BOVNAR · V1.1

Bovnar Python Bindings

Bovnar (BVNR) v1.1 documentation · Python Bindings · 2026-06-21

> **Version:** 1.1

Pure-`ctypes` Python bindings for the **Bovnar (BVNR)** typed serialisation library (spec v1.1).

No compiled extension module is needed — the bindings load `libbvnr.so` at import time via the standard `ctypes.CDLL` machinery.

Requirements

Requirement	Notes
Python ≥ 3.10	`dataclasses`, `enum.IntEnum`, union-type annotations (`X | Y`)
`libbvnr.so`	Runtime only; see Library discovery below
`numpy` ≥ 1.24	**Optional** — only for the NumPy bridge (`pip install bovnar[numpy]`)
`pint` ≥ 0.22	**Optional** — only for the pint bridge (`pip install bovnar[pint]`)
pytest ≥ 7	Test suite only (`pip install bovnar[dev]`)

`numpy` and `pint` are imported **lazily, on first use** of their respective bridge functions — importing `bovnar` never requires either to be installed.

Installation

```
# Editable install from source (recommended during development)
pip install -e ".[dev]"
```

Optional extras pull in the dependencies for the bridges:

```
pip install "bovnar[numpy]"   # NumPy bridge
pip install "bovnar[pint]"    # pint bridge
pip install "bovnar[all]"     # both numpy and pint
```

Library discovery

The bindings search for `libbvnr.so` in this order:

1. **`LIBBOVNAR_PATH`** — absolute path to the `.so` file, e.g. `bash export LIBBOVNAR_PATH=/opt/bovnar/lib/libbvnr.so`
2. **`LIBBOVNAR_DIR`** — directory that contains the `.so`, e.g. `bash export LIBBOVNAR_DIR=/opt/bovnar/lib`
3. **`ctypes.util.find_library('bovnar')`** — standard `ldconfig` / `LD_LIBRARY_PATH` search.
4. **In-tree build paths** — `../../build/`, `../../build/release/`, `../../`, `.` (resolved relative to the `_ffi.py` file; useful when building Bovnar alongside the bindings from a mono-repo).

If the library cannot be found a `BovnarLibraryNotFound` exception is raised with the list of searched paths.

Quick-start

High-level API (`loads` / `dumps`)

```python
import bovnar

# Serialise a Python dict to BVNR bytes
data = {
    "sensor_id": 42,
    "temperature": -3.7,
    "label": "outdoor",
    "active": True,
    "payload": None,
}
bvnr_bytes = bovnar.dumps(data)

# Deserialise back to a Python dict
recovered = bovnar.loads(bvnr_bytes)
assert recovered["sensor_id"] == 42
```

`loads` accepts an optional `typed` flag that wraps typed values in `Quantity` objects instead of decoding them to native Python scalars. This preserves the original text representation, exact type width, numeral base, and physical unit for lossless round-trips:

```python
bvnr = b".pressure = <float:32,Pa> 101325.0;"
doc  = bovnar.loads(bvnr, typed=True)
q    = doc["pressure"]     # Quantity('101325.0', FLOAT [Pa])
print(q.raw)               # '101325.0'
print(q.unit_str())        # 'Pa'

# dumps() accepts Quantity values — annotation and raw text are re-emitted as-is
out = bovnar.dumps(doc)
assert bovnar.loads(out, typed=True) == doc
```

`dumps()` starts with a 4 MiB write buffer and doubles it automatically on overflow, up to 256 MiB. The `cap` keyword argument that existed in earlier versions is no longer accepted.

SAX-style streaming reader

The verified callback receives exactly **two** positional arguments: the event code and the data payload. There is no userdata/context argument — capture external state via closure instead.

```python
from bovnar.reader import Reader
from bovnar import Event

def on_event(ev, data):
    if ev == Event.ASSIGNMENT_START:
        print("key:", data.raw_str())
    elif ev == Event.DATA:
        print("value bytes:", data.raw_bytes())
    return True   # returning False stops parsing (raises BovnarParseError)

with Reader() as r:
    r.read_mem(bvnr_bytes, on_verified=on_event)
```

Generator / iterator interface

```python
from bovnar.reader import Reader

with Reader() as r:
    for payload in r.iter_mem(bvnr_bytes):
        print(payload.event, payload.text)
```

DOM (random-access) API

```python
import bovnar

doc = bovnar.dom_parse(bvnr_bytes)

# Top-level key lookup
node = doc["sensor_id"]
print(node.as_i64())        # → 42
print(node.value_type)      # ValueTypeSpec(family=UINT, width=64, base=0)

# Struct traversal
cfg = doc["config"]
host = cfg["host"].as_str()

# Array access
arr = doc["values"]
for i in range(len(arr)):
    print(arr[i].as_float())

# Convert entire document to a plain Python dict (drops type/unit info)
d = doc.to_dict()
```

Low-level writer

```python
from bovnar.writer import Writer
from bovnar.enums  import BaseUnit, SIPrefix

with Writer.to_mem() as w:
    w.write_float("velocity", 9.81, width=64,
                  unit_si_base=BaseUnit.METER,
                  unit_si_prefix=SIPrefix.NONE)
    w.write_uint("count", 1024, width=32)

output: bytes = w.get_output()
```

Streaming / framing (`bovnar.stream`)

Bindings for the streaming layer (see Streaming, Framing & Multiplexing for the full treatment):

```python
from bovnar import stream

# Multi-document record framing
blob = stream.dump_documents([{"id": 1}, {"id": 2}])   # list[dict] -> bytes
docs = stream.load_documents(blob)                      # bytes -> list[dict]
docs = stream.load_documents(blob, continue_past_failed=True)  # bad docs -> None

# Octet multiplexing: (channel, payload) of any size, interleaved & reassembled
multiplexed = stream.mux_dump([(1, b"hello"), (42, b"world")])
messages    = stream.mux_load(multiplexed)              # [(1, b"hello"), (42, b"world")]

# Document-in-document
outer = stream.embed_document(bovnar.dumps({"v": 1}), key="payload")
inner = stream.parse_embedded(bovnar.loads(outer)["payload"])   # {"v": 1}
```

Unit helpers

```python
import bovnar

# Parse a unit string into a ValueUnit struct
vu = bovnar.parse_unit("k~g·m/s²")

# Convert a ValueUnit back to its canonical string
s = bovnar.unit_to_str(vu)     # → "k~g·m/s²"

# Scalar SI/IEC prefix factor for a unit string
f = bovnar.unit_factor("M~Hz") # → 1_000_000.0
```

