```julia
# truncation.jl
#
# Implements truncation schemes for truncating a tensor with svd, leftorth or
  rightorth
abstract type TruncationScheme end

struct NoTruncation <: TruncationScheme
end
notrunc() = NoTruncation()

struct TruncationError{T<:Real} <: TruncationScheme
    ϵ::T
end
truncerr(epsilon::Real) = TruncationError(epsilon)

struct TruncationDimension <: TruncationScheme
    dim::Int
end
truncdim(d::Int) = TruncationDimension(d)

struct TruncationSpace{S<:ElementarySpace} <: TruncationScheme
    space::S
end
truncspace(space::ElementarySpace) = TruncationSpace(space)

struct TruncationCutoff{T<:Real} <: TruncationScheme
    ϵ::T
    add_back::Int
end
truncbelow(epsilon::Real, add_back::Int=0) = TruncationCutoff(epsilon, add_back)


# For a single vector
function _truncate!(v::AbstractVector, ::NoTruncation, p::Real = 2)
    return v, zero(real(eltype(v)))
end

function _truncate!(v::AbstractVector, trunc::TruncationError, p::Real = 2)
    S = real(eltype(v))
    truncerr = zero(S)
    dmax = length(v)
    dtrunc = dmax
    while dtrunc > 0
        dtrunc -= 1
        prevtruncerr = truncerr
        truncerr = norm(view(v, dtrunc+1:dmax), p)
        if truncerr > trunc.ϵ
            dtrunc += 1
            truncerr = prevtruncerr
            break
        end
    end
    resize!(v, dtrunc)
    return v, truncerr
```

```julia
end

function _truncate!(v::AbstractVector, trunc::TruncationDimension, p::Real = 2)
    S = real(eltype(v))
    dtrunc = min(length(v), trunc.dim)
    truncerr = norm(view(v, dtrunc+1:length(v)), p)
    resize!(v, dtrunc)
    return v, truncerr
end

_truncate!(v::AbstractVector, trunc::TruncationSpace, p::Real = 2) =
    _truncate!(v, truncdim(dim(trunc.space)), p)

########################
########################
function _truncate!(v::AbstractVector, trunc::TruncationCutoff, p::Real = 2)
    S = real(eltype(v))
    dtrunc = findlast(Base.Fix2(>, trunc.ϵ), v)
    if dtrunc === nothing
        dtrunc = 0
    end
    dtrunc = min(dtrunc + trunc.add_back, length(v))
    truncerr = norm(view(v, dtrunc+1:length(v)), p)
    resize!(v, dtrunc)
    return v, truncerr
end

# For SectorDict
const SectorVectorDict{I<:Sector} = SectorDict{I, <:AbstractVector}

function _findnexttruncvalue(V::SectorVectorDict{I},
                                truncdim::SectorDict{I, Int}, p::Real) where
{I<:Sector}
    S = real(eltype(valtype(V)))
    q = convert(S, p)
    it = keys(V)
    next = iterate(it)
    next === nothing && nothing
    c, s = next
    while truncdim[c] == 0
        next = iterate(it, s)
        next === nothing && return nothing
        c, s = next
    end
    cmin = c
    vmin::S = convert(S, dim(c))^inv(q)*V[c][truncdim[c]]
    next = iterate(it, s)
    while next !== nothing
        c, s = next
        if truncdim[c] > 0
            v = dim(c)^inv(q)*V[c][truncdim[c]]
            if v < vmin
                cmin, vmin = c, v
            end
        end
        next = iterate(it, s)
    end
    return cmin
end

function _truncate!(V::SectorVectorDict, ::NoTruncation, p = 2)
    S = real(eltype(valtype(V)))
    return V, zero(S)
end
function _truncate!(V::SectorVectorDict, trunc::TruncationError, p = 2)
    I = keytype(V)
    S = real(eltype(valtype(V)))
    truncdim = SectorDict{I, Int}(c=>length(v) for (c, v) in V)
    truncerr = zero(S)
    while true
        cmin = _findnexttruncvalue(V, truncdim, p)
