HOOX · PYNE · DOCUMENTATION

RUNTIME ENGINEERS · STRATEGY AUTHORS

RUNTIME MANUAL

Evaluator, series model, builtins, strategy, compiler

Open-source Pine Script evaluation framework

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

CONTENTS

01 Overview

02 Series And History

03 Expressions Statements

04 Overview

05 Technical

06 Strategy

07 Collections

08 Drawing Plotting

09 Request Input

10 Libraries

11 Events

12 Alerts

13 Overview

14 Numba

15 Strategy Broker

OVERVIEW

title: "Runtime" description: "Bar-loop evaluator, series model, builtins, strategy events, and the optional Numba compile path."

RUNTIME

Abstract

PYNE's runtime is the deterministic **bar-loop** that turns a parsed ASDL AST into series values, drawing side-effects, and strategy events. It is deliberately host-agnostic: the same visitor mixins power the library API, the Flask Pro API (`Runtime.run`), browser Pyodide, and edge workers. A second path—source-to-source compilation into a Numba or object-mode bar loop—trades interpretive flexibility for throughput on long histories.

This section documents **semantics**, not the chart. AXIS consumes plots; HOOX consumes strategy events; neither is required to evaluate a script.

Conceptual model

```
flowchart TB
  subgraph host [Host]
    OHLCV[OHLCV bars]
    IN[input overrides / data_feed]
  end
  subgraph runtime [Runtime]
    CTX[Context + PineSeries]
    EV[NodeLiteralEvaluator]
    BUILTIN[Builtin dispatch]
    STRAT[StrategyState]
    PLOT[PlotRegistry / DrawingRegistry]
  end
  subgraph out [Outputs]
    SERIES[Plot series]
    EVENTS[StrategyEvent list]
    META[inputs / drawings]
  end
  OHLCV --> CTX
  IN --> EV
  CTX --> EV
  EV --> BUILTIN
  BUILTIN --> STRAT
  BUILTIN --> PLOT
  EV --> SERIES
  STRAT --> EVENTS
  PLOT --> META
```

Two execution modes share the same parse tree:

Mode	Entry	Bar loop	Typical use
Interpret	`NodeLiteralEvaluator.visit(tree)` per bar	Host updates `PineSeries` / context, re-visits AST	Full language surface, strategy, drawings
Compile	`pynescript.compiler.engine.compile_script`	Generated `for __bar_idx in range(n)`	Numeric indicators (Numba) or object-mode subset

Interface surface

Concern	Start here
Series history, `[]`, `var` / `varip`, `na`	Series & history
Expressions, statements, control flow	Expressions & statements
Builtin namespaces (`ta.*`, `strategy.*`, ...)	Builtins hub
Library `export` / `import`	Libraries
`StrategyEvent` parity contract	Events
Numba / object-mode compiler	Compiler overview

Library callers typically use `NodeLiteralEvaluator` (or the Pro API `Runtime`) rather than wiring the bar loop by hand:

```
from pynescript.ast.evaluator import NodeLiteralEvaluator

ev = NodeLiteralEvaluator(context={"close": [1.0, 2.0, 3.0], "bar_index": 2})
result = ev.evaluate_script('//@version=5\nindicator("x")\nplot(close)')
```

Hosts that own OHLCV (Pro API, workers) implement the loop: push bar → fill pending orders → `visit(tree)` → drain events → collect plots. See `backend/runtime.py` for the reference implementation.

Internals

Path	Role
`src/pynescript/ast/evaluator/`	Mixin evaluators (base, names, expressions, statements, libraries, events)
`src/pynescript/ast/evaluator/builtins/`	Builtin dispatch tables
`src/pynescript/compiler/`	`CompilerVisitor`, Numba builtins, strategy broker, engine façade
`backend/runtime.py`	Production bar loop + `mode="compile"` bridge
`tests/test_evaluator.py`, `tests/test_parity.py`, `tests/test_strategy_*.py`	Semantic oracles

Composition of the full interpreter:

```
BaseEvaluator
  + LiteralEvaluator
  + NameEvaluator
  + ExpressionEvaluator
  + StatementEvaluator
  + BuiltinEvaluator
  → NodeLiteralEvaluator
```

`BuiltinEvaluator` itself aggregates mixins (`TechnicalAnalysisMixin`, `StrategyBuiltinsMixin`, `PlottingFunctionsMixin`, ...) into one dispatch map built at construction time.

Invariants & edge cases

1. **Bar re-entrancy.** The AST is visited once per bar. Function/type definitions lock after bar 0 (`_pine_defs_locked`) so multi-dispatch tables do not grow (O(\text{bars}^2)).
2. **Order of broker steps.** Pending limit/stop fills run **before** script body evaluation on each bar (interpreter and compile object-mode agree).
3. **`na` is `None`.** Unresolved missing data, OOB history, and arithmetic with missing operands produce Python `None`, not IEEE NaN—except on the Numba path, which uses `np.nan` as the array sentinel.
4. **Strategy state is per-evaluator.** `StrategyState` lives on the instance (`evaluator._strategy_state`), never as a process-global singleton—required for concurrent runs and parity tests.
5. **Drawing/plot registries are side channels.** Visual objects do not participate in pure expression values except where `plot()` returns a plot id for `fill()`.

Worked example — minimal bar loop mental model

```
for bar_index, bar in ohlcv:
    open/high/low/close.update(bar)
    context[bar_index, time, barstate, …] = …
    process_pending_orders(OHLC)   # broker sim
    evaluator.visit(ast)           # script body
    events += drain_events()
    series.append(plot values)
```

Compile mode collapses this into one generated function over full arrays, with `__bar_idx` standing in for `bar_index`.

Failure modes

Symptom	Likely cause
`unexpected type of node: …`	AST node not implemented in any mixin's `visit_*`
`Unknown built-in function: …`	Missing dispatch key or wrong qualified-name resolution
Divergent strategy events vs TV	Partial-fill / OCA / risk settings, or fill timing vs bar
Numba path raises / falls back	Script uses UDT/map/drawing → object mode; or Numba not installed
Plots empty but no error	Host forgot to collect `plot_outputs` / `PlotRegistry` after visit

See also

- Language core — grammar, AST, types before evaluation
- Pro API runtime bridge
- Numerical validation
- pine-worker parity
- AXIS docs — optional AXIS for series/drawings

SERIES AND HISTORY

title: "Series and history" description: "Pine series model: history operator, na propagation, var/varip persistence, and bar-mode scalars."

SERIES AND HISTORY

Abstract

Pine's core data structure is the **series**: a value that evolves bar by bar, with random access to past values via the history operator `[]`. PYNE implements that model with host-managed series wrappers (`PineSeries` in the Pro API runtime), plain Python lists in unit tests, and flat `numpy` arrays on the compile path. Understanding history indexing and `na` is prerequisite to every builtin and strategy claim.

Conceptual model

```
flowchart LR
  subgraph bar_t [Bar t]
    C0["series[0] current"]
  end
  subgraph bar_tm1 [Bar t-1]
    C1["series[1]"]
  end
  subgraph bar_tm2 [Bar t-2]
    C2["series[2]"]
  end
  C0 -.->|history| C1
  C1 -.->|history| C2
```

Pine indices are **most-recent-first**:

Pine	Meaning	List representation (chronological)
`close[0]`	Current bar	`list[-1]`
`close[1]`	Previous bar	`list[-2]`
`close[n]` with (n \ge \text{len})	Out of range	`na` (`None`)

Negative offsets are **not** supported; they raise rather than wrap.

Interface surface

History operator (`visit_Subscript`)

Implemented in `NameEvaluator.visit_Subscript`:

- Float indices coerce to `int` (e.g. `depth / 2`); NaN index → `na`.
- Lists map Pine index (i) to Python index `-(i + 1)`.
- Scalars: `x[0]` is `x`; `x[i]` for (i > 0) is `na` (no history buffer).
- Matrices use `[row, col]` (tuple/list of length 2), not series semantics.

Series wrappers

Hosts expose OHLCV as objects with:

- `.current` — scalar for the active bar
- `.history` — most-recent-first buffer (deque or list)

Arithmetic in `ExpressionEvaluator` coerces such wrappers via `_as_scalar_operand` so `close + 1` does not attempt object addition.

`var` / `varip` / `const`

Qualifier	Semantics in PYNE
`var` / `varip`	Initializer runs on first **execution** of that declaration site (tracked in `_var_declarations`), not only `bar_index == 0`. Later bars skip re-init so the value carries.
`const` (v6)	Always initializes when the statement runs; not a cross-bar carry lock like `var`.
bare assign	Re-evaluates every bar; series "history" is the host series or prior list values.

Nested `var` inside `if barstate.islast` or a function therefore initializes on first path-taken bar—matching Pine's execution-based persistence.

`na` propagation

Operation	Rule
Binary arithmetic / comparison	Any `None` operand → `None` (or element-wise for lists)
Division by zero	→ `None` (not exception)
Soft type errors (`"a" + 1`)	→ `None`
Unary	`None` stays `None`
History OOB	→ `None`
Bare name `na`	Builtin / sentinel resolving to `None`
`nz(x, r)` (Numba path)	`nan` → replacement

List-valued series apply operators element-wise, aligning on the **trailing edge** when lengths differ (pad leading `None`).