Extended unit functions

The following functions operate on `ValueUnit` objects and are available both from the top-level `bovnar` namespace and from `bovnar.units`.

```
from bovnar import (
    unit_valid, unit_prefix_factor, unit_prefix_exponent,
    prefix_unit_valid,
    unit_to_si_factor, units_compatible,
    unit_convert_factor, unit_dimension_vector,
    unit_reduce, unit_to_str_ex,
    exponent_to_int, int_to_exponent,
    convert_value,
    UnitFlags,
)

# Validate a ValueUnit struct
ok = unit_valid(vu)                   # True when vu is structurally valid

# Prefix scale factor (SI or IEC) for a ValueUnit
f = unit_prefix_factor(vu)            # e.g. 1000.0 for k~m, 2**30 for Gi~B

# Prefix exponent (base-10 for SI, base-2 for IEC)
e = unit_prefix_exponent(vu)          # e.g. 3 for kilo, -3 for milli, 30 for gibi

# Validate a prefix for a base unit (IEC prefixes are only valid on bit/byte)
from bovnar import ValueUnitPrefix, IECPrefix, BaseUnit
p = ValueUnitPrefix.make_iec(IECPrefix.GIBI)
ok = prefix_unit_valid(p, BaseUnit.BYTE)   # True
ok = prefix_unit_valid(p, BaseUnit.METER)  # False

# Full SI conversion including affine terms (e.g. Celsius → Kelvin)
conv = unit_to_si_factor(vu)
# conv.factor, conv.is_affine, conv.affine_offset

# Check dimensional compatibility
ok = units_compatible(vu_a, vu_b)     # True if same SI dimension vector

# Conversion factor between two compatible units
c = unit_convert_factor(vu_from, vu_to)
# c.factor, c.requires_affine

# 7-element SI dimension exponent vector [m, kg, s, A, K, mol, cd]
dims = unit_dimension_vector(vu)       # e.g. [1, 0, -1, 0, 0, 0, 0] for m/s

# Reduce a compound unit to its canonical named SI unit
r = unit_reduce(vu)                   # r.unit, r.scale

# Convert a scalar value between units (handles affine conversions)
kelvin = convert_value(25.0, vu_celsius, vu_kelvin)

# Serialise with formatting options (see UnitFlags below)
s = unit_to_str_ex(vu, UnitFlags.ASCII_EXP)   # use ^N instead of Unicode superscripts
s = unit_to_str_ex(vu, UnitFlags.REDUCE)      # reduce to canonical named unit first
s = unit_to_str_ex(vu, UnitFlags.REDUCE | UnitFlags.ASCII_EXP)

# Exponent enum ↔ integer conversions
n   = exponent_to_int(Exponent.NEG_SQUARE)  # → -2
exp = int_to_exponent(-2)                    # → Exponent.NEG_SQUARE
```

`SI_DIM_NAMES` is the ordered tuple `('m', 'kg', 's', 'A', 'K', 'mol', 'cd')` — the index positions used by `unit_dimension_vector`.

`UnitFlags`

```python
from bovnar import UnitFlags   # also from bovnar.units

UnitFlags.NONE      # 0 — no special formatting
UnitFlags.REDUCE    # reduce to a canonical named SI unit before serialising
UnitFlags.ASCII_EXP # use ^N exponent notation instead of Unicode superscripts
```

`UnitFlags` is an `IntFlag` and its values may be OR-combined:

```python
s = unit_to_str_ex(vu, UnitFlags.REDUCE | UnitFlags.ASCII_EXP)
```

`ValueUnitPrefix`

`ValueUnitPrefix` is the public mirror of the C `value_unit_prefix_t` struct. It can be constructed with class methods or extracted from a `ValueUnitComponent`:

```python
from bovnar import ValueUnitPrefix, SIPrefix, IECPrefix

p_si  = ValueUnitPrefix.make_si(SIPrefix.KILO)
p_iec = ValueUnitPrefix.make_iec(IECPrefix.GIBI)

vu   = bovnar.parse_unit("Gi~B")
comp = vu.components[0]
p    = comp.prefix   # ValueUnitPrefix extracted from a component
```

Inline unit suffix

In addition to the unit embedded in a type annotation (`<float:64,m/s>`), the Bovnar format supports an **inline unit suffix** placed directly after a scalar value, before the terminating `;`:

```
.speed = 9.81 m/s;           # inline suffix, no annotation
.mass  = <float:64> 70.5 k~g;# annotation without unit, inline adopted
.dist  = <float:64,m> 1.5 m; # annotation and inline both say 'm' — valid
```

From the Python layer, inline and annotation units are **identical**: both reach the application as `data.value_unit` inside the `Event.DATA` payload. No extra code is required to consume inline units.

The validator raises `ErrorCode.UNIT_MISMATCH` (38 / `BovnarParseError`) when an inline suffix is present and a type-annotation unit is also present but the two do not resolve to the same `value_unit_t`. Inline unit suffixes inside array elements always raise `ErrorCode.UNEXPECTED_INPUT_BYTE`.

`Quantity`

`Quantity` is a typed, unit-annotated scalar value that preserves the original text representation, type width, numeral base, and physical unit across a `loads` / `dumps` round-trip.

```
from bovnar import Quantity, ValueTypeSpec, ValueUnit
from bovnar.enums import ValueTypeFamily
```

Construction

`Quantity` is normally produced by `loads(..., typed=True)` rather than constructed by hand, but direct construction is supported:

```
from bovnar.structs import make_type_spec
from bovnar.enums   import ValueTypeFamily

vt = make_type_spec(ValueTypeFamily.FLOAT, 32)
q  = Quantity('101325.0', vt)          # dimensionless
q2 = Quantity('9.81',     vt, vu)      # vu is a ValueUnit, e.g. from parse_unit
```

For exact numeric input there is also the `Quantity.from_number` constructor, which stores the value as an exact decimal literal (so writing it with `dumps` is lossless to the format's precision):

```
from decimal import Decimal
from fractions import Fraction

Quantity.from_number(Decimal('19.99'))                          # float_dec:64
Quantity.from_number(Decimal('1.1'), family=ValueTypeFamily.FLOAT, width=128)
Quantity.from_number(Fraction(837, 256),
                     family=ValueTypeFamily.FLOAT_FIX, width=32, frac=8)
```

It accepts a `Decimal`, `Fraction` (must be a terminating decimal), `int`, `str` (a verbatim literal), or `float` (only as precise as the double), and validates the width for the chosen family.

