HOOX · PYNE · DOCUMENTATION

SCRIPT AUTHORS · LIBRARY CONSUMERS · EDITOR USERS

END USER MANUAL

Install, CLI, library API, editors, and Pro API as a consumer

Open-source Pine Script evaluation framework

2026-08-03 DIN A4 · HTTPS://HOOX.SH/PYNE/DOCS OPEN SOURCE

CONTENTS

OVERVIEW

title: "PYNE Documentation" description: "Open-source Pine Script™ evaluation framework — grammar, AST, bar-loop runtime, LSP, Pro API, and edge workers."

PYNE DOCUMENTATION

PYNE is the open evaluation stack for TradingView® Pine Script™: a formal ANTLR4 grammar, an ASDL algebraic AST, a deterministic bar-loop evaluator, a Language Server Protocol surface, a Flask Pro API, and edge workers that share one evaluate contract.

Abstract

Where closed hosts couple the language to a proprietary charting host, PYNE treats the language as an inspectable pipeline:

```
Source (.pine)
  → ANTLR4 lexer/parser (resource grammar)
  → ASDL AST (builder)
  → bar-loop evaluator (builtins, strategy, drawings)
  → plots / events / metrics
```

The same pipeline powers the desk CLI, the LSP binary, the Pro API, browser Pyodide (via AXIS), and Cloudflare® workers. Round-trip fidelity (`parse → unparse`) and a large builtin corpus are first-class invariants, not marketing claims — see compatibility and implementation status.

Product map

Surface	Role	Start here
Library + CLI	Parse, lint, unparse, evaluate on your machine	End User hub
Language core	Grammar, AST, visitors, type system	Core
Runtime	Series model, builtins, strategy, compiler	Runtime
LSP	Diagnostics, completion, hover, format	LSP
Pro API	HTTP `/run`, preview, backtest	API
Ops	CI, Docker, Nuitka, secrets, GCP	DevOps
AXIS (separate product)	Charting PWA AXIS	AXIS docs
HOOX	Edge trade mesh	HOOX docs

Conceptual model

```
flowchart LR
  S[Pine source] --> P[ANTLR4 parse]
  P --> A[ASDL AST]
  A --> E[Bar-loop evaluator]
  A --> L[LSP features]
  A --> U[Unparser]
  E --> R[Plots / strategy events]
  E --> API[Pro API / workers]
  R --> AXIS[AXIS optional]
  R --> HOOX[HOOX trade-worker]
```

Invariant: evaluation never requires a proprietary chart host. AXIS is an optional *AXIS*; HOOX is an optional *execution mesh*.

Tracks

End User

Install the package, run the CLI, embed the library API, wire an editor, or call the Pro API as a consumer.

→ End User hub

Core / Runtime / LSP / API

Contributor and systems manuals: how the front-end, semantics, editor protocol, and HTTP bridge are built.

DevOps

CI matrices, Fernet metadata encryption, Nuitka LSP binaries, containers, Cloud Build.

→ DevOps hub

Reference

Compatibility guarantees, missing surface, numerical validation, TypeScript `pine-worker` parity tool.

Sister surfaces

- **pyne-worker** (separate repo) — Python Cloudflare® Worker for production `/run`.
- **pine-worker** (this monorepo, `pine-worker/`) — TypeScript port + converter; see pine-worker.
- **AXIS** (`frontend/`) — installable PWA; documented at hoox.sh/axis/docs.

Offline & agent exports

Auto-generated like HOOX manuals (`bun run docs:exports`):

Kind	Path
Full-corpus LLM pack	`llm.txt`
LLM site map (llmstxt.org)	`llms.txt`
Track PDFs (A4)	`/exports/pyne-*-manual.pdf`

See also

- Installation
- Quick start
- Grammar pipeline
- Evaluate contract

OVERVIEW

title: "End User Hub" description: "Choose-your-path guide for installing PYNE, running the CLI, embedding the library, wiring editors, and calling the Pro API."

END USER HUB

This track is for people who **consume** PYNE: parse and reformat Pine Script™, lint before upload, evaluate expressions or full scripts against OHLCV, wire an editor via LSP, or call the HTTP evaluate contract. Contributor internals (grammar regeneration, AST builder, bar-loop semantics) live under Core, Runtime, and DevOps.

Abstract

PYNE exposes one language pipeline through several thin surfaces:

```
.pine source
  → parse (ANTLR4 + ASDL AST)
  → optional: dump | unparse | lint | walk/transform
  → optional: evaluate (literal_eval | bar-loop Runtime)
  → plots / events / drawings / metrics
```

The **desk path** is the `pynescript` Click CLI and `pynescript.ast` library API. The **editor path** is `pynescript-lsp` (stdio). The **HTTP path** is the Flask Pro API (`POST /run` , preview, backtest). The optional chart is AXIS; the optional trade mesh is HOOX. Evaluation does not require either.

Conceptual model

```
flowchart TB
  subgraph consumers [Consumer surfaces]
    CLI[pynescript CLI]
    LIB[Python library]
    LSP[pynescript-lsp]
    HTTP[Pro API /run]
  end
  subgraph pipeline [Shared pipeline]
    P[parse]
    A[ASDL AST]
    U[unparse / dump]
    L[lint]
    E[evaluator / Runtime]
  end
  CLI --> P
  LIB --> P
  LSP --> P
  HTTP --> E
  P --> A
  A --> U
  A --> L
  A --> E
  E --> OUT[plots / series / events]
  OUT --> AXIS[AXIS]
  OUT --> HOOX[HOOX trade-worker]
```

Surface	Entry	Typical job
CLI	`pynescript …`	One-shot parse, format, lint, market data
Library	`from pynescript.ast import parse, unparse`	Embed parse/transform/eval in tools
LSP	`pynescript-lsp`	Diagnostics, completion, hover in editors
Pro API	`POST /run`	Shared evaluate contract for AXIS / workers
AXIS	separate product	Chart PWA AXIS over evaluate results
HOOX	separate product	Edge execution mesh after strategy events

Choose your path

1. Getting started

New install, first parse, and environment knobs:

- **Installation** — PyPI, extras (`lsp`, `data`), Hatch checkout, smoke checks.
- **Quick start** — Parse → dump → unparse → lint → `literal_eval` in under ten minutes.
- **Configuration** — Extras, console scripts, Pro API env vars, editor settings.

2. Operational guides

Daily workflows against the same pipeline:

- **CLI** — `parse-and-dump`, `parse-and-unparse`, `lint`, `data`, `download-builtin-scripts`.
- **Library API** — `parse`, `unparse`, `dump`, `walk`, visitors, transformers, linter.
- **Evaluate scripts** — Expression eval, bar-loop Runtime, strategy events, mock vs live data.
- **Editors** — VS Code, Neovim, Zed, Emacs, Helix; `clients/` snippets.
- **Pro API usage** — `/run`, `/run/batch`, preview, backtest as a **consumer**.
- **Troubleshooting** — Parse failures, recursion limits, missing extras, CORS, auth.

3. Reference

- **CLI commands** — Full option trees for every Click command.
- **Glossary** — AST, series, bar-loop, Runtime, ASDL, and related terms.
- **FAQ** — Version support, TradingView parity, licensing, AXIS/HOOX boundaries.

Interface surface (consumer map)

Want	Command / import	Docs
Install core	`pip install pynescript`	Installation
Install LSP	`pip install "pynescript[lsp]"`	Editors
Install market data deps	`pip install "pynescript[data]"`	CLI data
Parse file	`pynescript parse-and-dump path.pine`	CLI
Normalize source	`pynescript parse-and-unparse path.pine`	CLI
Lint	`pynescript lint path.pine`	CLI
Library parse	`from pynescript.ast import parse, unparse`	Library API
Expression eval	`from pynescript.ast.helper import literal_eval`	Evaluate
Start LSP	`pynescript-lsp` (stdio)	Editors
Local Pro API	`make run` / `python -m backend.app`	Pro API usage

Two console scripts are registered in `pyproject.toml`:

Script	Module	Role
`pynescript`	`pynescript.__main__:cli`	Desk CLI (Click group)
`pynescript-lsp`	`pynescript.langserver.__main__:main`	Language server (pygls)

Do not conflate them: lint and parse live on `pynescript`; editor features live on `pynescript-lsp`.

Internals (repo paths)

Consumer-relevant code only — deeper design is linked from each guide:

Path	Role
`src/pynescript/__main__.py`	Click CLI: parse-and-dump, parse-and-unparse, lint, data, download-builtin-scripts
`src/pynescript/ast/helper.py`	Public `parse`, `unparse`, `dump`, `walk`, `literal_eval`
`src/pynescript/ast/linter.py`	`lint_script` / `PineLinter`
`src/pynescript/ast/evaluator/`	Bar-aware and literal evaluators
`src/pynescript/langserver/`	LSP features and server
`src/pynescript/util/data.py`	Historical data providers (mock, yahoo, alphavantage, ccxt)
`backend/app.py`	Flask Pro API entry (`/run`, auth, blueprints)
`backend/runtime.py`	HTTP-facing `Runtime.run` bar loop
`examples/`	Worked scripts (parse, RSI strategy, datafeed)
`clients/`	Editor client configs (Neovim, Zed, Emacs)
`vscode-extension/`	VS Code extension package

Invariants & edge cases

- **Round-trip is first-class.** `parse` → `unparse` should preserve semantics; use it to normalize formatting, not as a semantic rewrite.
- **Mode split.** `parse(..., mode="exec")` yields a script tree; `mode="eval"` parses a single expression (used by `literal_eval`).
- **No proprietary host required.** Evaluation works offline with mock or supplied OHLCV; AXIS/HOOX are optional.
- **Extras are opt-in.** Core parse/lint needs only base deps (`antlr4-python3-runtime`, `click`, …). LSP needs `[lsp]`; CCXT paths need `[data]` / `[datafeed]`.
- **Parametrized corpus.** Tests under `tests/data/builtin_scripts/` are large; a fixture using `pinescript_filepath` can explode CI runtime if misused.

Worked examples

Minimal library round-trip:

```
from pynescript.ast import parse, unparse

source = """
//@version=5
indicator("My RSI")
plot(ta.rsi(close, 14))
"""
tree = parse(source)
print(unparse(tree))
```

Minimal CLI:

```
pip install pynescript
pynescript parse-and-dump examples/rsi_strategy.pine
pynescript lint examples/rsi_strategy.pine
```

Minimal HTTP evaluate (local server running):

```
curl -s http://127.0.0.1:5002/ \
  | python -m json.tool
```

Failure modes

Symptom	Likely cause	Where to go
`pynescript: command not found`	Package not on PATH / venv inactive	Installation
`pynescript-lsp` missing	Installed without `[lsp]` extra	Editors
`SyntaxError` on parse	Grammar mismatch or truncated source	Troubleshooting
`/run` 400 `NO_DATA`	Missing OHLCV list	Pro API usage
Numerical drift vs TradingView	Builtin / series edge case	Compatibility

See also

- PYNE product map — full stack overview
- Installation
- Quick start
- Evaluate contract (API)
- AXIS docs — optional chart
- HOOX docs — optional edge trade mesh

INSTALLATION

title: "Installation" description: "Install pynescript from PyPI or a source checkout, select extras for LSP and market data, and verify the CLI and library imports."

INSTALLATION

Abstract

PYNE ships as the Python package `pynescript` (src-layout under `src/pynescript/`). The default install gives you the Click CLI (`pynescript`) and the AST library (`parse`, `unparse`, `dump`, `walk`, `literal_eval`, linter). Optional extras add the Language Server (`pynescript-lsp`) and CCXT-backed market data. A Hatch-managed checkout is the supported path for contributors who need tests, grammar tooling, or the Flask Pro API in-tree.

