s2, c, map(isdual, dom.spaces))
                r = (offset2 + 1):(offset2 + dim(dom, s2))
                push!(colrc, f2 => r)
                offset2 = last(r)
            end
        end
        (haskey(data, c) && size(data[c]) == (offset1, offset2)) ||
            throw(DimensionMismatch())
        push!(rowr, c=>rowrc)
        push!(colr, c=>colrc)
    end
    if !isreal(G) && eltype(valtype(data)) <: Real
        b = valtype(data)(undef, (0,0))
        V = typeof(complex(b))
        K = keytype(data)
        data2 = SectorDict{K,V}((c=>complex(data[c])) for c in blockiterator)
        A = typeof(data2)
        return TensorMap{S, N₁, N₂, G, A, F₁, F₂}(data2, codom, dom, rowr, colr)
    else
        V = valtype(data)
        K = keytype(data)
        data2 = SectorDict{K,V}((c=>data[c]) for c in blockiterator)
        A = typeof(data2)
        return TensorMap{S, N₁, N₂, G, A, F₁, F₂}(data2, codom, dom, rowr, colr)
    end
end

# without data: generic constructor from callable:
function TensorMap(f, codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂}) where
{S<:IndexSpace, N₁, N₂}
    G = sectortype(S)
    if G == Trivial
        d1 = dim(codom)
        d2 = dim(dom)
        data = f((d1,d2))
        A = typeof(data)
        return TensorMap{S, N₁, N₂, Trivial, A, Nothing, Nothing}(data, codom, dom)
    else
        F₁ = fusiontreetype(G, StaticLength(N₁))
        F₂ = fusiontreetype(G, StaticLength(N₂))
        # TODO: the current approach is not very efficient and somewhat wasteful
        sampledata = f((1,1))
        if !isreal(G) && eltype(sampledata) <: Real
            A = typeof(complex(sampledata))
        else
            A = typeof(sampledata)
        end
        data = SectorDict{G,A}()
        rowr = SectorDict{G, FusionTreeDict{F₁, UnitRange{Int}}}()
```

```
        colr = SectorDict{G, FusionTreeDict{F₂, UnitRange{Int}}}()
        for c in blocksectors(codom ← dom)
            rowrc = FusionTreeDict{F₁, UnitRange{Int}}()
            colrc = FusionTreeDict{F₂, UnitRange{Int}}()
            offset1 = 0
            for s1 in sectors(codom)
                for f1 in fusiontrees(s1, c, map(isdual, codom.spaces))
                    r = (offset1 + 1):(offset1 + dim(codom, s1))
                    push!(rowrc, f1 => r)
                    offset1 = last(r)
                end
            end
            dim1 = offset1
            offset2 = 0
            for s2 in sectors(dom)
                for f2 in fusiontrees(s2, c, map(isdual, dom.spaces))
                    r = (offset2 + 1):(offset2 + dim(dom, s2))
                    push!(colrc, f2 => r)
                    offset2 = last(r)
                end
            end
            dim2 = offset2
            push!(data, c=>f((dim1, dim2)))
            push!(rowr, c=>rowrc)
            push!(colr, c=>colrc)
        end
        return TensorMap{S, N₁, N₂, G, SectorDict{G,A}, F₁, F₂}(data, codom, dom,
rowr, colr)
    end
end

TensorMap(f,
            ::Type{T},
            codom::ProductSpace{S},
            dom::ProductSpace{S}) where {S<:IndexSpace, T<:Number} =
    TensorMap(d->f(T, d), codom, dom)

TensorMap(::Type{T},
            codom::ProductSpace{S},
            dom::ProductSpace{S}) where {S<:IndexSpace, T<:Number} =
    TensorMap(d->Array{T}(undef, d), codom, dom)

TensorMap(::UndefInitializer,
            ::Type{T},
            codom::ProductSpace{S},
            dom::ProductSpace{S}) where {S<:IndexSpace, T<:Number} =
    TensorMap(d->Array{T}(undef, d), codom, dom)

TensorMap(::UndefInitializer,
            codom::ProductSpace{S},
            dom::ProductSpace{S}) where {S<:IndexSpace} =
    TensorMap(undef, Float64, codom, dom)
```