Properties and methods

Name	Type	Description
`q.raw`	`str | None`	Original text token as it appeared in the BVNR stream
`q.vtype`	`ValueTypeSpec`	Type family, bit width, and numeral base
`q.unit`	`ValueUnit`	Physical unit (`BVN_UNIT_NONE` when dimensionless)
`q.value`	property	Decode `raw` to the closest native Python scalar (`int` , `float` , `str` , `bool`) — **lossy** for `float_dec` / `float_fix` / `float:128` + (goes through a C `double`)

Name	Type	Description
`q.unit_str()`	`str`	Canonical unit string (e.g. `'m/s²'`), or `''` when dimensionless
`q.decimal()`	`Decimal`	**Exact** value as `decimal.Decimal` from the verbatim literal — lossless at any width; raises for non-numeric families
`q.fraction()`	`Fraction`	Exact value as `fractions.Fraction` (for `float_fix`, the exact `mantissa / 2**frac`)
`q.fixed_point()`	`(int, int)`	`(mantissa, frac_bits)` of a `float_fix` value; the value is `mantissa / 2**frac_bits`
`q.stored_value()`	`Decimal`	The value materialised into the declared IEEE/fixed format (round-to-nearest-even) — differs from `decimal()` only when the literal carries more precision than the format holds
`q.ieee_bits()`	`bytes`	IEEE-754 interchange bytes (binary16...256 for `float`, decimal16...256 for `float_dec`), little-endian word order
`q.epoch_name`	`str | None`	For a `datetime`, the epoch name (`"unix"`, `"tai"`, ...); `None` otherwise
`q.epoch_mjd`	`int | None`	For a `datetime`, the epoch's Modified Julian Day; `None` otherwise

Lossless numeric access (`float_dec`, `float_fix`, `float:128`/`256`)

`q.value` decodes through a C `double`, which loses precision for the decimal-float, fixed-point, and wide binary-float families. The accessors above instead use the verbatim literal text (and bovnar's arbitrary-precision `bvn_float`), so the full precision the format carries is reachable from Python:

```
q = bovnar.loads(b'.p=<float_dec:64> 3.14159265358979323846264338327950 3;',
                 typed=True)['p']
q.value          # 3.141592653589793   (lossy C double)
q.decimal()      # Decimal('3.14159265358979323846264338327950 3')  (exact literal)
q.stored_value() # Decimal('3.141592653589793')  (the decimal64-rounded value)

f = bovnar.loads(b'.x=<float_fix:32,q8> 3.27;', typed=True)['x']
f.fraction()     # Fraction(837, 256)
f.fixed_point()  # (837, 8)
```

These materialise over the **full** representable range, including exponents the C parser cannot otherwise reach (beyond ~1e9865). `decimal()` / `fraction()` work at every binary width bovnar allows (16, or a multiple of 32 up to 32768); the bit-exact `stored_value()` / `ieee_bits()` apply to the IEEE encodings (`float:16/32/64/128/256`), which are also exposed directly as `bovnar.BvnFloat`.

`dumps()` integration

`_emit_value` dispatches on `Quantity` before the plain `int` / `float` path, so any dict that came from `loads(..., typed=True)` can be passed directly to `dumps()` and will produce identical output:

```
bvnr  = b".speed = <float:32,m/s> 9.81;"
doc   = bovnar.loads(bvnr, typed=True)
out   = bovnar.dumps(doc)
assert out.strip() == bvnr.strip()
```

The annotation is suppressed when `_needs_annotation` returns `False` — i.e. when the type is `FLOAT:64` with no unit and no non-decimal base (matching the C library's own default-annotation omission rules).

`dumps()` also accepts bare `decimal.Decimal` (written as an exact `float_dec:64` literal) and terminating `fractions.Fraction` values directly — a non-terminating fraction (e.g. `Fraction(1, 3)`) raises `BovnarArgumentError`.

`write_array` helper

`write_array` is a high-level helper exported from the top-level `bovnar` namespace. It handles both flat and multi-dimensional arrays.

```
from bovnar import write_array, Writer
from bovnar.structs import make_type_spec, make_unit_si
from bovnar.enums import ValueTypeFamily, BaseUnit

with Writer.to_mem() as w:
    # Single-row array
    write_array(w, "primes", [2, 3, 5, 7])

    # Multi-row array (rows separated by /)
    write_array(w, "matrix", [[1, 2, 3], [4, 5, 6]])

    # Typed array: whole-array type annotation
    vt = make_type_spec(ValueTypeFamily.UINT, 16)
    write_array(w, "ports", [80, 443, 8080], vt=vt)
```

Element types accepted per element: `int`, `float`, `str`, `bool`, `None`, `dict` (written as a struct), or nested `list` / `tuple` (written as a nested array). When rows is a flat sequence it is treated as a single-row array; when all top-level elements are themselves `list` or `tuple` it is treated as a multi-row array.

---

pint bridge

bovnar units interoperate with pint through a hand-verified translation table (`bovnar._pint_units`). pint is an **optional** dependency, imported lazily on first use; importing `bovnar` never requires it. Install with `pip install "bovnar[pint]"` (or `bovnar[all]`).

The four bridge functions are exported from the top-level `bovnar` namespace (and from `bovnar._pint_bridge`):

Function	Direction	Description
`to_pint(value, vu, *, ureg=None)`	bovnar → pint	Wrap a scalar/ndarray + `ValueUnit` in a pint `Quantity`
`to_pint_unit(vu, *, ureg=None)`	bovnar → pint	`ValueUnit` → pint `Unit` (dimensionless when no real unit)
`from_pint(qty, *, ureg=None, validate=True)`	pint → bovnar	`Quantity` → `(magnitude, ValueUnit)`
`from_pint_unit(unit, *, ureg=None, validate=True)`	pint → bovnar	`Unit` / `Quantity` / `str` → `ValueUnit`

```python
import bovnar

vu  = bovnar.parse_unit("k~m")        # kilometre
qty = bovnar.to_pint(5.0, vu)         # <Quantity(5.0, 'kilometer')>

mag, vu2 = bovnar.from_pint(qty)      # (5.0, ValueUnit for km)
vu3      = bovnar.from_pint_unit("newton")     # str/Unit/Quantity all accepted
```

Prefixes ride in the unit, never the magnitude

bovnar's `k~m` maps to pint `kilometer` — the prefix is kept inside the unit name, never folded into the magnitude. A wrapped value is therefore returned unscaled, so a wrapped numpy array is never silently rescaled.

Affine temperature units

Offset/affine scales (`°C`, `°F`, Réaumur, Delisle, Newton, Rømer) cannot carry a prefix or exponent — pint forbids it and bovnar never emits it. A prefixed or exponentiated affine unit raises `BovnarArgumentError` rather than a cryptic pint error.

Validation

`from_pint` / `from_pint_unit` validate the resulting `ValueUnit` by default (`validate=True`); a pint unit that maps to a structurally invalid bovnar unit (e.g. a prefix not permitted on that base) raises `BovnarArgumentError`. pint units with more than 8 components, non-integer exponents, or exponents outside `[-9, 9]` also raise.