Conceptual model

```
flowchart LR
  PYPI[PyPI pynescript] --> CORE[Core wheel]
  GIT[Git checkout] --> EDIT[pip install -e]
  CORE --> CLI[pynescript]
  CORE --> LIB[pynescript.ast]
  EDIT --> CLI
  EDIT --> LIB
  EXTRA_LSP["extra: lsp"] --> LSP[pynescript-lsp]
  EXTRA_DATA["extra: data / datafeed"] --> CCXT[ccxt providers]
  CORE --> EXTRA_LSP
  CORE --> EXTRA_DATA
```

Install flavor	What you get	When
`pip install pynescript`	CLI + library	Desk use, scripts, CI lint
`pip install "pynescript[lsp]"`	+ pygls LSP server	Editors
`pip install "pynescript[data]"`	+ ccxt	Live/historical exchange data
`pip install -e ".[dev-lsp]"`	Editable + LSP test harness	Contributing to LSP
Hatch env	Matrix tests, lint, docs	Full monorepo workflow

Interface surface

Prerequisites

- **Python ≥ 3.10** (classifiers declare 3.10–3.12; 3.13 may work in CI — pin to 3.10+ for production)
- `pip` or an equivalent installer; virtualenv strongly recommended
- For Pro API locally: `flask`, `flask-cors`, `numpy`, `matplotlib` (see `backend/requirements.txt`)
- For VS Code extension build: Node 22+ under `vscode-extension/` (optional)

Core install (PyPI / local package)

From a release or built wheel:

```
pip install pynescript
```

From a cloned monorepo root (non-editable):

```
pip install .
```

Editable (preferred for development):

```
pip install -e .
# or with Make:
make install
```

Extras

Defined in `pyproject.toml` under `[project.optional-dependencies]`:

```
# Language Server (pygls + lsprotocol)
pip install "pynescript[lsp]"

# CCXT for exchange data / datafeed
pip install "pynescript[data]"
# alias:
pip install "pynescript[datafeed]"

# LSP + pytest-lsp for protocol tests
pip install -e ".[dev-lsp]"
```

Combine extras:

```
pip install -e ".[lsp,data]"
```

Hatch (contributor)

```
# Install Hatch if needed, then:
hatch shell
hatch run test:test
hatch run lint:style
```

Make targets used in-repo (see `AGENTS.md`):

```
make install         # pip install -e ".[lsp]"
make test            # pytest tests/ -v --tb=short
make lint            # ruff check src/ tests/ backend/
make run             # python -m backend.app
make run-lsp         # python -m pynescript.langserver
make build-check     # fast import check without Nuitka
```

Console scripts

After install, two entry points should resolve:

Command	Purpose
`pynescript`	Click CLI group
`pynescript-lsp`	Language server (requires `[lsp]`)

```
pynescript --version
pynescript --help
which pynescript-lsp   # only after [lsp]
```

Internals (repo paths)

Path	Role
`pyproject.toml`	Package metadata, deps, optional extras, scripts
`src/pynescript/`	Library + CLI + LSP package tree
`src/pynescript/__about__.py`	`__version__` (e.g. `0.2.0`)
`src/pynescript/__main__.py`	CLI implementation
`src/pynescript/langserver/__main__.py`	`pynescript-lsp` entry
`backend/requirements.txt`	Pro API server deps (not in core extras)
`Makefile`	Convenience install / test / run targets
`CONTRIBUTING.md`	Hatch-based contributor workflow

Base runtime dependencies (core):

- `antlr4-python3-runtime>=4.13.1`
- `click>=8.1.7`
- `requests`
- `tqdm`

Invariants & edge cases

- **License.** Package license is **AGPL-3.0-or-later** (`pyproject.toml`). Network use triggers AGPL source-offer obligations (see GNU AGPL §13); consult your counsel for commercial embedding.
- **Src layout.** Imports are `pynescript.*`, not a flat package. Editable install requires hatchling to map `src/`.
- **LSP is not a CLI subcommand of `pynescript` in the current Click group.** README sometimes says `pynescript lsp`; the registered script is **`pynescript-lsp`**.
- **Pro API is not a pip extra.** Run from monorepo with backend deps, or call a hosted endpoint. Local: `make run` binds `HOST` / `PORT` (default `127.0.0.1:5002`).
- **Anaconda / Nuitka.** Building the onefile LSP binary may need `libpython-static` or `--static-libpython=no` (see build docs). End users rarely need this.
- **Generated artifacts.** Do not hand-edit `src/pynescript/ast/grammar/antlr4/generated/` or ASDL generated modules.

Worked examples

Smoke-test CLI and library

```
python - <<'PY'
from pynescript.ast import parse, unparse
from pynescript import __about__
print("version", __about__.__version__)
src = '//@version=5\nindicator("t")\nplot(close)\n'
print(unparse(parse(src)))
PY

pynescript parse-and-dump examples/rsi_strategy.pine | head
pynescript lint examples/rsi_strategy.pine
```

Fresh venv recipe

```
python3.12 -m venv .venv
source .venv/bin/activate
pip install -U pip
pip install "pynescript[lsp,data]"
pynescript --version
pynescript-lsp --help 2>/dev/null || true  # server may only speak stdio
```

Monorepo desk + Pro API

```
git clone https://github.com/jango-blockchained/pynescript.git
cd pynescript
pip install -e ".[lsp]"
pip install -r backend/requirements.txt
make run   # Flask on :5002
```

CI-style install

```
pip install .
# or matrix:
# pip install -e ".[dev-lsp]"
pytest tests/test_parse_and_unparse.py -q
```

Failure modes

Failure	Diagnosis	Fix
`ModuleNotFoundError: pynescript`	Wrong env / not installed	Activate venv; reinstall
`No module named 'pygls'`	LSP import without extra	`pip install "pynescript[lsp]"`
`No module named 'ccxt'`	data CLI/provider path	`pip install "pynescript[data]"`
`pynescript` not found	Scripts not on PATH	Use `python -m pynescript` or fix venv `bin/`
Import error from editable	Incomplete install / broken path	Re-run `pip install -e .` from repo root
Backend `ImportError: flask`	Pro API without backend deps	`pip install -r backend/requirements.txt`
ANTLR / grammar mismatch on odd syntax	Older package vs newer scripts	Upgrade package; see Troubleshooting

See also

- Quick start
- Configuration
- CLI guide
- Editors / LSP
- Pro API usage
- End User hub

QUICK START

title: "Quick Start" description: "Parse, dump, unparse, lint, and evaluate Pine Script™ with the pynescript CLI and library in under ten minutes."

QUICK START

Abstract

This page is a linear path from a working install to the five desk operations most users need: **parse**, **dump**, **unparse**, **lint**, and **evaluate**. Full option trees live in CLI commands; deeper evaluation semantics live in Evaluate scripts.

Conceptual model

```
sequenceDiagram
  participant U as User
  participant CLI as pynescript CLI
  participant AST as ASDL AST
  participant EV as literal_eval / Runtime
  U->>CLI: parse-and-dump script.pine
  CLI->>AST: parse(source)
  AST-->>CLI: tree
  CLI-->>U: dump text
  U->>CLI: parse-and-unparse
  CLI->>AST: parse + unparse
  AST-->>U: normalized .pine
  U->>CLI: lint
  CLI-->>U: LintWarning list
  U->>EV: literal_eval / Runtime.run
  EV-->>U: values / plots / events
```

Interface surface

Assumes:

```
pip install hoox-pyne
# optional for data:
# pip install "hoox-pyne[data]"
```

Sample script in-tree: `examples/rsi_strategy.pine`.

Internals (repo paths)

Step	Implementation
CLI parse/dump	`src/pynescript/__main__.py` → `parse_and_dump`
CLI unparse	`parse_and_unparse`
CLI lint	`lint` → `pynescript.ast.linter.lint_script`
Library helpers	`src/pynescript/ast/helper.py`
Example scripts	`examples/parse_dump_unparse.py`, `examples/evaluate_expressions.py`

Invariants & edge cases

- Always declare `//@version=5` or `//@version=6` at the top; the linter warns (`W001`) if missing.
- `parse-and-unparse` is a **formatter via AST**, not a semantic optimizer.
- `literal_eval` is for expressions / built-in calls with optional series context — not a full multi-bar strategy backtester. For bar loops, use `backend.runtime.Runtime` or the Pro API `/run`.
- Filename arguments to the CLI must exist and be readable; lint may also read stdin (`-` or no file).

Worked examples

1. Parse and dump the AST

```
pynescript parse-and-dump examples/rsi_strategy.pine
```

Pretty-print with indent and optional file output:

```
pynescript parse-and-dump examples/rsi_strategy.pine --indent 2 --output-file /tmp/rsi.ast.txt
```

Library equivalent:

```
from pynescript.ast import parse, dump

with open("examples/rsi_strategy.pine", encoding="utf-8") as f:
    tree = parse(f.read(), "examples/rsi_strategy.pine")
print(dump(tree, indent=2))
```

2. Round-trip unparse (normalize)

```
pynescript parse-and-unparse examples/rsi_strategy.pine
pynescript parse-and-unparse messy.pine --output-file clean.pine
```

```
from pynescript.ast import parse, unparse

source = """
//@version=5
indicator("My RSI")
rsi(close, 14)
"""
print(unparse(parse(source)))
```

Canonical demo: `examples/parse_dump_unparse.py`.

3. Lint before upload

```
pynescript lint examples/rsi_strategy.pine
pynescript lint --fail-on warnings examples/rsi_strategy.pine
echo '//@version=5\nindicator("x")\nplot(close)' | pynescript lint -
```

Library:

```
from pynescript.ast.linter import lint_script

issues = lint_script(open("examples/rsi_strategy.pine").read(), "rsi_strategy.pine")
for w in issues:
    print(w)  # severity: [CODE] message at line N
```

4. Evaluate expressions

```
from pynescript.ast.helper import literal_eval

print(literal_eval("1 + 2 * 3"))  # 7
print(literal_eval("math.sqrt(16)"))
prices = [100, 102, 101, 103, 105, 104, 106, 108, 107, 110]
print(literal_eval(f"ta.rsi({prices}, 9)"))

# Series history with context
ctx = {
    "close": [100, 102, 101, 103, 105],
    "open": [99, 101, 100, 102, 104],
}
print(literal_eval("close[0] - close[1]", ctx))
```

Broader demo: `examples/evaluate_expressions.py`.

5. Fetch sample market data (CLI)

```
# mock provider (no network)
pynescript data AAPL --provider mock

# Yahoo (network)
pynescript data AAPL --provider yahoo --period 6mo --interval 1d

# CCXT (needs pynescript[data])
pynescript data BTC/USDT --provider ccxt --exchange binance
```

6. Optional: local Pro API `/run`

```
# from monorepo with backend deps
make run
```

```
curl -s -X POST http://127.0.0.1:5002/run \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=5\nindicator(\"t\")\nplot(close)",
    "data": [
      {"time": 1, "open": 1, "high": 2, "low": 0.5, "close": 1.5, "volume": 100},
      {"time": 2, "open": 1.5, "high": 2.5, "low": 1.0, "close": 2.0, "volume": 120}
    ]
  }' | python -m json.tool
```

See Pro API usage for schemas and batch mode.

7. Optional: start LSP for editors

```
pip install "hoox-pyne[lsp]"
pynescript-lsp   # stdio; normally launched by the editor
```

Wire Neovim / Zed / Emacs using Editors and `clients/`.

Failure modes

Step fails	Check
Dump raises `SyntaxError`	Script version / unsupported construct — try a minimal `indicator` + `plot(close)`
Unparse output differs cosmetically	Expected; compare semantic structure via dump
Lint exits non-zero	`--fail-on` threshold; inspect codes `E001`, `W001`, …
`literal_eval` `NotImplementedError`	Expression not in literal/builtin subset; use full Runtime
`data` provider error	Network, API key, missing `ccxt` extra

See also

- Installation
- Configuration
- CLI guide
- Library API
- Evaluate scripts
- CLI command reference

CONFIGURATION

title: "Configuration" description: "Package extras, console scripts, Pro API environment variables, data providers, and editor LSP settings for PYNE."

CONFIGURATION

Abstract

PYNE is mostly **convention-over-config** at the library layer: parse and evaluate take explicit arguments rather than a global config file. Configuration appears at three boundaries: **install extras** (what code is available), **process environment** (Pro API host, CORS, API keys store), and **editor / LSP client settings** (diagnostics, formatting, binary path). This page inventories those knobs for end users.