        cmin === nothing && break
        truncdim[cmin] -= 1
        prevtruncerr = truncerr
        truncerr = _norm((c=>view(v, truncdim[c]+1:length(v)) for (c, v) in V), p,
zero(S))
        if truncerr > trunc.ϵ
            truncdim[cmin] += 1
            truncerr = prevtruncerr
            break
        end
    end
    for (c, v) in V
        resize!(v, truncdim[c])
    end
    return V, truncerr
end
function _truncate!(V::SectorVectorDict, trunc::TruncationDimension, p = 2)
    I = keytype(V)
    S = real(eltype(valtype(V)))
    truncdim = SectorDict{I, Int}(c=>length(v) for (c, v) in V)
    while sum(dim(c)*d for (c, d) in truncdim) > trunc.dim
        cmin = _findnexttruncvalue(V, truncdim, p)
        cmin === nothing && break
        truncdim[cmin] -= 1
    end
    truncerr = _norm((c=>view(v, truncdim[c]+1:length(v)) for (c, v) in V), p,
zero(S))
    for (c, v) in V
        resize!(v, truncdim[c])
    end
    return V, truncerr
end
function _truncate!(V::SectorVectorDict, trunc::TruncationSpace, p = 2)
    I = keytype(V)
    S = real(eltype(valtype(V)))
    truncdim = SectorDict{I, Int}(c=>min(length(v), dim(trunc.space, c)) for (c,
v) in V)
```

```
    truncerr = _norm((c=>view(v, truncdim[c]+1:length(v)) for (c, v) in V), p,
zero(S))
    for c in keys(V)
        resize!(V[c], truncdim[c])
    end
    return V, truncerr
end
########################
########################
function _truncate!(V::SectorVectorDict, trunc::TruncationCutoff, p = 2)
    I = keytype(V)
    S = real(eltype(valtype(V)))
    truncdim = SectorDict{I, Int      }(c=>length(v) for (c, v) in V)
    next_val = SectorDict{I, Array{S} }(c=>[zero(S)] for (c, v) in V)
    for (c, v) in V
        newdim = findlast(Base.Fix2(>, trunc.ϵ), v)
        if newdim === nothing
            truncdim[c] = 0
        next_val[c] = v
    else
        truncdim[c] = newdim
            next_val[c] = v[newdim+1:end]
        end
    end
    for i in 1:trunc.add_back
        key_max = argmax(next_val)
    length(next_val[key_max]) == 0 && break
        truncdim[key_max] += 1
        next_val[key_max] = next_val[key_max][2:end]
    end
    truncerr = _norm((c=>view(v, truncdim[c]+1:length(v)) for (c, v) in V), p,
zero(S))
    for (c, v) in V
        resize!(v, truncdim[c])
    end
    return V, truncerr
end

# Combine truncations
struct MultipleTruncation{T<:Tuple{Vararg{<:TruncationScheme}}} <: TruncationScheme
    truncations::T
end
Base.:&(a::MultipleTruncation, b::MultipleTruncation) =
    MultipleTruncation((a.truncations..., b.truncations...))
Base.:&(a::MultipleTruncation, b::TruncationScheme) =
    MultipleTruncation((a.truncations..., b))
Base.:&(a::TruncationScheme, b::MultipleTruncation) =
    MultipleTruncation((a, b.truncations...))
Base.:&(a::TruncationScheme, b::TruncationScheme) = MultipleTruncation((a, b))

function _truncate!(v, trunc::MultipleTruncation, p::Real = 2)
    v, truncerrs = __truncate!(v, trunc.truncations, p)
    return v, norm(truncerrs, p)
end
```

```julia
function __truncate!(v, trunc::Tuple{Vararg{<:TruncationScheme}}, p::Real = 2)
    v, truncerr1 = _truncate!(v, first(trunc), p)
    v, truncerrtail = __truncate!(v, tail(trunc), p)
    return v, (truncerr1, truncerrtail...)
end
function __truncate!(v, trunc::Tuple{<:TruncationScheme}, p::Real = 2)
    v, truncerr1 = _truncate!(v, first(trunc), p)
    return v, (truncerr1,)
end
```