Spenso

Typed tensors, symbolic structures, and executable tensor networks

---

1 Overview

Spenso represents and evaluates tensors with abstract indices. It separates a tensor's structure—representations, slots, dimensions, names, and index order—from its stored data. Dense and sparse storage, symbolic and parametric values, automatic index matching, and network execution build on that separation.

> **Two questions, two layers**
> Tensor structure answers "which indices and representations does this object carry?" Tensor data answers "which values occupy that structure?" Contraction first matches compatible slots, then combines data. Keep these phases distinct when diagnosing a shape, duality, or contraction error.

1.1 Choose a task

- To install Symbolica and construct a typed tensor with the bundled Python module, use the Python guide.
- To verify slot duality with one numerical Rust contraction, follow the Rust guide, then continue with the first-contraction tutorial and the exact `Contract` reference.
- To assemble and execute several tensor/scalar nodes, use the tensor-network guide with the `Network` reference.
- To construct tensors or execute a network from Python, follow the Python tensor-workflow guide with the exact `TensorNetwork` reference.
- To parse symbolic tensor functions or apply Dirac/color identities, review the feature and interface guide, then continue with Idenso for representation-aware rewrites.

1.2 From tensors to networks

The core package supports three related levels of work:

- tensor structures describe slots and abstract indices;
- dense, sparse, numeric, symbolic, and parametric tensors attach data to those structures;
- tensor networks represent a collection of tensors and choose an execution or contraction strategy.

Networks can be assembled directly. With the `shadowing` feature, they can also be inferred from Symbolica expressions whose functions carry recognizable tensor structure. Parsing is not mere string conversion: representation initialization, index variance, names, and the tensor library determine the inferred network.

```
[dependencies]
spenso = "0.6"
```

Enable `shadowing` only when Symbolica-backed parsing or symbolic operations are needed. It pulls in the Symbolica integration and corresponding Linnet support. The default core remains useful for typed tensors and contractions without that layer.

1.3 Companion packages

> **Component ownership**
> `spenso` owns the tensor model and execution engine. `spenso-macros` owns the `SimpleRepresentation` derive

used to declare custom index representations. `spenso-hep-lib` owns concrete high-energy-physics tensor-library data such as gamma matrices and projectors. The `spynso3` adapter owns the Python community module. Their versions and feature gates are independent, so select and upgrade each component explicitly.

Spenso uses Linnet for the underlying network graph. Idenso is the symbolic-identity layer for Spenso-formatted Symbolica expressions. GammaLoop consumes these components inside a broader collider workflow.

The interface guide, native Rustdoc, and generated Python reference list the public modules, traits, feature gates, and import paths.

Contributors changing structures, contraction, parsing, storage, or network scheduling should also read the source-audited Spenso implementation architecture.

2 Build your first Spenso tensor

Spenso is available as a native Rust library and as a community module bundled with Symbolica's Python package. Both interfaces preserve representation, variance, and abstract-index structure alongside tensor data.

Python
Start here for notebooks and exploratory calculations. The published Symbolica wheel includes the native Spenso community module and needs no GammaLoop checkout.

Use Spenso from Python →

Rust
Start here when a Rust application owns tensor storage, contractions, or network execution. The published `spenso` crate exposes the complete native API.

Use Spenso from Rust →

Choose Python for the shortest installation and Rust for the most complete library surface. The interface guide maps both routes to their generated reference.

3 Using Spenso from Python

The shortest path into Spenso is the community module bundled with Symbolica `2.2.0`. This example creates a typed two-dimensional tensor and checks its diagonal entries; it needs no GammaLoop checkout.

3.1 Install and verify the module

Create an isolated environment, install the version used by this manual, and verify the community import:

```
python -m venv .venv
. .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install "symbolica==2.2.0"
python -c "import symbolica.community.spenso as spenso; print(spenso.__name__)"
```

There is no separate `spenso` Python wheel. The installed `symbolica` distribution owns the native `symbolica.community.spenso` module. Symbolica's installation and license terms apply; its free restricted mode is enough for this single-process example.

3.2 Construct a dense tensor

Save this as `spenso_quickstart.py`:

```
from symbolica.community.spenso import Representation, Tensor, TensorIndices

rep = Representation.euc(2)
matrix = Tensor.dense(
    TensorIndices(rep("i"), rep("j")),
    [1.0, 0.0, 0.0, 1.0],
)

assert matrix[0, 0] == 1.0
assert matrix[1, 1] == 1.0
matrix.to_sparse()
print(matrix)
```

Run `python spenso_quickstart.py`. `TensorIndices` gives the data a representation-aware structure, while the four row-major values provide its concrete storage. Converting to sparse storage changes only that storage; it does not change the slots or re-index the tensor.

> **Structure comes before storage**
> Equal dimensions alone do not make two tensor slots compatible. Their representations, variance, and abstract indices remain part of the value. Diagnose those properties before changing dense or sparse storage.

Continue with the Python tensor-workflow guide, or use the Rust guide to perform a checked native contraction.

4 Using Spenso from Rust

This example constructs two dense tensors, contracts one pair of dual indices, and checks one output component. It uses a two-dimensional Euclidean representation as a compact software exercise rather than a Lorentz-space physics example.

4.1 Create a Rust project

Use Rust 1.85 or newer:

```
cargo new spenso-quickstart
cd spenso-quickstart
cargo add spenso@0.6.0
```

Replace `src/main.rs` with:

```
use spenso::{
    contraction::Contract,
    structure::{
        OrderedStructure, PermutedStructure,
        representation::{Euclidean, LibraryRep, RepName},
        slot::{DualSlotTo, IsAbstractSlot},
    },
    tensors::data::{DenseTensor, GetTensorData, SetTensorData},
};

fn main() {
    let rep = Euclidean {};
    let left_free = rep.new_slot(2, 0).to_lib();
    let right_free = rep.new_slot(2, 2).to_lib();
    let shared = rep.new_slot(2, 10).to_lib();

    let left_structure: OrderedStructure<LibraryRep> =
        PermutedStructure::from_iter([left_free, shared]).structure;
```

```
    let right_structure: OrderedStructure<LibraryRep> =
        PermutedStructure::from_iter([right_free, shared.dual()]).structure;

    let mut left = DenseTensor::<i32, _>::zero(left_structure);
    let mut right = DenseTensor::<i32, _>::zero(right_structure);
    left.set(&[0, 1], 2).unwrap();
    right.set(&[0, 1], 5).unwrap();

    let result = left.contract(&right).unwrap();
    assert_eq!(*result.get_ref([0, 0]).unwrap(), 10);
    println!("result structure: {}", result.structure);
    println!("result data: {:?}", result.data);
}
```

Run `cargo run`. Success means the assertion passes and the printed result contains `10` at the component selected by the two remaining free indices. Spenso matches index identity and duality, not merely equal dimensions.

> **The docs run this example**
> The documentation test compiles and executes this program against the current workspace. The contraction itself needs neither Symbolica nor Spenso's optional `shadowing` feature.

Continue with the first tensor workflow for a more deliberate explanation of structure and storage, then use the tensor-network guide when more than two tensors should be planned and executed together.

5 Tutorial

This tutorial performs one concrete tensor contraction in Rust. The example makes the tensor structure, dual index matching, storage, and numerical result visible—the same layers that a larger Spenso network coordinates automatically. It uses a two-dimensional Euclidean representation as a small linear-algebra example; it is not a Lorentz-space physics example.

For the shortest installation and first-run path, begin with Using Spenso from Rust. Then return here for a closer look at the index structure and contraction result.

5.1 Prerequisites

Use Rust 1.85 or newer (Spenso uses Rust 2024 edition), then create a binary project:

```
cargo new spenso-first-contraction
cd spenso-first-contraction
cargo add spenso@0.6
```

This first contraction uses dense integer tensors and needs neither Symbolica nor the `shadowing` feature.

5.2 Contract one shared index

Replace `src/main.rs` with:

```
use spenso::{
    contraction::Contract,
    structure::{
        OrderedStructure, PermutedStructure,
        representation::{Euclidean, LibraryRep, RepName},
        slot::{DualSlotTo, IsAbstractSlot},
    },
    tensors::data::{DenseTensor, GetTensorData, SetTensorData},
```

```
};

fn main() {
    let rep = Euclidean {};
    let left_free = rep.new_slot(2, 0).to_lib();
    let right_free = rep.new_slot(2, 2).to_lib();
    let shared = rep.new_slot(2, 10).to_lib();

    let left_structure: OrderedStructure<LibraryRep> =
        PermutedStructure::from_iter([left_free, shared]).structure;
    let right_structure: OrderedStructure<LibraryRep> =
        PermutedStructure::from_iter([right_free, shared.dual()]).structure;

    let mut left = DenseTensor::<i32, _>::zero(left_structure);
    let mut right = DenseTensor::<i32, _>::zero(right_structure);
    left.set(&[0, 1], 2).unwrap();
    right.set(&[0, 1], 5).unwrap();

    let result = left.contract(&right).unwrap();
    assert_eq!(*result.get_ref([0, 0]).unwrap(), 10);
    println!("result structure: {}", result.structure);
    println!("result data: {:?}", result.data);
}
```

Run `cargo run`. Success means the assertion passes and the printed result contains `10` at the component selected by the two remaining free indices. Spenso matched `shared` with `shared.dual()`, summed that axis, and preserved `left_free` and `right_free` in the output structure. The contraction follows index identity and duality, not merely equal dimensions.

> **Verification scope and cost**
> The docs harness compiles and runs this Rust program; it syntax-checks the setup commands without creating a project or using the network. Success is the asserted component value `10` with two free output indices. The contraction is tiny; a clean external Cargo build can take minutes, while the program itself should finish in well under a second.

> **Structure is part of the value**
> The data vectors alone do not say which axes may contract. Preserve the `OrderedStructure` with the tensor, and inspect the resulting structure before interpreting a flat component offset. A dimension match with unrelated abstract indices produces an exterior product, not the contraction shown above.

5.3 From two tensors to a network

Pairwise `Contract` is the clearest first success. For a larger expression, place tensors in a `Network`, inspect its graph, choose or benchmark a contraction strategy, and execute it. Use `DataTensor` when individual nodes may be dense or sparse. Enable `shadowing` only when the network must mix concrete data with Symbolica-backed parametric tensors.

5.4 Troubleshooting and next steps

- If `contract` leaves both shared-looking axes free, print their slots and check that one is the dual of the other; dimensions alone are insufficient.
- If `set` rejects a component, compare the coordinate count and each coordinate with the structure's rank and dimensions.
- If a sparse contraction reports an empty input, distinguish an intentionally zero tensor from a tensor whose nonzero entries were never populated.

- Continue with the tensor-structures and networks manual before selecting an automatic contraction strategy.

6 Tensor data, contraction, and network execution

Spenso keeps tensor structure and tensor data separate so the same structural operations work with dense, sparse, symbolic, or parametric storage. A structure supplies ordered slots, representations, dimensions, and abstract indices. Data supplies values and a storage-specific access strategy. Contraction first establishes structural compatibility and only then chooses a data operation.

6.1 Dense, sparse, and parametric data

Dense storage is appropriate when most components are populated and a predictable contiguous layout matters. Sparse storage is appropriate when zero structure dominates, but contraction cost depends on index overlap and lookup behavior rather than only the number of stored values. Parametric and symbolic tensors defer numeric substitution; keep their parameter environment with the tensor when serializing or moving a calculation between stages.

Conversions must preserve the structure's slot order. A tensor with the same dimensions but a different representation or variance is not interchangeable until an explicit structural operation establishes that relationship.

6.2 Contraction and planning

Pairwise contraction matches compatible dual slots, determines the output structure, and then contracts the selected data backends. A tensor network generalizes this to many nodes connected through abstract indices. The current public API selects a contraction strategy while executing the network; it does not expose a separately stored, reusable plan object.

The cheapest-looking local pair is not always the cheapest complete order. Intermediate tensor dimensions, sparsity, symbolic expression growth, and reusable subexpressions can dominate. Record the strategy and step bound with performance results. A structural change can lead the same strategy to choose a different order.

> **Strategy selection is not a cached plan**
> Replacing values without changing slots preserves the contraction problem, but the public API still runs the selected strategy during execution. Adding a tensor, changing a representation, or rewiring an index changes that problem and requires a fresh execution decision.