Conceptual model

```
flowchart TB
  subgraph install [Install-time]
    EX[optional-dependencies]
  end
  subgraph runtime [Process env]
    ENV[HOST PORT ALLOWED_ORIGINS ADMIN_TOKEN API_KEY_STORE]
  end
  subgraph editor [Editor client]
    LS[pynescript-lsp settings]
  end
  EX --> BIN[console scripts + imports]
  ENV --> API[backend.app Flask]
  LS --> FEAT[diagnostics / format / completion]
  BIN --> CLI[pynescript]
  BIN --> LSPBIN[pynescript-lsp]
```

There is no required `pynescript.toml` for core use.

Interface surface

Package extras (`pyproject.toml`)

Extra	Dependencies	Enables
(none)	antlr4-runtime, click, requests, tqdm	CLI + AST + linter + literal_eval
`lsp`	pygls, lsprotocol	`pynescript-lsp`
`dev-lsp`	pygls, lsprotocol, pytest-lsp	LSP + protocol tests
`data`	ccxt	Exchange historical data
`datafeed`	ccxt	Same as `data` (alias for realtime feed paths)

Console scripts

Name	Entry point	Notes
`pynescript`	`pynescript.__main__:cli`	Always with core install
`pynescript-lsp`	`pynescript.langserver.__main__:main`	Needs importable pygls

Invoke without scripts on PATH:

```
python -m pynescript --help
python -m pynescript.langserver
```

CLI flags that act as “config”

Per-invocation only (no persistence):

Command	Notable options
`parse-and-dump`	`--encoding`, `--indent`, `--output-file`
`parse-and-unparse`	`--encoding`, `--output-file`
`lint`	`--encoding`, `--fix`, `--fail-on {errors,warnings,all}`
`data`	`--provider`, `--period`, `--interval`, `--api-key`, `--secret`, `--exchange`
`download-builtin-scripts`	`--script-dir` (required)

`--fix` on `lint` is accepted by Click; auto-fix behavior may be incomplete — .

Pro API environment variables

Consumed by `backend/app.py` and middleware:

Variable	Default	Purpose
`HOST`	`127.0.0.1`	Bind address when running `__main__`
`PORT`	`5002`	Bind port
`ALLOWED_ORIGINS`	`https://pynescript.ai`, `https://app.pynescript.ai`, localhost regex	CORS allowlist (comma-separated; may include regex strings)
`ADMIN_TOKEN`	unset	Required for `POST /auth/create_key`; if unset, create returns 403
`API_KEY_STORE`	`/root/pynescript/data/api_keys.json`	Path for key store (override for local)

Also relevant:

Variable	Context
`MAX_CONTENT_LENGTH`	Hardcoded 5 MiB on Flask app config (not env) — large OHLCV POSTs fail above this

Local run:

```
python -m backend.app
# or: make run
```

Data providers

CLI `pynescript data` / library providers:

Provider	Auth	Notes
`mock`	none	Deterministic offline bars (default)
`yahoo`	none	Yahoo Finance path
`alphavantage`	`--api-key` (falls back to `demo`)	Limited with demo key
`ccxt`	optional key/secret; `--exchange`	Requires `[data]` extra

Pro API `/run` optional fields:

Field	Values	Role
`data_source`	`""`, `mock`, `ccxt`, `ccxtpro`, `yahoo`, `alphavantage`	Wires `request.*` resolution
`data_options`	object	exchange, api_key, seed, …
`mode`	`interpret` (default), `compile`	AST walker vs compiled subset

Editor / LSP client settings

VS Code extension keys (from `clients/README.md`):

Setting	Default	Meaning
`pynescript.lsp.enabled`	true	Toggle server
`pynescript.lsp.command`	`pynescript-lsp`	Binary path
`pynescript.formatting.enabled`	true	Format document
`pynescript.diagnostics.enabled`	true	Lint squiggles
`pynescript.completion.snippets`	true	Snippet completions

Neovim (`clients/neovim.lua`):

```
settings = {
  pinescript = {
    formatting = { enabled = true },
    diagnostics = { enabled = true },
    completion = { snippets = true },
  },
}
```

Zed: `language_servers.pynescript.command` / `arguments: ["--stdio"]` — see Editors.

AXIS / frontend coupling (optional)

When running SuperChart Lite PWA against local Flask:

Make target	Port (typical)
`make run`	API `:5002`
`make run-frontend`	PWA `:8081`
`make worker-dev`	CF Worker `:8787`

CORS must allow the PWA origin via `ALLOWED_ORIGINS`.

Internals (repo paths)

Path	Config surface
`pyproject.toml`	extras, scripts, Python requires
`src/pynescript/__main__.py`	CLI options
`backend/app.py`	Flask, CORS, HOST/PORT, body size
`backend/middleware/auth.py`	`API_KEY_STORE`, key tiers
`backend/middleware/schemas.py`	Request field defaults (`mode`, `symbol`, …)
`clients/*.json	lua
`vscode-extension/`	Extension contribution points

Invariants & edge cases

- **Unknown JSON fields on Pro API are rejected** (`UNKNOWN_FIELDS`) — schemas are strict.
- **CORS defaults exclude arbitrary production domains** — set `ALLOWED_ORIGINS` deliberately.
- **Key store default path is production-oriented** (`/root/...`) — always override locally.
- **Lint** `--fail-on` only affects process exit code; it does not change which rules run.
- **Recursion limit** during parse is raised to at least 5000 temporarily inside `helper._parse` for deep nests; not user-configurable.

Worked examples

Desk-only machine

```
python -m venv .venv && source .venv/bin/activate
pip install hoox-pyne
# no env vars required
pynescript lint strategy.pine --fail-on errors
```

Editor workstation

```
pip install "hoox-pyne[lsp]"
# ensure which pynescript-lsp
# paste clients/neovim.lua or clients/zed.json as documented
```

Local API + AXIS

```
pip install -e ".[lsp]"
pip install -r backend/requirements.txt

make run
# other terminal:
make run-frontend
```

Scripted Alpha Vantage fetch

```
pynescript data EUR/USD --provider alphavantage --api-key "$AV_KEY" --period 3mo
```

Failure modes

Symptom	Config fix
Browser CORS error on `/run`	Add origin to `ALLOWED_ORIGINS`
`create_key` always 403	Set `ADMIN_TOKEN` and send `X-Admin-Token`
API keys vanish / permission error	Point `API_KEY_STORE` to writable path
Payload too large	Split bars or stay under 5 MiB body limit
Editor “server failed to start”	Fix `pynescript.lsp.command` PATH; install `[lsp]`
`ccxt` import errors	Install `[data]` extra

See also

- Installation
- Pro API usage
- Editors
- CLI commands
- API auth (systems)
- AXIS docs

CLI

title: "CLI Guide" description: "Use the pynescript Click CLI to dump ASTs, round-trip format, lint scripts, fetch market data, and download builtin corpora."

CLI GUIDE

Abstract

The `pynescript` console script is a thin Click group over the same helpers used by the library API. It is optimized for **shell pipelines, CI gates, and one-shot inspection** — not for multi-bar evaluation (use the library `Runtime` or Pro API for that). This guide covers workflows; exhaustive flags are in CLI commands.

Conceptual model

```
flowchart LR
  F[.pine file / stdin] --> C[pynescript]
  C --> D[parse-and-dump]
  C --> U[parse-and-unparse]
  C --> L[lint]
  C --> M[data]
  C --> B[download-builtin-scripts]
  D --> OUT1[AST text]
  U --> OUT2[normalized source]
  L --> OUT3[diagnostics + exit code]
  M --> OUT4[bar summary]
  B --> OUT5[corpus .pine files]
```

Commands mirror `pynescript.ast.helper` and friends:

CLI	Library
`parse-and-dump`	`parse` + `dump`
`parse-and-unparse`	`parse` + `unparse`
`lint`	`lint_script`
`data`	`pynescript.util.data.get_provider`
`download-builtin-scripts`	`pynescript.util.pine_facade.download_builtin_scripts`

Interface surface

```
pynescript --help
pynescript --version
pynescript <command> --help
```

Command	Input	Output
`parse-and-dump PATH`	file	AST dump (stdout or `--output-file`)
`parse-and-unparse PATH`	file	Pine source
`lint [PATH|-]`	file or stdin	human diagnostics
`data SYMBOL`	symbol + provider opts	summary stats
`download-builtin-scripts`	`--script-dir`	downloads reference scripts

Not part of this Click group: starting the language server — use `pynescript-lsp`.

Internals (repo paths)

Path	Role
`src/pynescript/__main__.py`	All command implementations
`src/pynescript/ast/helper.py`	`parse` , `dump` , `unparse`
`src/pynescript/ast/linter.py`	`lint_script`
`src/pynescript/util/data.py`	providers + `DataProviderError`
`src/pynescript/util/pine_facade.py`	builtin script download
`examples/`	sample `.pine` inputs

Invariants & edge cases

- **Encoding default is UTF-8** for file reads/writes.
- **`--output-file -`** means stdout (Click `allow_dash=True`).
- **Lint without a path** reads stdin — useful in git hooks and `cat script.pine | pynescript lint`.
- **`--fail-on`** defaults to `errors` (syntax/`E001`); `warnings` fails on any warning severity too; `all` fails if any issue exists.
- **`data` default provider is `mock`** — offline-safe.
- **Alpha Vantage without key** prints a warning and uses `demo`.
- Download command requires an explicit directory; create parents as needed (implementation writes into the given path).

Worked examples

Inspect structure of a strategy

```
pynescript parse-and-dump examples/rsi_strategy.pine --indent 2 | less
```

Compare two scripts' shapes in CI:

```
pynescript parse-and-dump a.pine --output-file /tmp/a.ast
pynescript parse-and-dump b.pine --output-file /tmp/b.ast
diff -u /tmp/a.ast /tmp/b.ast
```

Format / normalize

```
pynescript parse-and-unparse messy.pine --output-file clean.pine
# in-place-ish via shell:
pynescript parse-and-unparse strategy.pine --output-file strategy.pine.tmp \
  && mv strategy.pine.tmp strategy.pine
```

Lint gate in CI

```
set -e
pynescript lint strategies/ --help  # note: single file PATH, not directory recursion
#
for f in strategies/*.pine; do
  pynescript lint --fail-on warnings "$f"
done
```

Stdin pipeline:

```
git show HEAD:strategies/rsi.pine | pynescript lint --fail-on errors -
```

Market data smoke checks

```
pynescript data AAPL
pynescript data AAPL --provider yahoo --period 1y --interval 1d
pynescript data BTC/USDT --provider ccxt --exchange binance
pynescript data EUR/USD --provider alphavantage --api-key "$AV_KEY"
```

Typical stdout shape (fields from `__main__.py`):

```
Symbol: AAPL
Bars: N
Date range: N bars
First close: ...
Last close: ...
Volume (avg): ...
```

Refresh builtin corpus (contributors / offline tests)

```
pynescript download-builtin-scripts --script-dir tests/data/builtin_scripts
```

Use sparingly: network + rate limits; CI may cache the corpus.

Compose with library for evaluation

CLI stops at parse/lint/data. Chain:

```
# normalize then evaluate in Python
pynescript parse-and-unparse strat.pine --output-file /tmp/s.pine
python - <<'PY'
from pathlib import Path
from pynescript.ast.helper import parse
# hand off to Runtime or your own visitor
print(parse(Path("/tmp/s.pine").read_text()).__class__.__name__)
PY
```

Failure modes

Exit / message	Cause	Mitigation
Click path errors	missing file	check `PATH` exists and is a file
Parse exception during dump	invalid syntax	fix source; use smaller repro
`Lint failed with errors.`	`--fail-on` threshold	read `E00x` / `W00x` lines
`DataProviderError` → ClickException	network / auth / missing dep	install `[data]`, fix keys
Empty dump / unexpected tree	`mode` always exec for CLI	expression-only needs library `mode="eval"`

See also

- CLI commands reference
- Library API
- Quick start
- Troubleshooting

LIBRARY API

title: "Library API" description: "Embed pynescript: parse, dump, unparse, walk, transform, and lint Pine Script™ from Python."

