```julia
# Basic algebra
#--------------
Base.copy(t::AbstractTensorMap) = Base.copy!(similar(t), t)

Base.:-(t::AbstractTensorMap) = mul!(similar(t), t, -one(eltype(t)))
function Base.:+(t1::AbstractTensorMap, t2::AbstractTensorMap)
    T = promote_type(eltype(t1), eltype(t2))
    return axpy!(one(T), t2, copy!(similar(t1, T), t1))
end
function Base.:-(t1::AbstractTensorMap, t2::AbstractTensorMap)
    T = promote_type(eltype(t1), eltype(t2))
    return axpy!(-one(T), t2, copy!(similar(t1, T), t1))
end

Base.:*(t::AbstractTensorMap, α::Number) =
    mul!(similar(t, promote_type(eltype(t), typeof(α))), t, α)
Base.:*(α::Number, t::AbstractTensorMap) =
    mul!(similar(t, promote_type(eltype(t), typeof(α))), α, t)
Base.:/(t::AbstractTensorMap, α::Number) = *(t, one(eltype(t))/α)
Base.:\(α::Number, t::AbstractTensorMap) = *(t, one(eltype(t))/α)

LinearAlgebra.normalize!(t::AbstractTensorMap, p::Real = 2) = rmul!(t, inv(norm(t,
p)))
LinearAlgebra.normalize(t::AbstractTensorMap, p::Real = 2) =
    mul!(similar(t), t, inv(norm(t, p)))

Base.:*(t1::AbstractTensorMap, t2::AbstractTensorMap) =
    mul!(similar(t1, promote_type(eltype(t1), eltype(t2)),
codomain(t1)←domain(t2)), t1, t2)
Base.exp(t::AbstractTensorMap) = exp!(copy(t))
Base.:^(t::AbstractTensorMap, p::Integer) =
    p < 0 ? Base.power_by_squaring(inv(t), -p) : Base.power_by_squaring(t, p)

# Special purpose constructors
#-----------------------------
Base.zero(t::AbstractTensorMap) = fill!(similar(t), 0)
function Base.one(t::AbstractTensorMap)
    domain(t) == codomain(t) ||
        throw(SectorMismatch("no identity if domain and codomain are different"))
    return one!(similar(t))
end
function one!(t::AbstractTensorMap)
    domain(t) == codomain(t) ||
        throw(SectorMismatch("no identity if domain and codomain are different"))
    for (c, b) in blocks(t)
        _one!(b)
    end
    return t
end

"""
    id([A::Type{<:DenseMatrix} = Matrix{Float64},] space::VectorSpace) -> TensorMap

Construct the identity endomorphism on space `space`, i.e. return a `t::TensorMap`
```

```
with `domain(t) == codomain(t) == V`, where `storagetype(t) = A` can be specified.
"""
id(A, V::ElementarySpace) = id(A, ProductSpace(V))
id(V::VectorSpace) = id(Matrix{Float64}, V)
id(::Type{A}, P::ProductSpace) where {A<:DenseMatrix} =
    one!(TensorMap(s->A(undef, s), P, P))

"""
    isomorphism([A::Type{<:DenseMatrix} = Matrix{Float64},]
                    cod::VectorSpace, dom::VectorSpace)
    -> TensorMap

Return a `t::TensorMap` that implements a specific isomorphism between the
codomain `cod`
and the domain `dom`, and for which `storagetype(t)` can optionally be chosen to
be of type
`A`. If the two spaces do not allow for such an isomorphism, and are thus not
isomorphic,
and error will be thrown. When they are isomorphic, there is no canonical choice
for a
specific isomorphism, but the current choice is such that
`isomorphism(cod, dom) == inv(isomorphism(dom, cod))`.

See also [`unitary`](@ref) when `spacetype(cod) isa EuclideanSpace`.