Bar mode vs full-series mode

Technical helpers (`technical_submodules/core.py`) distinguish:

- **Full-series mode** (unit tests with explicit lists): `ta.sma` may return a full list of values.
- **Bar mode** (`_pine_bar_mode`): returns the **current scalar** so expressions like `ta.ema(close,12) - ta.ema(close,26)` stay numeric per bar.

History buffers for indicators are truncated to a rolling window (`_SERIES_MAX = 256` for wrapper histories) to avoid (O(n^2)) full-history recomputation every bar.

Internals

Path	Role
`src/pynescript/ast/evaluator/names.py`	`visit_Name`, `visit_Attribute`, `visit_Subscript`
`src/pynescript/ast/evaluator/expressions.py`	NA-safe binary/unary ops, series coerce
`src/pynescript/ast/evaluator/statements.py`	`var` / `varip` / `const` assign
`src/pynescript/ast/evaluator/builtins/technical_submodules/core.py`	`_as_series`, bar mode finalize
`src/pynescript/ast/evaluator/builtins/utility.py`	`max_bars_back`, `last_bar_index`, `na` helpers
`backend/runtime.py`	`PineSeries.update` per bar

Compile path: series are `np.full(n_bars, np.nan)` written at `__bar_idx`; history is `arr[__bar_idx - n]` (see Compiler overview).

Invariants & edge cases

1. **Chronology of `.history`.** Most-recent-first; reverse when converting to chronological lists for `ta.*` / `array.*`.
2. **No negative Pine indices.** Explicit error—different from Python lists.
3. **`None` vs `float('nan')`.** Interpreter prefers `None`; Numba kernels use `np.nan`. Cross-mode comparisons must normalize.
4. **`max_bars_back`.** Declares intended depth; PYNE does not hard-cap OHLCV history the way a hosted chart might, but indicator helpers still bound rolling windows for cost.
5. **Tuple returns from multi-value `ta.*`.** Unpacking `[a, b, c] = ta.macd(...)` uses `current` when it is a sequence of matching arity; otherwise history heuristics apply.

Worked examples

History lag

```
//@version=5
indicator("lag")
delta = close - close[1]
plot(delta)
```

On bar 0, `close[1]` is `na` → `delta` is `na`. On bar 1+, arithmetic is ordinary floats (or series lists in batch tests).

`var` counter

```
//@version=5
indicator("count")
var int n = 0
n := n + 1
plot(n)
```

`n` initializes once; subsequent bars reassignment (`:=`) increments. Declaration sites are recorded in `_var_declarations`.

Element-wise NA

Given two list series of different lengths, `a + b` aligns tails and pads the longer prefix with `None`—preserving "most recent bars line up" intuition.

Failure modes

Symptom	Cause
All `na` after first bar	Host not updating series / `bar_index` between visits
`var` re-inits every bar	Fresh evaluator or `reset_var_declarations` each bar incorrectly
TypeError on `close + 1`	Series wrapper missing `.current` / not coerced
Compile vs interpret plot mismatch	`nan` vs `None`, or seed differences in EMA/RSI kernels

See also

- Expressions & statements
- Technical builtins
- Numba path
- Runtime hub

EXPRESSIONS STATEMENTS

title: "Expressions and statements" description: "How the AST walker evaluates operations, calls, control flow, assignments, and user definitions."

EXPRESSIONS AND STATEMENTS

Abstract

Evaluation is pure visitor dispatch: each ASDL node type maps to a `visit_*` method on a mixin. Expressions produce values (with Pine `na` and series rules); statements mutate context, register types/functions, or emit side effects via builtins. This page is the operational semantics of the interpreter path—what runs inside `evaluator.visit(tree)` each bar.

Conceptual model

```
flowchart TB
  Script[Script body] --> Stmt[Statements]
  Stmt --> Assign[Assign / ReAssign]
  Stmt --> If[If / For / While]
  Stmt --> Def[FunctionDef / TypeDef / EnumDef]
  Stmt --> ExprStmt[Expression statements]
  ExprStmt --> Expr[Expressions]
  Expr --> BinOp[BinOp / BoolOp / Unary / Compare]
  Expr --> Call[Call / Attribute / Name / Subscript]
  Call --> Builtin[Builtin map]
  Call --> UserFn[User function / method]
  Assign --> Ctx[(context dict)]
  UserFn --> Ctx
```

Interface surface

Expressions (`ExpressionEvaluator`)

Node	Behavior
`BoolOp` (`and` / `or`)	Short-circuit via Python `all` / `any` on visited values
`BinOp`	NA-safe `+ - * / %` ; division by zero → `na`
`UnaryOp`	`not` , unary `+` / `-` with NA and list map
`Compare`	Chained comparisons with NA-safe operators
`IfExp`	Ternary: condition, then, else
`Call`	Builtin dispatch, user functions, methods, multi-dispatch overloads
`Tuple` / lists	Literal sequences for unpacking and multi-arg returns

Binary ops coerce `PineSeries` -like objects to `.current` before operating. List operands broadcast or zip with trailing alignment (see Series & history).

Names (`NameEvaluator`)

Node	Behavior
`Name`	Context lookup; bare series builtins (`na` , `year` , ...); else string id (lazy)
`Attribute`	Qualified context keys, builtins (`strategy.position_size`), library modules, UDT fields/methods, array/drawing/matrix method markers, Python getattr fallback
`Subscript`	Series history / array / matrix indexing

Critical: qualified names like `strategy.opentrades.entry_price` are resolved via AST path construction (`ast_qualified_name`) **without** evaluating intermediate zero-arg series that would collapse the path to an int.

Statements (`StatementEvaluator`)

Construct	Behavior
`Script`	Walk body; finalize library exports if `library(...)` active
`Assign`	`var` / `varip` once-per-declaration; `const` ; ordinary assign; tuple unpack; `export` registration
`ReAssign` (`:=`)	Update existing binding (and UDT fields)
`If` / `For` / `While`	Standard control flow; loop variables in context
`FunctionDef`	Store callable in context; multi-dispatch overload lists; `export` ; methods tagged `__pine_method__`
`TypeDef` / `EnumDef`	Type registry / enum member dicts
`Import`	Resolve `LibraryRegistry` → bind alias to `LibraryModule`
Declarations	`indicator` / `strategy` / `library` via declaration builtins

Function invocation binds parameters into context (with restore of prior bindings on exit) so nested calls and bar-local state interact cleanly. After bar 0, hosts set `_pine_defs_locked` so re-visiting the script does not re-register overload tables.

Calls and multi-dispatch

Method markers returned from attribute evaluation:

Marker	Meaning
`("_method_call", instance, name)`	UDT method
`("_array_method", list, name)`	`array.<name>(self, …)`
`("_ns_method", obj, "label.get_text")`	Drawing/matrix namespace method
`("_ext_method", receiver, name)`	Free `method` with receiver as first arg

Overloads with typed first parameters (e.g. `matrix.float` vs `array.label`) select via coarse type tags; `na` receivers deliberately exclude matrix/array/drawing tags so optional UDT fields do not mis-dispatch to `tostring` on collections.

Internals

Path	Role
`src/pynescript/ast/evaluator/expressions.py`	Ops, calls, conditionals
`src/pynescript/ast/evaluator/names.py`	Names, attributes, subscripts
`src/pynescript/ast/evaluator/statements.py`	Script structure, assign, defs, import
`src/pynescript/ast/evaluator/literals.py`	Constants, colors, strings
`src/pynescript/ast/evaluator/base.py`	Context, math constants, library registry hook
`src/pynescript/ast/evaluator/types.py`	`EvaluatorProtocol` typing for mixins

Invariants & edge cases

1. **Context is the single store.** Builtins read OHLCV and strategy series from `self.context`; hosts must keep it coherent each bar.
2. **String fallback names.** An unbound `Name` returns its id string so later attribute/call resolution can still hit the builtin map (`"ta.sma"` patterns via attribute chains).
3. **Parameter unbind.** Missing-before-bind params use a sentinel so unbind `pop`s rather than leaving `None` ghosts.
4. **`na` normalization.** String `"na"` / `"nan"` / `"none"` may coerce to `None` at dispatch boundaries for UDT optional args.
5. **User functions re-enter the visitor.** Recursive and higher-order patterns are limited by Python stack, not a Pine-specific trampoline.

Worked examples

Ternary and comparison chain

```
//@version=5
indicator("cmp")
v = close > open ? close - open : open - close
plot(v)
```

Each operator path is NA-safe: if `close` or `open` were missing, comparisons and arithmetic yield `na`.

User function with series

```
//@version=5
indicator("fn")
f(x, n) =>
    ta.sma(x, n)
plot(f(close, 14))
```

`f` is stored as a Python callable that rebinds `x` / `n` and visits the function body AST each call.

Method multi-dispatch sketch

```
method log(this, string s) => ...
method log(this, float x) => ...
```

Both register under the same name with `__pine_overloads__`; call sites pick by receiver/arg type tags.