Registry control: `build_registry`

A module-level default `pint.UnitRegistry` is built on first use. To share a registry across calls — or to register bovnar's custom units onto your own — pass `ureg=`:

```
from bovnar._pint_units import build_registry, is_currency_unit

ureg = build_registry()                        # fresh registry with bovnar units
ureg = build_registry(my_existing_registry)    # extend an existing one
ureg = build_registry(with_currencies=False)   # skip the currency dimensions

qty  = bovnar.to_pint(5.0, vu, ureg=ureg)
```

`build_registry` registers bovnar's custom physical-unit definitions — units where pint's own definition differs or is missing (e.g. `bvnr_gauss`, `bvnr_var`, the historical German units) — plus, by default, the ISO 4217 + crypto currencies as custom dimensions. `is_currency_unit(unit)` reports whether a pint unit involves a currency dimension (holds for products such as `USD/year`).

Semantic caveats

A few bovnar units map to a pint native unit whose semantics differ slightly; these are recorded in `bovnar._pint_units.SEMANTIC_CAVEATS` (e.g. `BYTE`, `DECIBEL`, `NEPER`). Consult that dict when exact round-trip semantics matter.

The mapping is **not 1:1** by construction (bovnar `b`=bit vs pint barn, `R`=roentgen vs pint's gas constant). The translation table is locked against silent drift by `TestUnitTableIntegrity`, which re-derives every physical unit's dimension and magnitude from bovnar and asserts the pint bridge reproduces both.

NumPy bridge

The NumPy bridge converts between bovnar arrays and `numpy.ndarray`. numpy is an **optional** dependency, imported lazily on first use. Install with `pip install "bovnar[numpy]"` (or `bovnar[all]`).

All five functions are exported from the top-level `bovnar` namespace (and from `bovnar._numpy`):

Function	Direction	Description
`to_numpy(src, *, dtype=None, return_unit=False)`	bovnar → numpy	Array → `ndarray` (optionally `(ndarray, unit_str)`)
`to_pint_array(src, *, dtype=None, ureg=None)`	bovnar → pint	Array → pint `Quantity` (ndarray data + unit)

Function	Direction	Description
`from_numpy(writer, key, arr, *, unit=None, float_format=None)`	numpy → bovnar	Write an `ndarray` into a `Writer`
`from_pint_array(writer, key, qty)`	pint → bovnar	Write a pint `Quantity` (magnitude + unit) into a `Writer`
`array_to_bvnr(key, arr, *, unit=None, pretty=True, float_format=None)`	numpy → bovnar	`ndarray` → bovnar bytes (convenience)

Reading: `to_numpy`

src is either a `DomNode` for an ARRAY (random-access, from `dom_parse`) or the nested list/tuple that `loads(..., typed=True)` produces. Both `/`-separated rows and bracket nesting collapse to the same `ndarray` shape.

```
import bovnar

# from a typed loads() result
doc = bovnar.loads(b'.a=<uint:8>[1,2,3];', typed=True)
arr = bovnar.to_numpy(doc['a'])               # dtype uint8

# with the whole-array unit
arr, unit = bovnar.to_numpy(
    bovnar.loads(b'.a=<float:32,m/s>[1,2,3]/[4,5,6];', typed=True)['a'],
    return_unit=True)                         # unit == 'm/s', arr.shape == (2, 3)

# from a DOM node
arr = bovnar.to_numpy(bovnar.dom_parse(b'.a=<sint:16>[10,-20,30];')['a'])
```

The bovnar `(family, width)` maps directly to the numpy dtype (`uint:8` → `uint8`, `float:32` → `float32`, ...). Pass `dtype=` to coerce.

The families with no exact native numpy dtype — `float_dec`, `float_fix`, and the wide binary floats `float:128`/`256` (and any width above 64) — decode to an **object array of exact `decimal.Decimal`** on the typed (`loads(..., typed=True)`) path, which is lossless; pass `dtype='float64'` for the lossy native conversion. The DOM (`dom_parse`) path only has a C `double` for these, so it raises and points you at the typed path or `dtype='float64'`. Integers wider than 64 bits likewise come back as an object array of Python ints (`dtype=object`), which is exact.

A **`datetime`** array on the **unix** epoch (spec 1.1) maps to/from `datetime64[s]` — the integer epoch-seconds carrier is unix-relative, so it round-trips losslessly. A non-unix epoch (`tai`, `gps`, ...) is not unix-relative, so `to_numpy` refuses it (mapping it to `datetime64` would mis-date every value) and points you at `dtype='int64'` for the raw seconds. On the write side a `datetime64[*]` array is coarsened to whole seconds and written as `<datetime:64,unix>`; `NaT` has no bovnar representation and is rejected, and `array_to_bvnr` prepends the `#! bovnar 1.1` directive so its output re-parses.

Strict by default:

- an array that mixes numeric encodings (e.g. `uint:8` and `float:64`) raises unless you pass `dtype=` to coerce to one;
- bovnar 1.0 arrays are rectangular and homogeneous, so the result is always a regular ndarray — there is no ragged case to handle;
- a `null` element cannot fill a dtype with no null slot — integer, `bool`, or string (which would otherwise silently become `0` / `False` / `'None'`); pass `dtype=float` (→ `NaN`) or `dtype=object` (→ `None`);
- the unit is a whole-array property (numpy has one dtype per array) — mixed units raise, as does mixing a dimensioned element with a dimensionless one; the unit is returned alongside the data, never baked into elements.

Writing: `from_numpy` / `array_to_bvnr`

```
import numpy as np
import bovnar
from bovnar import Writer

with Writer.to_mem() as w:
    bovnar.from_numpy(w, "velocity",
                      np.array([1.5, 2.5], dtype=np.float32), unit="m/s")
out = w.get_output()

# one-shot convenience
raw = bovnar.array_to_bvnr("matrix", np.arange(8).reshape(2, 2, 2))

# exact decimal/fixed/wide-binary float arrays via float_format
from decimal import Decimal
prices = np.array([Decimal("19.99"), Decimal("0.01")], dtype=object)
bovnar.array_to_bvnr("p", prices, unit="$USD")            # -> <float_dec:64,$USD>
bovnar.array_to_bvnr("a", prices, float_format=("float", 128))   # binary128
```

- `from_numpy` requires a 1-D-or-higher array; write a 0-D scalar with the scalar `Writer` API instead.
- unit may be a bovnar unit string, a `ValueUnit`, or a pint `Unit` / `Quantity` (anything else raises `BovnarArgumentError`).
- Units apply to **numeric** arrays only — a unit on a `bool` or string array raises `BovnarArgumentError`.
- An **object array of `Decimal` / `Fraction`** is written as an exact decimal/fixed/wide-binary float. `float_format=(family, width[, frac])` — family `'float'`, `'float_dec'` (the default for a Decimal object array), or `'float_fix'` — selects the target; it is required to disambiguate an object array that is not purely `Decimal` / `Fraction`.
- A **masked array** with any masked entry is rejected (it would otherwise serialise the underlying value of a masked cell); fill or unmask it first.
- `array_to_bvnr` grows its write buffer like `dumps()` (4 MiB, doubling up to 256 MiB).

pint arrays

`to_pint_array` and `from_pint_array` bridge straight through to pint Quantities backed by ndarrays, reusing the unit translation above:

```
q = bovnar.to_pint_array(
    bovnar.loads(b'.a=<float:64,k~m>[1,2,3];', typed=True)['a'])
# q is a pint Quantity: magnitude ndarray [1, 2, 3], unit 'kilometer'

with Writer.to_mem() as w:
    bovnar.from_pint_array(w, "dist", q)
```

A `datetime` array has no pint unit, so `to_pint_array` rejects it rather than wrap it as a meaningless dimensionless quantity — use `to_numpy` for the `datetime64` array (or `dtype='int64'` for the raw epoch seconds).