"""
isomorphism(cod::TensorSpace, dom::TensorSpace) = isomorphism(Matrix{Float64},
cod, dom)
isomorphism(P::TensorMapSpace) = isomorphism(codomain(P), domain(P))
isomorphism(A::Type{<:DenseMatrix}, P::TensorMapSpace) =
    isomorphism(A, codomain(P), domain(P))
isomorphism(A::Type{<:DenseMatrix}, cod::TensorSpace, dom::TensorSpace) =
    isomorphism(A, convert(ProductSpace, cod), convert(ProductSpace, dom))
function isomorphism(::Type{A}, cod::ProductSpace, dom::ProductSpace) where
{A<:DenseMatrix}
    cod ≅ dom || throw(SpaceMismatch("codomain $cod and domain $dom are not
isomorphic"))
    t = TensorMap(s->A(undef, s), cod, dom)
    for (c, b) in blocks(t)
        _one!(b)
    end
    return t
end

const EuclideanTensorSpace = TensorSpace{<:EuclideanSpace}
const EuclideanTensorMapSpace = TensorMapSpace{<:EuclideanSpace}
const AbstractEuclideanTensorMap = AbstractTensorMap{<:EuclideanTensorSpace}
const EuclideanTensorMap = TensorMap{<:EuclideanTensorSpace}

"""
    unitary([A::Type{<:DenseMatrix} = Matrix{Float64},] cod::VectorSpace,
dom::VectorSpace)
    -> TensorMap

Return a `t::TensorMap` that implements a specific unitary isomorphism between the
```

```
codomain
`cod` and the domain `dom`, for which `spacetype(dom)` (`== spacetype(cod)`) must
be a
subtype of `EuclideanSpace`. Furthermore, `storagetype(t)` can optionally be
chosen to be
of type `A`. If the two spaces do not allow for such an isomorphism, and are thus
not
isomorphic, and error will be thrown. When they are isomorphic, there is no
canonical choice
for a specific isomorphism, but the current choice is such that
`unitary(cod, dom) == inv(unitary(dom, cod)) = adjoint(unitary(dom, cod))`.
"""
unitary(cod::EuclideanTensorSpace, dom::EuclideanTensorSpace) = isomorphism(cod,
dom)
unitary(P::EuclideanTensorMapSpace) = isomorphism(P)
unitary(A::Type{<:DenseMatrix}, P::EuclideanTensorMapSpace) = isomorphism(A, P)
unitary(A::Type{<:DenseMatrix}, cod::EuclideanTensorSpace,
dom::EuclideanTensorSpace) =
    isomorphism(A, cod, dom)

"""
    isometry([A::Type{<:DenseMatrix} = Matrix{Float64},] cod::VectorSpace,
dom::VectorSpace)
    -> TensorMap

Return a `t::TensorMap` that implements a specific isometry that embeds the domain
`dom`
into the codomain `cod`, and which requires that `spacetype(dom)` (`==
spacetype(cod)`) is
a subtype of `EuclideanSpace`. An isometry `t` is such that its adjoint `t'` is
the left
inverse of `t`, i.e. `t'*t = id(dom)`, while `t*t'` is some idempotent
endomorphism of
`cod`, i.e. it squares to itself. When `dom` and `cod` do not allow for such an
isometric
inclusion, an error will be thrown.
"""
isometry(cod::EuclideanTensorSpace, dom::EuclideanTensorSpace) =
    isometry(Matrix{Float64}, cod, dom)
isometry(P::EuclideanTensorMapSpace) = isometry(codomain(P), domain(P))
isometry(A::Type{<:DenseMatrix}, P::EuclideanTensorMapSpace) =
    isometry(A, codomain(P), domain(P))
isometry(A::Type{<:DenseMatrix}, cod::EuclideanTensorSpace,
dom::EuclideanTensorSpace) =
    isometry(A, convert(ProductSpace, cod), convert(ProductSpace, dom))
function isometry(::Type{A},
                    cod::ProductSpace{S},
                    dom::ProductSpace{S}) where {A<:DenseMatrix, S<:EuclideanSpace}
    dom ≾ cod || throw(SpaceMismatch("codomain $cod and domain $dom do not allow
for an isometric mapping"))
    t = TensorMap(s->A(undef, s), cod, dom)
    for (c, b) in blocks(t)
        _one!(b)
    end
```

```
    return t
end

# In-place methods
#------------------
import Base: copyto!