6.3 Execute a small tensor/scalar network

This self-contained network computes `2 a + 3 b` for two equally structured tensors. The dummy libraries make the execution boundary explicit: every tensor is already concrete and any unresolved function key is an error.

```
use spenso::{
    network::{
        ExecutionResult, Network, Sequential, SmallestDegree,
        library::{DummyKey, DummyLibrary, DummyLibraryTensor, panicing::ErroringLibrary},
        store::NetworkStore,
    },
    structure::{
        OrderedStructure,
        representation::{Euclidean, RepName},
    },
    tensors::data::DenseTensor,
};

fn main() {
    type Tensor = DenseTensor<f64, OrderedStructure<Euclidean>>;
    type Store = NetworkStore<Tensor, f64>;
```

```
    type Net = Network<Store, DummyKey, DummyKey>;
    type LibraryTensor = DummyLibraryTensor<Tensor>;

    let structure = OrderedStructure::new(vec![Euclidean {}.new_slot(2, 1)]).structure;
    let a = DenseTensor::from_data(vec![1.0, 2.0], structure.clone()).unwrap();
    let b = DenseTensor::from_data(vec![3.0, 4.0], structure).unwrap();
    let tensors = DummyLibrary::new();
    let functions = ErroringLibrary::new();

    let mut network =
        Net::from_scalar(2.0) * Net::from_tensor(a) + Net::from_scalar(3.0) *
Net::from_tensor(b);
    network
        .execute::<Sequential, SmallestDegree, LibraryTensor, _, _>(&tensors, &functions)
        .unwrap();

    let ExecutionResult::Val(result) = network.result_tensor::<LibraryTensor,
_>(&tensors).unwrap()
    else {
        panic!("expected a concrete tensor result");
    };
    assert_eq!(result.data, vec![11.0, 16.0]);
}
```

The output invariant is the component vector `[11.0, 16.0]`; the original rank-one structure must also remain attached to the result. The documentation harness compiles the complete program; run it to execute the assertion in a provisioned Symbolica environment. See the exact `Network`, `DenseTensor`, and pairwise contraction Rustdoc.

> **Interpret execution failures by layer**
> `from_data` errors are rank/dimension or storage-length mismatches. An unresolved function-key error means the network contains a symbolic library node despite the concrete-only setup. A different value with the same shape points to scalar placement or node arithmetic; a different shape points to changed slots or unintended contraction, which requires replanning.

6.4 Symbolica shadowing

With `shadowing`, Spenso recognizes registered tensor functions inside Symbolica expressions and builds a network whose nodes shadow those subexpressions. Initialization of representations and the tensor library must happen before parsing. Unknown functions remain symbolic rather than silently acquiring tensor semantics.

Shadowing is useful for extracting a contraction problem while retaining a reversible link to the source expression. Keep the replacement map until the computed result has been substituted back. For rewrite identities and index cooking, use Idenso Spenso itself owns storage, contraction, and execution.

The implementation handoff from symbolic syntax into network nodes is traced in the Spenso parsing flow. The Symbolica syntax and rewrite guide records the contributor-facing matching and replacement conventions.

6.5 Custom representations and the HEP library

`SimpleRepresentation` derives the repetitive representation plumbing for a user enum. The derive input still defines domain semantics: duality, dimension, slot compatibility, and any Symbolica tags must agree. The resulting representations implement the ordinary Rust traits used by the rest of Spenso; the `spenso-macros` API reference lists the supported derive attributes.

`spenso-hep-lib` registers concrete high-energy-physics structures such as gamma matrices and projectors. Treat that library as shared domain data. GammaLoop uses the same definitions.

Graph and tensor ownership
Linnet owns connectivity, cuts, cycles, and graph mutation. Spenso owns which connected slots contract and how a plan is executed. Keeping that boundary explicit prevents a graph rewrite from being mistaken for a tensor-algebra identity.

7 Python tensor workflows

Spenso's Python interface is a native adapter over the same tensor structures, libraries, and network executor as the Rust crates. Use the generated reference for exact signatures and this guide for the object boundaries and execution sequence that those signatures do not explain.

7.1 Availability and version boundary

A Symbolica community module
Import Spenso as `symbolica.community.spenso`; there is no standalone `spenso` wheel. The published Symbolica wheel bundles this community module. Its Symbolica version determines which Spenso API it contains. A source checkout or generated `.pyi` file does not add the native module to an existing Python environment.

Install the current assembly and check the environment before building a workflow:

```
python -m pip install --upgrade symbolica
python -c "import symbolica.community.spenso as spenso; print(spenso.__name__)"
```

Source embedders can build a custom community-module assembly. Record the Symbolica assembly version with reproducible results; it is a more useful Python compatibility fact than the version of an unrelated local Rust checkout.

7.2 Choose the right object

- `Representation` and `Slot` define dimensions, duality, and abstract indices.
- `TensorIndices` carries concrete abstract indices; `TensorStructure` describes a reusable shape whose indices can be assigned later.
- `Tensor` owns ordinary dense or sparse data. `LibraryTensor` is the named form registered in a `TensorLibrary` for reuse by symbolic networks.
- `TensorNetwork` owns an expression graph. `ExecutionMode` selects the rewrite strategy: one smallest-degree rewrite per step, scalar work only, or the general smallest-degree strategy. Use `n_steps`, not the mode, to bound how many execution steps run.
- `TensorEvaluator` substitutes repeated numerical parameter batches into a symbolic tensor; its compiled counterpart owns the generated native evaluator.

The generated `Representation`, `Tensor`, and `TensorNetwork` entries are the exact versioned contracts. Do not infer index compatibility from two equal dimensions: names, representations, and duality remain part of the value.

7.3 Construct and inspect concrete data

This complete source creates a named rank-two tensor, verifies one component, and converts its storage in place. The documentation harness compiles the Python source without importing the native module.

```
from symbolica.community.spenso import Representation, Tensor, TensorIndices

rep = Representation.euc(2)
```

```
i = rep("i")
j = rep("j")
indices = TensorIndices(i, j)
matrix = Tensor.dense(indices, [1.0, 0.0, 0.0, 1.0])

assert len(matrix) == 4
assert matrix[0, 0] == 1.0
matrix.to_sparse()
assert matrix[1, 1] == 1.0
```

`Tensor.dense` requires row-major data whose length is the product of the structure dimensions. `Tensor.sparse` instead needs the element type and starts empty. `to_dense()` and `to_sparse()` mutate the storage representation; they do not change slots or re-index the tensor. See the dense constructor and conversion contract.

> **Diagnose structure before storage**
> A constructor failure usually means the data length and dimensions disagree. An unexpected contraction or exterior product is instead an index/duality problem. Print the structure and slots before changing dense/sparse storage, because a storage conversion cannot repair a structural mismatch.

7.4 Register data and execute a network

A symbolic network resolves named tensors through a library. Register the data first, construct the expression with the same name and compatible slots, then execute and request the result:

```
from symbolica.community.spenso import (
    ExecutionMode,
    LibraryTensor,
    Representation,
    TensorLibrary,
    TensorName,
    TensorNetwork,
    TensorStructure,
)

rep = Representation.euc(2)
A = TensorName("A")
structure = TensorStructure(rep, rep, name=A)
library = TensorLibrary()
library.register(LibraryTensor.dense(structure, [1.0, 0.0, 0.0, 1.0]))

network = TensorNetwork(A(rep("i"), rep("j")), library=library)
network.execute(library=library, mode=ExecutionMode.All)
result = network.result_tensor(library=library)
assert len(result) == 4
```

`ExecutionMode.Single` selects the smallest-degree single-rewrite strategy, but execution still continues while work remains unless it is bounded; use `n_steps=1` to inspect exactly one step. `Scalar` processes scalar work while retaining tensor structure, and `All` attempts the complete available execution. A successful `execute()` does not make `result_scalar()` appropriate for a tensor result; choose `result_tensor()` or `result_scalar()` according to the remaining structure. Exact behavior is linked from `execute`, `result_tensor`, and `ExecutionMode`.

> **Interpret a network failure by ownership**
> A missing-library error means a symbolic tensor name has no registered data. A result-kind error means the graph still has tensor structure when a scalar was requested, or vice versa. A changed network structure invalidates assumptions about contraction order; replacing only values with the same registered structure does not.

7.5 Repeated symbolic evaluation

Use `Tensor.evaluator()` when the tensor structure stays fixed and only symbolic parameters change across batches. Supply constants, custom functions, and parameters explicitly. Call `evaluate()` for real inputs and `evaluate_complex()` when coefficients or inputs are complex. `compile()` creates source and a shared library, so treat its filenames, compiler, architecture, and optimization level as reproducibility inputs rather than incidental arguments.

Control Symbolica-backed Rayon work with `SymbolicParallelism`: `Serial` keeps work on the calling thread, `Auto` checks license capability and applies workload heuristics, and `Parallel` forces Rayon without that safety choice. Configure the policy before benchmarking; otherwise a threading-policy change can be mistaken for an algorithmic improvement.

For exact parameters and defaults, use the evaluator reference and symbolic parallelism reference. The implementation starts in the Spenso Python adapter.

8 Rust, macros, and Python APIs

8.1 Core Rust package

The `spenso` crate organizes its public surface into `structure`, `tensors`, `contraction`, `network`, `iterators`, and `algebra`. The important abstractions form a progression:

- `TensorStructure` and related traits describe slots, names, and contraction compatibility;
- `DenseTensor`, `SparseTensor`, and the heterogeneous tensor enums own storage;
- contraction traits perform pairwise or multi-tensor operations;
- network stores and libraries bind symbolic tensor names to concrete data and execute a graph.

Trait implementations and generic constraints determine which combinations of structure and data support an operation. Consult the Rust orientation to choose the relevant crate, then use its revision-specific Rustdoc when a method is unavailable for a particular tensor type. The `shadowing` API and symbolic parallelism controls require their corresponding Cargo features.

8.2 Proc macros and HEP data

`spenso-macros` is a separate proc-macro crate. Its `SimpleRepresentation` derive generates the representation and duality boilerplate used by Spenso index types. A declaration supplies a symbolic name and chooses either `self_dual` or a `dual_name`:

```
use spenso_macros::SimpleRepresentation;
use spenso::structure::representation::RepName;

#[derive(SimpleRepresentation)]
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, PartialOrd, Ord, Default)]
#[representation(name = "flavor", dual_name = "AntiFlavor")]
struct Flavor {}
```

The `SimpleRepresentation` Rustdoc lists the derive's helper attributes and allowed targets. Macro expansion happens at compile time and produces ordinary Rust implementations.

`spenso-hep-lib` supplies domain data and tensor-library construction for high-energy physics. It is intentionally separate from the generic core. Users who need gamma matrices or physics projectors add that package; generic Spenso users do not inherit those conventions implicitly.

8.3 Python community module

An adapter, not a Spenso wheel
Python users import `symbolica.community.spenso`. The implementation is the `spynso3` Rust adapter and is distributed through the Symbolica community-module mechanism. Enabling Spenso's Rust `python` feature only enables conversion interoperability; it does not create a standalone importable `spenso` Python distribution.

Install the published Symbolica assembly with `python -m pip install --upgrade symbolica`, then verify `python -c "import symbolica.community.spenso"`. Module availability follows the Symbolica assembly version; a local Spenso source checkout does not add the module to an installed wheel. Source embedders must add `spynso3` to the external symbolica-community assembly and invoke its `SymbolicaCommunityModule` registration while building that extension; building the Rust crate alone does not inject the module into another Symbolica wheel.

```
from symbolica.community.spenso import (
    Representation,
    Tensor,
    TensorIndices,
)

rep = Representation.euc(2)
indices = TensorIndices(rep("i"), rep("j"))
identity = Tensor.dense(indices, [1.0, 0.0, 0.0, 1.0])
```

The Python tensor-workflow guide connects construction, libraries, network execution, evaluators, and symbolic-parallelism policy. Use it with the structured Python reference, whose exact signatures can differ from the generic Rust API because `spynso3` provides Python-specific conversions and defaults.

Source starting points are the core crate, the derive crate, the HEP library, and the Python adapter.

9 Version history

History coverage
This version of Spenso is 0.6.0, but the published changelog currently ends at 0.5.6. Release notes for 0.6.0 are not yet available, so the history below is incomplete for that version. `spenso-macros`, `spenso-hep-lib`, and `spynso3` have independent versions and changelogs.

9.1 Available histories

- Spenso core versions
- Spenso macros versions
- Spenso HEP library versions
- Spynso3 Python-adapter versions

When upgrading, inspect the history for tensor structure, contraction, network parsing, Symbolica integration, and Python API or signature changes. Also check each companion component used by the application: a core Spenso release does not imply a release of the macro, HEP, or Python adapter package.

9.2 Upgrade checklist

Compare tensor structure and contraction behavior, network parsing, Symbolica integration, feature flags, and Python signatures for every component used by the application.

9.3 Reproducibility record

For reproducible upgrades, record the versions of Spenso and every companion component used by the application. Do not assume that changes between 0.5.6 and 0.6.0 are fully represented in the available notes.

10 Spenso versions

This page renders the canonical Spenso changelog source as part of the Spenso website, navigation, and search.