```julia
function TensorMap(I::LinearAlgebra.UniformScaling,
                    ::Type{T},
                    codom::ProductSpace{S},
                    dom::ProductSpace{S}) where {S<:IndexSpace, T<:Number}
    Base.depwarn("`TensorMap(I, T, codomain, domain)`  using `LinearAlgebra.I` is
deprecated, use `id([A,], space)` for creating the identity tensor on a space, and
`isomorphism([A,], codomain, domain)` to construct a fixed invertible map between
two isomorphic spaces. When `spacetype(domain)<:EuclideanSpace`, one can also use
`unitary([A,], codomain, domain)` which is then equivalent to `isomorphism`.
`A<:AbstractMatrix` is the type of storage, and is by default chosen equal to
`Matrix{Float64}`.", ((Base.Core).Typeof(TensorMap)).name.mt.name)
    isomorphism(Matrix{T}, codom, dom)
end
TensorMap(I::LinearAlgebra.UniformScaling,
            codom::ProductSpace{S},
            dom::ProductSpace{S}) where {S<:IndexSpace} =
    TensorMap(I, Float64, codom, dom)

TensorMap(::Type{T},
            codom::TensorSpace{S},
            dom::TensorSpace{S}) where {T<:Number, S<:IndexSpace} =
    TensorMap(T, convert(ProductSpace, codom), convert(ProductSpace, dom))

TensorMap(dataorf, codom::TensorSpace{S}, dom::TensorSpace{S}) where
{S<:IndexSpace} =
    TensorMap(dataorf, convert(ProductSpace, codom), convert(ProductSpace, dom))

TensorMap(dataorf, ::Type{T},
            codom::TensorSpace{S},
            dom::TensorSpace{S}) where {T<:Number, S<:IndexSpace} =
    TensorMap(dataorf, T, convert(ProductSpace, codom), convert(ProductSpace, dom))

TensorMap(codom::TensorSpace{S}, dom::TensorSpace{S}) where {S<:IndexSpace} =
    TensorMap(Float64, convert(ProductSpace, codom), convert(ProductSpace, dom))

TensorMap(dataorf, T::Type{<:Number}, P::TensorMapSpace{S}) where {S<:IndexSpace} =
    TensorMap(dataorf, T, codomain(P), domain(P))

TensorMap(dataorf, P::TensorMapSpace{S}) where {S<:IndexSpace} =
    TensorMap(dataorf, codomain(P), domain(P))

TensorMap(T::Type{<:Number}, P::TensorMapSpace{S}) where {S<:IndexSpace} =
    TensorMap(T, codomain(P), domain(P))

TensorMap(P::TensorMapSpace{S}) where {S<:IndexSpace} = TensorMap(codomain(P),
domain(P))

Tensor(dataorf, T::Type{<:Number}, P::TensorSpace{S}) where {S<:IndexSpace} =
    TensorMap(dataorf, T, P, one(P))

Tensor(dataorf, P::TensorSpace{S}) where {S<:IndexSpace} = TensorMap(dataorf, P,
one(P))

Tensor(T::Type{<:Number}, P::TensorSpace{S}) where {S<:IndexSpace} = TensorMap(T,
```

```
P, one(P))

Tensor(P::TensorSpace{S}) where {S<:IndexSpace} = TensorMap(P, one(P))

# Efficient copy constructors
#-----------------------------
function Base.copy(t::TrivialTensorMap{S, N₁, N₂, A}) where {S, N₁, N₂, A}
    return TrivialTensorMap{S, N₁, N₂, A}(copy(t.data), t.codom, t.dom)
end
function Base.copy(t::TensorMap{S, N₁, N₂, G, A, F₁, F₂}) where {S, N₁, N₂, G, A,
F₁, F₂}
    return TensorMap{S, N₁, N₂, G, A, F₁, F₂}(deepcopy(t.data), t.codom, t.dom,
t.rowr, t.colr)
end

# Similar
#---------
Base.similar(t::AbstractTensorMap, T::Type, codomain::VectorSpace,
domain::VectorSpace) =
    similar(t, T, codomain←domain)
Base.similar(t::AbstractTensorMap, codomain::VectorSpace, domain::VectorSpace) =
    similar(t, codomain←domain)