Failure modes

Error	Meaning
`unexpected type of node`	Missing `visit_*` for an AST construct
`Unsupported binary operator`	Op class not in the dispatch table
`Negative indices not supported`	Pine forbids `series[-1]` Python style
Wrong overload / destroyed receiver	Historical `na` string path; now normalized—report residual mismatches as bugs
Stack overflow	Deep recursion in user functions

See also

- Series & history
- Builtins hub
- Libraries
- Type system (core)

OVERVIEW

title: "Builtins" description: "Dispatch architecture for ta.*, strategy.*, collections, plotting, request.*, and other standard namespaces."

BUILTINS

Abstract

Pine's standard library is not a single module: it is a **flat qualified-name dispatch table** (`"ta.sma"`, `"strategy.entry"`, `"array.push"`, …) assembled from category mixins. Each entry is a handler `(args, kwargs?) → value` with side effects when the language requires them (orders, plots, drawings, inputs). This hub orients the surface; child pages go deep on technical, strategy, collections, drawing/plotting, and request/input.

Conceptual model

```
flowchart LR
  Call[Call / Attribute] --> Qual[Qualified name]
  Qual --> Map["_build_builtin_map()"]
  Map --> Handler[BuiltinHandler]
  Handler --> Val[Return value]
  Handler --> Side[Side effects]
  Side --> Strat[StrategyState]
  Side --> Plot[PlotRegistry]
  Side --> Draw[DrawingRegistry]
  Side --> Inp[_input_declarations]
```

`BuiltinEvaluator` merges mixin maps in a fixed order and then registers ticker, logging, color, timeframe, and script-declaration functions. The `strategy()` declaration handler is wrapped so broker settings apply to `StrategyState`.

Interface surface

Namespace / area	Mixin / module	Docs
`ta.*`	`TechnicalAnalysisMixin` + `technical_submodules/*`	Technical
`strategy.*`	`StrategyBuiltinsMixin`, `StrategyConstantsMixin`	Strategy
`array.*` / `matrix.*` / `map.*`	`ArrayBuiltinsMixin`, matrix/map evaluators	Collections
`plot*` / `hline` / `line` / `label` / …	`PlottingFunctionsMixin`, `DrawingBuiltinsMixin`	Drawing & plotting
`request.*` / `input.*`	`RequestBuiltinsMixin`, `InputBuiltinsMixin`	Request & input
`math.*`, min/max, `nz`, …	`NumericBuiltinsMixin`	(this page)
`str.*`	`StringBuiltinsMixin`	(this page)
`color.*`	`register_color_functions`	(this page)
`syminfo` / `ticker.*`	`register_ticker_functions`	(this page)
`timeframe.*`	`register_timeframe_functions`	(this page)
`alert` / logging	`AlertsMixin`, `register_logging_functions`	Alerts
`indicator` / `strategy` / `library`	`register_script_declaration_functions`	Libraries
Footprint / volume_row	`FootprintBuiltinsMixin`	Request & input

Numeric and string essentials

- **Numeric:** `math.*` constants from `BaseEvaluator` (`math.pi`, `math.e`, golden-ratio helpers); functions cover abs, min/max, rounding, logs, trig, and NA helpers (`nz`, `na` predicates via utility).
- **String:** `str.*` formatting, conversion (`str.tostring` respects `format.*` constants), and manipulation aligned with common Pine scripts in the builtin corpus.

Color, ticker, timeframe

- **Colors:** hex constants (`color.red` → `#F23645`, etc.) plus composition helpers.

- **syminfo / ticker:** host-provided `Syminfo` object or flat context keys; ticker construction for `request.security` symbols.
- **timeframe:** period flags (`isintraday` , `isdaily` , ...) default daily in base constants; hosts override from bar spacing.

Alerts

`alert(message, freq)` and `alertcondition(condition, title, message)` record structured firings (`message` , `freq` , `bar_index` , `time` , `source`) with TV-style frequency rules (`once_per_bar` , `once_per_bar_close` , `all`). Hosts export them on `/run` and can POST last-bar batches to webhooks (**L2**). Side channel only — **not** strategy events.

Full detail: Alerts.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/__init__.py`	`BuiltinEvaluator` composition
`src/pynescript/ast/evaluator/builtins/base.py`	Dispatch mixin, `_call_builtin` , registration
`src/pynescript/ast/evaluator/builtins/*.py`	Category implementations
`scripts/generate_builtin_metadata.py`	LSP/metadata surface (not hand-edited JSON)

Handlers are looked up by the fully qualified string built during name/attribute evaluation. Zero-arg series (e.g. `strategy.equity`) are registered as builtins that ignore empty args.

Invariants & edge cases

1. **One map, many mixins.** New builtins require a map entry *and* (usually) metadata regeneration for LSP.
2. **Kwargs and positionals.** Handlers accept both; Pine named arguments arrive as kwargs when the call AST carries them.
3. **Bar mode.** Stateful `ta.crossover` / similar use per-bar call indices reset by the host (`_cross_call_i`).
4. **Mock fallbacks.** `request.*` without a feed still returns synthetic data so offline evaluation does not hard-fail—hosts must wire real providers for production accuracy.
5. **Compile path subset.** Numba builtins reimplement only a fraction of this table; object mode covers more via Python helpers—see Compiler.

Worked example — resolution path

```
plot(ta.sma(close, 14))
```

1. Attribute/call chain builds qualified name `ta.sma` .
2. `_call_builtin("ta.sma", [close_series, 14])` .
3. Handler coerces series → list, computes SMA, finalizes scalar or list.
4. `plot` registers a `Plot` with that series value and returns the plot id.

Failure modes

Symptom	Fix
`Unknown built-in function`	Typo, version surface gap, or failed attribute chain
Always mock prices	Wire `data_feed` / `data_provider` into evaluator context
LSP knows a function runtime lacks	Metadata ahead of implementation—check missing features
Handler arity errors	Positional vs keyword mismatch in script

See also

- Technical
- Strategy
- Collections
- Drawing & plotting
- Request & input
- Alerts
- Builtin metadata (LSP)

TECHNICAL

title: "Technical analysis (ta.*)" description: "ta.* indicators: series helpers, bar mode, numerical parity, and submodule layout."

TECHNICAL ANALYSIS (`ta.*`)

Abstract

The `ta.*` namespace is the largest pure-compute surface in PYNE: moving averages, oscillators, volatility bands, volume studies, pattern helpers, and cross/rise/fall predicates. Implementations live under `technical_submodules/` and are composed into `TechnicalAnalysisMixin`. Numerical behavior is validated against TradingView reference series (see numerical validation); the design goal is IEEE-limit parity, not “approximate TA.”

Conceptual model

```
flowchart TB
  Args[Args: series, period, …] --> As[_as_series]
  As --> Chrono[Chronological list]
  Chrono --> Inc{bar mode + incremental?}
  Inc -->|yes hot path| State[Call-site state O1]
  Inc -->|no| Kernel[Full-history kernel]
  State --> Scalar[Current scalar]
  Kernel --> Fin[_finalize_series]
  Fin -->|bar mode| Scalar
  Fin -->|test mode| Full[Full list]
```

Hosts in production set bar mode so indicator calls return **scalars for the active bar**, matching how Pine expressions compose (`ta.ema(a) - ta.ema(b)`).

Incremental hot path (2026-07-28)

When `_pine_bar_mode` and `_pine_ta_incremental` are enabled (Runtime default; disable with `PYNE_TA_INCREMENTAL=0`), these builtins update **call-site state** once per bar instead of recomputing full history:

Builtin	State model
`ta.sma`	Rolling window (deque)
`ta.ema` / `ta.rma`	Recursive seed + step
`ta.rsi`	Dual RMA of gains/losses
`ta.macd`	Fast/slow/signal EMAs in one slot
`ta.atr`	TR stream; mean warm-up then EMA-of-TR (matches current full oracle)

Call sites are indexed like crossovers (`_ta_call_i` reset each bar). Nested forms such as `ta.ema(ta.sma(close, 14), 10)` stay correct because each call keeps its own slot. Golden suite: `tests/test_ta_incremental.py` (incremental last values ≡ full recompute).

Interface surface

Dispatch keys (non-exhaustive; map is authoritative in `technical.py`):

Family	Examples
Moving averages	`ta.sma`, `ta.ema`, `ta.wma`, `ta.rma`, `ta.hma`, `ta.vwma`, `ta.swma`, `ta.alma`
Oscillators	`ta.rsi`, `ta.stoch`, `ta.cci`, `ta.cmo`, `ta.mfi`, `ta.roc`, `ta.wpr`, `ta.tsi`
Trend / channels	`ta.macd`, `ta.adx`, `ta.dmi`, `ta.supertrend`, `ta.sar`, `ta.linreg`
Volatility	`ta.atr`, `ta.tr`, `ta.bb`, `ta.bbw`, `ta.kc`, `ta.kcw`, `ta.stdev`, `ta.variance`
Volume	`ta.obv`, `ta.vwap`, `ta.mfi` (shared), volume submodule helpers
Structure	`ta.highest` / `lowest` / `highestbars` / `lowestbars`, `ta.pivothigh` / `pivotlow`
Predicates	`ta.crossover`, `ta.crossunder`, `ta.cross`, `ta.rising`, `ta.falling`
Stats / other	`ta.change`, `ta.mom`, `ta.cum`, `ta.median`, `ta.mode`, percentiles, `ta.barssince`, `ta.valuewhen`, `ta.correlation`

Argument conventions

- Typical form: `ta.fn(series, length)`.
- Some functions allow **period-only** calls (e.g. `ta.highest(20)`) defaulting source to `high` / context series via `_expect_series(..., allow_period_only=True)`.
- Multi-value returns (e.g. MACD, BB, Supertrend) unpack as tuples/lists; assignment uses StatementEvaluator unpack rules.
- Fractional lengths floor to int (TradingView-compatible).