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

11.1 Unreleased

11.2 0.5.6 - 2026-01-08

11.2.1 Fixed

- fix `python_stub_gen` feature
- fix metric initialization and update to symbolica 1.2

11.2.2 Other

- update linnet

11.3 0.5.5 - 2025-12-15

11.3.1 Fixed

- fix canonization in presence of duals
- fix sum normalization in network parsing
- fix warnings
- fix overflowing slot order
- fix `add_assign` with sparse tensors

11.3.2 Other

- Update symbolica to 1.1.0
- update linnet
- add depth to parser, to stop at first mul
- remove more prints
- remove prints
- update linnet
- add readmes
- formatting and add ref for parametric
- update to 0.17 `pyo3_stub_gen`
- update linnet and make classattr to classmethods
- Release spenso-macros 0.3
- update to symbolica 1.0
- update sympolica to 0.20
- Properly precontract scalars
- Add back pyo3-stub-gen-derive
- Updated spenso with minor canonical string update for symbolica f2eb0c1ddcc61c342f2ff9616c2d129141d433dc
- Refactor coefficient conversions to support generic float types
- update to current dev symbolica and add pattern for pslash + m conjugate

- update to linnet 0.13.1
- working canonize (up to functions) and dirac adjoint
- spenso canonize buggy
- `robust_conj` but with lots of `gamma_0s`
- first try conj
- update python api
- working pow and fun execution
- working pow execution with wrapping
- support for executing pow and functions on tensors
- implement function library trait
- add function generic key

11.4 0.5.3 - 2025-08-18

11.4.1 Other

- update linnet to crates
- all tests pass
- Fix Multiple Heads Bug
- implement ExportNumber for spenso::Complex
- to Atom for ExpandedIndex
- remove printouts
- update linnet
- update linnet
- update linnet
- add infinite loop error test
- update parse problem and add spenso:: to all symbols
- remove expand

11.5 0.5.2 - 2025-07-15

11.5.1 Other

- remove printout

11.6 0.5.1 - 2025-07-15

11.6.1 Fixed

- fix gamma algebra≈

11.6.2 Other

- allow arbitrary arguments for to dots

12 Spenso macros versions

The procedural macros are versioned independently from Spenso. Their canonical version history remains the spenso-macros changelog source. This rendered copy integrates it with the manual, navigation, search, and product PDF.

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

13.1 Unreleased

13.2 0.3.2 - 2026-01-08

13.2.1 Fixed

- fix metric initialization and update to symbolica 1.2

13.2.2 Other

- update linnet

13.3 0.3.1 - 2025-12-15

13.3.1 Other

- Update symbolica to 1.1.0
- update linnet
- update linnet
- add readmes
- update linnet and make classattr to classmethods

13.4 0.2.1 - 2025-08-18

13.4.1 Other

- update linnet to crates
- update linnet
- update linnet
- update linnet

14 Spenso HEP library versions

The HEP helper library is versioned independently from Spenso. Its canonical version history remains the spenso-hep-lib changelog source. This rendered copy integrates it with the manual, navigation, search, and product PDF.

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

15.1 Unreleased

15.2 0.1.7 - 2026-01-14

15.2.1 Other

- update versions

15.3 0.1.6 - 2026-01-08

15.3.1 Fixed

- fix `python_stub_gen` feature
- fix metric initialization and update to symbolica 1.2

15.4 0.1.5 - 2025-12-15

15.4.1 Fixed

- fix warnings
- fix `add_assign` with sparse tensors

15.4.2 Other

- Update symbolica to 1.1.0
- update linnet
- add readmes
- Release spenso-macros 0.3
- update to symbolica 1.0
- update sympolica to 0.20
- Properly precontract scalars
- Refactor coefficient conversions to support generic float types
- working pow and fun execution
- working pow execution with wrapping
- add function generic key

15.5 0.1.3 - 2025-08-18

15.5.1 Other

- all tests pass
- update linnet

15.6 0.1.2 - 2025-07-15

15.6.1 Fixed

- fix gamma algebra again

15.6.2 Other

- update versions

15.7 0.1.1 - 2025-07-15

15.7.1 Fixed

- fix gamma algebra≈

15.7.2 Other

- update release action
- allow arbitrary arguments for to dots

16 Spynso3 versions

The Python adapter is versioned independently from Spenso. Its canonical version history remains the spynso3 changelog source. This rendered copy integrates it with the manual, navigation, search, and product PDF.

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

17.1 Unreleased

17.2 0.1.2 - 2026-01-14

17.2.1 Other

- update versions

17.3 0.1.1 - 2025-12-15

17.3.1 Fixed

- fix sum normalization in network parsing
- fix warnings
- fix `add_assign` with sparse tensors
- fix projp
- fix param docs and iter return type
- fix `pyo3_stubs`

17.3.2 Other

- Update implementation to pyo3-stub-gen 0.17
- Update symbolica to 1.1.0
- update linnet
- add readmes
- update to 0.17 `pyo3_stub_gen`
- update linnet and make classattr to classmethods
- move initialize to api
- Release spenso-macros 0.3
- update to symbolica 1.0
- update sympolica to 0.20
- Properly precontract scalars
- Add back pyo3-stub-gen-derive
- update to current dev symbolica and add pattern for pslash + m conjugate
- first try conj
- update python api
- add function generic key
- again
- return any because python is dumb
- update `__iter__` method
- use numpy doc format
- add overloaded pyi stub gen

Python API reference

The supported Python packages available in this version are listed below. The HTML manual provides a separate page for every public class and function; this printable edition keeps a compact inventory. Rust APIs use canonical Rustdoc in the HTML manual.

`spynso3`

Exports registered by spynso3.

`CompiledTensorEvaluator`

A compiled and optimized evaluator for maximum performance tensor evaluation.

```
A compiled and optimized evaluator for maximum performance tensor evaluation.

This class wraps a compiled C++ shared library for extremely fast numerical
evaluation of tensor expressions. It only supports complex-valued evaluation
as this is the most general case.

A compiled and optimized evaluator for maximum performance tensor evaluation.

This class wraps a compiled C++ shared library for extremely fast numerical
evaluation of tensor expressions. It only supports complex-valued evaluation
as this is the most general case.

Create instances using the `TensorEvaluator.compile()` method.

Examples
--------
>>> compiled = evaluator.compile("eval_func", "code.cpp", "lib")
>>> results = compiled.evaluate_complex(large_input_batch)
```

```
class CompiledTensorEvaluator:
```

Requires features: `python_stubgen`

`evaluate_complex`

Kind: `Method`

```
def evaluate_complex(self, inputs: Sequence[Sequence[complex]]) -> list[Tensor]:
```

```
Evaluate the tensor expression for multiple complex-valued parameter inputs.

Uses the compiled C++ code for maximum performance evaluation with complex numbers.

Parameters
----------
inputs : list of list of complex
    List of parameter value lists, where each inner list contains
    complex values for all parameters in the same order as specified
    when creating the original evaluator

Returns
-------
list of Tensor
    List of evaluated tensors, one for each input parameter set

Examples
--------
>>> complex_inputs = [
```

```
...     [1.0+2.0j, 3.0+0.0j],
...     [0.0+1.0j, 2.0+1.0j]
... ]
>>> results = compiled_evaluator.evaluate_complex(complex_inputs)
```

inputs

Kind: Parameter

```
Sequence[Sequence[complex]]
```

List of parameter value lists, where each inner list contains complex values for all parameters in the same order as specified when creating the original evaluator

Exhaustive reference · Source

LibraryTensor

A library tensor class optimized for use in tensor libraries and networks.

A library tensor class optimized for use in tensor libraries and networks.

Library tensors are similar to regular tensors but use explicit keys for efficient lookup and storage in tensor libraries. They can be either dense or sparse and store data as floats, complex numbers, or symbolic expressions.

LibraryTensors are designed for:
- Registration in TensorLibrary instances
- Use in tensor networks where structure reuse is important
- Efficient symbolic manipulation and pattern matching

```
Examples
--------
>>> from symbolica.community.spenso import LibraryTensor, TensorStructure,
Representation
>>> rep = Representation.euc(3)
>>> structure = TensorStructure(rep, rep, name="T")
>>> data = [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
>>> tensor = LibraryTensor.dense(structure, data)
>>> sparse_tensor = LibraryTensor.sparse(structure, float)
```

```
class LibraryTensor:
```

Requires features: python_stubgen

structure

Kind: Method

```
def structure(self) -> TensorStructure:
```

sparse

Kind: AssociatedFunction

```
def sparse(structure: TensorStructure | list[Representation] | list[int], type_info:
type) -> LibraryTensor:
```

Create a new sparse empty library tensor with the given structure and data type.

Creates a sparse tensor that initially contains no non-zero elements.
Elements can be set individually using indexing operations.

```
Parameters
----------
```

```
structure : TensorStructure, list of Representations, or list of int
    The tensor structure defining shape and index properties
type_info : type
    The data type - either `float` or `Expression` class

Returns
-------
LibraryTensor
    A new sparse library tensor with all elements initially zero

Examples
--------
>>> import symbolica as sp
>>> from symbolica.community.spenso import LibraryTensor, TensorStructure,
Representation
>>> rep = Representation.euc(3)
>>> structure = TensorStructure(rep, rep)
>>> sparse_float = LibraryTensor.sparse(structure, float)
>>> sparse_sym = LibraryTensor.sparse(structure, sp.Expression)
>>> sparse_float[0, 0] = 1.0
>>> sparse_float[1, 1] = 2.0
```

structure

Kind: Parameter

```
TensorStructure | list[Representation] | list[int]
```

```
The tensor structure defining shape and index properties
```

type_info

Kind: Parameter

```
type
```

```
The data type - either `float` or `Expression` class
```

dense

Kind: AssociatedFunction

```
def dense(structure: TensorStructure | list[Representation] | list[int], data:
Sequence[Expression] | Sequence[float] | Sequence[complex]) -> LibraryTensor:
```

```
Create a new dense library tensor with the given structure and data.

Dense tensors store all elements explicitly in row-major order. The structure
defines the tensor's shape and indexing properties.

Parameters
----------
structure : TensorStructure, list of Representations, or list of int
    The tensor structure defining shape and index properties
data : list of float, complex, or Expression
    The tensor data in row-major order

Returns
-------
LibraryTensor
    A new dense library tensor with the specified data
```

```
Examples
--------
>>> from symbolica import S
>>> from symbolica.community.spenso import LibraryTensor, TensorStructure,
Representation
>>> rep = Representation.euc(2)
>>> sigma = S("sigma")
>>> structure = TensorStructure(rep, rep, name=sigma)
>>> data = [0.0, 1.0, 1.0, 0.0]
>>> tensor = LibraryTensor.dense(structure, data)
>>> x, y = S("x", "y")
>>> sym_data = [x, y, -y, x]
>>> sym_tensor = LibraryTensor.dense(structure, sym_data)
```

structure

Kind: Parameter

```
TensorStructure | list[Representation] | list[int]
```

The tensor structure defining shape and index properties

data

Kind: Parameter

```
Sequence[Expression] | Sequence[float] | Sequence[complex]
```

The tensor data in row-major order

one

Kind: AssociatedFunction

```
def one() -> LibraryTensor:
```

Create a scalar library tensor with value 1.0.

```
Returns
-------
LibraryTensor
    A scalar library tensor containing the value 1.0

Examples
--------
>>> from symbolica.community.spenso import LibraryTensor
>>> one = LibraryTensor.one()
```

zero

Kind: AssociatedFunction

```
def zero() -> LibraryTensor:
```

Create a scalar library tensor with value 0.0.

```
Returns
-------
LibraryTensor
    A scalar library tensor containing the value 0.0

Examples
--------
```

```
>>> from symbolica.community.spenso import LibraryTensor
>>> zero = LibraryTensor.zero()
```

to_dense

Kind: Method

```
def to_dense(self) -> None:
```

Convert this library tensor to dense storage format.

Converts sparse tensors to dense format in-place. Dense tensors are unchanged.
This allocates memory for all tensor elements.

Examples:
```python
from symbolica.community.spenso import LibraryTensor, TensorStructure, Representation

rep = Representation.cof(2)
structure = TensorStructure([rep, rep])
tensor = LibraryTensor.sparse(structure, float)
tensor[0, 0] = 1.0
tensor.to_dense()  # Now stores all 4 elements explicitly
```

to_sparse

Kind: Method

```
def to_sparse(self) -> None:
```

Convert this library tensor to sparse storage format.

Converts dense tensors to sparse format in-place, only storing non-zero elements.
This can save memory for tensors with many zero elements.