Base.similar(t::AbstractTensorMap{S}, ::Type{T},
                P::TensorMapSpace{S} = (domain(t) → codomain(t))) where {T,S} =
    TensorMap(d->similarstoragetype(t, T)(undef, d), P)
Base.similar(t::AbstractTensorMap{S}, ::Type{T}, P::TensorSpace{S}) where {T,S} =
    Tensor(d->similarstoragetype(t, T)(undef, d), P)
Base.similar(t::AbstractTensorMap{S},
                P::TensorMapSpace{S} = (domain(t) → codomain(t))) where {S} =
    TensorMap(d->storagetype(t)(undef, d), P)
Base.similar(t::AbstractTensorMap{S}, P::TensorSpace{S}) where {S} =
    Tensor(d->storagetype(t)(undef, d), P)

function Base.complex(t::AbstractTensorMap)
    if eltype(t) <: Complex
        return t
    elseif t.data isa AbstractArray
        return TensorMap(complex(t.data), codomain(t), domain(t))
    else
        data = SectorDict(c=>complex(d) for (c,d) in t.data)
        return TensorMap(data, codomain(t), domain(t))
    end
end

# Conversion between TensorMap and Dict, for read and write purpose
#------------------------------------------------------------------
function Base.convert(::Type{Dict}, t::AbstractTensorMap)
    d = Dict{Symbol,Any}()
    d[:codomain] = repr(codomain(t))
    d[:domain] = repr(domain(t))
    data = Dict{String,Any}()
    for (c,b) in blocks(t)
        data[repr(c)] = Array(b)
```

```julia
    end
    d[:data] = data
    return d
end
function Base.convert(::Type{TensorMap}, d::Dict{Symbol,Any})
    codomain = eval(Meta.parse(d[:codomain]))
    domain = eval(Meta.parse(d[:domain]))
    data = SectorDict(eval(Meta.parse(c))=>b for (c,b) in d[:data])
    return TensorMap(data, codomain, domain)
end

# Getting and setting the data
#------------------------------
hasblock(t::TrivialTensorMap, ::Trivial) = true
hasblock(t::TensorMap, s::Sector) = haskey(t.data, s)

block(t::TrivialTensorMap, ::Trivial) = t.data
function block(t::TensorMap, s::Sector)
    sectortype(t) == typeof(s) || throw(SectorMismatch())
    A = valtype(t.data)
    if haskey(t.data, s)
        return t.data[s]
    else # at least one of the two matrix dimensions will be zero
        return storagetype(t)(undef, (blockdim(codomain(t),s), blockdim(domain(t),
s)))
    end
end

blocks(t::TensorMap{<:IndexSpace,N₁,N₂,Trivial}) where {N₁,N₂} =
    SingletonDict(Trivial()=>t.data)
blocks(t::TensorMap) = t.data

fusiontrees(t::TrivialTensorMap) = ((nothing, nothing),)
fusiontrees(t::TensorMap) = TensorKeyIterator(t.rowr, t.colr)

@inline function Base.getindex(t::TensorMap{<:IndexSpace,N₁,N₂,G},
                               sectors::Tuple{Vararg{G}}) where {N₁,N₂,G<:Sector}

    FusionStyle(G) isa Abelian ||
        throw(SectorMismatch("Indexing with sectors only possible if abelian"))
    s1 = TupleTools.getindices(sectors, codomainind(t))
    s2 = TupleTools.getindices(sectors, domainind(t))
    c1 = length(s1) == 0 ? one(G) : (length(s1) == 1 ? s1[1] : first(⊗(s1...)))
    @boundscheck begin
        c2 = length(s2) == 0 ? one(G) : (length(s2) == 1 ? s2[1] : first(⊗(s1...)))
        c2 == c1 || throw(SectorMismatch())
        hassector(codomain(t), s1) && hassector(domain(t), s2)
    end
    f1 = FusionTree(s1,c1)
    f2 = FusionTree(s2,c1)
    @inbounds begin
        return t[f1,f2]
    end
end
```

```julia
@propagate_inbounds Base.getindex(t::TensorMap, sectors::Tuple) =
    t[map(sectortype(t), sectors)]