Currency helpers

The `bovnar.currency` submodule (exposed as `bovnar.currency`) provides metadata for the ISO 4217 fiat and cryptocurrency `BaseUnit` members. It is pure Python — no library or optional dependency required.

```
from bovnar import currency, BaseUnit

currency.is_currency(BaseUnit.USD)   # True
currency.is_fiat(BaseUnit.USD)       # True
currency.is_crypto(BaseUnit.BTC)     # True

info = currency.currency_info(BaseUnit.USD)
info.code           # 'USD'
info.numeric_code   # 840 (ISO 4217 numeric; 0 for crypto)
info.minor_unit     # 2   (decimal places: 1 major = 10^minor minor units)
info.name           # 'US Dollar'

currency.minor_unit(BaseUnit.JPY)    # 0
currency.currency_code(BaseUnit.EUR) # 'EUR'
currency.from_code('GBP')            # BaseUnit.GBP

for ci in currency.all_fiat():       # all_crypto() / all_currencies() also exist
    ...
```

`CurrencyInfo` is a frozen dataclass; `currency_info`, `minor_unit`, `currency_name`, `currency_code`, and `from_code` raise for non-currency bases or unknown codes.

`Reader` reference

Construction

```
with Reader() as r:
    ...
```

`Reader.__init__` calls `bvnr_reader_create` immediately. Use as a context manager or call `r.close()` explicitly to release the C object.

`read_mem(data, *, on_verified, on_unverified, max_file_size, continue_on_error, strict_version)`

Parse BVNR from a `bytes`, `bytearray`, or `memoryview` object.

Parameter	Type	Default	Description
`data`	`bytes | bytearray | memoryview`	—	Input buffer
`on_verified`	`Callable[[Event, BvnrData | None], bool] | None`	`None`	Callback for validated events
`on_unverified`	`Callable[[Event, BvnrData | None], bool] | None`	`None`	Callback for raw pre-validation events
`max_file_size`	`int`	`0` (unlimited)	Hard limit on bytes consumed
`continue_on_error`	`bool`	`False`	Enable resync mode
`strict_version`	`bool`	`False`	Reject a declared spec version newer than this build (`error_unsupported_spec_version`)

`read_fd(fd, *, on_verified, on_unverified, max_file_size, continue_on_error, strict_version)`

Parse BVNR from an open POSIX file descriptor. Parameters identical to `read_mem` except the first argument is a non-negative `int` fd.

`read_file(path, *, on_verified, on_unverified, max_file_size, continue_on_error, strict_version)`

Convenience wrapper: opens `path` with `os.O_RDONLY`, calls `read_fd`, closes the fd in a `finally` block. The `max_file_size` default is `MAX_FILESIZE_BYTES` (16 MiB) rather than unlimited.

`iter_mem(data, *, verified_only, max_file_size)`

Generator that collects all events from `read_mem` and yields `EventPayload(event, raw, value_type, value_unit)` objects.

Parameter	Default	Description
`verified_only`	`True`	When `False`, both callbacks fire and events may appear twice
`max_file_size`	`0`	Forwarded to `read_mem`

`EventPayload` fields:

Field	Type	Description
`event`	`Event`	The event code
`raw`	`bytes`	Raw token bytes
`value_type`	`ValueTypeSpec | None`	Type annotation if present
`value_unit`	`ValueUnit | None`	Unit if present
`text`	`str` (property)	`raw` decoded as UTF-8 (with `errors='replace'`)

Error-state properties

These properties query the C reader object after a failed parse.

Property	Type	Description
`error_code`	`ErrorCode`	Most recent error code
`error_line`	`int`	1-based line number of the error
`error_column`	`int`	1-based column of the error
`error_offset`	`int`	Byte offset of the error
`recovery_count`	`int`	Number of times resync was entered (incremented at error entry, not at resync completion)

`MAX_FILESIZE_BYTES`

```
from bovnar import MAX_FILESIZE_BYTES   # 16 * 1024 * 1024  (16 MiB)
```

Default `max_file_size` cap used by `Reader.read_file`.

`Writer` reference

Construction class methods

Method	Description
`Writer.to_mem(buf=None, cap=4194304, *, pretty=True)`	Write to an in-process buffer. `buf` may be a pre-allocated `bytearray`; when `None` an internal buffer of size `cap` is allocated.
`Writer.to_fd(fd, *, pretty=True)`	Write to an open POSIX file descriptor.
`Writer.to_file(path, *, pretty=True)`	Open `path` for writing (`O_WRONLY|O_CREAT|O_TRUNC`, mode `0o644`) and write to it; the fd is closed when the writer is finished or destroyed.

All three are used as context managers. On clean exit (`exc_type is None`) the context manager calls `finish()` automatically.

Output retrieval

Method / property	Description
`w.get_output() -> bytes`	Return the bytes written so far (mem writers only).
`w.bytes_written`	Number of bytes written (all writer modes).
`w.finish()`	Flush and seal the output. Raises `BovnarWriteError` if any struct is still open.
`w.destroy()`	Release the underlying C writer object immediately.

Scalar write methods

All scalar writers accept keyword-only unit parameters. Unit resolution priority: `unit_str` (parsed via `bvn_parse_unit_n`) → `unit_iec_base` → `unit_si_base` → `BVN_UNIT_NONE` (no annotation emitted).

`write_uint(key, value, *, width=64, base=10, unit_str=None, unit_si_base=None, unit_si_prefix=SIPrefix.NONE, unit_si_exp=Exponent.LINEAR, unit_iec_base=None, unit_iec_prefix=IECPrefix.NONE)`

Write an unsigned integer. `base` selects the numeral system (10 or 16 for inline values; any Bovnar-supported base for `write_bvni`).

`write_sint(key, value, *, width=64, base=10, unit_str=None, unit_si_base=None, unit_si_prefix=SIPrefix.NONE, unit_si_exp=Exponent.LINEAR)`

Write a signed integer.

`write_float(key, value, *, width=64, unit_str=None, unit_si_base=None, unit_si_prefix=SIPrefix.NONE, unit_si_exp=Exponent.LINEAR)`

Write an IEEE 754 binary float.