Examples:
```python
from symbolica.community.spenso import LibraryTensor, TensorStructure, Representation

rep = Representation.euc(2)
structure = TensorStructure(rep, rep)
data = [1.0, 0.0, 0.0, 2.0]
tensor = LibraryTensor.dense(structure, data)
tensor.to_sparse()  # Now only stores 2 non-zero elements
```

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

__str__

Kind: Method

```
def __str__(self) -> str:
```

__len__

Kind: Method

```
def __len__(self) -> int:
```

__getitem__

Kind: Method

__getitem__

Kind: Overload

```
def __getitem__(self, item: slice | int | list[int]) -> Any:
```

item

Kind: Parameter

```
slice | int | list[int]
```

__getitem__

Kind: Overload

```
def __getitem__(self, item: slice) -> list[Expression | complex | float]:
```

```
Get library tensor elements at the specified range of indices.

Parameters
----------
item : slice
    Slice object defining the range of indices

Returns
-------
list of float, complex, or Expression
    The tensor elements at the specified range
```

item

Kind: Parameter

```
slice
```

```
Slice object defining the range of indices
```

__getitem__

Kind: Overload

```
def __getitem__(self, item: Sequence[int] | int) -> Expression | complex | float:
```

```
Get library tensor element at the specified index or indices.

Parameters
----------
item : int or list of int
    Index specification (int for flat index, list of int for coordinates)

Returns
-------
float, complex, or Expression
    The tensor element at the specified index
```

item

Kind: Parameter

```
Sequence[int] | int
```

```
Index specification (int for flat index, list of int for coordinates)
```

__setitem__

Kind: Method

__setitem__

Kind: Overload

```
def __setitem__(self, item: Any, value: Any) -> None:
```

Set library tensor element(s) at the specified index or indices.

```
Parameters
----------
item : int or list of int
    Index specification (int for flat index, list of int for coordinates)
value : float, complex, or Expression
    The value to set

Examples
--------
>>> from symbolica.community.spenso import LibraryTensor, TensorStructure,
Representation
>>> rep = Representation.euc(2)
>>> structure = TensorStructure(rep, rep)
>>> tensor = LibraryTensor.sparse(structure, float)
>>> tensor[0] = 1.0
>>> tensor[1, 1] = 2.0
```

item

Kind: Parameter

```
Any
```

Index specification (int for flat index, list of int for coordinates)

value

Kind: Parameter

```
Any
```

The value to set

__setitem__

Kind: Overload

```
def __setitem__(self, item: Sequence[int] | int, value: Expression | complex | float)
-> None:
```

Set library tensor element(s) at the specified index or indices.

```
Parameters
----------
item : int or list of int
    Index specification (int for flat index, list of int for coordinates)
value : float, complex, or Expression
    The value to set

Examples
--------
>>> from symbolica.community.spenso import LibraryTensor, TensorStructure,
Representation
```

```
>>> rep = Representation.euc(2)
>>> structure = TensorStructure(rep, rep)
>>> tensor = LibraryTensor.sparse(structure, float)
>>> tensor[0] = 1.0
>>> tensor[1, 1] = 2.0
```

item

Kind: `Parameter`

```
Sequence[int] | int
```

```
Index specification (int for flat index, list of int for coordinates)
```

value

Kind: `Parameter`

```
Expression | complex | float
```

```
The value to set
```

scalar

Kind: `Method`

```
def scalar(self) -> Expression:
```

```
Extract the scalar value from a rank-0 (scalar) library tensor.

Returns
-------
Expression
    The scalar expression contained in this tensor

Raises
------
RuntimeError
    If the tensor is not a scalar

Examples
--------
>>> from symbolica.community.spenso import LibraryTensor
>>> scalar_tensor = LibraryTensor.one()
>>> value = scalar_tensor.scalar()
```

Exhaustive reference · Source

Representation

```
A representation in the sense of group representation theory for tensor indices.
```

```
A representation in the sense of group representation theory for tensor indices.

Representations define the transformation properties of tensor indices under group
operations.
They specify the dimension and duality structure, determining which indices can
contract.

Key concepts:
- **Self-dual**: Indices can contract with other indices of the same representation
- **Dualizable**: Indices can only contract with their dual representation
- **Dimension**: Size of the representation space
```

```
Predefined representations are available as class methods:
- `Representation.euc(d)`: Euclidean space (self-dual)
- `Representation.mink(d)`: Minkowski space (self-dual)
- `Representation.bis(d)`: Bispinor (self-dual)
- `Representation.cof(d)`: Color fundamental (dualizable)
- `Representation.coad(d)`: Color adjoint (self-dual)
- `Representation.cos(d)`: Color sextet (dualizable)

# Examples:
```python
from symbolica.community.spenso import Representation

# Standard representations
euclidean = Representation.euc(4)      # 4D Euclidean
lorentz = Representation.mink(4)       # 4D Minkowski
color = Representation.cof(3)          # SU(3) fundamental
adjoint = Representation.coad(8)       # SU(3) adjoint

# Custom representation
custom = Representation("MyRep", 5, is_self_dual=True)

# Create slots with indices
mu_slot = euclidean('mu')              # Euclidean index μ
a_slot = color('a')                    # Color index a

# Generate metric tensors
metric = euclidean.g('mu', 'nu')       # g_μν
```

class Representation:
```

Requires features: `python_stubgen`

`__call__`
Kind: `Method`

`__call__`
Kind: `Overload`

```
def __call__(self, aind: int | Expression | str) -> Slot:

Create a slot from this representation, by specifying an index.

Parameters
----------
aind : int, str, or Symbol
    The index specification

Returns
-------
Slot
    A new Slot object with the specified index

Examples
--------
>>> from symbolica.community.spenso import Representation
>>> import symbolica as sp
>>> rep = Representation.euc(3)
```

```
>>> slot1 = rep('mu')
>>> slot2 = rep(1)
>>> slot3 = rep(sp.S('nu'))
```

aind

Kind: Parameter

```
int | Expression | str
```

The index specification

__call__

Kind: Overload

```
def __call__(self, aind: Expression) -> Expression | Slot:
```

Create a slot or symbolic expression from this representation.

```
Parameters
----------
aind : Expression
    The index specification (Expression creates symbolic representation)

Returns
-------
Expression
    A symbolic expression representing this representation

Examples
--------
>>> from symbolica.community.spenso import Representation
>>> import symbolica as sp
>>> rep = Representation.euc(3)
>>> expr = rep(sp.E("cos(x)"))
```

aind

Kind: Parameter

```
Expression
```

The index specification (Expression creates symbolic representation)

__new__

Kind: AssociatedFunction

```
def __new__(cls, name: str, dimension: int | Expression | str, is_self_dual: bool =
True) -> Representation:
```

Create and register a new representation with specified properties.

```
# Parameters:
- name: String name for the representation
- dimension: Size of the representation (int or symbolic)
- is_self_dual: If True, creates self-dual representation; if False, creates
dualizable pair

# Examples:
```python
from symbolica.community.spenso import Representation
import symbolica as sp
```
```

```
# Self-dual representation (indices contract with themselves)
euclidean = Representation("Euclidean", 4, is_self_dual=True)

# Dualizable representation (needs dual partner for contraction)
vector_up = Representation("VectorUp", 4, is_self_dual=False)

# Symbolic dimension
n = sp.S('n')
general = Representation("General", n, is_self_dual=True)
```

name

Kind: Parameter

str

String name for the representation

dimension

Kind: Parameter

int | Expression | str

Size of the representation (int or symbolic)

is_self_dual

Kind: Parameter

bool

Default: True

If True, creates self-dual representation; if False, creates dualizable pair

dual

Kind: Method

def dual(self) -> Representation:

g

Kind: Method

def g(self, i: int | Expression | str, j: int | Expression | str) -> TensorIndices:

Create a metric tensor for this representation.

```
# Parameters:
- i: First index
- j: Second index

# Examples:
```python
rep = Representation.mink(4)
metric = rep.g('mu', 'nu')  # Minkowski metric g_μν
```
```

i

Kind: Parameter

int | Expression | str

First index

j

Kind: Parameter

int | Expression | str

Second index

flat

Kind: Method

def flat(self, i: int | Expression | str, j: int | Expression | str) ->
TensorIndices:

Create a musical isomorphism tensor for this representation.

Parameters:
- i: First index
- j: Second index

Examples:
```python
rep = Representation.mink(4)
flat = rep.flat('mu', 'nu')  # Flat isomorphism ♭_μν
```

i

Kind: Parameter

int | Expression | str

First index

j

Kind: Parameter

int | Expression | str

Second index

id

Kind: Method

def id(self, i: int | Expression | str, j: int | Expression | str) -> TensorIndices:

Create an identity tensor for this representation.

Parameters:
- i: First index
- j: Second index

Examples:
```python
rep = Representation.cof(3)
identity = rep.id('a', 'b')  # Color identity δ_ab
```

i

Kind: Parameter

```
int | Expression | str
```

First index

j

Kind: Parameter

```
int | Expression | str
```

Second index

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

__str__

Kind: Method

```
def __str__(self) -> str:
```

to_expression

Kind: Method

```
def to_expression(self) -> Expression:
```

Convert the representation to a symbolic expression.

bis

Kind: AssociatedFunction

```
def bis(dimension: int | Expression | str) -> Representation:
```

Create a bispinor representation.

```
# Parameters:
- dimension: The dimension of the bispinor space
```

dimension

Kind: Parameter

```
int | Expression | str
```

The dimension of the bispinor space

euc

Kind: AssociatedFunction

```
def euc(dimension: int | Expression | str) -> Representation:
```

Create a Euclidean space representation.

```
# Parameters:
- dimension: The dimension of the Euclidean space
```

dimension

Kind: Parameter

```
int | Expression | str
```

The dimension of the Euclidean space

mink

Kind: AssociatedFunction

```
def mink(dimension: int | Expression | str) -> Representation:
```

Create a Minkowski space representation.

```
# Parameters:
- dimension: The dimension of the Minkowski space
```

dimension

Kind: Parameter

```
int | Expression | str
```

The dimension of the Minkowski space

cof

Kind: AssociatedFunction

```
def cof(dimension: int | Expression | str) -> Representation:
```

Create a color fundamental representation.

```
# Parameters:
- dimension: The dimension of the color group (e.g., 3 for SU(3))
```

dimension

Kind: Parameter

```
int | Expression | str
```

The dimension of the color group (e.g., 3 for SU(3))

coad

Kind: AssociatedFunction

```
def coad(dimension: int | Expression | str) -> Representation:
```

Create a color adjoint representation.

```
# Parameters:
- dimension: The dimension of the adjoint representation (e.g., 8 for SU(3))
```

dimension

Kind: Parameter

```
int | Expression | str
```

The dimension of the adjoint representation (e.g., 8 for SU(3))

cos

Kind: AssociatedFunction

```
def cos(dimension: int | Expression | str) -> Representation:
```

Create a color sextet representation.

```
# Parameters:
- dimension: The dimension of the sextet representation (e.g., 6 for SU(3))
```

dimension

Kind: Parameter

```
int | Expression | str
```

The dimension of the sextet representation (e.g., 6 for SU(3))

Exhaustive reference · Source

Slot

A tensor index slot combining a representation with an abstract index.

A tensor index slot combining a representation with an abstract index.

Slots are the building blocks for tensor structures, pairing a representation
(which defines transformation properties) with an abstract index identifier.
Slots with matching representations and indices can be contracted.

Examples:
```python
from symbolica.community.spenso import Slot, Representation
import symbolica as sp

# Create representation and slots
rep = Representation.euc(3)
slot1 = rep('mu')           # Slot with string index
slot2 = rep(1)              # Slot with integer index
slot3 = rep(sp.S('nu'))     # Slot with symbolic index

# Create custom slot
custom_slot = Slot("MyRep", 4, 'alpha', dual=False)

# Use in tensor structures
from symbolica.community.spenso import TensorIndices
tensor_structure = TensorIndices(slot1, slot2)
```

```
class Slot:
```

Requires features: `python_stubgen`

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

__str__

Kind: Method

```
def __str__(self) -> str:
```

dual

Kind: Method

```
def dual(self) -> Slot:
```

__new__

Kind: AssociatedFunction

```
def __new__(cls, name: str, dimension: int, aind: int | Expression | str, dual: bool
= False) -> Slot:
```

Create a new slot with a custom representation and index.

```
# Parameters:
- name: String name for the representation
- dimension: Size of the representation space
- aind: The abstract index (int, str, or Symbol)
- dual: If True, creates dualizable representation; if False, self-dual

# Examples:
```python
from symbolica.community.spenso import Slot
import symbolica as sp

# Self-dual slot
euclidean_slot = Slot("Euclidean", 4, 'mu', dual=False)

# Dualizable slot
vector_slot = Slot("Vector", 4, 'nu', dual=True)

# With symbolic index
sym_index = sp.S('alpha')
symbolic_slot = Slot("Custom", 3, sym_index, dual=False)
```
```

name

Kind: Parameter

str

String name for the representation

dimension

Kind: Parameter

int

Size of the representation space

aind

Kind: Parameter

int | Expression | str

The abstract index (int, str, or Symbol)

dual

Kind: Parameter

bool

Default: False

If True, creates dualizable representation; if False, self-dual

to_expression

Kind: Method

def to_expression(self) -> Expression:

Convert the slot to a symbolic expression.

```
# Examples:
```python
```
```

```
rep = Representation.euc(3)
slot = rep('mu')
expr = slot.to_expression()  # Symbolic representation of the slot
```

Exhaustive reference · Source

Tensor

```
A tensor class that can be either dense or sparse with flexible data types.
```

```
A tensor class that can be either dense or sparse with flexible data types.

The tensor can store data as floats, complex numbers, or symbolic expressions.
Tensors have an associated structure that defines their shape and index properties.

Examples
--------
>>> from symbolica.community.spenso import Tensor, TensorIndices, Representation
>>> structure = TensorIndices(Representation.euc(4)(1))
>>> data = [1.0, 2.0, 3.0, 4.0]
>>> tensor = Tensor.dense(structure, data)
>>> sparse_tensor = Tensor.sparse(structure, float)
```

```
class Tensor:
```