Base.@deprecate(
    copyto!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap),
    copy!(tdst, tsrc))

# Wrapping the blocks in a StridedView enables multithreading if JULIA_NUM_THREADS
  > 1
# Copy, adjoint! and fill:
function Base.copy!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap)
    space(tdst) == space(tsrc) || throw(SpaceMismatch())
    for c in blocksectors(tdst)
        copy!(StridedView(block(tdst, c)), StridedView(block(tsrc, c)))
    end
    return tdst
end
function Base.fill!(t::AbstractTensorMap, value::Number)
    for (c, b) in blocks(t)
        fill!(b, value)
    end
    return t
end
function LinearAlgebra.adjoint!(tdst::AbstractEuclideanTensorMap,
                                tsrc::AbstractEuclideanTensorMap)
    space(tdst) == adjoint(space(tsrc)) || throw(SpaceMismatch())
    for c in blocksectors(tdst)
        adjoint!(StridedView(block(tdst, c)), StridedView(block(tsrc, c)))
    end
    return tdst
end

# Basic vector space methods: addition and scalar multiplication
LinearAlgebra.rmul!(t::AbstractTensorMap, α::Number) = mul!(t, t, α)
LinearAlgebra.lmul!(α::Number, t::AbstractTensorMap) = mul!(t, α, t)

function LinearAlgebra.mul!(t1::AbstractTensorMap, t2::AbstractTensorMap,
α::Number)
    space(t1) == space(t2) || throw(SpaceMismatch())
    for c in blocksectors(t1)
        mul!(StridedView(block(t1, c)), StridedView(block(t2, c)), α)
    end
    return t1
end
function LinearAlgebra.mul!(t1::AbstractTensorMap, α::Number,
t2::AbstractTensorMap)
    space(t1) == space(t2) || throw(SpaceMismatch())
    for c in blocksectors(t1)
        mul!(StridedView(block(t1, c)), α, StridedView(block(t2, c)))
    end
    return t1
```

```julia
end
function LinearAlgebra.axpy!(α::Number, t1::AbstractTensorMap,
t2::AbstractTensorMap)
    space(t1) == space(t2) || throw(SpaceMismatch())
    for c in blocksectors(t1)
        axpy!(α, StridedView(block(t1, c)), StridedView(block(t2, c)))
    end
    return t2
end
function LinearAlgebra.axpby!(α::Number, t1::AbstractTensorMap,
                                β::Number, t2::AbstractTensorMap)
    space(t1) == space(t2) || throw(SpaceMismatch())
    for c in blocksectors(t1)
        axpby!(α, StridedView(block(t1, c)), β, StridedView(block(t2, c)))
    end
    return t2
end

# inner product and norm only valid for spaces with Euclidean inner product
function LinearAlgebra.dot(t1::AbstractEuclideanTensorMap,
t2::AbstractEuclideanTensorMap)
    space(t1) == space(t2) || throw(SpaceMismatch())
    T = promote_type(eltype(t1), eltype(t2))
    s = zero(T)
    for c in blocksectors(t1)
        s += convert(T, dim(c)) * dot(block(t1, c), block(t2, c))
    end
    return s
end

LinearAlgebra.norm(t::AbstractEuclideanTensorMap, p::Real = 2) =
    _norm(blocks(t), p, float(zero(real(eltype(t)))))
function _norm(blockiter, p::Real, init::Real)
    if p == Inf
        return mapreduce(max, blockiter; init = init) do (c, b)
            isempty(b) ? init : oftype(init, LinearAlgebra.normInf(b))
        end
    elseif p == 2
        return sqrt(mapreduce(+, blockiter; init = init) do (c, b)
            isempty(b) ? init : oftype(init, dim(c)*LinearAlgebra.norm2(b)^2)
        end)
    elseif p == 1
        return mapreduce(+, blockiter; init = init) do (c, b)
            isempty(b) ? init : oftype(init, dim(c)*sum(abs, b))
        end
    elseif p > 0
        s = mapreduce(+, blockiter; init = init) do (c, b)
            isempty(b) ? init : oftype(init, dim(c)*LinearAlgebra.normp(b, p)^p)