LIBRARY API

Abstract

The public Python surface for language work lives primarily under `pynescript.ast`: parse source into an ASDL-backed tree, serialize it (`dump`), regenerate source (`unparse`), traverse (`walk`, visitors), rewrite (`NodeTransformer`), and lint (`lint_script`). Expression evaluation (`literal_eval`) and full script execution (evaluator / `Runtime`) sit adjacent; this page focuses on **structural** APIs. Evaluation workflows are covered in Evaluate scripts.

Conceptual model

```
flowchart LR
  S[str source] --> P[parse]
  P --> T[AST root Script or Expression]
  T --> D[dump]
  T --> U[unparse]
  T --> W[walk / NodeVisitor]
  T --> X[NodeTransformer]
  S --> L[lint_script]
  T --> E[literal_eval / NodeEvaluator]
```

Design intent mirrors Python's `ast` module: **stable tree operations**, location attributes, and visitor protocols — specialized for Pine's script model (annotations, series-aware semantics live at evaluation time).

Interface surface

Imports

```
# Re-exported convenience (node classes, helpers, visitors, transformers)
from pynescript.ast import parse, unparse, dump, walk
from pynescript.ast import NodeVisitor, NodeTransformer
from pynescript.ast.helper import literal_eval, iter_child_nodes, get_source_segment
from pynescript.ast.linter import lint_script, lint_file, PineLinter, LintWarning
```

`pynescript.ast.__init__` star-imports `helper`, `node`, `visitor`, `transformer`, and `error`.

`parse(source, filename="<unknown>", mode="exec") -> AST`

Arg	Meaning
`source`	Full script or expression text
`filename`	Error reporting; absolutized if file exists
`mode`	`"exec"` → script statements; `"eval"` → single expression

Raises `ValueError` on bad mode; `SyntaxError` (via error listener) on grammar failure.

```
tree = parse('//@version=5\nindicator("x")\nplot(close)\n')
expr = parse("close > open", mode="eval")
```

`unparse(node) -> str`

Round-trips an AST through `NodeUnparser`.

```
normalized = unparse(parse(messy_source))
```

`dump(node, *, annotate_fields=True, include_attributes=False, indent=None) -> str`

Human-readable tree serialization (debug, tests, CLI).

```
print(dump(tree, indent=2, include_attributes=True))
```

`walk(node) -> Iterator[AST]`

Breadth-first (deque) traversal yielding every node.

```
from pynescript.ast import walk
names = [n for n in walk(tree) if n.__class__.__name__ == "Name"]
```

Location helpers

Function	Role
`copy_location(new, old)`	Copy lineno/col offsets
`fix_missing_locations(node)`	Fill missing positions
`increment_lineno(node, n=1)`	Shift lines
`get_source_segment(source, node, *, padded=False)`	Slice original text
`iter_fields` / `iter_child_nodes`	Schema-aware iteration

Visitors and transformers

```
from pynescript.ast import NodeVisitor, NodeTransformer, parse

class CallCounter(NodeVisitor):
    def __init__(self):
        self.calls = 0
    def visit_Call(self, node):
        self.calls += 1
        self.generic_visit(node)

class RenameClose(NodeTransformer):
    def visit_Name(self, node):
        #
        if getattr(node, "id", None) == "close":
            node.id = "price"
        return node

tree = parse(source)
CallCounter().visit(tree)
tree2 = RenameClose().visit(tree)
```

Linter

```
from pynescript.ast.linter import lint_script, LintWarning

warnings: list[LintWarning] = lint_script(source, "strat.pine")
for w in warnings:
    print(w.code, w.severity, w.line, w.message)
```

Code family	Examples
`E001`	Syntax error (error severity)
`W001` – `W002`	Missing / old `@version`
`W101` – `W103`	Deprecated patterns
`C001` – `C004`	Naming / style (line length 120, trailing newline, ...)

`PineLinter.lint` runs: syntax → version → deprecated → naming → style.

`literal_eval` (bridge to evaluation)

```
from pynescript.ast.helper import literal_eval

literal_eval("ta.sma([1,2,3,4,5], 3)")
literal_eval("close[1]", {"close": [10.0, 11.0, 12.0]})
```

Optional `data_feed` / `data_provider` for `request.*` in literal contexts.

Internals (repo paths)

Path	Role
`src/pynescript/ast/helper.py`	Public parse/dump/walk/unparse/literal_eval
`src/pynescript/ast/builder.py`	Parse tree → ASDL nodes
`src/pynescript/ast/unparser.py`	`NodeUnparser`
`src/pynescript/ast/visitor.py`	`NodeVisitor`
`src/pynescript/ast/transformer.py`	`NodeTransformer`
`src/pynescript/ast/node.py`	Node re-exports / helpers
`src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl`	Algebraic schema
`src/pynescript/ast/linter.py`	Static rules
`src/pynescript/ast/error.py`	Error types
`examples/parse_dump_unparse.py`	Round-trip demo

Pipeline inside `parse`:

1. `InputStream` / `FileStream`
2. `PinescriptLexer` + `PinescriptParser` (error listener)
3. `PinescriptASTBuilder.visit`
4. In `exec` mode: `StatementCollector`, comment collection, `@` annotation attach

Invariants & edge cases

- **Annotation comments** (`//@version`, etc.) attach to script / function / type / assign nodes by kind suffix — see `_add_annotations` in `helper.py`.
- **Deep nesting:** temporary `sys.setrecursionlimit(max(old, 5000))` during parse.

- **`dump` requires a true AST node** — `TypeError` otherwise.
- **Transformers mutate or replace nodes** depending on your visit methods; prefer returning new nodes if you need purity.
- **Round-trip fidelity** is tested against a large corpus but is not a proof of bit-identical source; whitespace and some sugar may normalize.
- **Do not edit generated grammar modules**; extend via resource grammars if contributing.

Worked examples

End-to-end inspect

```
from pynescript.ast import parse, dump, unparse, walk

source = open("examples/rsi_strategy.pine", encoding="utf-8").read()
tree = parse(source, "examples/rsi_strategy.pine")
print(dump(tree, indent=2)[:500], "...")
print("---")
print(unparse(tree))
print("node count", sum(1 for _ in walk(tree)))
```

Collect all call names

```
from pynescript.ast import parse, NodeVisitor

class Calls(NodeVisitor):
    def __init__(self):
        self.names = []
    def visit_Call(self, node):
        func = node.func
        # Attribute vs Name depending on ta.rsi vs f()
        self.names.append(func)
        self.generic_visit(node)

c = Calls()
c.visit(parse(source))
```

Lint programmatically with fail semantics

```
from pynescript.ast.linter import lint_script

def assert_clean(path: str) -> None:
    text = open(path, encoding="utf-8").read()
    issues = lint_script(text, path)
    errors = [i for i in issues if i.severity == "error"]
    if errors:
        raise SystemExit("\n".join(map(str, errors)))
```

Expression mode for tooling

```
from pynescript.ast import parse, dump

print(dump(parse("ta.rsi(close, 14)", mode="eval"), indent=2))
```

Failure modes

Exception / result	Meaning
`SyntaxError`	Lexer/parser rejection
`ValueError: invalid argument mode`	mode not in `{exec,eval}`
Empty / odd tree	Empty body scripts; annotations-only edge cases
Unparse mismatch	Node kinds without unparser support — file issue against unparser
Linter false positives	Style rules are heuristic (regex naming) — treat `C00x` as advisory

See also

- Evaluate scripts
- CLI guide
- Helper API (core)
- Visitor / transformer (core)
- Linter (core)

EVALUATE SCRIPTS

title: "Evaluate Scripts" description: "Run Pine expressions and multi-bar scripts with literal_eval, the AST evaluator, backend Runtime, data providers, and strategy events."

EVALUATE SCRIPTS

Abstract

Evaluation is the second half of the PYNE pipeline: given an AST (or source) and **market context**, produce plots, series, strategy events, and drawings. End users typically choose among three graded tools:

1. `literal_eval` — pure/builtin expressions, optional series context
2. **AST evaluator** (`NodeLiteralEvaluator` / full evaluator mixins) — script-shaped evaluation in-process

3. `backend.runtime.Runtime` or **Pro API** `POST /run` — bar-loop over OHLCV with shared evaluate contract

This guide stays at the consumer level; series semantics and builtin inventories are under Runtime.

Conceptual model

```
flowchart TB
  SRC[Source] --> PARSE[parse]
  PARSE --> AST[AST]
  AST --> LE[literal_eval]
  AST --> NE[NodeEvaluator / bar visit]
  OHLCV[OHLCV bars] --> RT[Runtime.run]
  AST --> RT
  RT --> PLOTS[plots / series / plot_meta]
  RT --> EV[strategy events]
  RT --> DR[drawings]
  DF[data_feed / data_provider] --> RT
  DF --> LE
```

Mode	Bar loop	Strategy	Typical use
`literal_eval`	no (single shot)	limited	TA on arrays, math, strings
Evaluator in tests / tools	optional	yes via state	Unit tests, notebooks
`Runtime.run` / `/run`	yes	yes	Charts, AXIS, backtests
`mode="compile"`	yes (subset)	limited	Faster SMA/EMA/RSI-style paths

Interface surface

Expression evaluation

```
from pynescript.ast.helper import literal_eval

literal_eval("1 + 2 * 3")
literal_eval("math.max(1, 5, 3)")
literal_eval('str.upper("hi")')
literal_eval("array.size([1, 2, 3])")

prices = [100, 102, 101, 103, 105, 104, 106, 108, 107, 110]
literal_eval(f"ta.sma({prices}, 5)")
literal_eval(f"ta.rsi({prices}, 9)")
bb = literal_eval(f"ta.bb({prices}, 5, 2)")  # middle, upper, lower
```

Series history context (Pine-style `[0]` current / `[1]` previous as implemented):

```
context = {
    "close": [100, 102, 101, 103, 105],
    "open": [99, 101, 100, 102, 104],
    "high": [101, 103, 102, 104, 106],
    "low": [98, 100, 99, 101, 103],
}
literal_eval("close[0]", context)
literal_eval("close[0] - close[1]", context)
```

Optional live/historical wiring:

```
literal_eval(expr, context, data_feed=feed, data_provider=provider)
```

Script evaluation helper

```
from pynescript.ast.evaluator import NodeLiteralEvaluator

ev = NodeLiteralEvaluator()
result = ev.evaluate_script(
    """
//@version=5
indicator("demo")
// body depends on what the evaluator implements for statements
"""
)
```

Libraries:

```
ev.register_library_source(namespace="MyNs", name="Lib", version=1, source=lib_source)
mod = ev.lookup_library(namespace="MyNs", name="Lib", version=1)
```

Bar-loop Runtime (HTTP contract shape)

```
from backend.runtime import Runtime  # monorepo; not always installed as package path

runtime = Runtime(symbol="AAPL")
ohlcv = [
    {"time": 1_700_000_000, "open": 100, "high": 101, "low": 99, "close": 100.5, "volume": 1_000},
    {"time": 1_700_086_400, "open": 100.5, "high": 102, "low": 100, "close": 101.5, "volume": 1_200
]
result = runtime.run(
    source_code='//@version=5\nindicator("t")\nplot(close)\n',
    ohlcv_data=ohlcv,
    mode="interpret",  # or "compile" for supported subset
)
# result: plots, series, plot_meta, events, drawings, script_id, run_id, count, ...
# or {"error": "..."}
```

Pro API wraps the same runtime — see Pro API usage.