Requires features: `python_stubgen`

structure

Kind: `Method`

```
def structure(self) -> TensorIndices:
```

sparse

Kind: `AssociatedFunction`

```
def sparse(structure: TensorIndices | list[Slot], type_info: type) -> Tensor:
```

```
Create a new sparse empty tensor with the given structure and data type.

Parameters
----------
structure : TensorIndices or list of Slots
    The tensor structure defining shape and index properties
type_info : type
    The data type - either `float` or `Expression` class

Returns
-------
Tensor
    A new sparse tensor with all elements initially zero

Examples
--------
>>> from symbolica.community.spenso import Tensor, TensorIndices, Representation as R
>>> structure = TensorIndices(R.euc(3)(1), R.euc(3)(2))
>>> sparse_float = Tensor.sparse(structure, float)
>>> sparse_sym = Tensor.sparse(structure, symbolica.Expression)
```

structure

Kind: `Parameter`

```
TensorIndices | list[Slot]
```

The tensor structure defining shape and index properties

type_info

Kind: Parameter

```
type
```

The data type - either `float` or `Expression` class

dense

Kind: AssociatedFunction

```
def dense(structure: TensorIndices | list[Slot], data: Sequence[Expression] |
Sequence[float] | Sequence[complex]) -> Tensor:
```

Create a new dense tensor with the given structure and data.

```
Parameters
----------
structure : TensorIndices or list of Slots
    The tensor structure defining shape and index properties
data : list of float, complex, or Expression
    The tensor data in row-major order

Returns
-------
Tensor
    A new dense tensor with the specified data

Examples
--------
>>> from symbolica import S
>>> from symbolica.community.spenso import Tensor, TensorIndices, Representation as R
>>> structure = TensorIndices(R.euc(2)(1), R.euc(2)(2))
>>> data = [1.0, 2.0, 3.0, 4.0]
>>> tensor = Tensor.dense(structure, data)
>>> x, y = S("x", "y")
>>> sym_data = [x, y, x * y, x + y]
>>> sym_tensor = Tensor.dense(structure, sym_data)
```

structure

Kind: Parameter

```
TensorIndices | list[Slot]
```

The tensor structure defining shape and index properties

data

Kind: Parameter

```
Sequence[Expression] | Sequence[float] | Sequence[complex]
```

The tensor data in row-major order

one

Kind: AssociatedFunction

```
def one() -> Tensor:
```

```
Create a scalar tensor with value 1.0.

Returns
-------
Tensor
    A scalar tensor containing the value 1.0

Examples
--------
>>> from symbolica.community.spenso import Tensor
>>> one = Tensor.one()
```

zero

Kind: `AssociatedFunction`

```
def zero() -> Tensor:

Create a scalar tensor with value 0.0.

Returns
-------
Tensor
    A scalar tensor containing the value 0.0

Examples
--------
>>> from symbolica.community.spenso import Tensor
>>> zero = Tensor.zero()
```

to_dense

Kind: `Method`

```
def to_dense(self) -> None:

Convert this tensor to dense storage format.

Converts sparse tensors to dense format in-place. Dense tensors are unchanged.
This allocates memory for all tensor elements.

Examples
--------
>>> from symbolica.community.spenso import Tensor, TensorIndices, Representation as R
>>> structure = TensorIndices(R.euc(4)(2))
>>> tensor = Tensor.sparse(structure, float)
>>> tensor[0] = 1.0
>>> tensor.to_dense()
```

to_sparse

Kind: `Method`

```
def to_sparse(self) -> None:

Convert this tensor to sparse storage format.

Converts dense tensors to sparse format in-place, only storing non-zero elements.
This can save memory for tensors with many zero elements.

Examples
--------
```

```
>>> from symbolica.community.spenso import Tensor, TensorIndices, Representation as R
>>> structure = TensorIndices(R.euc(2)(2), R.euc(2)(1))
>>> data = [1.0, 0.0, 0.0, 2.0]
>>> tensor = Tensor.dense(structure, data)
>>> tensor.to_sparse()
```

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

__str__

Kind: Method

```
def __str__(self) -> str:
```

__len__

Kind: Method

```
def __len__(self) -> int:
```

__getitem__

Kind: Method

__getitem__

Kind: Overload

```
def __getitem__(self, item: slice | int | list[int]) -> Any:
```

item

Kind: Parameter

```
slice | int | list[int]
```

__getitem__

Kind: Overload

```
def __getitem__(self, item: slice) -> list[Expression | complex | float]:
```

```
Get tensor elements at the specified range of indices.

Parameters
----------
item : slice
    Slice object defining the range of indices

Returns
-------
list of float, complex, or Expression
    The tensor elements at the specified range
```

item

Kind: Parameter

```
slice
```

```
Slice object defining the range of indices
```

__getitem__

Kind: Overload

```
def __getitem__(self, item: Sequence[int] | int) -> Expression | complex | float:
```

```
Get tensor element at the specified index or indices.

Parameters
----------
item : int or list of int
    Index specification (int for flat index, list of int for coordinates)

Returns
-------
float, complex, or Expression
    The tensor element at the specified index
```

item

Kind: Parameter

```
Sequence[int] | int
```

```
Index specification (int for flat index, list of int for coordinates)
```

__setitem__

Kind: Method

__setitem__

Kind: Overload

```
def __setitem__(self, item: Any, value: Any) -> None:
```

```
Set tensor element(s) at the specified index or indices.

Parameters
----------
item : int or list of int
    Index specification (int for flat index, list of int for coordinates)
value : float, complex, or Expression
    The value to set

Examples
--------
>>> from symbolica.community.spenso import Tensor, TensorIndices, Representation as R
>>> structure = TensorIndices(R.euc(2)(2), R.euc(2)(1))
>>> tensor = Tensor.sparse(structure, float)
>>> tensor[0] = 4.0
>>> tensor[1, 1] = 1.0
```

item

Kind: Parameter

```
Any
```

```
Index specification (int for flat index, list of int for coordinates)
```

value

Kind: Parameter

```
Any
```

```
The value to set
```

__setitem__

Kind: Overload

```
def __setitem__(self, item: int | Sequence[int], value: Expression | complex | float)
-> None:
```

Set tensor element at the specified index.

```
Parameters
----------
item : int or list of int
    Index specification (int for flat index, list of int for coordinates)
value : float, complex, or Expression
    The value to set

Examples
--------
>>> from symbolica.community.spenso import Tensor, TensorIndices, Representation
>>> rep = Representation.euc(2)
>>> structure = TensorIndices(rep(1), rep(2))
>>> tensor = Tensor.sparse(structure, float)
>>> tensor[0] = 1.0
>>> tensor[1, 1] = 2.0
```

item

Kind: Parameter

```
int | Sequence[int]
```

Index specification (int for flat index, list of int for coordinates)

value

Kind: Parameter

```
Expression | complex | float
```

The value to set

evaluator

Kind: Method

```
def evaluator(self, constants: Mapping[Expression, Expression], funs:
Mapping[tuple[Expression, str, Sequence[Expression]], Expression], params:
Sequence[Expression], iterations: int = 100, n_cores: int = 4, verbose: bool = False)
-> TensorEvaluator:
```

Create an optimized evaluator for symbolic tensor expressions.

```
Compiles the symbolic expressions in this tensor into an optimized evaluation tree
that can efficiently compute numerical values for different parameter inputs.

Parameters
----------
constants : dict
    Dict mapping symbolic expressions to their constant numerical values
funs : dict
    Dict mapping function signatures to their symbolic definitions
params : list of Expression
    List of symbolic parameters that will be varied during evaluation
iterations : int, optional
```

```
    Number of optimization iterations for Horner scheme (default: 100)
n_cores : int, optional
    Number of CPU cores to use for optimization (default: 4)
verbose : bool, optional
    Whether to print optimization progress (default: False)

Returns
-------
TensorEvaluator
    An optimized evaluator for efficient numerical evaluation

Examples
--------
>>> from symbolica import S
>>> from symbolica.community.spenso import Tensor, TensorIndices, Representation as R
>>> x, y = S("x", "y")
>>> structure = TensorIndices(R.euc(2)(1))
>>> tensor = Tensor.dense(structure, [x * y, x + y])
>>> evaluator = tensor.evaluator(constants={}, funs={}, params=[x, y], iterations=50)
>>> results = evaluator.evaluate_complex([[1.0, 2.0], [3.0, 4.0]])
```

constants

Kind: Parameter

Mapping[Expression, Expression]

Dict mapping symbolic expressions to their constant numerical values

funs

Kind: Parameter

Mapping[tuple[Expression, str, Sequence[Expression]], Expression]

Dict mapping function signatures to their symbolic definitions

params

Kind: Parameter

Sequence[Expression]

List of symbolic parameters that will be varied during evaluation

iterations

Kind: Parameter

int

Default: 100

Number of optimization iterations for Horner scheme (default: 100)

n_cores

Kind: Parameter

int

Default: 4

Number of CPU cores to use for optimization (default: 4)

verbose

Kind: `Parameter`

```
bool
```

Default: `False`

```
Whether to print optimization progress (default: False)
```

scalar

Kind: `Method`

```
def scalar(self) -> Expression:
```

```
Extract the scalar value from a rank-0 (scalar) tensor.

Returns
-------
Expression
    The scalar expression contained in this tensor

Raises
------
RuntimeError
    If the tensor is not a scalar

Examples
--------
>>> from symbolica.community.spenso import Tensor
>>> scalar_tensor = Tensor.one()
>>> value = scalar_tensor.scalar()
```

__iter__

Kind: `Method`

```
def __iter__(self) -> Iterator[Any]:
```

```
Iterator
```

Exhaustive reference · Source

TensorEvaluator

```
An optimized evaluator for symbolic tensor expressions.
```

```
An optimized evaluator for symbolic tensor expressions.

This class provides efficient numerical evaluation of symbolic tensor expressions
after optimization. It supports both real and complex-valued evaluations.

Create instances using the `Tensor.evaluator()` method rather than directly.

Examples
--------
>>> evaluator = my_tensor.evaluator(constants={}, funs={}, params=[x, y])
>>> results = evaluator.evaluate([[1.0, 2.0], [3.0, 4.0]])
```

```
class TensorEvaluator:
```

Requires features: `python_stubgen`

evaluate

Kind: Method

```
def evaluate(self, inputs: Sequence[Sequence[float]]) -> list[Tensor]:
```

```
Evaluate the tensor expression for multiple real-valued parameter inputs.

Parameters
----------
inputs : list of list of float
    List of parameter value lists, where each inner list contains
    numerical values for all parameters in the same order as specified
    when creating the evaluator

Returns
-------
list of Tensor
    List of evaluated tensors, one for each input parameter set

Raises
------
ValueError
    If the evaluator contains complex coefficients

Examples
--------
>>> results = evaluator.evaluate([[1.0, 2.0], [3.0, 4.0]])
```

inputs

Kind: Parameter

```
Sequence[Sequence[float]]
```

```
List of parameter value lists, where each inner list contains numerical values for
all parameters in the same order as specified when creating the evaluator
```

evaluate_complex

Kind: Method

```
def evaluate_complex(self, inputs: Sequence[Sequence[complex]]) -> list[Tensor]:
```

```
Evaluate the expression for multiple inputs and return the results.
```

inputs

Kind: Parameter

```
Sequence[Sequence[complex]]
```

compile

Kind: Method

```
def compile(self, function_name: str, filename: str, library_name: str, inline_asm:
str = 'default', optimization_level: int = 3, compiler_path: Optional[str] = None,
custom_header: Optional[str] = None) -> CompiledTensorEvaluator:
```

```
Compile the evaluator to a shared library using C++ for maximum performance.

Generates optimized C++ code with optional inline assembly and compiles it
into a shared library that can be loaded for extremely fast evaluation.
```

```
Parameters
----------
function_name : str
    Name for the generated C++ function
filename : str
    Path for the generated C++ source file
library_name : str
    Name for the compiled shared library
inline_asm : str, optional
    Type of inline assembly optimization ("default", "x64", "aarch64", "none")
optimization_level : int, optional
    Compiler optimization level 0-3 (default: 3)
compiler_path : str, optional
    Path to specific C++ compiler (default: system default)
custom_header : str, optional
    Additional C++ header code to include

Returns
-------
CompiledTensorEvaluator
    A compiled evaluator for maximum performance evaluation

Examples
--------
>>> compiled = evaluator.compile(
...     function_name="fast_eval",
...     filename="tensor_eval.cpp",
...     library_name="tensor_lib",
...     optimization_level=3,
... )
>>> results = compiled.evaluate_complex([[1.0, 2.0], [3.0, 4.0]])
```

function_name

Kind: Parameter

str

Name for the generated C++ function

filename

Kind: Parameter

str

Path for the generated C++ source file

library_name

Kind: Parameter

str

Name for the compiled shared library

inline_asm

Kind: Parameter

str

Default: `'default'`

Type of inline assembly optimization ("default", "x64", "aarch64", "none")

optimization_level

Kind: Parameter

int

Default: 3

Compiler optimization level 0-3 (default: 3)

compiler_path

Kind: Parameter

Optional[str]

Default: None

Path to specific C++ compiler (default: system default)

custom_header

Kind: Parameter

Optional[str]

Default: None

Additional C++ header code to include

Exhaustive reference · Source

TensorIndices

A tensor structure with abstract indices for symbolic tensor operations.