Stateful crosses

`ta.crossover` / `ta.crossunder` need previous-bar pairs. In bar-mode runs the host resets a call-site index (`_cross_call_i`) each bar so multiple cross calls in one script keep independent state.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/technical.py`	Dispatch map aggregation
`.../technical_submodules/core.py`	Series coerce, expect helpers, bar finalize, **incremental kernels**
`.../moving_averages.py`, `oscillators.py`, `volatility.py`, `volume.py`, …	Kernels + inc wiring
`.../common.py`, `basic.py`, `advanced.py`, `patterns.py`, `strategies.py`, `synthesizer.py`, `economics.py`	Additional families
`tests/test_ta_indicators_*.py`, `tests/test_indicators.py`	Regression
`tests/test_ta_incremental.py`	Inc ≡ full recompute golden
`docs/numerical_validation_report.md`	Published precision summary

History for wrapper series is reversed to chronological order and may be truncated (`_SERIES_MAX`) before full kernels run. Incremental path only needs `series[-1]` per call, so truncation does not freeze state.

Invariants & edge cases

1. **Warm-up → `na`.** Insufficient bars (e.g. SMA length (N) needs (N) samples) yield `None` / leading `na` in full-series mode.
2. **NA in window.** Many kernels propagate `na` if any window element is missing (mirrors Pine strictness for SMA-like sums).
3. **Default sources.** `ta.atr` pulls high/low/close from `current_series` / context when not passed explicitly.
4. **Compile path kernels differ.** Numba `numba_sma` / `numba_ema` / `numba_rsi` / highest/lowest are separate implementations—parity tests should cover both modes for scripts that compile.
5. **No chart look-ahead.** Kernels only see history available at the current bar index; `request.security` lookahead is a separate concern.
6. **Incremental ≡ full recompute (oracle).** Changing seed rules (e.g. ATR Wilder vs EMA-of-TR) is a correctness project, not a silent perf flag.

Worked examples

Classic overlay

```
//@version=5
indicator("SMA 14")
v = ta.sma(close, 14)
plot(v)
```

In bar mode each visit returns one float (or `na`); the host stacks them into a series for the response envelope.

Cross entry signal

```
//@version=5
strategy("X")
fast = ta.ema(close, 12)
slow = ta.ema(close, 26)
if ta.crossover(fast, slow)
    strategy.entry("L", strategy.long)
```

Cross state is bar-local; ensure the runtime resets cross call indices when reusing an evaluator.

Multi-value unpack

```
[macdLine, signal, hist] = ta.macd(close, 12, 26, 9)
plot(hist)
```

Failure modes

Symptom	Cause
Always `na`	Length longer than available history; or series not updated
Wrong vs TV by large margin	Wrong source series; mock OHLCV; or non-bar-mode list composition bug
Cross never fires	Call-index state not reset / shared incorrectly across bars
Compile diverges	Numba RSI/EMA seed conventions—check `numba_builtins.py`

See also

- Series & history
- Strategy builtins
- Numba builtins
- Numerical validation

STRATEGY

title: "Strategy builtins" description: "Orders, fills, OCA, commission, risk gates, position metrics, and StrategyState."

STRATEGY BUILTINS

Abstract

PYNE's strategy layer is a **per-run broker simulation** driven by `strategy.*` calls during the bar loop. It is not a live exchange adapter: orders become fills against bar OHLC (market immediately; limit/stop via pending book), commissions and slippage adjust PnL, and risk helpers can block entries. Structured StrategyEvent records form the parity contract with `pine-worker` and the HOOX trade mesh.

Conceptual model

```
sequenceDiagram
  participant Host as Bar loop
  participant Pend as Pending orders
  participant Script as strategy.* body
  participant State as StrategyState
  participant Ev as Event buffer
  Host->>Pend: process_pending_orders(OHLC)
  Pend->>State: fills / OCA
  Pend->>Ev: order/entry/cancel…
  Host->>Script: visit(AST)
  Script->>State: entry/exit/close/order/risk
  Script->>Ev: StrategyEvent
  Host->>Ev: drain_events()
```

Fill-before-script matches the interpreter `Runtime` and the compile-path strategy broker.

Interface surface

Order placement

Builtin	Role
`strategy.entry`	Open/add (or reverse) by direction; market or pending limit/stop
`strategy.exit`	Bracket-style exit (limit/stop) against position
`strategy.close` / `strategy.close_all`	Flatten by id or all
`strategy.order`	Lower-level order with OCA name/type, partial fill cap
`strategy.cancel` / `strategy.cancel_all`	Remove pending orders

Directions accept Pine constants (`strategy.long` / `strategy.short`) and common string aliases.

Position and performance series

Zero-arg builtins include `strategy.position_size` (signed: +long / −short), `position_avg_price` , `position_entry_name` , `opentrades` / `closedtrades` counts, `netprofit` , `openprofit` , `equity` , `cash` , gross win/loss stats, averages, max drawdown/runup, and max contracts held.

Trade queries

Indexed accessors:

- `strategy.closedtrades.entry_*` / `exit_*` / `profit` / `size` / `commission`
- `strategy.opentrades.entry_*` / `size` / `profit` / `commission`

Risk

Builtin	Effect
`strategy.risk.max_position_size`	Cap size as % of equity
`strategy.risk.max_intraday_loss`	Intraday loss gate
`strategy.risk.max_intraday_filled_orders`	Order count cap
`strategy.risk.max_drawdown`	Absolute / percent drawdown halt
`strategy.risk.max_cons_loss_days`	Consecutive losing calendar days → `entries_blocked`
`strategy.risk.allow_entry_in`	`"all"` | `"long"` | `"short"`

Blocked entries still emit a diagnostic-style event with comment `risk_blocked` where implemented.

Declaration

`strategy(title, …)` applies broker settings onto `StrategyState`: `initial_capital`, commission type/value, slippage ticks, pyramiding, etc.

Internals

`StrategyState` (`strategy.py`)

Per-evaluator instance fields include:

- Position: direction (`flat` / `long` / `short`), size (non-negative internal), entry metadata
- Books: `pending_orders`, `open_trades`, `closed_trades`
- Economics: capital, commission model, slippage, mintick
- Risk: max position %, drawdown, consecutive loss days, `entries_blocked`
- Equity curve peak/trough for max DD / runup
- `_events: list[StrategyEvent]` drained each bar

`signed_position_size()` implements Pine's signed `strategy.position_size`.

`Order` and fills

```
Order: market | limit | stop | stop-limit
  + oca_name / oca_type ∈ {none, cancel, reduce}
  + max_fill_per_bar (0 = fill remaining)
```

`process_pending_orders(open, high, low, close)` walks the book, computes trigger prices from OHLC (gap-aware open logic for limits/stops), applies commission/slippage, updates trades, runs OCA side effects, and emits events.

OCA

oca_type	On fill of a group member
`cancel`	Cancel other pending in group
`reduce`	Reduce sibling quantities by fill size
`none`	Independent

Covered by `tests/test_oca_commission.py` and order-fill suites.

Invariants & edge cases

1. **Isolation.** Never use class-level strategy state; concurrent runs must not share books.
2. **Pyramiding.** Additional entries respect `pyramiding` from the declaration.
3. **Partial fills.** `max_fill_per_bar` / defaults allow multi-bar completion of large orders.
4. **Calendar buckets for cons-loss.** Exit timestamps in ms vs seconds vs bar index are normalized in `note_closed_trade_day`.
5. **Compile path.** Object-mode uses `CompileStrategyBroker`—aligned fill rules, smaller surface than full interpreter metrics. Prefer interpret mode for full `strategy.closedtrades.*` analytics unless compile coverage is verified.

Worked examples

Market long / close

```
//@version=5
strategy("Demo", overlay=true, initial_capital=100000)
if bar_index == 5
    strategy.entry("L", strategy.long, qty=10)
if bar_index == 10
    strategy.close("L")
```

Parity fixtures under `tests/fixtures/parity/pine/` encode expected event sequences for such scripts.

Pending stop entry

```
strategy.entry("Break", strategy.long, stop=high[1])
```

Creates a pending stop; subsequent bars' `process_pending_orders` fill when `high` trades through the stop, using open-gap-aware fill prices.

OCA bracket sketch

```
strategy.order("TP", strategy.short, qty=1, limit=tp, oca_name="br", oca_type=strategy.oca.cancel)
strategy.order("SL", strategy.short, qty=1, stop=sl, oca_name="br", oca_type=strategy.oca.cancel)
```

First fill cancels the sibling.