`write_float_fix(key, value, *, width=64, q=0, unit_str=None, unit_si_base=None, unit_si_prefix=SIPrefix.NONE, unit_si_exp=Exponent.LINEAR)`

Write a Q-format fixed-point value. `q` is the number of fractional bits (`0 ≤ q < width`).

`write_float_dec(key, value, *, width=64, unit_str=None, unit_si_base=None, unit_si_prefix=SIPrefix.NONE, unit_si_exp=Exponent.LINEAR)`

Write an IEEE 754-2008 decimal float.

`write_string(key, value)`

Write a bare quoted UTF-8 string with no type annotation:

```
.host = "localhost";
```

To emit an explicit `<utf8>` annotation, use the low-level `emit` API with `Event.TYPE_ANNOTATION_START` / `TYPE_ANNOTATION_END` before `Event.DATA`.

`write_bool(key, value)`

Write a `bool` value (`token_is_bool`) — serialized as the bare keyword `true` or `false`. On read-back it decodes to a Python `bool`.

`write_null(key)`

Write a null value (empty slot).

Extended integer writers

`write_bvni(key, value, *, width=64, base=10, signed=None, unit_str=None, unit_si_base=None, unit_si_prefix=SIPrefix.NONE, unit_si_exp=Exponent.LINEAR)`

Arbitrary-width integer writer that supports all Bovnar numeral bases (**2–62, 64, and 85**). Non-decimal values are formatted using Python's own big-integer arithmetic and emitted as quoted strings. `signed` defaults to `True` when `value < 0`.

`write_bvnf_base(key, value_str, *, width=0, base=10, unit_str=None, unit_si_base=None, unit_si_prefix=SIPrefix.NONE, unit_si_exp=Exponent.LINEAR)`

Write a float from a pre-formatted string in base 10 or 16. `base` must be 10 or 16; any other value raises `BovnarArgumentError`.

Struct helpers

```
w.begin_struct(key)   # emit ASSIGNMENT_START + STRUCT_START, increment depth
w.end_struct()        # emit STRUCT_END, decrement depth
```

`finish()` verifies that the struct depth is zero; an unclosed struct raises `BovnarWriteError(GOT_INCOMPLETE_BVNR_STREAM)`.

Array helpers

```
w.begin_array_row()   # emit ARRAY_ROW_START
w.end_array_row()     # emit ARRAY_ROW_END
w.new_array_dim()     # emit ARRAY_DIM_START (the / separator between rows)
```

Low-level `emit`

```
w.emit(event, *, key=None, value=None, vt=None, vu=None)
```

Send an arbitrary event to the C writer. `key` and `value` are encoded as UTF-8. When both `vt` and `value` are supplied the token type is inferred: `_TOKEN_IS_STRING` for `ValueTypeFamily.UTF8`, `_TOKEN_IS_NUMBER` otherwise.

DOM API

The DOM API parses a complete BVNR document into an in-memory tree for random-access queries without writing a SAX callback.

`dom_parse(data) -> DomDoc`

Top-level convenience function (mirrors `DomDoc.parse`).

```
import bovnar
doc = bovnar.dom_parse(bvnr_bytes)
```

`DomDoc`

Owning wrapper around `bvn_dom_doc_t`. Destroying the object frees the entire tree; any `DomNode` derived from it becomes invalid after that point.

Method / property	Description
`DomDoc.parse(data)`	Class method. Parse `bytes | bytearray | memoryview`.
`DomDoc.parse_fd(fd)`	Class method. Parse from an open file descriptor.

> **File-size limits:** `DomDoc.parse_fd` and `DomDoc.parse_file` apply an internal ceiling of `BVN_DOM_FD_MAX_BYTES` (256 MiB). This is 16× larger than `Reader.read_file`'s `MAX_FILESIZE_BYTES` (16 MiB). Use `bvn_dom_parse_fd_ex` directly if you need a different limit. | `DomDoc.parse_file(path)` | Class method. Open path and parse (fd closed in `finally`). Applies an internal hard cap of `BVN_DOM_FD_MAX_BYTES` (256 MiB) via the C function `bvn_dom_parse_fd`. This ceiling is distinct from `Reader.read_file`'s `MAX_FILESIZE_BYTES` (16 MiB). | | `doc.parse_error` | `ErrorCode` — `NONE` on success. | | `doc[key]` | Return top-level `DomNode` by key; raises `KeyError` when absent. | | `key in doc` | `True` when the top-level key exists. | | `len(doc)` | Number of top-level entries. | | `iter(doc)` | Iterate over `(key, DomNode)` pairs. | | `doc.entries()` | Return all top-level `(key, DomNode)` pairs as a list. | | `doc.lookup(path)` | Dot-separated path lookup, e.g. `'server.tls.cert'`. Returns `None` when absent. | | `doc.to_dict()` | Convert entire document to a plain Python dict, dropping type and unit info. |

`DomNode`

Non-owning view into a `bvn_dom_node_t`. The parent `DomDoc` must remain alive for as long as any derived `DomNode` is in use.

Property / method	Description
`node.dom_type`	`DomType` enum value
`node.value_type`	`ValueTypeSpec`
`node.unit`	`ValueUnit`
`node.unit_str`	Unit as a string via `bvn_dom_get_unit_string`, or `''`
`node.is_null()`	`True` for null values
`node.value_in_base_units()`	Numeric value scaled to SI base units (`float`)
`node.as_i64()` / `as_u64()`	Signed / unsigned 64-bit integer
`node.as_i32()` / `as_u32()`	Signed / unsigned 32-bit integer
`node.as_i16()` / `as_u16()`	Signed / unsigned 16-bit integer
`node.as_i8()` / `as_u8()`	Signed / unsigned 8-bit integer
`node.as_float()`	`float` (64-bit)
`node.as_str()`	Python `str` for `STRING`, `SYMBOL`, or `REFERENCE` nodes
`node.as_bytes()`	`bytes` for `OCTET_STREAM` nodes
`node.as_int_str(base=10)`	Integer value as a string in the given base; result C string is freed before return
`node.datetime_fraction`	`str` of the verbatim ISO sub-second digits for a `datetime` written as a literal with a fraction (spec 1.1), else `None`; the carrier value is unchanged and still read via `to_python()`
`node[key]`	Child `DomNode` by string key (STRUCT) or integer index (ARRAY)

Property / method	Description
`key in node`	Membership test for STRUCT nodes
`len(node)`	Element count for STRUCT or ARRAY nodes
`iter(node)`	For STRUCT: iterate `(key, DomNode)` pairs; for ARRAY: iterate elements
`node.entries()`	List of `(key, DomNode)` pairs (STRUCT nodes only)
`node.array_dims()`	Number of `/`-separated dimensions (ARRAY nodes only)
`node.to_python()`	Recursively convert to a native Python value (drops type/unit info)

`DomType`

```
NULL=0  INT=1  FLOAT=2  STRING=3  SYMBOL=4
REFERENCE=5  STRUCT=6  ARRAY=7  OCTET_STREAM=8  BOOL=9
```

Running the test suite

```
# Run everything (library-dependent tests are skipped when libbvnr.so is absent)
pytest

# Run only the pure-Python tests (no library required)
pytest -m "not needs_lib"

# Run only integration tests (requires libbvnr.so)
pytest -m needs_lib

# Verbose with short tracebacks (already the default via pyproject.toml)
pytest -v --tb=short
```

The NumPy- and pint-bridge tests `pytest.importorskip` their dependency, so they are skipped automatically when `numpy` / `pint` is not installed. Install `bovnar[all]` to run the full suite.