A tensor structure with abstract indices for symbolic tensor operations.

TensorIndices represents the index structure of tensors with named abstract indices
that can be contracted, manipulated symbolically, and converted to expressions.
It maintains both the representation structure and index assignments.

Examples

```
>>> from symbolica.community.spenso import TensorIndices, Representation, TensorName
>>> rep = Representation.euc(3)
>>> mu = rep('mu')
>>> nu = rep('nu')
>>> indices = TensorIndices(mu, nu)
>>> T = TensorName("T")
>>> named_indices = T(mu, nu)
>>> expr = named_indices.to_expression()
```

class TensorIndices:

Requires features: python_stubgen

set_name

Kind: Method

def set_name(self, name: TensorName | str | Expression) -> None:

Set the tensor name for this structure.

```
Parameters
----------
name : TensorName
    The tensor name to assign

Examples
--------
>>> from symbolica.community.spenso import TensorIndices, TensorName, Representation
>>> rep = Representation.euc(3)
>>> structure = TensorIndices(rep('mu'), rep('nu'))
>>> T = TensorName("T")
>>> structure.set_name(T)
```

name

Kind: Parameter

```
TensorName | str | Expression
```

```
The tensor name to assign
```

get_name

Kind: Method

```
def get_name(self) -> Optional[TensorName]:
```

```
Get the tensor name of this structure.

Returns
-------
TensorName or None
    The tensor name if set, None otherwise

Examples
--------
>>> name = structure.get_name()
```

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

__str__

Kind: Method

```
def __str__(self) -> str:
```

to_expression

Kind: Method

```
def to_expression(self) -> Expression:
```

```
Convert the tensor indices to a symbolic expression.

Creates a symbolic representation of the tensor with its indices that can be
used in algebraic manipulations and pattern matching.

Returns
-------
Expression
    A symbolic Expression representing this indexed tensor
```

```
Raises
------
RuntimeError
    If the tensor structure has no name

Examples
--------
>>> from symbolica.community.spenso import TensorName, Representation
>>> T = TensorName("T")
>>> rep = Representation.euc(3)
>>> mu = rep('mu')
>>> nu = rep('nu')
>>> indices = T(mu, nu)
>>> expr = indices.to_expression()
```

__len__

Kind: Method

```
def __len__(self) -> int:
```

__add__

Kind: Method

```
def __add__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression) -> Expression:
```

Add this expression to `other`, returning the result.

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression
```

__radd__

Kind: Method

```
def __radd__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression) -> Expression:
```

Add this expression to `other`, returning the result.

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression
```

__sub__

Kind: Method

```
def __sub__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression) -> Expression:
```

Subtract `other` from this expression, returning the result.

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression
```

__rsub__

Kind: Method

```
def __rsub__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression) -> Expression:
```

Subtract this expression from `other`, returning the result.

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression
```

__mul__

Kind: Method

```
def __mul__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression) -> Expression:
```

Add this expression to `other`, returning the result.

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression
```

__rmul__

Kind: Method

```
def __rmul__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression) -> Expression:
```

Add this expression to `other`, returning the result.

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression
```

__new__

Kind: AssociatedFunction

```
def __new__(cls, *slots: TensorIndices | list[Slot], name: TensorName | str |
Expression | None = None) -> TensorIndices:
```

Create tensor structure from slots and optional arguments.

```
Parameters
----------
*additional_args : Slot or Expression
    Mixed arguments (Slot objects and Expressions for additional arguments)
name : TensorName, optional
    Optional tensor name to assign to the structure

Returns
-------
TensorIndices
    A new TensorIndices object

Examples
--------
```

```
>>> from symbolica import S
>>> from symbolica.community.spenso import TensorIndices, Representation, TensorName
>>> rep = Representation.euc(3)
>>> mu = rep('mu')
>>> nu = rep('nu')
>>> structure = TensorIndices(mu, nu)
>>> x = S('x')
>>> structure_with_args = TensorIndices(mu, nu, x)
>>> T = TensorName("T")
>>> named_structure = TensorIndices(mu, nu, name=T)
```

slots

Kind: Parameter

```
TensorIndices | list[Slot]
```

name

Kind: Parameter

```
TensorName | str | Expression | None
```

Default: None

```
Optional tensor name to assign to the structure
```

__getitem__

Kind: Method

__getitem__

Kind: Overload

```
def __getitem__(self, item: slice) -> list[Expression | complex | float]:
```

```
Get expanded indices at the specified range of flattened indices.

Parameters
----------
item : slice
    Slice object defining the range of indices

Returns
-------
list of list of int
    List of expanded indices
```

item

Kind: Parameter

```
slice
```

```
Slice object defining the range of indices
```

__getitem__

Kind: Overload

```
def __getitem__(self, item: Sequence[int]) -> Expression | complex | float:
```

```
Get flattened index associated to this expanded index.

Parameters
----------
```

```
item : list of int
    Multi-dimensional index coordinates

Returns
-------
int
    The flat index
```

item

Kind: Parameter

```
Sequence[int]
```

```
Multi-dimensional index coordinates
```

__getitem__

Kind: Overload

```
def __getitem__(self, item: int) -> Expression | complex | float:
```

```
Get expanded index associated to this flat index.

Parameters
----------
item : int
    Flat index into the tensor

Returns
-------
list of int
    Multi-dimensional index coordinates
```

item

Kind: Parameter

```
int
```

```
Flat index into the tensor
```

Exhaustive reference · Source

TensorLibrary

```
A library for registering and managing tensor templates and structures.
```

```
A library for registering and managing tensor templates and structures.

The TensorLibrary provides a centralized registry for tensor definitions that can be
reused across tensor networks and expressions. It manages tensor structures with
their
associated names and can resolve symbolic references to registered tensors.

```python
import symbolica
from symbolica.community.spenso import TensorLibrary, LibraryTensor, TensorStructure,
Representation

lib = TensorLibrary()
rep = Representation.euc(3)
name = symbolica.S("my_tensor")
structure = TensorStructure(rep, rep, name=name)
```
```

```
tensor = LibraryTensor.dense(structure, [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0])
lib.register(tensor)
tensor_ref = lib[name]
```

```
class TensorLibrary:
```

Requires features: `python_stubgen`

`__new__`

Kind: `AssociatedFunction`

```
def __new__(cls) -> TensorLibrary:
```

```
Create a new empty tensor library.

Initializes an empty library ready for registering tensor structures.
The library automatically manages internal tensor IDs.

Returns
-------
TensorLibrary
    A new empty tensor library

Examples
--------
>>> from symbolica.community.spenso import TensorLibrary
>>> lib = TensorLibrary()
```

`construct`

Kind: `AssociatedFunction`

```
def construct() -> TensorLibrary:
```

```
Create a new empty tensor library (static method).

Initializes an empty library ready for registering tensor structures.
The library automatically manages internal tensor IDs.

Returns
-------
TensorLibrary
    A new empty tensor library

Examples
--------
>>> from symbolica.community.spenso import TensorLibrary
>>> lib = TensorLibrary.construct()
```

`register`

Kind: `Method`

```
def register(self, tensor: Tensor | LibraryTensor) -> None:
```

```
Register a tensor in the library.

Adds a tensor template to the library that can be referenced by name
in tensor networks and symbolic expressions. The tensor must have a name
set in its structure.
```

```
Parameters
----------
tensor : LibraryTensor or Tensor
    The tensor to register - can be a LibraryTensor or regular Tensor

Examples
--------
>>> import symbolica
>>> from symbolica.community.spenso import TensorLibrary, LibraryTensor,
TensorStructure, Representation
>>> lib = TensorLibrary()
>>> rep = Representation.euc(3)
>>> name = symbolica.S("my_tensor")
>>> structure = TensorStructure(rep, rep, name=name)
>>> tensor = LibraryTensor.dense(structure, [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0])
>>> lib.register(tensor)
>>> tensor_ref = lib[name]
```

tensor

Kind: Parameter

```
Tensor | LibraryTensor
```

```
The tensor to register - can be a LibraryTensor or regular Tensor
```

__getitem__

Kind: Method

```
def __getitem__(self, key: Expression | int | str | float | complex | str) ->
TensorStructure:
```

```
Retrieve a registered tensor structure by name.

Looks up a previously registered tensor by its name and returns
a reference structure that can be used to create new tensor instances.

Parameters
----------
key : str or Expression
    The tensor name - can be a string or symbolic expression

Returns
-------
TensorStructure
    A TensorStructure representing the registered tensor template

Raises
------
RuntimeError
    If the tensor name is not found in the library

Examples
--------
>>> structure = lib["T"]
```

key

Kind: Parameter

```
Expression | int | str | float | complex | str
```

```
The tensor name - can be a string or symbolic expression
```

hep_lib

Kind: AssociatedFunction

```
def hep_lib() -> TensorLibrary:
```

```
Create a library pre-loaded with High Energy Physics tensor definitions.

Returns a library containing standard HEP tensors such as gamma matrices,
color generators, metric tensors, and other commonly used structures in
particle physics calculations. They are floating point tensors with f64 precision.

Returns
-------
TensorLibrary
    A TensorLibrary pre-populated with HEP tensor definitions

Examples
--------
>>> import symbolica
>>> from symbolica.community.spenso import TensorLibrary, TensorName
>>> hep_lib = TensorLibrary.hep_lib()
>>> gamma_structure = hep_lib[symbolica.S("spenso::gamma")]
```

hep_lib_atom

Kind: AssociatedFunction

```
def hep_lib_atom() -> TensorLibrary:
```

```
Create a library pre-loaded with High Energy Physics tensor definitions.

Returns a library containing standard HEP tensors such as gamma matrices,
color generators, metric tensors, and other commonly used structures in
particle physics calculations. They are tensors with atom numeric entries.

Returns
-------
TensorLibrary
    A TensorLibrary pre-populated with HEP tensor definitions

Examples
--------
>>> import symbolica
>>> from symbolica.community.spenso import TensorLibrary, TensorName
>>> hep_lib = TensorLibrary.hep_lib_atom()
>>> gamma_structure = hep_lib[symbolica.S("spenso::gamma")]
```

Exhaustive reference · Source

TensorName

```
A symbolic name for tensor functions and structures.
```


```
A symbolic name for tensor functions and structures.
```

```
TensorName represents named tensor functions that can be called with indices and
arguments
to create tensor structures. Names can have various mathematical properties like
symmetry,
antisymmetry, and custom normalization or printing behavior.

Examples
--------
>>> from symbolica.community.spenso import TensorName, Slot, Representation
>>> T = TensorName("T")
>>> symmetric_T = TensorName("S", is_symmetric=True)
>>> antisymmetric_T = TensorName("A", is_antisymmetric=True)
>>> rep = Representation.cof(3)
>>> mu = rep('mu')
>>> nu = rep('nu')
>>> tensor_structure = T(mu, nu)
```

```
class TensorName:
```

Requires features: `python_stubgen`

`__new__`

Kind: `AssociatedFunction`

```
def __new__(cls, name: str, is_symmetric: Optional[bool] = None, is_antisymmetric:
Optional[bool] = None, is_cyclesymmetric: Optional[bool] = None, is_linear:
Optional[bool] = None, custom_normalization: Optional[Transformer] = None) ->
TensorName:
```

```
Create a new tensor name with optional mathematical properties.

Parameters
----------
name : str
    The string name for the tensor function
is_symmetric : bool, optional
    If True, tensor is symmetric under index permutation
is_antisymmetric : bool, optional
    If True, tensor is antisymmetric under index permutation
is_cyclesymmetric : bool, optional
    If True, tensor is symmetric under cyclic permutations
is_linear : bool, optional
    If True, tensor is linear in its arguments
custom_normalization : Transformer, optional
    Custom normalization function (advanced)

Returns
-------
TensorName
    A new TensorName with the specified properties

Examples
--------
>>> from symbolica.community.spenso import TensorName
>>> T = TensorName("T")
>>> g = TensorName("g", is_symmetric=True)
>>> F = TensorName("F", is_antisymmetric=True)
>>> D = TensorName("D", is_linear=True)
```

name
Kind: Parameter

str

The string name for the tensor function

is_symmetric
Kind: Parameter

Optional[bool]

Default: None

If True, tensor is symmetric under index permutation

is_antisymmetric
Kind: Parameter

Optional[bool]

Default: None

If True, tensor is antisymmetric under index permutation

is_cyclesymmetric
Kind: Parameter

Optional[bool]

Default: None

If True, tensor is symmetric under cyclic permutations

is_linear
Kind: Parameter

Optional[bool]

Default: None

If True, tensor is linear in its arguments

custom_normalization
Kind: Parameter

Optional[Transformer]

Default: None

Custom normalization function (advanced)

__repr__
Kind: Method

```
def __repr__(self) -> str:
```

__str__
Kind: Method

```
def __str__(self) -> str:
```

to_expression

Kind: Method

```
def to_expression(self) -> Expression:
```

```
Convert the tensor name to a symbolic expression.

Returns
-------
Expression
    A symbolic Expression representing this tensor name

Examples
--------
>>> T = TensorName("T")
>>> expr = T.to_expression()
```

g

Kind: AssociatedFunction

```
def g() -> TensorName:
```

```
Predefined metric tensor name.
```

flat

Kind: AssociatedFunction

```
def flat() -> TensorName:
```

```
Predefined musical isomorphism tensor name. This enables dualizing self dual indices.
```

gamma

Kind: AssociatedFunction

```
def gamma() -> TensorName:
```

```
Predefined gamma matrix name.
```

gamma5

Kind: AssociatedFunction

```
def gamma5() -> TensorName:
```

```
Predefined gamma5 matrix name.
```

projm

Kind: AssociatedFunction

```
def projm() -> TensorName:
```

```
Predefined left chiral projector name.
```

projp

Kind: AssociatedFunction

```
def projp() -> TensorName:
```

```
Predefined right chiral projector name.
```

sigma

Kind: AssociatedFunction

```
def sigma() -> TensorName:
```

Predefined sigma matrix name.

f

Kind: AssociatedFunction

def f() -> TensorName:

Predefined color structure constant name.

t

Kind: AssociatedFunction

def t() -> TensorName:

Predefined color generator name.