Failure modes

Symptom	Cause
Entry never fills	Pending stop/limit; OHLC never trades through; or risk block
Double entries	Pyramiding > 0 or missing close
Events missing `script_id`	Host forgot to stamp after `drain_events` (Runtime does this)
PnL off vs TV	Commission type, slippage ticks, mintick, or fill price model
Interpret vs compile event drift	Broker subset / timing—diff with `tests/test_compiler_strategy.py`

See also

- Events
- Strategy broker (compile)
- Parity / pine-worker
- Pro API backtest

COLLECTIONS

title: "Collections" description: "array.*, matrix.*, and map.* runtime semantics, method dispatch, and type tags."

COLLECTIONS

Abstract

Pine collections—**arrays**, **matrices**, and **maps**—are mutable values that live in the evaluator context (often under `var` so identity persists across bars). PYNE maps them to Python `list`, a dedicated `Matrix` type, and a `Map` / `dict`-like type respectively, with builtin functions and instance-method sugar (`a.push(x)` → `array.push(a, x)`).

Conceptual model

```
flowchart LR
  Pine[array / matrix / map API] --> Disp[Qualified builtins]
  Pine --> Meth[Instance method markers]
  Meth --> Disp
  Disp --> List[list]
  Disp --> Mat[Matrix]
  Disp --> MapT[Map / dict]
```

Interface surface

Arrays (`array.*`)

Represented as Python lists. Highlights from the dispatch map:

Category	Functions
Construct	`array.new`, `array.new_<type>`, `array.from`
Mutate	`push`, `pop`, `shift`, `unshift` (if present), `insert`, `remove`, `set`, `fill`, `clear`, `reverse`, `sort`
Access	`get`, `first`, `last`, `size`, `includes`, `indexof`, `lastindexof`, `slice`
Aggregate	`sum`, `avg`, `min`, `max`, `median`, `mode`, `range`, `stdev`, `variance`, covariance, percentiles, percentrank
Search	`binary_search`, `binary_search_leftmost`, `binary_search_rightmost`
Higher-order	`every`, `some`, `copy`, `concat`, `join`, `abs`

Indexing: Pine array indices are **0-based from the start of the list** (unlike series history). Negative indices are rejected for series; array paths use list semantics consistent with the builtin handlers.

Series-like values passed into array ops unwrap via `.history` (reversed to chronological) when duck-typed.

Matrices (`matrix.*`)

`Matrix` supports:

- Construction: `matrix.new`
- Element access: `matrix.get` / `set`, and `m[row, col]` via subscript
- Shape: `rows`, `columns`, `elements_count`
- Row/column ops: add/remove/copy, sum/avg/min/max/mode, fill
- Aggregates: sum/avg/min/max/mode all
- Transforms: `transpose`, fill diagonal, and further linear helpers in the evaluator map

Method sugar: `m.rows()` → `matrix.rows(m)`.

Maps (`map.*`)

Function	Role
`map.new`	Empty map
`put` / `put_all` / `get` / `remove` / `clear`	Mutation
`contains` / `keys` / `values` / `size` / `copy`	Query

Keys follow Pine map rules as implemented (hashable Python keys). Compile object-mode lowers maps to ordinary Python `dict`s.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/arrays.py`	Array builtins
`src/pynescript/ast/evaluator/builtins/matrix.py`	`Matrix` type
`src/pynescript/ast/evaluator/builtins/matrix_evaluator.py`	matrix.* handlers
`src/pynescript/ast/evaluator/builtins/map.py`	`Map` type
`src/pynescript/ast/evaluator/builtins/map_evaluator.py`	map.* handlers
`src/pynescript/ast/evaluator/names.py`	Method markers for list/Matrix
`tests/test_collections*.py`, `test_map_collections.py`, `test_matrix_*.py`	Coverage

Multi-dispatch type tags (`array.float`, `matrix.string`, …) interact with method overloading and `na` exclusion lists so `na`-typed optionals do not bind to collection `tostring` overloads.

Invariants & edge cases

1. **Identity under `var`.** `var a = array.new_float(0)` keeps one list across bars; reassigning `a := array.new_float(0)` replaces it.
2. **Drawing-typed arrays.** `array.new_label` etc. store drawing object references; delete semantics remain drawing-registry concerns.
3. **Empty array overload match.** Empty arrays match any `array.*` element tag in dispatch (no sample element).
4. **Matrix unpack hazard.** StatementEvaluator refuses to treat matrices as general iterables for tuple unpack—prevents corrupting multi-assign from row iteration.
5. **Compile mode.** Maps/UDTs force **object mode**; pure array numeric work may still numeric-compile depending on visitor detection—verify generated code when performance-critical.

Worked examples

Rolling buffer

```
//@version=5
indicator("buf")
var floats = array.new_float(0)
array.push(floats, close)
if array.size(floats) > 20
    array.shift(floats)
plot(array.avg(floats))
```

Matrix element

```
var m = matrix.new<float>(2, 2, 0.0)
matrix.set(m, 0, 1, close)
plot(matrix.get(m, 0, 1))
```

Map counter

```
var counts = map.new<string, int>()
map.put(counts, "n", nz(map.get(counts, "n")) + 1)
```

Failure modes

Symptom	Cause
`map.get requires map and key`	Wrong arity or non-Map receiver
Index errors	OOB get/set without Pine `na` guard in that handler
Method not found	Typo or non-list receiver after series unwrap failed
Shared mutation across runs	Reused evaluator context without reset

See also

- Expressions & statements
- Drawing & plotting — arrays of labels/lines
- Compiler object mode

DRAWING PLOTTING

title: "Drawing and plotting" description: "plot* side effects, DrawingRegistry objects, and export shapes for hosts and AXIS."

DRAWING AND PLOTTING

Abstract

Visual effects in Pine are **side channels**: they do not change pure arithmetic results, but they are first-class runtime outputs. PYNE records plots in `PlotRegistry` and drawing primitives (lines, boxes, labels, tables, polylines, linefills) in `DrawingRegistry`, then serializes them for the Pro API and optional AXIS. Evaluation never requires a UI—tests assert on registries alone.

Conceptual model

```
flowchart TB
  Script[plot / line.new / label.new] --> Reg[Registries]
  Reg --> PlotR[PlotRegistry]
  Reg --> DrawR[DrawingRegistry]
  PlotR --> Host[Host series envelope]
  DrawR --> API[export_for_api]
  API --> AXIS[AXIS / chart clients]
```

Interface surface

Plotting (`PlottingFunctionsMixin`)

Function	Kind	Notes
`plot`	`plot`	Returns `Plot` id for `fill(plot1, plot2)`
`plotshape` / `plotchar` / `plotarrow`	markers	location, char, text metadata
`plotbar` / `plotcandle`	OHLC visuals	open/high/low/close fields
`hline`	horizontal	price level
`bgcolor` / `barcolor`	coloring	series-driven colors
`fill`	fill	references two plot ids

Style constants: `plot.linestyle_solid` / `dashed` / `dotted`.

Each call constructs a `Plot` dataclass (`kind` , `series` , `title` , `color` , linewidth, text formatting, force_overlay, ...) and appends it to `PlotRegistry.plots` .

Hosts often **also** capture per-bar plot values on the evaluator (`plot_outputs`) for time-aligned series maps—see `backend/runtime.py` .

Drawing objects (`DrawingBuiltinsMixin`)

Object types: `Line` , `Box` , `Label` , `Table` , `Polyline` , `LineFill` , `ChartPoint` .

Typical factories: `line.new` , `box.new` , `label.new` , `table.new` , `polyline.new` , `linefill.new` , plus getters/setters/deletes and `*.all` / last-bar helpers where implemented.

Instance methods resolve through namespace markers:

```
la.get_text()  →  label.get_text(la)
```

`DrawingRegistry.export_for_api(bar_times)` maps `xloc=bar_index` coordinates to wall times for chart clients, normalizes colors, and skips deleted objects.

Registry lifecycle

```
DrawingRegistry.reset()  # clears drawings + PlotRegistry
```

Hosts must reset at run start so labels from a previous evaluation do not leak.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/plotting.py`	`Plot`, `PlotRegistry`, plot* handlers
`src/pynescript/ast/evaluator/builtins/drawing.py`	Drawing types, registry, builtins, export
`src/pynescript/ast/evaluator/names.py`	`_DRAWING_METHOD_NS`
`tests/test_plotting_effects.py`, `test_drawing_all_and_last_bar.py`	Behavior
Compiler object mode	Accumulates `__drawings` event list instead of full registry parity

Invariants & edge cases

1. **`plot` returns an id.** Required for `fill`; treating it as "void" breaks band fills.
2. **Deleted flag.** Soft-delete keeps history for debugging; `active()` filters deleted plots.
3. **Series in drawings.** Prices may be series wrappers—export coerces `.current` and maps NaN → omit.
4. **Compile path.** Numeric mode supports `plot` arrays; drawings force object mode with a simplified event list—not full delete/style parity with the interpreter.
5. **No GPU/canvas in-process.** Registries are data; AXIS/HOOX render elsewhere.