Package layout

```
bovnar/
├── __init__.py      # loads() / dumps() / dom_parse() / unit helpers — public API
├── _ffi.py          # ctypes FFI: library discovery + argtypes/restype
├── _bvnfloat.py     # BvnFloat: arbitrary-precision float + IEEE binary/decimal/fixed
encoders
├── _numpy.py        # NumPy bridge: to_numpy/from_numpy/to_pint_array/array_to_bvnr
(numpy optional)
├── _pint_bridge.py  # pint bridge: to_pint/from_pint/to_pint_unit/from_pint_unit (pint
optional)
├── _pint_units.py   # verified bovnar↔pint unit table + build_registry()
├── currency.py      # ISO 4217 / crypto currency metadata (pure Python)
├── dom.py           # DomDoc, DomNode, DomType — random-access DOM
├── enums.py         # Python IntEnum mirrors of C enums
├── exceptions.py    # BovnarError hierarchy
├── quantity.py      # Quantity — typed, unit-annotated scalar for lossless round-trips
├── reader.py        # Reader class + EventPayload dataclass
├── structs.py       # ctypes Structure/Union definitions + ValueUnitPrefix + make_*
helpers
├── units.py         # unit_to_si_factor, unit_convert_factor, etc.
└── writer.py        # Writer class

tests/
├── conftest.py                   # shared fixtures, needs_lib marker
├── test_analytics.py             # analytic / benchmarking tests (needs_lib)
├── test_array_parser.py          # array parsing integration tests (needs_lib)
├── test_currency_units.py        # currency unit round-trip tests (needs_lib)
├── test_dom.py                   # DOM API integration tests (needs_lib)
├── test_enums.py                 # pure-Python enum tests
├── test_example_numpy_roundtrip.py # NumPy bridge end-to-end example (needs_lib + numpy)
├── test_lossless_floats.py       # exact decimal/fixed/float128-256 round-trips
(needs_lib)
├── test_numpy_bridge.py          # NumPy bridge tests (needs_lib + numpy)
├── test_pint_bridge.py           # pint bridge + unit-table integrity (needs_lib + pint)
├── test_reader.py                # integration: Reader (needs_lib)
├── test_structs.py               # pure-Python struct / helper tests
├── test_unit_physics.py          # unit physics / conversion tests (needs_lib)
├── test_units.py                 # mixed: unit parsing / serialisation (needs_lib for
FFI)
├── test_write_array.py           # write_array integration tests (needs_lib)
└── test_writer.py                # integration: Writer (needs_lib)
```

Error handling

All errors surface as subclasses of `BovnarError`:

Exception	When raised
`BovnarLibraryNotFound`	`libbvnr.so` not found at import
`BovnarParseError`	Parse error in `Reader` (carries `code`, `line`, `column`, `offset`, `byte`)
`BovnarWriteError`	Write error in `Writer` (carries `code`, `offset`)
`BovnarArgumentError`	Invalid argument passed to a helper (e.g. bad unit string, closed reader/writer)

`unit_convert_factor` **error semantics:** The C function uses the `(ok, requires_affine)` pair to signal three distinct outcomes:

ok	requires_affine	Meaning
`True`	`False`	Multiplicative conversion; `factor` is ready to use
`False`	`True`	Affine conversion required (e.g. °C ↔ K); `factor` is still valid but a plain multiply is insufficient
`False`	`False`	Units are dimensionally incompatible → `BovnarArgumentError`

`BovnarArgumentError` is raised **only** when both `ok=False` and `requires_affine=False`. When `requires_affine=True`, call `convert_value` which handles the two-step affine path automatically, or call `unit_to_si_factor` on both units and perform the conversion manually.

Callbacks returning `False`: When an `on_verified` or `on_unverified` callback returns `False`, the C parser stops and `bvnr_read` returns failure. The Python layer then calls `_raise_error()` and raises `BovnarParseError` with whatever error code the reader recorded. `BovnarCallbackAbort` is defined in `exceptions.py` but is not raised by the current implementation.

Callbacks raising exceptions: If the callback itself raises a Python exception, that exception is captured, the C callback returns `False` to stop the parse, and the original exception is re-raised from `read_mem` / `read_fd` after the C call returns.

FFI details

`ON_ERROR_FUNC` signature

The error callback type matches the C `bvnr_on_error_fn` signature exactly:

```
void (*bvnr_on_error_fn)(
    void*        userdata,
    error_code_t err,
    uint64_t     line,
    uint64_t     column,
    uint32_t     byte,
    uint64_t     offset);
```

In Python this is declared as:

```
ON_ERROR_FUNC = ctypes.CFUNCTYPE(
    None,
    ctypes.c_void_p,   # userdata
    ctypes.c_int,      # error_code_t err
    ctypes.c_uint64,   # line
    ctypes.c_uint64,   # column
    ctypes.c_uint32,   # byte
    ctypes.c_uint64,   # offset
)
```

`BvnrWriteFlags` layout

`BvnrWriteFlags` mirrors the C `bvnr_write_flags_s` struct in full, including the trailing `unit_flags` field (`bvn_unit_flags_t`, a `uint32_t`):

Flag constant (C)	Value	Effect
`BVN_UNIT_FLAGS_NONE`	`0`	Default: full Unicode exponent characters
`BVN_UNIT_REDUCE`	`1 << 0`	Reduce compound units before serialising
`BVN_UNIT_ASCII_EXP`	`1 << 1`	Use `^N` ASCII exponent notation instead of Unicode superscripts

The writer respects the `unit_flags` stored inside the C writer object when serialising unit annotations. `_emit_annotation` retrieves the live flags via `bvnr_writer_unit_flags(w)` and passes them to `bvn_unit_to_string_ex`.

`write_string` behaviour

`Writer.write_string` emits a bare quoted string with no type annotation, matching `bvnr_write_string` in the C library:

```
.host = "localhost";
```

To write a string with an explicit `<utf8>` annotation, use the low-level `emit` API with `Event.TYPE_ANNOTATION_START` / `TYPE_ANNOTATION_END` events before the `Event.DATA` event.