__call__

Kind: Method

__call__

Kind: Overload

def __call__(self, *args: Slot | Expression | int | str | float | complex) -> TensorIndices:

Call the tensor name with arguments to create tensor structures.

Accepts a mix of slots and symbolic expressions (for additional arguments).

Parameters
----
*args : Slot or Expression
    Slot objects and Expressions for additional arguments

Returns

TensorIndices
    A new TensorIndices object

Examples

>>> from symbolica.community.spenso import TensorName, Slot, Representation
>>> import symbolica as sp
>>> T = TensorName("T")
>>> rep = Representation.euc(3)
>>> mu = rep("mu")
>>> nu = rep("nu")
>>> indexed_tensor = T(mu, nu)

args

Kind: Parameter

Slot | Expression | int | str | float | complex

__call__

Kind: Overload

def __call__(self, *args: Representation | Expression) -> TensorStructure:

```
Call the tensor name with arguments to create a TensorStructure.

Accepts a mix of representations and symbolic expressions (for additional arguments).

Parameters
----------
*args : Representation or Expression
    Representation objects and Expressions for additional arguments

Returns
-------
TensorStructure
    A new TensorStructure object

Examples
--------
>>> from symbolica.community.spenso import TensorName, Slot, Representation
>>> import symbolica as sp
>>> T = TensorName("T")
>>> rep = Representation.euc(3)
>>> structure_tensor = T(rep, rep)
```

args

Kind: `Parameter`

```
Representation | Expression
```

Exhaustive reference · Source

TensorNetwork

```
A graph of tensor operations that can be simplified and executed.
```


```
A graph of tensor operations that can be simplified and executed.
```


```
Named tensor expressions are resolved through a `TensorLibrary`. Register concrete
data
before constructing and executing a network; an expression alone supplies structure,
not
component values.

Examples
--------
>>> from symbolica.community.spenso import (
...     ExecutionMode,
...     LibraryTensor,
...     Representation,
...     TensorLibrary,
...     TensorName,
...     TensorNetwork,
...     TensorStructure,
... )
>>> rep = Representation.euc(2)
>>> A = TensorName("A")
>>> structure = TensorStructure(rep, rep, name=A)
>>> library = TensorLibrary()
>>> library.register(
...     LibraryTensor.dense(structure, [1.0, 0.0, 0.0, 1.0])
... )
```

```
>>> network = TensorNetwork(
...     A(rep("i"), rep("j")),
...     library=library,
... )
>>> network.execute(library=library, mode=ExecutionMode.All)
>>> result = network.result_tensor(library=library)
>>> len(result)
4
```

```
class TensorNetwork:
```

Requires features: `python_stubgen`

`__new__`

Kind: `AssociatedFunction`

```
def __new__(cls, expr: Expression | int | str | float | complex | TensorIndices |
Expression, library: Optional[TensorLibrary] = None) -> TensorNetwork:
```

```
Create a tensor network by parsing an arithmetic expression.

Parses symbolic expressions containing tensor operations and converts them
into an optimizable computational graph representation.

Parameters
----------
expr : ArithmeticStructure
    The arithmetic expression or tensor structure to parse
library : TensorLibrary, optional
    Optional tensor library for resolving named tensor references

Returns
-------
TensorNetwork
    A new TensorNetwork representing the parsed expression
```

`expr`

Kind: `Parameter`

```
Expression | int | str | float | complex | TensorIndices | Expression
```

```
The arithmetic expression or tensor structure to parse
```

`library`

Kind: `Parameter`

```
Optional[TensorLibrary]
```

Default: `None`

```
Optional tensor library for resolving named tensor references
```

`one`

Kind: `AssociatedFunction`

```
def one() -> TensorNetwork:
```

```
Create a tensor network representing the scalar value 1.

Returns
-------
```

```
TensorNetwork
    A TensorNetwork containing only the scalar 1

Examples
--------
>>> from symbolica.community.spenso import TensorNetwork
>>> one_net = TensorNetwork.one()
>>> result = one_net.result_scalar()
```

bracket

Kind: AssociatedFunction

```
def bracket() -> Expression:
```

broadcast

Kind: AssociatedFunction

```
def broadcast(str: str) -> Expression:
```

str

Kind: Parameter

```
str
```

zero

Kind: AssociatedFunction

```
def zero() -> TensorNetwork:
```

```
Create a tensor network representing the scalar value 0.

Returns
-------
TensorNetwork
    A TensorNetwork containing only the scalar 0

Examples
--------
>>> from symbolica.community.spenso import TensorNetwork
>>> zero_net = TensorNetwork.zero()
>>> result = zero_net.result_scalar()
```

replace

Kind: Method

```
def replace(self, pattern: Expression | int | str | float | complex, rhs: Expression
| int | str | float | complex | HeldExpression | Callable[[dict[Expression,
Expression]], Expression] | int | float | complex | decimal.Decimal, _cond:
Optional[PatternRestriction | Condition] = None, non_greedy_wildcards:
Optional[Sequence[Expression]] = None, level_range: Optional[tuple[int,
Optional[int]]] = None, level_is_tree_depth: Optional[bool] = None,
allow_new_wildcards_on_rhs: Optional[bool] = None, rhs_cache_size: Optional[int] =
None, repeat: Optional[bool] = None) -> TensorNetwork:
```

```
Replace patterns in the tensor network using symbolic pattern matching.

Applies pattern-based transformations to the network structure, allowing for
symbolic simplifications, substitutions, and algebraic manipulations.
```

```
Parameters
----------
pattern : Expression
    The symbolic pattern to match against
rhs : Expression
    The replacement expression or pattern
non_greedy_wildcards : list of Expression, optional
    List of wildcard symbols to match non-greedily
level_range : tuple of int, optional
    Tuple specifying depth range for pattern matching
level_is_tree_depth : bool, optional
    Whether level refers to tree depth or expression depth
allow_new_wildcards_on_rhs : bool, optional
    Allow new wildcards in replacement pattern
rhs_cache_size : int, optional
    Size of cache for replacement pattern compilation
repeat : bool, optional
    Whether to repeatedly apply the replacement until no more matches

Returns
-------
TensorNetwork
    A new TensorNetwork with the replacements applied
```

pattern

Kind: Parameter

```
Expression | int | str | float | complex
```

```
The symbolic pattern to match against
```

rhs

Kind: Parameter

```
Expression | int | str | float | complex | HeldExpression |
Callable[[dict[Expression, Expression]], Expression] | int | float | complex |
decimal.Decimal
```

```
The replacement expression or pattern
```

_cond

Kind: Parameter

```
Optional[PatternRestriction | Condition]
```

Default: None

non_greedy_wildcards

Kind: Parameter

```
Optional[Sequence[Expression]]
```

Default: None

```
List of wildcard symbols to match non-greedily
```

level_range

Kind: Parameter

```
Optional[tuple[int, Optional[int]]]
```

Default: None

Tuple specifying depth range for pattern matching

level_is_tree_depth
Kind: Parameter

Optional[bool]

Default: None

Whether level refers to tree depth or expression depth

allow_new_wildcards_on_rhs
Kind: Parameter

Optional[bool]

Default: None

Allow new wildcards in replacement pattern

rhs_cache_size
Kind: Parameter

Optional[int]

Default: None

Size of cache for replacement pattern compilation

repeat
Kind: Parameter

Optional[bool]

Default: None

Whether to repeatedly apply the replacement until no more matches

evaluate
Kind: Method

```
def evaluate(self, constants: Mapping[Expression, float], functions:
Mapping[Expression, Any]) -> TensorNetwork:
```

Evaluate symbolic expressions in the network with numerical values.

Substitutes symbolic constants and functions with numerical values,
converting symbolic parts of the network to concrete numerical tensors.

Parameters
----
constants : dict
    Dict mapping symbolic expressions to their numerical values
functions : dict
    Dict mapping function symbols to Python callable objects

Returns

TensorNetwork
    A new TensorNetwork with symbolic expressions evaluated

constants

Kind: Parameter

```
Mapping[Expression, float]
```

Dict mapping symbolic expressions to their numerical values

functions

Kind: Parameter

```
Mapping[Expression, Any]
```

Dict mapping function symbols to Python callable objects

execute

Kind: Method

```
def execute(self, library: Optional[TensorLibrary] = None, function_library: None =
None, n_steps: Optional[int] = None, mode: ExecutionMode = ExecutionMode.All) ->
None:
```

Execute the tensor network to perform tensor contractions and simplifications.

Processes the computational graph by executing tensor operations such as
contractions, additions, and multiplications. The execution can be controlled
by mode and step limits.

Parameters
----
library : TensorLibrary, optional
    Optional tensor library for resolving tensor operations
function_library : None, optional
    Reserved for an internally supplied function library
n_steps : int, optional
    Maximum number of execution steps (None for complete execution)
mode : ExecutionMode, optional
    Execution strategy. ExecutionMode.Single selects one smallest-degree rewrite per
    step; use n_steps to bound how many steps run.

Examples

>>> from symbolica.community.spenso import TensorNetwork, ExecutionMode,
TensorLibrary
>>> network = TensorNetwork(some_expression)
>>> network.execute()
>>> network.execute(n_steps=5)
>>> network.execute(mode=ExecutionMode.Scalar)
>>> lib = TensorLibrary.hep_lib()
>>> network.execute(library=lib)

library

Kind: Parameter

```
Optional[TensorLibrary]
```

Default: None

Optional tensor library for resolving tensor operations

function_library

Kind: Parameter

None

Default: None

Reserved for an internally supplied function library

n_steps

Kind: Parameter

Optional[int]

Default: None

Maximum number of execution steps (None for complete execution)

mode

Kind: Parameter

ExecutionMode

Default: ExecutionMode.All

Execution strategy. ExecutionMode.Single selects one smallest-degree rewrite per step; use n_steps to bound how many steps run.

result_tensor

Kind: Method

```
def result_tensor(self, library: Optional[TensorLibrary] = None) -> Tensor:
```

Extract the final tensor result from the executed network.