Worked examples

Dual plot + fill

```
//@version=5
indicator("BB mid")
basis = ta.sma(close, 20)
p1 = plot(basis)
p2 = plot(basis * 1.01)
fill(p1, p2, color=color.new(color.blue, 80))
```

Label on last bar

```
//@version=5
indicator("lbl", overlay=true)
if barstate.islast
    label.new(bar_index, high, str.tostring(close))
```

Hosts read `DrawingRegistry.labels` or API export after the run.

Failure modes

Symptom	Cause
Empty drawings in API	Forgot `DrawingRegistry.reset` timing / export; or script never called `*.new`
fill no-ops	plot ids not retained from `plot()` returns
Stale labels across HTTP runs	Missing registry reset between `Runtime.run` calls (Runtime resets—custom hosts must)
Object mode missing delete	Known compile limitation—use interpret mode

See also

- Builtins hub
- Pro API preview / chart renderer
- AXIS docs
- Compiler overview

REQUEST INPUT

title: "request.* and input.*" description: "Multi-symbol/timeframe requests, fundamental mocks, footprint, and input parameter resolution."

`request.*` AND `input.*`

Abstract

Two namespaces couple scripts to the **host environment**: `input.*` declares parameters (with defaults and UI metadata), and `request.*` pulls data from other symbols, timeframes, or fundamental/economic sources. PYNE implements both as builtins with explicit extension points— `data_feed`, `data_provider`, and `_input_overrides` — so the same script can run offline with mocks or online with real market data.

Conceptual model

```
flowchart TB
  subgraph inputs [Inputs]
    Def[defval in script]
    Over[_input_overrides by title]
    Def --> Res[Resolved value]
    Over --> Res
    Res --> Meta[_input_declarations]
  end
  subgraph req [request.*]
    Call[request.security / …] --> Feed[data_feed]
    Call --> Prov[data_provider]
    Feed --> Out[Series / scalar]
    Prov --> Out
    Call --> Mock[Mock fallback]
    Mock --> Out
  end
```

Interface surface

`input.*` (`InputBuiltinsMixin`)

Builtin	Purpose
`input`	Generic defval + title/tooltip/inline/group/confirm/active
`input.bool` / `int` / `float` / `string` / `color`	Typed scalars
`input.price` / `source` / `time` / `timeframe` / `session` / `symbol`	Domain inputs
`input.enum` / `input.text_area`	Enumerations and multiline text

Runtime value: each call returns the resolved value (override if title matches, else default). Metadata is appended to `_input_declarations` for settings panels and LSP-adjacent hosts.

Overrides live on the evaluator as `_input_overrides: dict[title, value]`.

`request.*` (`RequestBuiltinsMixin`)

Builtin	Role
`request.security`	Other symbol / timeframe expression
`request.security_lower_tf`	Lower-TF array expansion
`request.dividends` / `earnings` / `splits`	Corporate actions
`request.financial` / `economic` / `quandl`	Fundamentals / external series
`request.currency_rate`	FX conversion helper
`request.seed`	Deterministic pseudo-series
`request.footprint`	Volume footprint object (v6 surface)

`request.security(symbol, timeframe, expression, …)`

Resolution order (simplified):

1. Normalize symbol (series → last element) and timeframe.
2. Try `data_feed.fetch_latest_ohlcv` / ticker.
3. Try `data_provider.fetch`.
4. Fall back to **mock** synthetic prices so evaluation continues offline.

Expression may be a simple series name (`close`) or more complex forms depending on handler paths; production hosts should supply real data for multi-asset accuracy.

Footprint (`FootprintBuiltinsMixin`)

Types `Footprint` and `VolumeRow` expose buy/sell volume, delta, VAH/VAL/POC rows, and per-row imbalance helpers—aligned with the v6 footprint surface inventory.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/input.py`	Input handlers + declarations
`src/pynescript/ast/evaluator/builtins/request.py`	request.* + footprint types
`src/pynescript/ast/evaluator/base.py`	Injects `data_feed` / `data_provider` into context
`backend/runtime.py`	`resolve_request_sources` wiring from chart bars
`tests/test_request_data_feed.py`, `test_datafeed*.py`	Feed contracts

Invariants & edge cases

1. **Inputs are pure values at runtime.** Titles matter only for override keys and UI metadata—not for Pine type identity.
2. **Mocks are intentional.** Silent mock fallback is not a hard error; production must validate feed wiring.
3. **Dynamic symbols.** List/series symbols resolve to the latest element—supports loops constructing ticker ids.
4. **Lower TF.** `request.security_lower_tf` returns array-like structures; length scales with simulated lower-TF density when mocking.
5. **Compile mode.** `input.*` defaults lower into numeric compile; live `request.security` is interpret-centric—do not assume multi-asset compile coverage.

Worked examples

Parameterized length

```
//@version=5
indicator("len")
len = input.int(14, "Length", minval=1)
plot(ta.sma(close, len))
```

Host:

```
ev._input_overrides = {"Length": 21}
```

Multi-timeframe close

```
htf = request.security(syminfo.tickerid, "D", close)
plot(htf)
```

With `Runtime.run(..., data_provider=...)`, daily closes come from the provider; without it, mocks apply.

Failure modes

Symptom	Cause
Always ~100 mock prices	No feed/provider; or fetch exceptions swallowed
Input override ignored	Title string mismatch (including empty title)
Footprint fields zero	`request.footprint` without configured footprint data
Lookahead surprises	Host data alignment / gaps—not automatic TV replay guarantees

See also

- Builtins hub
- Pro API run endpoint
- Runtime hub

LIBRARIES

title: "Libraries" description: "In-process library export/import, LibraryRegistry, and evaluate-order requirements."

LIBRARIES

Abstract

TradingView libraries publish as `namespace/Name/version` with `export`ed members. PYNE implements an **in-process** analogue: evaluate a `library("Title")` script (or register source by path), collect exports (`export const`, exported functions, types), then `import` binds an alias to a `LibraryModule` for `alias.member` access. There is no network publish step—the registry is the host's responsibility.

Conceptual model

```
flowchart LR
  LibSrc[library script] --> Eval[Evaluate once]
  Eval --> Exp[pending exports]
  Exp --> Reg[LibraryRegistry]
  Imp[import ns/Name/version as alias] --> Reg
  Reg --> Mod[LibraryModule]
  Mod --> Use[alias.member in consumer]
```

Interface surface

Library script

```
//@version=5
library("MyMath")

    a + b
```

On script completion, `StatementEvaluator` finalizes `_pending_library_exports` onto the active `LibraryModule` and calls `LibraryRegistry.register`.

Consumer import

```
//@version=5
indicator("use")

plot(M.add(close, M.PHI))
```

Resolution (`LibraryRegistry.lookup`):

1. Exact `(namespace, name, version)` if provided and registered.
2. Else title-only match (`MyMath`) for local evaluation workflows without publisher path.

API on the evaluator

```
ev.register_library_source(namespace, name, version, source_str)
mod = ev.lookup_library(namespace=..., name=..., version=...)
```

`register_library_source` stores Pine text for **lazy** load on import; evaluating a library script eagerly populates exports.

`LibraryModule`

Dataclass with `title`, optional `namespace` / `version`, and `exports: dict[str, Any]`. Attribute access reads exports; missing members raise `AttributeError` with a clear message. `NameEvaluator` routes `alias.member` when the base value is a `LibraryModule`.

Internals

Path	Role
`src/pynescript/ast/evaluator/libraries.py`	`LibraryModule`, `LibraryRegistry`
`src/pynescript/ast/evaluator/statements.py`	export registration, import visit, library finalize
`src/pynescript/ast/evaluator/base.py`	registry fields on `BaseEvaluator`
`src/pynescript/ast/evaluator/__init__.py`	`register_library_source`, `lookup_library`
`tests/test_library_export_import.py`	Round-trip coverage

Exportable values include callables (user functions), constants, and type-related bindings depending on declaration paths. Exported methods participate in the same call machinery as local functions once retrieved from the module.

Invariants & edge cases

1. **Evaluate library before consumer** (or register source for lazy import). Empty registry → import failure.
2. **Same evaluator instance** (or shared registry) must see both scripts—registries are not process-global unless the host shares them.
3. **Version is part of the path key.** Mismatched version does not fall back to another version automatically when namespace is specified.
4. **Title registration** always updates `_by_title` so simple local titles work without publisher metadata.
5. **No TV account auth.** Publishing/versioning outside the process is out of scope.

Worked examples

Explicit registration

```
from pynescript.ast.evaluator import NodeLiteralEvaluator

ev = NodeLiteralEvaluator()
ev.evaluate_script(library_source)   # registers by title
ev.evaluate_script(consumer_source)  # import by title or path
```

Path registration without pre-eval

```
ev.register_library_source("user", "MyMath", 1, library_source)
# import user/MyMath/1 loads and evaluates source on demand
```

Failure modes

Symptom	Cause
Import resolves to nothing	Library never registered / wrong title
`Library 'X' has no exported member 'y'`	Missing `export` or typo
Stale exports	Re-evaluated library without re-import; host cache
Version conflict	Multiple libs same title—last `register` wins for title key

See also

- Expressions & statements
- Builtins declarations
- End-user library API guide

EVENTS

title: "Strategy events" description: "StrategyEvent shape, emission points, parity corpus, and host serialization."