        end
        return s^inv(oftype(s, p))
    else
        msg = "Norm with non-positive p is not defined for `AbstractTensorMap`"
        throw(ArgumentError(msg))
    end
```

```
end

# TensorMap trace
function LinearAlgebra.tr(t::AbstractTensorMap)
    domain(t) == codomain(t) ||
        throw(SpaceMismatch("Trace of a tensor only exist when domain ==
codomain"))
    return sum(dim(c)*tr(b) for (c, b) in blocks(t))
end

# TensorMap multiplication:
function LinearAlgebra.mul!(tC::AbstractTensorMap,
                            tA::AbstractTensorMap,
                            tB::AbstractTensorMap, α = true, β = false)
    if !(codomain(tC) == codomain(tA) && domain(tC) == domain(tB) && domain(tA) ==
codomain(tB))
        throw(SpaceMismatch())
    end
    for c in blocksectors(tC)
        if hasblock(tA, c) # then also tB should have such a block
            A = block(tA, c)
            B = block(tB, c)
            C = block(tC, c)
            if isbitstype(eltype(A)) && isbitstype(eltype(B)) &&
isbitstype(eltype(C))
                @unsafe_strided A B C mul!(C, A, B, α, β)
            else
                mul!(StridedView(C), StridedView(A), StridedView(B), α, β)
            end
        elseif β != one(β)
            rmul!(block(tC, c), β)
        end
    end
    return tC
end

# TensorMap inverse
function Base.inv(t::AbstractTensorMap)
    cod = codomain(t)
    dom = domain(t)
    for c in union(blocksectors(cod), blocksectors(dom))
        blockdim(cod, c) == blockdim(dom, c) ||
            throw(SpaceMismatch("codomain $cod and domain $dom are not isomorphic:
no inverse"))
    end
    if sectortype(t) === Trivial
        return TensorMap(inv(block(t, Trivial())), domain(t)←codomain(t))
    else
        data = empty(t.data)
        for (c, b) in blocks(t)
            data[c] = inv(b)
        end
        return TensorMap(data, domain(t)←codomain(t))
    end
```

```
end
function LinearAlgebra.pinv(t::AbstractTensorMap; kwargs...)
    if sectortype(t) === Trivial
        return TensorMap(pinv(block(t, Trivial()); kwargs...),
domain(t)←codomain(t))
    else
        data = empty(t.data)
        for (c, b) in blocks(t)
            data[c] = pinv(b; kwargs...)
        end
        return TensorMap(data, domain(t)←codomain(t))
    end
end
function Base.:(\)(t1::AbstractTensorMap, t2::AbstractTensorMap)
    codomain(t1) == codomain(t2) ||
        throw(SpaceMismatch("non-matching codomains in t1 \\ t2"))
    if sectortype(t1) === Trivial
        data = block(t1, Trivial()) \ block(t2, Trivial())
        return TensorMap(data, domain(t1)←domain(t2))
    else
        cod = codomain(t1)
        data = SectorDict(c=>block(t1, c) \ block(t2, c) for c in
blocksectors(codomain(t1)))
        return TensorMap(data, domain(t1)←domain(t2))
    end
end
function Base.:(/)(t1::AbstractTensorMap, t2::AbstractTensorMap)
    domain(t1) == domain(t2) ||
        throw(SpaceMismatch("non-matching domains in t1 / t2"))
    if sectortype(t1) === Trivial
        data = block(t1, Trivial()) / block(t2, Trivial())
        return TensorMap(data, codomain(t1)←codomain(t2))
    else
        data = SectorDict(c=>block(t1, c) / block(t2, c) for c in
blocksectors(domain(t1)))
        return TensorMap(data, codomain(t1)←codomain(t2))
    end
end

# TensorMap exponentation:
function exp!(t::TensorMap)
    domain(t) == codomain(t) ||
        error("Exponentional of a tensor only exist when domain == codomain.")