`_resolve_unit` default

When no unit arguments are supplied to `write_uint`, `write_sint`, `write_float`, etc., the unit resolves to `BVN_UNIT_NONE` (zero components), matching the C convenience helpers. No unit annotation is emitted in this case. Passing `unit_si_base` or `unit_iec_base` produces a unit with one component; the `_SENTINEL`-only `no_unit` keyword is only produced when the caller explicitly constructs a dimensionless `ValueUnit` with `BaseUnit.NONE`.

`BaseUnit` enum

The `BaseUnit` enum mirrors the full C `value_base_unit_e`:

Range	Members
0	`NONE`
1–2	`BIT`, `BYTE`
3–9	SI base units (`SECOND` … `CANDELA`)
10–28	Named SI derived units (`HERTZ` … `KATAL`)
29–44	Non-SI accepted units (`LITER` … `YEAR`)
45–54	Imperial/US length (`INCH` … `FATHOM`)
55–62	Imperial/US mass (`POUND` … `CARAT`)
63	`FAHRENHEIT`
64–67	Pressure (`ATMOSPHERE` … `PSI`)
68–71	Energy (`CALORIE` … `THERM`)
72	`HORSEPOWER`
73–75	Force (`POUND_FORCE`, `DYNE`, `KIP`)
76	`KNOT`
77–85	US volume (`GALLON` … `BARREL`)
86–87	Area (`ACRE`, `BARN`)
88–90	Angle (`ARCMINUTE`, `ARCSECOND`, `GRAD`)
91–98	CGS units (`POISE` … `GALILEO`)
99–101	Radiation (`CURIE`, `ROENTGEN`, `REM`)
102–103	Logarithmic (`NEPER`, `DECIBEL`)
104	`RANKINE`
105	`SLUG`
106	`THOU`
107–109	UK imperial volume (`PINT_UK`, `FLUID_OUNCE_UK`, `QUART_UK`)
110–111	Electrical power (`VAR`, `VOLT_AMPERE`)
112	`KILOGRAM_FORCE`
113	`INCH_HG`
114	`RPM`
115	`FOOT_POUND`
116–117	Mass additional (`DRAM`, `PENNYWEIGHT`)

Range	Members
118–119	Length additional (`CHAIN` , `ROD`)
120–121	Volume additional (`GILL` , `GILL_UK`)
122	`STANDARD_GRAVITY`
123	`METRIC_HORSEPOWER`
124	`REVOLUTION`
125–126	Time additional (`MONTH` , `FORTNIGHT`)
127	`ATMOSPHERE_TECHNICAL`
128–129	Textile linear density (`TEX` , `DENIER`)
130–133	Apothecary/dry volume (`FLUID_DRAM` , `MINIM` , `PECK` , `BUSHEL`)
134–297	ISO 4217 fiat currencies (`AED` … `ZWL`) — see `CURRENCY_FIAT_FIRST` / `CURRENCY_FIAT_LAST`
298–347	Cryptocurrencies (`BTC` … `RUNE`) — see `CURRENCY_CRYPTO_FIRST` / `CURRENCY_CRYPTO_LAST`
348–360	Historical German units (`PFUND` , `ZENTNER` , `LOT` , Prussian line/zoll/fuss/elle/rute, `KLAFTER` , `GERMAN_MILE` , `MORGEN` , `SCHEFFEL`)
361–367	Additional physical units (`SURVEY_FOOT` , `LEAGUE` , `CABLE` , `HAND` , `QUINTAL` , `SCRUPLE` , `BAUD`)
368–371	Temperature scales (`DELISLE` , `NEWTON_TEMP` , `REAUMUR` , `ROMER`)
372–377	Ratio/proportion units (`PERCENT` , `PER_MILLE` , `PER_MYRIAD` , `PER_CENT_MILLE` , `PPM` , `PPB`)
378–379	Appended fiat currencies (`ZWG` , `XCG`) — added past the unit block for ABI stability; see `CURRENCY_EXT_FIRST` / `CURRENCY_EXT_LAST`
380	`_SENTINEL` (internal bound; do not use)

Note on `CUP` : the Cuban-Peso currency is exposed as `BaseUnit.CUP_` (trailing underscore), not `BaseUnit.CUP` . The plain name `CUP` is the US-cup volume unit (enum value 81); since Python enum member names must be unique, the currency at value 167 takes the suffixed name. This affects the Python member name only — the `.bvnr` wire token is still the bare uppercase `CUP` , and case stays load-bearing (`cup` = volume, `CUP` = currency).

Spec 1.1 additions

These bindings target the **Bovnar spec (v1.1)**; spec 1.0 remains the frozen baseline. The 1.1 features (all gated on a `#!bovnar 1.1` directive — an unversioned document is treated as 1.0) are exposed as:

- **Version:** `bovnar.version()` (library version string), `bovnar.spec_version()` → `(major, minor)` of the highest spec understood, and `bovnar.peek_version(data)` → the `(major, minor)` a document declares, or `None`. `Reader.declared_version` gives the same after a read; `read_mem` / `read_fd` / `loads` accept `strict_version=True` to reject a too-new version (`ErrorCode.UNSUPPORTED_SPEC_VERSION`).
- **Richer escapes:** `\u{…}` and `\xHH` in string literals are decoded by the C reader, so `loads` returns the resulting text transparently. A surrogate / out-of-range `\u` is `ErrorCode.INVALID_CODEPOINT`.
- **Datetime family:** `loads` decodes a `<datetime:width,epoch>` value to a plain `int` (signed epoch seconds). With `typed=True` it is a `Quantity` whose `.epoch_name` (`"unix"`, `"tai"`, ...) and `.epoch_mjd` recover the epoch. `dumps()` emits the `<datetime:…>` annotation and prepends `#!bovnar 1.1` automatically when the object contains a datetime. With `typed=True` the sub-second fraction of an ISO-8601 literal (spec 1.1) is preserved on the `Quantity` and re-emitted, so a `loads(typed=True)`→`dumps()` round-trip is lossless (`…T12:00:00.5Z` survives verbatim). The plain (non-typed) value is only the whole-second `int` carrier, so a non-typed `loads`→`dumps` drops the fraction; to read it explicitly use the DOM tier (`DomNode.datetime_fraction`) or the streaming reader (`bvnr_data_t.frac_data` via a callback). `ValueTypeFamily.DATETIME` is the family enum member.
- **Reference array indexing:** `&.matrix[0][1]` paths are stored verbatim and resolved by `DomDoc.lookup("matrix[0][1]")` at the DOM layer.

`ErrorCode` adds `INVALID_SPEC_VERSION` (42), `UNSUPPORTED_SPEC_VERSION` (43), `INVALID_CODEPOINT` (44), `INVALID_DATETIME_LITERAL` (45), and `DATETIME_LITERAL_UNSUPPORTED_EPOCH` (46).