STRATEGY EVENTS

Abstract

A **strategy event** is the immutable, structured record of one `strategy.*` action (or fill side-effect) at a specific bar. Events are the wire format between PYNE's Python evaluator, the TypeScript `pine-worker` port, and downstream HOOX trade workers. Changing the dataclass without updating the TS mirror and parity fixtures is a contract break.

Conceptual model

```
flowchart LR
  Builtin[strategy.entry / order / …] --> Emit[Construct StrategyEvent]
  Fill[process_pending_orders] --> Emit
  Emit --> Buf[StrategyState._events]
  Host[Bar loop] --> Drain[drain_events]
  Buf --> Drain
  Drain --> Dict[to_dict + script_id/run_id]
  Dict --> PW[pine-worker parity]
  Dict --> HOOX[trade-worker]
```

Interface surface

`StrategyEvent` fields

Defined in `src/pynescript/ast/evaluator/events.py`:

Field	Type (logical)	Notes
`kind`	`entry` | `exit` | `close` | `close_all` | `cancel` | `cancel_all` | `order`	Discriminator
`id`	`str | None`	Order / entry id
`direction`	`long` | `short` | `None`	
`qty`	`float | None`	
`order_type`	`market` | `limit` | `stop` | `None`	Stop-limit may appear as composite handling elsewhere
`limit` / `stop`	`float | None`	Prices when relevant
`oca_name`	`str | None`	OCA group
`comment`	`str | None`	Includes system comments (`risk_blocked` , `fill:…`)
`bar_index`	`int`	From context at emit
`bar_time`	`int`	From context
`ohlc`	4-tuple → list in JSON	Bar snapshot
`script_id`	`str`	Host-stamped (source hash)
`run_id`	`str`	Host-stamped run uuid

Frozen dataclass → `to_dict()` always includes keys (JSON nulls for unspecified fields) so parity diffs are stable.

Emission sources

1. **Explicit builtins** — `strategy.entry` , `exit` , `close` , `close_all` , `cancel` , `cancel_all` , `order` push events as they execute.
2. **Broker fills** — `process_pending_orders` emits fill-related `order` / `entry` / `cancel` (OCA) events.
3. **Risk gates** — blocked entries may emit with `comment="risk_blocked"` .

Host collection pattern

```
evaluator.reset_events()
# ... process_pending_orders + visit(tree)
for ev in evaluator._strategy_state.drain_events():
    d = ev.to_dict()
    d["script_id"] = script_id
    d["run_id"] = run_id
    all_events.append(d)
```

`Runtime.run` implements this and returns events in the response envelope.

Compile path

`CompileStrategyBroker` accumulates plain dict events (`__events` in the return mapping) with the same field names for kinds it supports—use for throughput; full kind coverage trails the interpreter.

Internals

Path	Role
`src/pynescript/ast/evaluator/events.py`	`StrategyEvent`
`src/pynescript/ast/evaluator/builtins/strategy.py`	Emit sites + `drain_events`
`src/pynescript/compiler/strategy_broker.py`	Compile-mode `_emit`
`tests/test_strategy_events.py`	Unit behavior
`tests/test_parity.py` + `tests/fixtures/parity/`	Cross-port oracle
`pine-worker/src/evaluator/events.ts`	TS twin

Parity corpus scripts (`strategy_01_entry_long.pine` , ...) regenerate expected JSON via:

```
python tests/fixtures/parity/generate_fixtures.py
```

Tests strip `script_id` / `run_id` before compare.

Invariants & edge cases

1. **Immutability.** Do not mutate events after construction; drain copies and clears the buffer.
2. **Per-bar reset.** `reset_events` / drain prevents cross-bar leakage in tests that share evaluators.
3. **OHLC list vs tuple.** `to_dict` converts to list for JSON round-trips.
4. **Kind vocabulary is closed.** New kinds require TS + fixtures + this doc.
5. **Not alerts.** `alert()` / `alertcondition()` use a separate alerts channel—not `StrategyEvent` .

Worked examples

Expected single entry

Script enters long on a fixed bar; fixture JSON lists one event:

```
{
  "kind": "entry",
  "id": "L",
  "direction": "long",
  "qty": 1.0,
  "order_type": "market",
  "bar_index": 5
}
```

(plus nullables and ohlc—see real fixtures for full keys).

Cancel after OCA

Fill of one OCA sibling yields `cancel` events for others with `comment` reflecting `oca_cancel` / `oca_reduce`.

Failure modes

Symptom	Cause
Empty events with live strategy	Forgot drain; or strategy never called
Parity flake on ids	Comparing `script_id` / `run_id`
TS/Python mismatch	Field rename without dual update
Duplicate events	Not clearing buffer; double `process_pending_orders`

See also

- Strategy builtins
- Strategy broker
- pine-worker
- HOOX docs

ALERTS

title: "Alerts" description: "alert() / alertcondition() engine, host export, and L2 HTTP webhooks (Pro API + pyne-worker)."

ALERTS

Abstract

Pine `alert()` and `alertcondition()` fire into a **host side-channel**, not strategy events. The evaluator records structured firings with TradingView-style frequency rules. Hosts (Pro API and **pyne-worker**) export them on evaluate responses and can **POST** them to HTTP webhooks (roadmap **L2**).

Conceptual model

```
flowchart LR
  P[Pine alert / alertcondition] --> E[AlertsMixin]
  E --> L[_triggered_alerts]
  L --> R[Runtime.export]
  R --> API["/run alerts[]"]
  R --> WH[HTTP webhook batch]
```

Layer	Module
Engine	`src/pynescript/ast/evaluator/builtins/alerts.py`
Pro API host	`backend/runtime.py` + `backend/alert_forwarder.py`
Edge host	pyne-worker `runtime.py` + `alert_engine.py` + `alert_forwarder.py`

Pine surface

`alert(message, freq)`

- Default `freq`: `alert.freq_once_per_bar` (canonical token `once_per_bar`).
- Constants (also bare strings): `alert.freq_once_per_bar`, `alert.freq_once_per_bar_close`, `alert.freq_all`.
- **once_per_bar** — first fire per bar for the same (source, title, message, freq) key.
- **once_per_bar_close** — only when the host marks the bar confirmed (`barstate.isconfirmed` / `islast`; hosts without barstate treat bars as confirmed).
- **all** — every call.

`alertcondition(condition, title, message)`

- Records each evaluation in `alert_conditions` (debug / UI).
- When `condition` is true, also **fires** a host alert with `source: "alertcondition"` and once-per-bar semantics.

Event shape (`to_dict` / API)

```
{
  "message": "cross up",
  "freq": "once_per_bar",
  "bar_index": 42,
  "time": 1700000000000,
  "title": "optional",
  "source": "alert"
}
```

`source` is `"alert"` or `"alertcondition"`. Hosts stamp `script_id` / `run_id` (and edge cron may add `symbol` / `timeframe`).

Host export

Pro API (`POST /run`)

- Interpret path: full history of firings in `alerts` (and optional `alert_conditions`).

- Compile path: `alerts: []` today (alert side effects are interpret-only).
- See POST /run.

pyne-worker

- `POST /run` returns the same `alerts` array.
- Cron filters to the **last closed bar** before webhook delivery so historical bars do not re-fire every minute.
- Health: `features.alerts`, `features.alert_webhooks`.

L2 webhooks

Both hosts POST JSON to an HTTPS endpoint.

Host	Default URL	Per-request / per-job
Pro API	env `ALERT_WEBHOOK_URL`	body `webhook_url`
pyne-worker	secret/var `ALERT_WEBHOOK_URL`	script/job `webhook_url`

Pro API request flags

Field	Default	Meaning
`webhook_url`	`""`	Override destination
`forward_alerts`	`true`	Skip POST when false
`alert_last_bar`	`true`	Only firings on the last OHLCV bar
`alert_batch`	`true`	One batch body vs one POST per alert

Optional timeout: env `ALERT_WEBHOOK_TIMEOUT` (seconds, default 10).

Batch body

```
{
  "type": "pine_alert_batch",
  "source": "pyne-pro-api",
  "count": 1,
  "content": "cross up",
  "alerts": [
    {
      "type": "pine_alert",
      "source": "pyne-pro-api",
      "message": "cross up",
      "freq": "once_per_bar",
      "alert_source": "alert",
      "bar_index": 42,
      "time": 1700000000000
    }
  ]
}
```

Edge worker uses `"source": "pyne-worker"`. Discord-friendly `content` is set when a message is present.

Response meta

Successful `/run` may include:

```
"alert_forward": {
  "forwarded": 1,
  "failed": 0,
  "filter": "last_bar",
  "url": "https://hooks.example.com/pine",
  "batch": true,
  "count": 1
}
```

Webhook failures never fail the evaluate status; inspect `alert_forward` / `alert_forward_error`.