    for (c, b) in blocks(t)
        copy!(b, LinearAlgebra.exp!(b))
    end
    return t
end

# Sylvester equation with TensorMap objects:
function LinearAlgebra.sylvester(A::AbstractTensorMap,
                                    B::AbstractTensorMap,
                                    C::AbstractTensorMap)
    (codomain(A) == domain(A) == codomain(C) && codomain(B) == domain(B) ==
```

```
    domain(C)) ||
        throw(SpaceMismatch())
    cod = domain(A)
    dom = codomain(B)
    sylABC(c) = sylvester(block(A, c), block(B, c), block(C, c))
    data = SectorDict(c=>sylABC(c) for c in blocksectors(cod ← dom))
    return TensorMap(data, cod ← dom)
end

# functions that map ℝ to (a subset of) ℝ
for f in (:cos, :sin, :tan, :cot, :cosh, :sinh, :tanh, :coth, :atan, :acot, :asinh)
    sf = string(f)
    @eval function Base.$f(t::AbstractTensorMap)
        domain(t) == codomain(t) ||
            error("$sf of a tensor only exist when domain == codomain.")
        I = sectortype(t)
        T = similarstoragetype(t, float(eltype(t)))
        if sectortype(t) === Trivial
            local data::T
            if eltype(t) <: Real
                data = real($f(block(t, Trivial())))
            else
                data = $f(block(t, Trivial()))
            end
            return TensorMap(data, codomain(t), domain(t))
        else
            if eltype(t) <: Real
                datadict = SectorDict{I, T}(c=>real($f(b)) for (c, b) in blocks(t))
            else
                datadict = SectorDict{I, T}(c=>$f(b) for (c, b) in blocks(t))
            end
            return TensorMap(datadict, codomain(t), domain(t))
        end
    end
end
# functions that don't map ℝ to (a subset of) ℝ
for f in (:sqrt, :log, :asin, :acos, :acosh, :atanh, :acoth)
    sf = string(f)
    @eval function Base.$f(t::AbstractTensorMap)
        domain(t) == codomain(t) ||
            error("$sf of a tensor only exist when domain == codomain.")
        I = sectortype(t)
        T = similarstoragetype(t, complex(float(eltype(t))))
        if sectortype(t) === Trivial
            data::T = $f(block(t, Trivial()))
            return TensorMap(data, codomain(t), domain(t))
        else
            datadict = SectorDict{I, T}(c=>$f(b) for (c, b) in blocks(t))
            return TensorMap(datadict, codomain(t), domain(t))
        end
    end
end

# concatenate tensors
```

```
function catdomain(t1::AbstractTensorMap{S, N₁, 1}, t2::AbstractTensorMap{S, N₁,
1}) where {S, N₁}
    codomain(t1) == codomain(t2) || throw(SpaceMismatch())

    V1, = domain(t1)
    V2, = domain(t2)
    isdual(V1) == isdual(V2) ||
        throw(SpaceMismatch("cannot horizontally concatenate tensors whose domain
has non-matching duality"))

    V = V1 ⊕ V2
    t = TensorMap(undef, promote_type(eltype(t1), eltype(t2)), codomain(t1), V)
    for c in sectors(V)
        block(t, c)[:, 1:dim(V1, c)] .= block(t1, c)
        block(t, c)[:, dim(V1, c) .+ (1:dim(V2, c))] .= block(t2, c)
    end
    return t
end
function catcodomain(t1::AbstractTensorMap{S, 1, N₂}, t2::AbstractTensorMap{S, 1,
N₂}) where {S, N₂}
    domain(t1) == domain(t2) || throw(SpaceMismatch())

    V1, = codomain(t1)
    V2, = codomain(t2)
    isdual(V1) == isdual(V2) ||
        throw(SpaceMismatch("cannot vertically concatenate tensors whose codomain
has non-matching duality"))

    V = V1 ⊕ V2
    t = TensorMap(undef, promote_type(eltype(t1), eltype(t2)), V, domain(t1))
    for c in sectors(V)
        block(t, c)[1:dim(V1, c), :] .= block(t1, c)
        block(t, c)[dim(V1, c) .+ (1:dim(V2, c)), :] .= block(t2, c)
    end
    return t
end

# tensor product of tensors
"""
    ⊗(t1::AbstractTensorMap{S}, t2::AbstractTensorMap{S}, ...) -> TensorMap{S}

Compute the tensor product between two `AbstractTensorMap` instances, which
results in a
new `TensorMap` instance whose codomain is `codomain(t1) ⊗ codomain(t2)` and whose
domain
is `domain(t1) ⊗ domain(t2)`.