After network execution, retrieves the computed tensor result. The network should be executed before calling this method.

```
Parameters
----------
library : TensorLibrary, optional
    Optional tensor library for resolving tensor structures

Returns
-------
Tensor
    The computed tensor result

Raises
------
RuntimeError
    If the network execution resulted in an error

Examples
--------
>>> from symbolica.community.spenso import TensorNetwork, TensorLibrary
>>> network = TensorNetwork(tensor_expression)
>>> network.execute()
>>> result = network.result_tensor()
```

```
>>> lib = TensorLibrary.hep_lib()
>>> result_with_lib = network.result_tensor(library=lib)
```

library

Kind: Parameter

```
Optional[TensorLibrary]
```

Default: None

```
Optional tensor library for resolving tensor structures
```

result_scalar

Kind: Method

```
def result_scalar(self) -> Expression:
```

```
Extract the final scalar result from the executed network.

For networks that evaluate to scalar expressions, retrieves the computed
scalar value. The network should be executed before calling this method.

Returns
-------
Expression
    The computed scalar expression

Raises
------
RuntimeError
    If the network execution resulted in an error

Examples
--------
>>> from symbolica.community.spenso import TensorNetwork
>>> network = TensorNetwork(scalar_expression)
>>> network.execute()
>>> scalar_result = network.result_scalar()
```

__str__

Kind: Method

```
def __str__(self) -> str:
```

```
Return a string representation of the network structure.

Generates a DOT format representation of the computational graph that can be
visualized using graphviz or similar tools.
```

__add__

Kind: Method

```
def __add__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression | TensorNetwork | Tensor) -> TensorNetwork:
```

```
Add two tensor networks element-wise.

Parameters
----------
rhs : TensorNetwork
```

```
    The tensor network to add (right-hand side)

Returns
-------
TensorNetwork
    A new TensorNetwork representing the sum

Examples
--------
>>> net1 = TensorNetwork(expr1)
>>> net2 = TensorNetwork(expr2)
>>> sum_net = net1 + net2
```

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression | TensorNetwork
| Tensor
```

```
The tensor network to add (right-hand side)
```

__radd__

Kind: Method

```
def __radd__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression | TensorNetwork | Tensor) -> TensorNetwork:
```

```
Add two tensor networks element-wise (right-hand addition).
```

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression | TensorNetwork
| Tensor
```

__sub__

Kind: Method

```
def __sub__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression | TensorNetwork | Tensor) -> TensorNetwork:
```

```
Subtract one tensor network from another element-wise.

Parameters
----------
rhs : TensorNetwork
    The tensor network to subtract (right-hand side)

Returns
-------
TensorNetwork
    A new TensorNetwork representing the difference

Examples
--------
>>> net1 = TensorNetwork(expr1)
>>> net2 = TensorNetwork(expr2)
>>> diff_net = net1 - net2
```

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression | TensorNetwork
| Tensor
```

The tensor network to subtract (right-hand side)

__rsub__

Kind: Method

```
def __rsub__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression | TensorNetwork | Tensor) -> TensorNetwork:
```

Subtract one tensor network from another (right-hand subtraction).

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression | TensorNetwork
| Tensor
```

__mul__

Kind: Method

```
def __mul__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression | TensorNetwork | Tensor) -> TensorNetwork:
```

Multiply two tensor networks.

```
Parameters
----------
rhs : TensorNetwork
    The tensor network to multiply with (right-hand side)

Returns
-------
TensorNetwork
    A new TensorNetwork representing the product

Examples
--------
>>> net1 = TensorNetwork(expr1)
>>> net2 = TensorNetwork(expr2)
>>> product_net = net1 * net2
```

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression | TensorNetwork
| Tensor
```

The tensor network to multiply with (right-hand side)

__rmul__

Kind: Method

```
def __rmul__(self, rhs: Expression | int | str | float | complex | TensorIndices |
Expression | TensorNetwork | Tensor) -> TensorNetwork:
```

Multiply two tensor networks (right-hand multiplication).

rhs

Kind: Parameter

```
Expression | int | str | float | complex | TensorIndices | Expression | TensorNetwork
| Tensor
```

Exhaustive reference · Source

TensorStructure

A tensor structure without abstract indices, defined purely by representations.

A tensor structure without abstract indices, defined purely by representations.

TensorStructure represents the shape and representation structure of tensors
without specific index assignments. It's used for defining tensor templates
in libraries and for creating indexless tensor computations.

Examples:
```python
from symbolica.community.spenso import TensorStructure, Representation, TensorName

# Create from representations
rep = Representation.euc(3)
structure = TensorStructure(rep, rep)  # 3x3 matrix structure

# With name for library registration
T = TensorName("T")
named_structure = TensorStructure(rep, rep, name=T)

# Use to create indexed tensor
indices = structure.index('mu', 'nu')  # Assign specific indices

# Create symbolic expression
expr = structure.symbolic('a', 'b')  # T(a, b)
```

```
class TensorStructure:
```

Requires features: python_stubgen

set_name

Kind: Method

```
def set_name(self, name: TensorName | str | Expression) -> None:
```

name

Kind: Parameter

```
TensorName | str | Expression
```

get_name

Kind: Method

```
def get_name(self) -> Optional[TensorName]:
```

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

__str__

Kind: Method

```
def __str__(self) -> str:
```

__len__

Kind: Method

```
def __len__(self) -> int:
```

__new__

Kind: AssociatedFunction

```
def __new__(cls, *reps_and_additional_args: TensorIndices | list[Slot], name:
TensorName | str | Expression | None = None) -> TensorStructure:
```

Construct a new TensorStructure with the given representations.

```
Parameters
----------
*reps_and_additional_args : Representation or Expression
    Mixed arguments (Representation objects and Expressions for additional arguments)
name : TensorName, optional
    Optional tensor name to assign to the structure

Returns
-------
TensorStructure
    A new TensorStructure object

Examples
--------
>>> from symbolica import S
>>> from symbolica.community.spenso import TensorStructure, Representation,
TensorName
>>> rep = Representation.euc(3)
>>> structure = TensorStructure(rep, rep)
>>> x = S('x')
>>> structure_with_args = TensorStructure(rep, rep, x)
>>> T = TensorName("T")
>>> named_structure = TensorStructure(rep, rep, name=T)
```

reps_and_additional_args

Kind: Parameter

```
TensorIndices | list[Slot]
```

name

Kind: Parameter

```
TensorName | str | Expression | None
```

Default: None

```
Optional tensor name to assign to the structure
```

__getitem__

Kind: Method

__getitem__

Kind: Overload

```
def __getitem__(self, item: slice) -> list[Expression | complex | float]:
```

```
Get expanded indices at the specified range of flattened indices.

Parameters
----------
item : slice
    Slice object defining the range of indices

Returns
-------
list of list of int
    List of expanded indices
```

item

Kind: Parameter

```
slice
```

```
Slice object defining the range of indices
```

__getitem__

Kind: Overload

```
def __getitem__(self, item: Sequence[int]) -> Expression | complex | float:
```

```
Get flattened index associated to this expanded index.

Parameters
----------
item : list of int
    Multi-dimensional index coordinates

Returns
-------
int
    The flat index
```

item

Kind: Parameter

```
Sequence[int]
```

```
Multi-dimensional index coordinates
```

__getitem__

Kind: Overload

```
def __getitem__(self, item: int) -> Expression | complex | float:
```

```
Get expanded index associated to this flat index.

Parameters
----------
item : int
    Flat index into the tensor

Returns
```

```
-------
list of int
    Multi-dimensional index coordinates
```

item

Kind: Parameter

```
int
```

```
Flat index into the tensor
```

__call__

Kind: Method

```
def __call__(self, *args: int | Expression | str, extra_args: Sequence[Expression |
int | str | float | complex] | None = None) -> Expression:
```

```
Convenience method for creating symbolic expressions.

This is a shorthand for calling `symbolic(*args, extra_args=extra_args)`.
Creates a symbolic Expression representing this tensor structure.

Parameters
----------
*args : int, str, Symbol, or Expression
    Positional arguments (indices and additional args)
extra_args : list of Expression, optional
    Optional list of additional non-tensorial arguments

Returns
-------
Expression
    A symbolic Expression representing the tensor

Examples
--------
>>> structure = TensorStructure(rep, rep, name="T")
>>> expr = structure('mu', 'nu')
```

args

Kind: Parameter

```
int | Expression | str
```

extra_args

Kind: Parameter

```
Sequence[Expression | int | str | float | complex] | None
```

Default: None

```
Optional list of additional non-tensorial arguments
```

symbolic

Kind: Method

```
def symbolic(self, *args: int | Expression | str, extra_args: Sequence[Expression |
int | str | float | complex] | None = None) -> Expression:
```

```
Create a symbolic expression representing this tensor structure.
```

```
Builds a symbolic tensor expression with the specified indices. Arguments can be
separated using a semicolon (';') to distinguish between additional arguments
and tensor indices.

Parameters
----------
*args : int, str, Symbol, Expression, or ';'
    Positional arguments (int, str, Symbol, Expression for indices, ';' for
separator)
extra_args : list of Expression, optional
    Optional list of additional non-tensorial arguments

Returns
-------
Expression
    A symbolic Expression representing the tensor with indices

Examples
--------
>>> import symbolica as sp
>>> from symbolica.community.spenso import TensorStructure, Representation,
TensorName
>>> rep = Representation.euc(3)
>>> T = TensorName("T")
>>> structure = TensorStructure([rep, rep], name=T)
>>> expr = structure.symbolic('mu', 'nu')
>>> x = sp.S('x')
>>> expr = structure.symbolic(x, ';', 'mu', 'nu')
>>> expr = structure.symbolic('mu', 'nu', extra_args=[x])
```

args

Kind: `Parameter`

```
int | Expression | str
```

extra_args

Kind: `Parameter`

```
Sequence[Expression | int | str | float | complex] | None
```

Default: `None`

```
Optional list of additional non-tensorial arguments
```

index

Kind: `Method`

```
def index(self, *args: int | Expression | str, extra_args: Sequence[Expression] |
None = None, cook_indices: bool = False) -> TensorIndices:
```

```
Create an indexed tensor (TensorIndices) from this structure.

Converts this structure template into a concrete indexed tensor by assigning
specific abstract indices to each representation slot.

Parameters
----------
*args : int, str, Symbol, Expression, or ';'
    Positional arguments (indices and ';' separator between additional args and
```

```
indices)
extra_args : list of Expression, optional
    Optional list of additional non-tensorial arguments
cook_indices : bool, optional
    If True, attempt to convert expressions to valid indices

Returns
-------
TensorIndices
    A TensorIndices object with concrete index assignments

Examples
--------
>>> import symbolica as sp
>>> from symbolica.community.spenso import TensorStructure, Representation,
TensorName
>>> rep = Representation.cof(3)
>>> T = TensorName("T")
>>> structure = TensorStructure([rep, rep], name=T)
>>> indices = structure.index('mu', 'nu')
>>> x = sp.S('x')
>>> indices = structure.index(x, ';', 'mu', 'nu')
```

args

Kind: `Parameter`

`int | Expression | str`

extra_args

Kind: `Parameter`

`Sequence[Expression] | None`

Default: `None`

`Optional list of additional non-tensorial arguments`

cook_indices

Kind: `Parameter`

`bool`

Default: `False`

`If True, attempt to convert expressions to valid indices`

Exhaustive reference · Source

ExecutionMode

`Execution modes for tensor network evaluation.`

`Execution modes for tensor network evaluation.`

```
Controls how the tensor network execution engine processes the computational graph.

Variants
--------
Single : Select one smallest-degree rewrite per step; without `n_steps`, continue
until no work remains
```

```
Scalar : Only contract scalar operations, leaving tensor structure intact
All : Execute all possible contractions for complete evaluation
```

```
class ExecutionMode(enum.Enum):
```

Requires features: `python_stubgen`

`Single`

Kind: `Variant`

```
Select one smallest-degree rewrite per step; without `n_steps`, continue until no
work remains
```

`Scalar`

Kind: `Variant`

```
Only contract scalar operations, leaving tensor structure intact
```

`All`

Kind: `Variant`

```
Execute all possible contractions for complete evaluation
```

Exhaustive reference · Source

`SymbolicParallelism`

```
Policy for Rayon operations that manipulate Symbolica expressions.
```

```
Policy for Rayon operations that manipulate Symbolica expressions.
```

```
class SymbolicParallelism(enum.Enum):
```

Requires features: `python_stubgen`

`Auto`

Kind: `Variant`

```
Permit Rayon when licensed and use workload heuristics where available.
```

`Serial`

Kind: `Variant`

```
Keep symbolic operations on the calling thread.
```

`Parallel`

Kind: `Variant`

```
Force Rayon without `Auto`'s Symbolica license safety check.
```

Exhaustive reference · Source

`set_symbolica_rayon_enabled`

```
Configure whether Spenso may use Rayon for Symbolica operations.
```

```
Configure whether Spenso may use Rayon for Symbolica operations.

`Auto` checks the Symbolica license once during this call. When licensed,
operations may use Rayon and apply workload heuristics where available;
when unlicensed, symbolic operations remain serial. The returned boolean
reports whether the resolved policy permits Rayon, not whether every
operation will use it.
```

```
def set_symbolica_rayon_enabled(policy: SymbolicParallelism) -> bool:
```

Requires features: `python_stubgen`

Exhaustive reference · Source

Version information

Save these identifiers with published results and include them in issue reports so that collaborators can reproduce the same software environment.

- **Documentation channel:** `latest`
- **Documentation route:** `/products/spenso/latest/`
- **Source revision:** `e51747446aa724c3ffac5c42c4103d090c09c6b0`

Package versions and enabled features

- `gammaloop/gammalooprs` — `0.3.4` (`no optional features`)
- `gammaloop/gammaloop-api` — `0.1.0` (`features: cli`)
- `gammaloop/gammaloop` — `0.3.4` (`features: python_api, python_abi, python_stubgen`)
- `linnet/linnet` — `0.17.0` (`features: nodestore-vec, serde, bincode, drawing, symbolica`)
- `linnet/linnet-py` — `0.1.0` (`features: extension-module, abi3-py310`)
- `spenso/spenso` — `0.6.0` (`features: shadowing`)
- `spenso/spenso-macros` — `0.3.2` (`features: shadowing`)
- `spenso/spenso-hep-lib` — `0.1.7` (`no optional features`)
- `spenso/spynso3` — `0.1.2` (`features: python_stubgen`)
- `idenso/idenso` — `0.3.0` (`features: bincode, reference-cases`)
- `idenso/idenso` — `0.3.0` (`features: python, python_stubgen`)
- `vakint/vakint` — `0.1.2` (`no optional features`)
- `vakint/vakint` — `0.1.2` (`features: symbolica_community_module, python_stubgen`)