@inline function Base.getindex(t::TensorMap{<:IndexSpace,N₁,N₂,G},
        f1::FusionTree{G,N₁}, f2::FusionTree{G,N₂}) where {N₁,N₂,G<:Sector}

    c = f1.coupled
    @boundscheck begin
        c == f2.coupled || throw(SectorMismatch())
        haskey(t.rowr[c], f1) || throw(SectorMismatch())
        haskey(t.colr[c], f2) || throw(SectorMismatch())
    end
    @inbounds begin
        d = (dims(codomain(t), f1.uncoupled)..., dims(domain(t), f2.uncoupled)...)
        return sreshape(StridedView(t.data[c])[t.rowr[c][f1], t.colr[c][f2]], d)
    end
end
@propagate_inbounds Base.setindex!(t::TensorMap{<:IndexSpace,N₁,N₂,G}, v,
f1::FusionTree{G,N₁}, f2::FusionTree{G,N₂}) where {N₁,N₂,G<:Sector} =
copyto!(getindex(t, f1, f2), v)

# For a tensor with trivial symmetry, allow no argument indexing
@inline Base.getindex(t::TrivialTensorMap) =
    sreshape(StridedView(t.data), (dims(codomain(t))..., dims(domain(t))...))
@inline Base.setindex!(t::TrivialTensorMap, v) = copyto!(getindex(t), v)

# For a tensor with trivial symmetry, fusiontrees returns (nothing,nothing)
@inline Base.getindex(t::TrivialTensorMap, ::Tuple{Nothing,Nothing}) = getindex(t)
@inline Base.setindex!(t::TrivialTensorMap, v, ::Tuple{Nothing,Nothing}) =
setindex!(t, v)

# For a tensor with trivial symmetry, allow direct indexing
@inline function Base.getindex(t::TrivialTensorMap, I::Vararg{Int})
    data = t[]
    @boundscheck checkbounds(data, I...)
    @inbounds v = data[I...]
    return v
end
@inline function Base.setindex!(t::TrivialTensorMap, v, I::Vararg{Int})
    data = t[]
    @boundscheck checkbounds(data, I...)
    @inbounds data[I...] = v
    return v
end

# Show
#------
function Base.summary(t::TensorMap)
    print("TensorMap(", codomain(t), " ← ", domain(t), ")")
end
function Base.show(io::IO, t::TensorMap{S}) where {S<:IndexSpace}
    if get(io, :compact, false)
        print(io, "TensorMap(", codomain(t), " ← ", domain(t), ")")
        return
```

```
    end
    println(io, "TensorMap(", codomain(t), " ← ", domain(t), "):")
    if sectortype(S) == Trivial
        Base.print_array(io, t[])
        println(io)
    elseif FusionStyle(sectortype(S)) isa Abelian
        for (f1,f2) in fusiontrees(t)
            println(io, "* Data for sector ", f1.uncoupled, " ← ", f2.uncoupled,
":")
            Base.print_array(io, t[f1,f2])
            println(io)
        end
    else
        for (f1,f2) in fusiontrees(t)
            println(io, "* Data for fusiontree ", f1, " ← ", f2, ":")
            Base.print_array(io, t[f1,f2])
            println(io)
        end
    end
end

# Real and imaginary parts
#---------------------------
function Base.real(t::AbstractTensorMap{S}) where {S}
    # `isreal` for a `Sector` returns true iff the F and R symbols are real. This
      guarantees
    # that the real/imaginary part of a tensor `t` can be obtained by just taking
    # real/imaginary part of the degeneracy data.
    if isreal(sectortype(S))
        realdata = Dict(k => real(v) for (k, v) in blocks(t))
        return TensorMap(realdata, codomain(t), domain(t))
    else
        msg = "`real` has not been implemented for `AbstractTensorMap{$(S)}`."
        throw(ArgumentError(msg))
    end
end

function Base.imag(t::AbstractTensorMap{S}) where {S}
    # `isreal` for a `Sector` returns true iff the F and R symbols are real. This
      guarantees
    # that the real/imaginary part of a tensor `t` can be obtained by just taking
    # real/imaginary part of the degeneracy data.
    if isreal(sectortype(S))
        imagdata = Dict(k => imag(v) for (k, v) in blocks(t))
        return TensorMap(imagdata, codomain(t), domain(t))
    else
        msg = "`imag` has not been implemented for `AbstractTensorMap{$(S)}`."
        throw(ArgumentError(msg))
    end
end

# Conversion and promotion:
#---------------------------
# TODO
```