"""
function ⊗(t1::AbstractTensorMap{S}, t2::AbstractTensorMap{S}) where S
    cod1, cod2 = codomain(t1), codomain(t2)
    dom1, dom2 = domain(t1), domain(t2)
    cod = cod1 ⊗ cod2
    dom = dom1 ⊗ dom2
    t = TensorMap(zeros, promote_type(eltype(t1), eltype(t2)), cod, dom)
    if sectortype(S) === Trivial
```

```
        d1 = dim(cod1)
        d2 = dim(cod2)
        d3 = dim(dom1)
        d4 = dim(dom2)
        m1 = reshape(t1[], (d1, 1, d3, 1))
        m2 = reshape(t2[], (1, d2, 1, d4))
        m = reshape(t[], (d1, d2, d3, d4))
        m .= m1 .* m2
    else
        for (f1l, f1r) in fusiontrees(t1)
            for (f2l, f2r) in fusiontrees(t2)
                c1 = f1l.coupled # = f1r.coupled
                c2 = f2l.coupled # = f2r.coupled
                for c in c1 ⊗ c2
                    degeneracyiter = FusionStyle(c) isa GenericFusion ?
                                        (1:Nsymbol(c1, c2, c)) : (nothing,)
                    for μ in degeneracyiter
                        for (fl, coeff1) in merge(f1l, f2l, c, μ)
                            for (fr, coeff2) in merge(f1r, f2r, c, μ)
                                d1 = dim(cod1, f1l.uncoupled)
                                d2 = dim(cod2, f2l.uncoupled)
                                d3 = dim(dom1, f1r.uncoupled)
                                d4 = dim(dom2, f2r.uncoupled)
                                m1 = reshape(t1[f1l, f1r], (d1, 1, d3, 1))
                                m2 = reshape(t2[f2l, f2r], (1, d2, 1, d4))
                                m = reshape(t[fl, fr], (d1, d2, d3, d4))
                                m .+= coeff1 .* coeff2 .* m1 .* m2
                            end
                        end
                    end
                end
            end
        end
    end
    return t
end

# deligne product of tensors
function ⊠(t1::AbstractTensorMap{<:EuclideanSpace{ℂ}},
            t2::AbstractTensorMap{<:EuclideanSpace{ℂ}})
    S1 = spacetype(t1)
    I1 = sectortype(S1)
    S2 = spacetype(t2)
    I2 = sectortype(S2)
    codom1 = codomain(t1) ⊠ one(S2)
    dom1 = domain(t1) ⊠ one(S2)
    data1 = SectorDict{I1 ⊠ I2, storagetype(t1)}(c ⊠ one(I2) => b for (c,b) in
blocks(t1))
    t1′ = TensorMap(data1, codom1, dom1)
    codom2 = one(S1) ⊠ codomain(t2)
    dom2 = one(S1) ⊠ domain(t2)
    data2 = SectorDict{I1 ⊠ I2, storagetype(t2)}(one(I1) ⊠ c => b for (c,b) in
blocks(t2))
    t2′ = TensorMap(data2, codom2, dom2)
```

```
    return t1′ ⊗ t2′
end
```