```
# FibonacciAnyons
"""
    struct FibonacciAnyon <: Sector
    FibonacciAnyon(s::Union{Symbol,Integer})

Represents the Fibonacci fusion category. It can take two values, corresponding to
the
trivial sector `FibonacciAnyon(:I) == FibonacciAnyon(0)` and the non-trivial sector
`FibonacciAnyon(:τ) = FibonacciAnyon(1)` with fusion rules ``τ ⊗ τ = 1 ⊕ τ``.
"""
struct FibonacciAnyon <: Sector
    isone::Bool
    function FibonacciAnyon(s::Symbol)
        s == :I || s == :τ || throw(ArgumentError("Unknown FibonacciAnyon $s."))
        new(s === :I)
    end
end
Fibonacci(i::Integer) = iszero(i) ? Fibonacci(:I) :
                            (isone(i) ? Fibonacci(:τ) : error("unkown Fibonacci
anyon"))

Base.IteratorSize(::Type{SectorValues{FibonacciAnyon}}) = HasLength()
Base.length(::SectorValues{FibonacciAnyon}) = 2
Base.iterate(::SectorValues{FibonacciAnyon}, i = 0) =
    i == 0 ? (FibonacciAnyon(:I), 1) : (i == 1 ? (FibonacciAnyon(:τ), 2) : nothing)
function Base.getindex(S::SectorValues{FibonacciAnyon}, i)
    if i == 1
        return FibonacciAnyon(:I)
    elseif i == 2
        return FibonacciAnyon(:τ)
    else
        throw(BoundsError(S, i))
    end
end
findindex(::SectorValues{FibonacciAnyon}, s::FibonacciAnyon) = 2 - s.isone

Base.convert(::Type{FibonacciAnyon}, s::Symbol) = FibonacciAnyon(s)
Base.convert(::Type{Symbol}, a::FibonacciAnyon) = a.s ? (:I) : (:τ)
Base.one(::Type{FibonacciAnyon}) = FibonacciAnyon(:I)
Base.conj(s::FibonacciAnyon) = s

const _goldenratio = (1 + sqrt(5)) / 2
dim(a::FibonacciAnyon) = isone(a) ? one(_goldenratio) : _goldenratio

Base.@pure FusionStyle(::Type{FibonacciAnyon}) = SimpleNonAbelian()
Base.@pure BraidingStyle(::Type{FibonacciAnyon}) = Anyonic()
Base.isreal(::Type{FibonacciAnyon}) = false

⊗(a::FibonacciAnyon, b::FibonacciAnyon) = FibonacciIterator(a,b)

struct FibonacciIterator
    a::FibonacciAnyon
    b::FibonacciAnyon
end
```

```
Base.IteratorSize(::Type{FibonacciIterator}) = Base.HasLength()
Base.IteratorEltype(::Type{FibonacciIterator}) = Base.HasEltype()
Base.length(iter::FibonacciIterator) = (isone(iter.a) || isone(iter.b)) ? 1 : 2
Base.eltype(::Type{FibonacciIterator}) = FibonacciAnyon
function Base.iterate(iter::FibonacciIterator, state = 1)
    I = FibonacciAnyon(:I)
    τ = FibonacciAnyon(:τ)
    if state == 1 # first iteration
        iter.a == I && return (iter.b, 2)
        iter.b == I && return (iter.a, 2)
        return (I, 2)
    elseif state == 2
        (iter.a == iter.b == τ) && return (τ, 3)
        return nothing
    else
        return nothing
    end
end

Nsymbol(a::FibonacciAnyon, b::FibonacciAnyon, c::FibonacciAnyon) =
    isone(a) + isone(b) + isone(c) != 2 # zero if one tau and two ones

function Fsymbol(a::FibonacciAnyon, b::FibonacciAnyon, c::FibonacciAnyon,
                 d::FibonacciAnyon, e::FibonacciAnyon, f::FibonacciAnyon)
    Nsymbol(a, b, e) || return zero(_goldenratio)
    Nsymbol(e, c, d) || return zero(_goldenratio)
    Nsymbol(b, c, f) || return zero(_goldenratio)
    Nsymbol(a, f, d) || return zero(_goldenratio)

    I = FibonacciAnyon(:I)
    τ = FibonacciAnyon(:τ)
    if a == b == c == d == τ
        if e == f == I
            return +1/_goldenratio
        elseif e == f == τ
            return -1/_goldenratio
        else
            return +1/sqrt(_goldenratio)
        end
    else
        return one(_goldenratio)
    end
end

function Rsymbol(a::FibonacciAnyon, b::FibonacciAnyon, c::FibonacciAnyon)
    Nsymbol(a, b, c) || return 0*exp((0π/1)*im)
    if isone(a) || isone(b)
        return exp((0π/1)*im)
    else
        return isone(c) ? exp(+(4π/5)*im) : exp(-(3π/5)*im)
    end
end

Base.show(io::IO, ::Type{FibonacciAnyon}) = print(io, "FibonacciAnyon")
```

```
function Base.show(io::IO, a::FibonacciAnyon)
    s = isone(a) ? ":I" : ":τ"
    return get(io, :typeinfo, nothing) === FibonacciAnyon ?
        print(io, s) : print(io, "FibonacciAnyon(", s, ")")
end

Base.hash(a::FibonacciAnyon, h::UInt) = hash(a.isone, h)
Base.isless(a::FibonacciAnyon, b::FibonacciAnyon) = isless(!a.isone, !b.isone)
```