Data providers and feeds

Historical CLI/library:

```
from pynescript.util.data import get_provider

prov = get_provider("mock")
# or yahoo / alphavantage / ccxt with kwargs
bars = prov.fetch("AAPL", "6mo", "1d")
# bars: dict with close/open/... lists
```

Realtime (requires ccxt / pro):

```
# examples/realtime_datafeed.py
from pynescript.util.datafeed import get_datafeed

feed = get_datafeed("ccxtpro", exchange="binance")
# async with feed: async for candle in feed.watch_ohlcv("BTC/USDT", "1m"): ...
```

Educational bar executor

`examples/execute_script.py` implements a **teaching** RSI strategy executor with a custom visitor and pandas history (`examples/historical_data.py` / yfinance). It is not the production `Runtime`, but demonstrates bar iteration patterns.

Internals (repo paths)

Path	Role
`src/pynescript/ast/helper.py`	`literal_eval`
`src/pynescript/ast/evaluator/base.py`	Context, series, visitor base
`src/pynescript/ast/evaluator/__init__.py`	`NodeLiteralEvaluator` composition
`src/pynescript/ast/evaluator/builtins/`	`ta.*`, `strategy.*`, arrays, …
`src/pynescript/ast/evaluator/events.py`	Event emission hooks
`backend/runtime.py`	Multi-bar `Runtime.run`
`backend/series.py`	`PineSeries` history model
`src/pynescript/util/data.py`	Providers + `resolve_request_sources`
`src/pynescript/util/datafeed.py`	Realtime feeds
`examples/evaluate_expressions.py`	Expression gallery
`examples/execute_script.py`	Didactic strategy loop
`examples/rsi_strategy.pine`	Sample strategy source

Invariants & edge cases

- **Determinism.** Same source + same OHLCV + same mode should yield reproducible series (mock providers seedable via options where supported).
- **`na` / warmup.** TA functions need enough bars; early bars may be `na`/None-like — guard like Pine (`not na(x)`).
- **`mode="compile"`** only supports a **subset** (docs in runtime: sma/ema/rsi, plots, math). Unsupported constructs fall back or error — .
- **History indexing.** Confirm `[0]` / `[1]` semantics against your evaluator version before porting TV scripts that assume closed-bar rules.
- **Strategy events** include run/script ids when stamped by Runtime — useful for HOOX downstream.
- **Body size / bar count.** HTTP path caps payload size (5 MiB); very long histories should be downsampled or chunked.

Worked examples

RSI on a synthetic series

```
from pynescript.ast.helper import literal_eval

closes = [float(x) for x in range(100, 130)]
print(literal_eval(f"ta.rsi({closes}, 14)"))
```

Multi-indicator expressions

```
highs = [c + 1 for c in closes]
lows = [c - 1 for c in closes]
macd, signal, hist = literal_eval(f"ta.macd({closes}, 12, 26, 9)")
atr = literal_eval(f"ta.atr({highs}, {lows}, {closes}, 14)")
```

Full script via Runtime (monorepo)

```
from pathlib import Path
from backend.runtime import Runtime

script = Path("examples/rsi_strategy.pine").read_text(encoding="utf-8")
# Build ohlcv list from your provider...
rt = Runtime(symbol="EXAMPLE")
out = rt.run(script, ohlcv_data=ohlcv, mode="interpret")
if "error" in out:
    raise RuntimeError(out["error"])
print(out.get("count"), len(out.get("events", [])), out.get("series", {}).keys())
```

Strategy event inspection

```
for ev in out.get("events", []):
    # shape depends on StrategyState / event schema
    print(ev)
```

request.* with resolved sources

When calling `/run` or Runtime, pass `data_source` / providers so `request.security` and friends resolve; without configuration they may use chart bars or mocks.

Failure modes

Symptom	Interpretation
`NotImplementedError` / incomplete builtin	Expression outside supported surface — check missing features
`{"error": "Parse Error: ..."}`	Runtime caught parse failure
`EXECUTION_ERROR` via HTTP	Exception during bar loop
Empty plots	No `plot` / `plotshape` executed; wrong declaration type
Numerical mismatch vs TV	Warmup, float path, or builtin parity gap — numerical validation
`DataProviderError`	Provider misconfig
Async feed errors	Missing ccxt.pro / network

See also

- Pro API usage
- Library API
- Runtime series model
- Strategy builtins
- Evaluate contract
- AXIS — visualizing series
- HOOX — consuming strategy events on the edge

EDITORS

title: "Editors & LSP" description: "Wire pynescript-lsp into VS Code, Neovim, Zed, Emacs, Helix, and other LSP clients for diagnostics, completion, hover, and formatting."

EDITORS & LSP

Abstract

`pynescript-lsp` is a **stdio Language Server** (pygls) over the same parse/lint/metadata stack as the desk tools. Editors do not reimplement Pine intelligence; they speak LSP. This guide covers install, client configs in `clients/`, and feature expectations. Server architecture is documented under LSP.

Conceptual model

```
sequenceDiagram
  participant Ed as Editor
  participant LSP as pynescript-lsp
  participant AST as parse / linter / metadata
  Ed->>LSP: initialize (stdio)
  Ed->>LSP: textDocument/didOpen
  LSP->>AST: parse + lint + symbols
  LSP-->>Ed: publishDiagnostics
  Ed->>LSP: completion / hover / definition
  LSP->>AST: builtin metadata + AST index
  LSP-->>Ed: LSP responses
```

Feature	Role
Diagnostics	Lint rules + parse errors as squiggles
Completion	Builtins (`ta.*` , `strategy.*` , …) + snippets
Hover	Signature / docs from metadata
Definition / references	Symbol index in document
Document symbols	Outline / breadcrumb
Formatting	Full document + range via unparser path

Builtin metadata is generated (`scripts/generate_builtin_metadata.py`); encrypted blobs may appear in release builds.

Interface surface

Install server

```
pip install "pynescript[lsp]"
# monorepo:
pip install -e ".[lsp]"
# or: make install

command -v pynescript-lsp
python -m pynescript.langserver  # equivalent entry
```

The process speaks **LSP over stdio** by default (`PynescriptLanguageServer.start_io()`). Do not expect a useful interactive TTY UI.

VS Code / Cursor / Antigravity

1. Build or install the extension from `vscode-extension/` :

```
cd vscode-extension
npm ci
npm run compile
# package: npx vsce package  (or make build-vscode)
```

2. Open a `.pine` / `.pinev5` / `.pinev6` file — LSP activates when enabled.

Settings:

Key	Default	Notes
`pynescript.lsp.enabled`	`true`	Master switch
`pynescript.lsp.command`	`pynescript-lsp`	Absolute path if not on PATH
`pynescript.formatting.enabled`	`true`	
`pynescript.diagnostics.enabled`	`true`	
`pynescript.completion.snippets`	`true`	

Marketplace availability may lag git — .

Neovim (nvim-lspconfig)

```
require("lspconfig").pynescript.setup({})
```

Manual config from `clients/neovim.lua`:

```
return {
  cmd = { "pynescript-lsp" },
  filetypes = { "pinescript" },
  root_dir = function(fname)
    return vim.fs.root(fname, { ".git", "*.pine", "*.pinev5", "*.pinev6", "pyproject.toml" })
      or vim.fn.getcwd()
  end,
  settings = {
    pinescript = {
      formatting = { enabled = true },
      diagnostics = { enabled = true },
      completion = { snippets = true },
    },
  },
}
```

Suggested maps: `gd` definition, `gr` references, `K` hover, format via `vim.lsp.buf.format`.

Ensure filetype detection maps `*.pine` → `pinescript` if your distro does not ship it.

Zed

Merge `clients/zed.json` or:

```
{
  "languages": {
    "Pine Script": {
      "language_servers": ["pynescript"]
    }
  },
  "language_servers": {
    "pynescript": {
      "command": "pynescript-lsp",
      "arguments": ["--stdio"]
    }
  }
}
```

Emacs (lsp-mode)

```
(use-package lsp-mode
  :hook ((pinescript-mode . lsp))
  :config
  (lsp-register-client
   (make-lsp-client
    :new-connection (lsp-stdio-connection '("pynescript-lsp" "--stdio"))
    :major-modes '(pinescript-mode)
    :server-id 'pynescript)))
```

Or `(load-file "clients/emacs.el")` from a checkout.

Helix

`~/.config/helix/languages.toml`:

```
[[language]]
name = "pinescript"
scope = "source.pinescript"
file-types = ["pine", "pinev5", "pinev6"]
roots = ["pyproject.toml"]
command = "pynescript-lsp"
args = ["--stdio"]
```

Sublime Text (LSP package)

```
{
  "clients": {
    "pynescript": {
      "command": ["pynescript-lsp", "--stdio"],
      "selector": "source.pinescript",
      "initializationOptions": {}
    }
  }
}
```

Generic

Any LSP client:

```
Command: pynescript-lsp
Args:    (none or --stdio depending on client)
Transport: stdio
```

Internals (repo paths)

Path	Role
`src/pynescript/langserver/__main__.py`	CLI entry `main()`
`src/pynescript/langserver/server.py`	`PynescriptLanguageServer`
`src/pynescript/langserver/features/`	diagnostics, completion, hover, …
`clients/README.md`	Canonical client matrix
`clients/neovim.lua`, `zed.json`, `emacs.el`	Drop-in snippets
`vscode-extension/`	VS Code extension
`scripts/generate_builtin_metadata.py`	Completion/hover corpus

Invariants & edge cases

- **Python 3.10+** required on the machine running the server.
- **Server must be on PATH** or configured with absolute command.
- **One server per workspace** is typical; large monorepos with many `.pine` files still parse on demand.
- **Formatting** uses the unparser pipeline — expect normalization, not clang-format style knobs.
- **Diagnostics** reuse linter codes (`E001`, `W001`, …); severity maps to LSP DiagnosticSeverity.
- **Nuitka binary** (`dist/lsp/pynescript-lsp`) may replace the Python entry in packaged distributions — point the editor at that binary if you ship it.

Worked examples

Verify server starts (stdio smoke)

```
# Editors start this; manual smoke is limited.
python -c "from pynescript.langserver.server import PynescriptLanguageServer; print(PynescriptLangu
```

Neovim minimal `init.lua` fragment

```
vim.api.nvim_create_autocmd("FileType", {
  pattern = "pinescript",
  callback = function()
    vim.lsp.start({
      name = "pynescript",
      cmd = { "pynescript-lsp" },
      root_dir = vim.fn.getcwd(),
    })
  end,
})
```

Force a specific venv binary in VS Code

```
{
  "pynescript.lsp.command": "/home/you/project/.venv/bin/pynescript-lsp"
}
```

Failure modes

Symptom	Fix
Client: executable not found	Install `[lsp]`; fix PATH / command setting
`No module named pygls`	Reinstall extra in the same env as the command
No diagnostics	Enable diagnostics; confirm filetype; check server logs
Stale completions	Regenerate metadata in dev builds; restart server
Format does nothing	Enable formatting; ensure document is valid enough to parse
High CPU on huge files	Split libraries; wait for parse; check for pathological nesting

See also

- Installation
- LSP architecture
- VS Code extension
- LSP clients (systems)
- Troubleshooting

PRO API USAGE

title: "Pro API Usage" description: "Call the PYNE Flask Pro API as a consumer: health, /run, /run/batch, chart preview, indicator preview, quick backtest, and auth."

PRO API USAGE

Abstract

The Pro API is the **HTTP evaluate contract** for PYNE: POST Pine source plus OHLCV, receive plots, series, events, drawings, and **alerts**. Free-tier evaluate endpoints (`/run` , `/run/batch`) are usable without a key; optional **L2 alert webhooks** POST last-bar firings to `webhook_url` or `ALERT_WEBHOOK_URL` . Pro preview/backtest routes track usage via API keys. This page is for **callers** (AXIS, scripts, workers). Server lifecycle and auth design are under API.

Conceptual model

```
flowchart LR
  C[Client] -->|POST JSON| F[Flask app]
  F --> V[schema validate]
  V --> R[Runtime.run]
  R --> J[JSON plots/series/events]
  F --> PV[preview / backtest services]
  PV --> PNG[base64 charts / metrics]
  AK[API key middleware] --> PV
```

Endpoint	Auth	Purpose
`GET /`	none	Health + endpoint map
`POST /run`	none (free)	Single script evaluate
`POST /run/batch`	none (free)	≤8 scripts, shared OHLCV
`POST /preview/chart`	Pro usage	Chart thumbnail
`POST /preview/indicator`	Pro usage	Indicator-style preview
`POST /backtest/quick`	Pro usage	Equity / summary
`POST /auth/create_key`	admin token	Mint key
`GET /auth/usage`	Pro	Usage stats
`POST /auth/validate`	body key	Validate key

Default local bind: `127.0.0.1:5002`.

Interface surface

Run the server (local)

```
pip install -r backend/requirements.txt

python -m backend.app
# or: make run
```

Health

```
curl -s http://127.0.0.1:5002/ | python -m json.tool
```

`POST /run`

Schema (`RUN_SCHEMA`):

Field	Type	Required	Default
`script`	string	yes	
`data`	list (OHLCV bars)	yes	
`symbol`	string	no	`"CHART"`
`data_source`	string	no	`""`
`data_options`	object	no	`{}`
`mode`	string	no	`"auto"`
`inputs`	object	no	`{}`
`webhook_url`	string	no	`""` (else env `ALERT_WEBHOOK_URL`)
`forward_alerts`	bool	no	`true`
`alert_last_bar`	bool	no	`true`
`alert_batch`	bool	no	`true`

Bar objects are expected to carry at least open/high/low/close/time (volume optional depending on script).

```
curl -s -X POST http://127.0.0.1:5002/run \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=5\nindicator(\"Demo\")\nplot(ta.sma(close, 3))",
    "symbol": "AAPL",
    "mode": "interpret",
    "data": [
      {"time": 1, "open": 10, "high": 11, "low": 9, "close": 10.5, "volume": 1000},
      {"time": 2, "open": 10.5, "high": 12, "low": 10, "close": 11.5, "volume": 1100},
      {"time": 3, "open": 11.5, "high": 12.5, "low": 11, "close": 12, "volume": 1200}
    ]
  }'
```

Success shape (conceptual):

```
{
  "status": "success",
  "plots": [],
  "series": {},
  "plot_meta": {},
  "events": [],
  "drawings": [],
  "alerts": [],
  "script_id": "...",
  "run_id": "...",
  "count": 3,
  "mode": "interpret",
  "data_source": "chart"
}
```

When a webhook URL is configured, the response may also include `alert_forward` (delivery counts). See Alerts.

Error codes include `NO_SCRIPT`, `NO_DATA`, `DATA_SOURCE_ERROR`, `EXECUTION_ERROR`, plus schema codes `INVALID_BODY`, `MISSING_FIELD`, `INVALID_FIELD`, `UNKNOWN_FIELDS`.

`POST /run/batch`

Shared OHLCV, multiple scripts (max **8**). Envelope:

```
{
  "scripts": [
    {"id": "sma", "script": "//@version=5\nindicator(\"s\")\nplot(ta.sma(close, 5))"},
    {"id": "rsi", "script": "//@version=5\nindicator(\"r\")\nplot(ta.rsi(close, 14))"}
  ],
  "data": [/* bars */],
  "symbol": "AAPL",
  "mode": "interpret"
}
```

String entries in `scripts` are accepted and auto-id'd as `script_0`, ... Per-script errors do not necessarily fail the whole HTTP status (route returns 200 with per-item status — verify against deployment).

`POST /preview/chart` (Pro)

```
{
  "script": "",
  "data": {"close": [100, 101, 102], "volume": [1, 2, 3]},
  "options": {
    "type": "line",
    "color": "#2196F3",
    "width": 600,
    "height": 300,
    "show_volume": false
  }
}
```

Response includes base64 PNG `chart` + `meta`. Width/height clamped (e.g. max 1200×600).

`POST /preview/indicator` (Pro)

```
{
  "expression": "ta.sma(close, 20)",
  "data": {"close": [/* ... */]},
  "options": {}
}
```

`POST /backtest/quick` (Pro)

```
{
  "script": "//@version=5\nstrategy(\"x\")\n...",
  "data": {},
  "initial_capital": 10000.0,
  "mock_data": true,
  "mock_bars": 252
}
```

Auth helpers

```
# Mint key (admin)
curl -s -X POST http://127.0.0.1:5002/auth/create_key \
  -H "Content-Type: application/json" \
  -H "X-Admin-Token: $ADMIN_TOKEN" \
  -d '{"tier":"hobby"}'

# Validate
curl -s -X POST http://127.0.0.1:5002/auth/validate \
  -H "Content-Type: application/json" \
  -d '{"api_key":"pyn_..."}'
```

Send Pro requests with the key as required by middleware (typically `Authorization` bearer or documented header —).

Hosted SDK (if published)

README sketches:

```
from pynescript.api import PynescriptAPI  #

api = PynescriptAPI(api_key="pyn_...")
chart = api.preview.chart(data={"close": [...]}, options={"type": "line"})
result = api.backtest.quick(script="...", mock_data=True, mock_bars=252)
```

If the client module is not present in your install, use raw HTTP as above.

Internals (repo paths)

Path	Role
`backend/app.py`	Routes `/`, `/run`, `/run/batch`, auth
`backend/api/preview.py`	`/preview/*`, `/backtest/quick`
`backend/middleware/schemas.py`	Strict request schemas
`backend/middleware/auth.py`	Keys, usage, admin token
`backend/runtime.py`	Evaluate bridge
`backend/services/chart_renderer.py`	PNG rendering
`backend/services/backtest.py`	Quick backtest + mock OHLCV

Invariants & edge cases

- **Max body 5 MiB** — oversized JSON → Flask 413.
- **Strict schemas** — unknown fields rejected.
- **CORS** — browser clients need listed origins; server-to-server without `Origin` is fine.
- **`mode=compile`** — subset only; prefer `interpret` unless you know the script is in the compiled surface.
- **Batch cap 8** — `TOO_MANY_SCRIPTS` if exceeded.
- **Free `/run` is still CPU-expensive** — rate-limit at your reverse proxy in production.

- AXIS multi-indicator UIs prefer `/run/batch` to share bar payloads.

Worked examples

Python `requests` client for `/run`

```
SCRIPT = """
//@version=5
indicator("SMA", overlay=true)
plot(ta.sma(close, 5))
"""

bars = [
    {"time": i, "open": 100 + i, "high": 101 + i, "low": 99 + i, "close": 100.5 + i, "volume": 1000
    for i in range(30)
]

r = requests.post(
    "http://127.0.0.1:5002/run",
    json={"script": SCRIPT, "data": bars, "symbol": "DEMO", "mode": "interpret"},
    timeout=60,
)
r.raise_for_status()
payload = r.json()
assert payload["status"] == "success", payload
print(payload["count"], list(payload.get("series", {})))
```

Batch two indicators

```
r = requests.post(
    "http://127.0.0.1:5002/run/batch",
    json={
        "scripts": [
            {"id": "sma", "script": "//@version=5\nindicator(\"s\")\nplot(ta.sma(close, 5))"},
            {"id": "ema", "script": "//@version=5\nindicator(\"e\")\nplot(ta.ema(close, 5))"},
        ],
        "data": bars,
    },
    timeout=120,
)
print(r.json())
```

Wire optional CCXT data source

```
{
  "script": "...",
  "data": [/* chart bars */],
  "symbol": "BTC/USDT",
  "data_source": "ccxt",
  "data_options": {"exchange": "binance"}
}
```

Requires server environment with `ccxt` installed.

Failure modes

Code / HTTP	Meaning	Action
400 `MISSING_FIELD`	No `script` / `data`	Fix body
400 `UNKNOWN_FIELDS`	Extra keys	Strip undocumented fields
400 `DATA_SOURCE_ERROR`	Provider config	Fix `data_options` / deps
500 `EXECUTION_ERROR`	Runtime exception	Simplify script; check message
403 on create_key	`ADMIN_TOKEN` unset/wrong	Export token
413	Body too large	Fewer bars / compress strategy
CORS browser error	Origin not allowed	Update `ALLOWED_ORIGINS`
Empty series	Script never plotted	Add `plot` / check declaration

See also

- Evaluate scripts
- Configuration
- API contract
- Auth and keys
- AXIS — AXIS over `/run` results
- HOOX — edge mesh after strategy events

TROUBLESHOOTING

title: "Troubleshooting" description: "Diagnose pynescript install, parse, lint, evaluation, LSP, data providers, and Pro API failures with concrete checks."

TROUBLESHOOTING

Abstract

Failures cluster by layer: **environment**, **parse**, **lint**, **evaluation**, **data I/O**, **LSP**, **HTTP**. Work top-down: confirm the binary/module, then a minimal script, then full strategy complexity. This page is a runbook; deep protocol and runtime docs are linked per section.

Conceptual model

```
flowchart TD
  S[Symptom] --> E{Import / PATH?}
  E -->|no| I[Fix install / venv]
  E -->|yes| P{Parses?}
  P -->|no| G[Grammar / version / encoding]
  P -->|yes| L{Lint gate?}
  L --> V{Evaluates?}
  V -->|no| R[Builtin / series / Runtime error]
  V -->|yes| X{LSP or HTTP?}
  X --> D[Client config / CORS / schema]
```

Interface surface (diagnostic commands)

```
python -c "import pynescript; from pynescript import __about__; print(__about__.__version__)"
python -c "from pynescript.ast import parse, unparse; print(unparse(parse('//@version=5\nindicator(
pynescript --version
pynescript lint - <<< $'//@version=5\nindicator("t")\nplot(close)\n'
python -c "import pygls"  # LSP extra
python -c "import ccxt"   # data extra
curl -s http://127.0.0.1:5002/ | head
```

Internals (where errors originate)

Layer	Path
CLI	`src/pynescript/__main__.py`
Parse	`src/pynescript/ast/helper.py`, ANTLR error listener
Lint	`src/pynescript/ast/linter.py`
Eval	`src/pynescript/ast/evaluator/`, `backend/runtime.py`
Data	`src/pynescript/util/data.py`
LSP	`src/pynescript/langserver/`
HTTP	`backend/app.py`, `backend/middleware/schemas.py`

Invariants & edge cases

- Minimal green script for isolation:

```
//@version=5
indicator("t")
plot(close)
```

- Encoding must be UTF-8 unless you pass `--encoding`.
- Pine v4 and earlier trigger linter deprecation (`W002`); parser support may still differ by construct.
- Recursion limit is raised only during parse; extreme generated scripts can still fail.

Worked examples (by symptom)

1. `pynescript: command not found`

```
which python
python -m pynescript --help
# If module works but script does not:
python -m pip show pynescript
ls "$(python -c 'import sysconfig; print(sysconfig.get_path("scripts"))')"
```

Fix: activate the venv used for install; reinstall `pip install pynescript`.

2. `ModuleNotFoundError: pynescript`

Wrong interpreter (IDE using system Python). Point the IDE at the venv; reinstall editable from repo root: `pip install -e .`.

3. Parse / SyntaxError

```
pynescript parse-and-dump broken.pine
```

Checks:

- `//@version=5` or `6` present
- Balanced parentheses / indentation (Pine uses line structure)
- Multiline strings / v6 features need a package version that includes them
- Binary/null bytes in file

Reduce to a minimal repro; if TV accepts it and PYNE does not, consult missing features and file a grammar issue.

4. Round-trip looks “wrong”

`unparse` normalizes formatting. Compare dumps:

```
pynescript parse-and-dump a.pine > /tmp/a.ast
pynescript parse-and-unparse a.pine | pynescript parse-and-dump /dev/stdin
# stdin may not work for parse-and-dump (file path required) — use a temp file:
pynescript parse-and-unparse a.pine --output-file /tmp/a2.pine
pynescript parse-and-dump /tmp/a2.pine > /tmp/a2.ast
diff -u /tmp/a.ast /tmp/a2.ast
```

Structural diff empty ⇒ cosmetic only.

5. Lint noise / CI red

Code	Action
`E001`	Fix syntax first
`W001`	Add `//@version=5`
`W002`	Upgrade version pragma
`C001` – `C004`	Style; use `--fail-on errors` if style should not gate CI

```
pynescript lint script.pine --fail-on errors
```

6. `literal_eval` fails

- Use `mode="eval"`-compatible **expressions**, not full `strategy()` scripts
- Pass series as Python lists inside the expression or via `context`
- Missing builtin → incomplete implementation

Escalate to `Runtime.run` for multi-bar scripts.

7. Runtime / `/run` execution error

```
curl -s -X POST http://127.0.0.1:5002/run -H 'Content-Type: application/json' -d '{"script":"//@ver
```

If minimal works but strategy fails: remove `request.*`, drawings, then strategy entries until isolated. Check `mode`; try `"interpret"`.

8. Schema validation errors

`UNKNOWN_FIELDS` means you sent a key not in the schema — remove it. Do not rely on servers ignoring extras.

9. CORS from AXIS / browser

```

```

Restart Flask after changing env.

10. LSP will not start

```
pip install "pynescript[lsp]"
python -c "import pygls, lsprotocol"
command -v pynescript-lsp
# Run with editor logging enabled; check stderr for import traces
```

Ensure the editor's `command` matches the venv. For Neovim, confirm `filetypes` include your buffer's filetype.

11. Data provider failures

Provider	Checklist
`mock`	Should always work offline
`yahoo`	Network; symbol format
`alphavantage`	API key; demo limits
`ccxt`	`pip install "pynescript[data]"`; exchange id; symbol `BTC/USDT`

```
pynescript data AAPL --provider mock
```

12. Version / parity confusion

PYNE is independent of TradingView®. Behavioral gaps are expected for edge builtins, broker emulators, and UI-only calls. Use compatibility and implementation status.

Failure modes (quick matrix)

Symptom	Layer	First check
Command not found	env	`python -m pynescript`
Import error pygls	extra	`[lsp]`
Import error ccxt	extra	`[data]`
SyntaxError	parse	minimal script + version
Lint CI fail	lint	`--fail-on` level
Empty plots	eval	`plot()` present; enough bars
400 UNKNOWN_FIELDS	HTTP	schema
403 create_key	HTTP	`ADMIN_TOKEN`
413	HTTP	body size
Squiggles missing	LSP	diagnostics enabled + filetype
Numerical drift	eval	warmup / parity docs

See also

- Installation
- Configuration
- CLI guide
- Evaluate scripts
- Pro API usage
- Editors
- FAQ

CLI COMMANDS

#!/usr/bin/env bash

COPYRIGHT (C) 2024-2026 JANGO_BLOCKCHAINED
THIS FILE IS PART OF PYNESCRIPT.
PYNESCRIPT IS FREE SOFTWARE: YOU CAN REDISTRIBUTE IT AND/OR MODIFY
IT UNDER THE TERMS OF THE GNU AFFERO GENERAL PUBLIC LICENSE AS PUBLISHED BY
THE FREE SOFTWARE FOUNDATION, EITHER VERSION 3 OF THE LICENSE, OR
(AT YOUR OPTION) ANY LATER VERSION.
PYNESCRIPT IS DISTRIBUTED IN THE HOPE THAT IT WILL BE USEFUL,
BUT WITHOUT ANY WARRANTY; WITHOUT EVEN THE IMPLIED WARRANTY OF
MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. SEE THE
GNU AFFERO GENERAL PUBLIC LICENSE FOR MORE DETAILS.
YOU SHOULD HAVE RECEIVED A COPY OF THE GNU AFFERO GENERAL PUBLIC LICENSE
ALONG WITH PYNESCRIPT. IF NOT, SEE HTTPS://WWW.GNU.ORG/LICENSES/.
SPDX-LICENSE-IDENTIFIER: AGPL-3.0-OR-LATER

---

title: "CLI Commands Reference" description: "Complete option reference for the pynescript Click CLI: parse-and-dump, parse-and-unparse, lint, data, and download-builtin-scripts."

CLI COMMANDS REFERENCE

Abstract

Machine-oriented reference for every command registered on the `pynescript` Click group in `src/pynescript/__main__.py`. For workflows and pipelines, see the CLI guide. The language server is a **separate** console script (`pynescript-lsp`) and is documented under Editors.

Conceptual model

```
pynescript [--version] COMMAND [ARGS]...
```

Command	Short help (Click)
`parse-and-dump`	Parse pinescript file to AST tree.
`parse-and-unparse`	Parse pinescript file and unparse back to pinescript.
`download-builtin-scripts`	Download builtin scripts.
`lint`	Lint Pine Script file for issues.
`data`	Fetch market data from providers.

Global:

```
pynescript --help
pynescript --version
```

Version source: package version option on the Click group (`@click.version_option()`), aligned with `pynescript.__about__.__version__` .

Interface surface

`pynescript parse-and-dump`

Parse a file and print an AST dump.

Kind	Name	Type / default	Description
argument	`PATH`	existing readable file	Input `.pine` path
option	`--encoding`	str, default `utf-8`	File text encoding
option	`--indent`	int, default `2`	Indent width for dump
option	`--output-file`	path or `-` , default `-`	Output path; `-` = stdout

Semantics: `parse(text, filename)` then `dump(node, indent=indent)` ; write with same encoding.

Examples:

```
pynescript parse-and-dump strategy.pine
pynescript parse-and-dump strategy.pine --indent 4 --output-file tree.txt
pynescript parse-and-dump strategy.pine --encoding utf-8 --output-file -
```

Exit codes: Click/path errors non-zero; uncaught parse exceptions propagate (non-zero).

`pynescript parse-and-unparse`

Parse a file and emit regenerated source.

Kind	Name	Type / default	Description
argument	`PATH`	existing readable file	Input path
option	`--encoding`	str, default `utf-8`	Text encoding
option	`--output-file`	path or `-`, default `-`	Output path; `-` = stdout

Semantics: `parse` → `unparse` → write.

Examples:

```
pynescript parse-and-unparse messy.pine
pynescript parse-and-unparse messy.pine --output-file clean.pine
```

`pynescript lint`

Lint a file or stdin.

Kind	Name	Type / default	Description
argument	`PATH`	optional string	File path; omit or `-` for stdin
option	`--encoding`	str, default `utf-8`	File encoding (ignored for stdin)
option	`--fix`	flag	Attempt to fix issues where possible
option	`--fail-on`	choice `errors` | `warnings` | `all`, default `errors`	Non-zero exit threshold

Semantics:

1. Read source (stdin if `PATH` is missing or `-`).
2. `warnings = lint_script(source, filename)`.
3. If empty → print `No issues found.` and return.
4. Print each issue with emoji prefix by severity; print summary count.
5. Raise `click.ClickException` when threshold met:

`--fail-on`	Fails when
`errors`	any `severity == "error"`
`warnings`	any error or warning
`all`	any issue

Examples:

```
pynescript lint strategy.pine
pynescript lint --fail-on warnings strategy.pine
pynescript lint --fail-on all strategy.pine
pynescript lint -
cat strategy.pine | pynescript lint
pynescript lint --encoding utf-8 strategy.pine
```

Issue line format (via `LintWarning.__str__`):

```
{severity}: [{code}] {message} at line {n}|unknown location
```

CLI prefixes ❌ (error) or ⚠️ (other).

`pynescript data`

Fetch market data for a symbol and print a short summary.

Kind	Name	Type / default	Description
argument	`SYMBOL`	string	Ticker / market symbol
option	`--provider`	`mock` | `yahoo` | `alphavantage` | `ccxt`, default `mock`	Backend
option	`--period`	str, default `1y`	e.g. `1d`, `1w`, `1mo`, `3mo`, `6mo`, `1y`, `2y`, `5y`
option	`--interval`	str, default `1d`	e.g. `1m`, `5m`, `15m`, `30m`, `60m`, `1d`, `1w`
option	`--api-key`	str, default `""`	Alpha Vantage or CCXT
option	`--secret`	str, default `""`	CCXT secret
option	`--exchange`	str, default `binance`	CCXT exchange id

Semantics:

1. Build provider kwargs (AV demo key warning if missing; CCXT exchange/key/secret).
2. `get_provider(provider, **kwargs).fetch(symbol, period, interval)`.
3. Print symbol, bar count, first/last close, average volume.

Examples:

```
pynescript data AAPL
pynescript data AAPL --provider=yahoo --period=6mo
pynescript data EUR/USD --provider=alphavantage --api-key=YOUR_KEY
pynescript data BTC/USDT --provider=ccxt --exchange=binance
```

Errors: `DataProviderError` → `ClickException` with message.

Deps: `ccxt` providers require `pip install "pynescript[data]"`.

`pynescript download-builtin-scripts`

Download TradingView® reference scripts used for tests/corpus.

Kind	Name	Type / default	Description
option	`--script-dir`	writable directory path	**Required.** Destination (e.g. `tests/data/builtin_scripts`)

Semantics: `pynescript.util.pine_facade.download_builtin_scripts(script_dir)`.

Examples:

```
pynescript download-builtin-scripts --script-dir tests/data/builtin_scripts
pynescript download-builtin-scripts --script-dir /tmp/pine_corpus
```

Notes: Network-dependent; not required for ordinary desk use.

Related: `pynescript-lsp` (not a subcommand)

```
pynescript-lsp
python -m pynescript.langserver
```

Entry: `pynescript.langserver.__main__:main` → `PynescriptLanguageServer().start_io()`.

Internals (repo paths)

Item	Path
CLI group	`src/pynescript/__main__.py`
Script entry	`pyproject.toml` → `[project.scripts] pynescript = ...`
parse/dump/unparse	`src/pynescript/ast/helper.py`
lint	`src/pynescript/ast/linter.py`
data	`src/pynescript/util/data.py`
download	`src/pynescript/util/pine_facade.py`

Invariants & edge cases

- File arguments for parse commands use `click.Path(exists=True, file_okay=True, dir_okay=False)` — **no directory recursion**.
- `lint` deliberately accepts a free string path so stdin modes work; invalid paths surface as open errors.
- Dump indent is passed as integer spaces to `dump(..., indent=indent)`.
- Output files are opened with Click `open_file` (creates/truncates as applicable).
- There is currently **no** `pynescript evaluate` / `pynescript run` CLI subcommand — evaluation is library or HTTP.

Worked examples

CI lint gate

```
set -euo pipefail
for f in "$@"; do
  pynescript lint --fail-on errors "$f"
done
```

Normalize and re-dump for structural snapshot tests

```
pynescript parse-and-unparse in.pine --output-file /tmp/norm.pine
pynescript parse-and-dump /tmp/norm.pine --indent 2 --output-file /tmp/norm.ast
```

Offline data smoke

```
pynescript data TEST --provider mock --period 1mo --interval 1d
```

Failure modes

Situation	Result
Missing input file	Click path validation error
Parse error in dump/unparse	Python traceback / non-zero
Lint threshold exceeded	`Error: Lint failed with ...`
Provider failure	`Error: <DataProviderError message>`
Missing `--script-dir`	Click missing option error
Unknown provider	Click choice validation error

See also

- CLI guide
- Library API
- Configuration
- FAQ

GLOSSARY

title: "Glossary" description: "Definitions of PYNE end-user terms: AST, ASDL, bar-loop, series, Runtime, LSP, Pro API, extras, and related concepts."

GLOSSARY

Abstract

Shared vocabulary for the End User track. Terms are defined as **used in this repo**, not as MarketingView marketing copy. Cross-links point at deeper systems manuals where useful.

Conceptual model

Terms fall into four buckets:

1. **Language representation** — grammar, AST, unparser
2. **Execution** — series, bar-loop, builtins, strategy events
3. **Surfaces** — CLI, library, LSP, Pro API
4. **Ecosystem** — AXIS, HOOX, pine-worker, pyne-worker

```
mindmap
  root((PYNE terms))
    Representation
      Grammar
      ASDL
      AST
      Unparse
    Execution
      Series
      Bar-loop
      Builtin
      Runtime
    Surfaces
      CLI
      LSP
      Pro API
    Ecosystem
      AXIS
      HOOX
```

Interface surface

This page is pure reference — no commands required.

Internals (repo paths)

Term cluster	Primary paths
Parse / AST	`src/pynescript/ast/helper.py`, `builder.py`, `grammar/`
Evaluate	`src/pynescript/ast/evaluator/`, `backend/runtime.py`
Lint	`src/pynescript/ast/linter.py`
LSP	`src/pynescript/langserver/`
HTTP	`backend/app.py`

Invariants & edge cases

- **Pine Script™ / TradingView®** are trademarks of TradingView, Inc. PYNE is independent (see product index).
- “Parity” is empirical (tests/corpus), not a legal claim of equivalence.

Terms

ASDL

Abstract Syntax Description Language. Schema language used to generate Python AST node classes (`Pinescript.asdl` → generated node module). Gives a single algebraic description of script structure.

Annotation

Special comments such as `//@version=5` attached to script or declaration nodes during parse (`_add_annotations` in `helper.py`).

AST

Abstract Syntax Tree. In-memory tree of ASDL-generated nodes produced by `parse`. Root is typically a `Script` (`mode="exec"`) or `Expression` (`mode="eval"`).

AXIS

Optional **charting PWA "AXIS"** product (`frontend/`, docs at `/axis/docs`). Consumes evaluate results (series/plots); not required for evaluation.

Bar / bar-loop

A discrete OHLCV time step. A **bar-loop** evaluator re-executes script logic once per bar with updated series history (`Runtime.run`).

Builtin

Host-provided namespace function or value (`ta.rsi`, `strategy.entry`, `math.sqrt`, ...). Implemented under `ast/evaluator/builtins/` and catalogued for LSP metadata.

CLI

The `pynescript` Click command group (`parse-and-dump`, `parse-and-unparse`, `lint`, `data`, `download-builtin-scripts`).

Compile mode

`mode="compile"` on Runtime / `/run`: attempts a **Numba (or similar) compiled subset** of bar logic for speed. Not full language coverage.

Data provider

Historical OHLCV source (`mock`, `yahoo`, `alphavantage`, `ccxt`) via `pynescript.util.data`.

Data feed

Realtime stream abstraction (`datafeed`, e.g. CCXT Pro) for live candles/trades.

Dump

`dump(node)` — pretty textual representation of an AST for debugging and tests (not executable Pine).

Evaluate / evaluator

Execution of AST nodes to values. Ranges from `literal_eval` (expression-safe) to full statement/builtin evaluators and HTTP `Runtime`.

Extra (package extra)

Optional dependency set in `pyproject.toml`: `lsp`, `dev-lsp`, `data`, `datafeed`.

HOOX

Edge **trade execution mesh** (separate monorepo / docs under `/docs`). Can consume strategy events from evaluate pipelines; optional.

Interpret mode

Default Runtime mode: AST walking per bar without the compiled subset path.

`literal_eval`

Helper that parses an expression (or accepts an AST) and evaluates literals plus supported builtins/series context — analogous in spirit to Python's `ast.literal_eval` but Pine-aware and broader.

Lint / `LintWarning`

Static diagnostics with `code`, `message`, `line`, `severity` (`error` | `warning`). Entry: `lint_script`.

LSP

Language Server Protocol. `pynescript-lsp` process implementing diagnostics, completion, hover, navigation, formatting.

NodeVisitor / NodeTransformer

Patterns for traversing or rewriting ASTs (`visitor.py`, `transformer.py`).

OHLCV

Open, high, low, close, volume — bar fields passed to Runtime / `/run` (plus `time`).

pine-worker

TypeScript port / worker tool colocated under `pine-worker/` for parity experiments and edge TS evaluate paths.

pyne-worker

Python Cloudflare Worker (sister repo) embedding pynescript for edge `/run`.

Plot / series / plot_meta

Evaluate outputs: plotted values, named multi-series maps, and metadata for AXIS UIs (AXIS).

Pro API

Flask application in `backend/` exposing `/run`, preview, backtest, and auth endpoints.

PYNE

Product name for this evaluation stack (docs under `/pyne/docs`); package name on PyPI remains **`pynescript`**.

Round-trip

`parse` → `unparse` (optionally → `parse` again). Used to normalize formatting and test fidelity.

Runtime

`backend.runtime.Runtime` — multi-bar execution engine used by Pro API.

Series

Time-indexed value stream with history access (`close[1]`). Implementation includes `PineSeries` and evaluator series objects.

Strategy event

Structured record of entries/exits/orders emitted during strategy evaluation for downstream systems (HOOX, analytics).

Unparse

`unparse(node)` — regenerate Pine source text from an AST.

`//@version`

Version pragma annotation; linter expects v5+ for modern scripts.

Warmup

Bars required before a TA function has enough history; early outputs may be `na`.

Worked examples

Using terms together:

```
Source --parse--> AST --unparse--> normalized source
AST + OHLCV --Runtime bar-loop--> series + strategy events
AST --lint_script--> LintWarnings
AST --LSP--> editor diagnostics/completion
series --AXIS--> chart
events --HOOX--> exchange orders
```

Failure modes

Misusing terms often causes support confusion:

Confused pair	Distinction
dump vs unparse	dump is debug AST text; unparse is Pine source
literal_eval vs Runtime	single-shot expressions vs multi-bar scripts
pynescript vs pynescript-lsp	desk CLI vs language server
AXIS vs PYNE	AXIS vs evaluate engine
data provider vs data feed	historical fetch vs realtime stream

See also

- End User hub
- FAQ
- PYNE product map
- AXIS docs
- HOOX docs

FAQ

title: "FAQ" description: "Frequently asked questions about pynescript install, Pine version support, TradingView parity, CLI vs library, LSP, Pro API, licensing, and ecosystem boundaries."

FAQ

Abstract

Short answers to questions that recur when adopting PYNE as a **consumer**. Where behavior is evolving, answers point at status docs rather than overclaiming parity.

Conceptual model

Questions group into: install & package shape, language coverage, evaluation fidelity, tooling surfaces, and product boundaries (AXIS / HOOX / workers).

Interface surface

No dedicated FAQ API — pointers only.

Internals (repo paths)

Answers cite:

- `pyproject.toml` — version, extras, license
- `src/pynescript/__main__.py` — CLI reality
- `docs/missing_features.md` / reference status pages — coverage
- `backend/app.py` — HTTP contract

Invariants & edge cases

- Prefer reading code + status docs over README marketing tables when they disagree; mark drift with issues.
- Trademark disclaimer lives on the product index once — not repeated here in full.

Questions

What is PYNE vs pynescript?

pynescript is the Python package / repo name. **PYNE** is the product documentation brand for the open evaluate stack (grammar → AST → runtime → LSP → Pro API → workers). You still `pip install pynescript`.

What Python versions are supported?

`requires-python = ">=3.10"`. Classifiers list 3.10–3.12. Newer CPython may work; CI matrices evolve — .

How do I install just the parser?

```
pip install pynescript
```

No extra required for `parse` / `unparse` / `lint` / basic `literal_eval`.

How do I install the LSP?

```
pip install "pynescript[lsp]"
```

Run `pynescript-lsp` (not a Click subcommand of `pynescript` in current code).

Why is `pynescript lsp` missing?

The registered entry point is the separate script **`pynescript-lsp`**. Some docs historically mentioned a `lsp` subcommand; trust `pyproject.toml` `[project.scripts]` and `__main__.py` for ground truth.

Which Pine versions are supported?

v5 is the primary surface; v6 features land incrementally (e.g. multiline strings). See implementation status, missing features, and pine v6 surface.

Is this affiliated with TradingView?

No. Independent open-source toolchain. Pine Script™ and TradingView® are trademarks of TradingView, Inc.

Will my TradingView strategy produce identical PnL?

Not guaranteed. Broker emulator rules, fill models, request.* data, and some builtins differ. Use PYNE for offline analysis, CI, and self-hosted evaluate; validate critical strategies numerically (numerical validation).

Can I format Pine like Black?

`parse-and-unparse` / LSP format normalize via the AST unparser. There is no rich style config comparable to Black's profile system.

Does the CLI evaluate strategies?

No evaluate subcommand today. Use:

- `literal_eval` for expressions
- `backend.runtime.Runtime` or `POST /run` for bar-loops
- `examples/execute_script.py` as a didactic sample

How do I lint in CI?

```
pynescript lint --fail-on errors path/to/script.pine
```

Loop files yourself; the CLI is single-file / stdin.

What lint rules exist?

Syntax (`E001`), version (`W001` – `W002`), deprecated patterns (`W101` –), style/naming (`C001` – `C004`). See `src/pynescript/ast/linter.py` and the library API guide.

How do I get market data?

```
pynescript data AAPL --provider yahoo
pip install "pynescript[data]"  # for ccxt
```

Or providers in Python via `pynescript.util.data.get_provider` .

What is the Pro API free tier?

`POST /run` and `POST /run/batch` are exposed as free evaluate endpoints (still your CPU if self-hosted). Preview/backtest routes are Pro-tier with API keys. Hosted pricing in README is product-side — .

How do I self-host the API?

Install backend requirements, set `API_KEY_STORE` / `ALLOWED_ORIGINS` / optional `ADMIN_TOKEN` , run `python -m backend.app` or `make run` . See configuration and Pro API usage.

What is AXIS?

Optional charting PWA that can call the evaluate contract and render series. Docs: AXIS. PYNE evaluates without AXIS.

What is HOOX?

Optional edge trading mesh (trade-worker, etc.). Docs: HOOX. Strategy events from PYNE can feed HOOX; not required for parsing/eval.

Alerts: `alert()` / `alertcondition()` firings export on Pro API and pyne-worker `/run`. Optional HTTP webhooks via `webhook_url` or `ALERT_WEBHOOK_URL` (last-bar by default). See Alerts.

What is pine-worker vs pyne-worker?

- **pine-worker** — TypeScript port / tool in this monorepo (`pine-worker/`)
- **pyne-worker** — Python Cloudflare Worker sister repo depending on pynescript

Both aim at edge evaluate; different runtimes.

What license is pynescript?

AGPL-3.0-or-later per `pyproject.toml`. Network use of modified versions requires offering corresponding source (AGPL §13). Proprietary embedding usually needs a commercial license; consult your counsel.

Can I use this commercially?

AGPL permits commercial use under its terms, but distribution or network service of modified versions requires source availability. Hosted Pro API terms (if any) are separate from the library license.

Why is parse slow on huge files?

ANTLR parse + AST build scale with file size and nesting. Deep expression nests also stress recursion (limit temporarily raised to ≥5000). Split libraries when possible.

Round-trip changed my spacing — is that a bug?

Usually no. Unparser emits a normalized style. File a bug if **semantics** change (identifiers, call structure, annotations lost).

Where is the builtin list for autocomplete?

Generated metadata used by the LSP (`scripts/generate_builtin_metadata.py`). Do not hand-edit encrypted release blobs.

How do I report a grammar bug?

Minimal `.pine` repro + expected TV behavior + PYNE version. Prefer failing pytest if contributing.

Is Jupyter supported?

README mentions magic commands — . You can always `parse` / `literal_eval` in notebooks with the library.

Can I transform scripts programmatically?

Yes — `NodeVisitor` / `NodeTransformer` on the AST, then `unparse`. See library API.

Worked examples

“Is my install healthy?”

```
python -c "from pynescript.ast import parse, unparse; print(unparse(parse('//@version=5\nindicator(
pynescript lint - <<< $'//@version=5\nindicator("t")\nplot(close)\n'
```

“Does HTTP evaluate work?”

```
curl -s http://127.0.0.1:5002/ | python -m json.tool
```

“Is LSP importable?”

```
python -c "import pygls; from pynescript.langserver.server import PynescriptLanguageServer"
```

Failure modes

If an FAQ answer appears wrong against your checkout:

1. Check package version (`pynescript --version`).
2. Read the cited source file.
3. Prefer `__main__.py` / `pyproject.toml` over secondary docs.
4. Open an issue or PR against the MDX page.

See also

- End User hub
- Installation
- Troubleshooting
- Glossary
- Compatibility
- CLI commands