Worked example

```
curl -sS -X POST http://127.0.0.1:5002/run \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=5\nindicator(\"a\")\nif bar_index == last_bar_index\n    alert(\"fire\")\
    "mode": "interpret",
    "webhook_url": "https://hooks.example.com/pine",
    "data": [
      {"time": 1, "open": 1, "high": 2, "low": 0.5, "close": 1.5, "volume": 10},
      {"time": 2, "open": 1.5, "high": 2.5, "low": 1.0, "close": 2.0, "volume": 12}
    ]
  }'
```

Invariants

1. **Not strategy events** — trade mesh uses `events[]`; alerts are independent.
2. **Per-run clear** — hosts call `clear_alerts()` so runs do not leak firings.
3. **Last-bar default for webhooks** — full-history still available in the `alerts` response field.
4. **Compile path** — prefer `mode: "interpret"` (or auto fallback) when you need alerts.

See also

- Strategy events
- POST /run
- Pro API usage
- Runtime bridge
- Roadmap (L2 ✅)

OVERVIEW

title: "Compiler overview" description: "Source-to-source compilation: CompilerVisitor, numeric vs object mode, and engine API."

COMPILER OVERVIEW

Abstract

The compile path lowers Pine's ASDL AST to **executable Python** that walks bars with an explicit index (`__bar_idx`) over contiguous `numpy` arrays—avoiding per-node visitor overhead. Pure numeric scripts wrap the loop in `@numba.njit`; scripts that touch UDTs, maps, or drawings select **object mode** (still a tight Python/numpy loop, no AST walk). The public façade is `pynescript.compiler.engine` (`transpile`, `compile_script`, `run_script`), also reachable as `Runtime.run(..., mode="compile")`.

This is an MVP with growing surface—not a full substitute for the interpreter on every script.

Conceptual model

```
flowchart LR
  Pine[Pine source] --> Parse[parse AST]
  Parse --> CV[CompilerVisitor]
  CV -->|no UDT/map/draw| Num["@numba.njit loop"]
  CV -->|UDT/map/draw/strategy| Obj[Object-mode loop]
  Num --> Exec[execute_script_compiled]
  Obj --> Exec
  OHLCV[OHLCV arrays] --> Exec
  Exec --> Out[plots / drawings / events]
```

Interface surface

```
from pynescript.compiler.engine import transpile, compile_script, run_script, has_numba

src = transpile(pine_text)           # generated Python string
cs = compile_script(pine_text)       # CompiledScript
result = cs.run(open, high, low, close, volume)
# result: dict of plot title → float64 array, optional __drawings, __events, …
```

API	Role
`transpile`	Parse + visit → source string (inspect/debug)
`compile_script`	`exec` generated code, warm-up call, return `CompiledScript`
`run_script`	One-shot compile+run (prefer cache `CompiledScript` for batches)
`has_numba`	Numeric mode prerequisite
`prewarm_numba_builtins` / `prewarm_scripts`	Host cold-start (H2 warm path)
`compile_cache_stats` / `compile_deploy_config`	Cache + deploy diagnostics

`CompiledScript` fields: `source`, `generated_code`, `execute`, `plot_titles`, `object_mode`, optional `nopython_fallback_reason`.

Warm compile (product defaults)

Pro API `/run` defaults to **`mode=auto`** (compile when safe, interpret on fallback). Disk IR cache is **on** (`PYNE_COMPILE_DISK_CACHE=1`); Docker sets `PYNE_COMPILE_CACHE_DIR=/data/compile-cache`. Operators call `POST /compile/prewarm` or `pynescript prewarm` so first interactive latency skips Numba cold JIT. See COMPILER_PLAN.md for SLOs and env flags.

Mode selection

`CompilerVisitor` sets `object_mode = True` when it sees:

- User-defined types / enums
- Map operations
- Drawing APIs / certain plot variants
- Strategy usage (object mode + `CompileStrategyBroker`)

Otherwise **numeric mode** requires Numba installed.

What numeric mode lowers today

Series assigns, arithmetic/logic, history (`close[n]`), `if` / `for` / `while`, selected `ta.*` (`sma`, `ema`, `rsi`, `highest`, `lowest`), scalar math helpers, `plot`, `input.*` defaults, `var` / `varip` carry—executed under `@numba.njit(cache=False)` (`cache=False` because code is `exec`'d from a string without a stable disk locator).

Object mode extras

- UDT instances as field dicts
- Maps as Python dicts
- `__drawings` list of structured events
- Optional `__strategy` broker: pending fills, position, `__events`, equity snapshots

Internals

Path	Role
`src/pynescript/compiler/compiler.py`	`CompilerVisitor`, emit numeric/object
`src/pynescript/compiler/numba_builtins.py`	JIT kernels
`src/pynescript/compiler/strategy_broker.py`	Compile broker
`src/pynescript/compiler/engine.py`	Façade + result normalize
`backend/runtime.py`	`_run_compiled` envelope → plots/series
`tests/test_compiler_numba.py`, `test_compiler_objects.py`, `test_compiler_strategy.py`	Coverage
`docs/COMPILER_PLAN.md`	Design history and remaining work

Data layout

Unlike interpreter `PineSeries` deques:

```
open_arr, high_arr, low_arr, close_arr, vol_arr   # inputs
user_arr = np.full(n_bars, np.nan)                 # or dtype=object for UDT
plot_i   = np.full(n_bars, np.nan)
for __bar_idx in range(n_bars):
    user_arr[__bar_idx] = …
    plot_i[__bar_idx] = …
```

`var` lowers to "init only when still na / first write" patterns so carry matches Pine without declaration sets.

Invariants & edge cases

1. **Same AST front-end.** No second parser—compile bugs are lowering bugs.
2. **OHLCV length equality** enforced in `CompiledScript.run`.
3. **Warm-up** ignores exceptions on a dummy 16-bar series (JIT or first-run).
4. **Result normalization** converts Numba typed maps to plain dicts; `__drawings` stays a Python list.
5. **Interpret remains default.** Full strategy analytics, request.*, and exotic builtins stay interpret-first until lowered.

Worked example

Pine:

```
//@version=5
indicator("c")
s = ta.sma(close, 14)
plot(s, "sma")
```

Generated shape (illustrative):

```
@numba.njit(cache=False)
def execute_script_compiled(open_arr, high_arr, low_arr, close_arr, vol_arr):
    n_bars = len(close_arr)
    s_arr = np.full(n_bars, np.nan)
    plot_0 = np.full(n_bars, np.nan)
    for __bar_idx in range(n_bars):
        s_arr[__bar_idx] = numba_sma(close_arr, 14, __bar_idx)
        plot_0[__bar_idx] = s_arr[__bar_idx]
    return {'sma': plot_0}
```

Failure modes

Symptom	Cause
`numba is required for numeric compile mode`	Install numba or simplify script into object mode triggers
Empty generated code	Visitor returned blank—unsupported top-level shape
Missing `execute_script_compiled`	Emit bug or failed `exec`
Silent semantic drift	Kernel seed differences—diff against interpret on fixtures
Strategy metrics missing	Use interpret or check `__events` / broker fields only

Remaining work (summary)

Expand Numba `ta.*` surface, richer drawings, nested UDT methods, disk cache of generated modules, and systematic interpret↔compile parity tests per lowered construct. See `docs/COMPILER_PLAN.md` for the living checklist.

See also

- Numba path
- Strategy broker
- Runtime hub
- Numerical validation

NUMBA

title: "Numba path" description: "JIT bar loops, numba_builtins kernels, NA-as-NaN, and performance characteristics."

NUMBA PATH

Abstract

Numeric compile mode emits a single `@numba.njit` function whose body is the script's bar loop. Pine builtins that cannot cross the JIT boundary are replaced by **hand-written kernels** in `numba_builtins.py` that take full arrays plus the current index. The result is large speedups on multi-thousand-bar series after a one-time JIT warm-up—at the cost of a restricted language subset and `np.nan` as the missing-value sentinel.

Conceptual model

```
flowchart TB
  CV[CompilerVisitor] --> Emit["@numba.njit execute_script_compiled"]
  Emit --> NB[numba_builtins]
  NB --> SMA[numba_sma]
  NB --> EMA[numba_ema]
  NB --> RSI[numba_rsi]
  NB --> HL[numba_highest / lowest]
  NB --> Math[numba_nz / abs / min / max]
  Emit --> Loop["for __bar_idx in range(n)"]
  Loop --> SMA
```

Interface surface

Kernels (all `@numba.njit(cache=True)` at definition site; generated entry uses `cache=False`):

Kernel	Semantics sketch
`numba_sma(arr, period, i)`	Mean of `arr[i-period+1 : i]`; `nan` if warm-up or any window `nan`
`numba_ema(arr, period, i)`	SMA seed over first `period` bars, then EMA to `i`
`numba_rsi(arr, period, i)`	Average gain/loss form; 100 if avg loss 0
`numba_highest` / `numba_lowest`	Window extrema; window clamps at 0
`numba_nz(val, replacement)`	Replace `nan`
`numba_abs` / `numba_min` / `numba_max`	Scalar helpers

Generated code imports `from pynescript.compiler.numba_builtins import *` so call sites are bare names.

History access

`close[n]` lowers to array indexing at `__bar_idx - n` with bounds → `nan` (parallel to interpreter `None`, different sentinel).

Control flow

`if` / `for` / `while` emit Python control flow **inside** the njit function—supported Numba subset only (no heap objects, no Python lists of mixed types).

Internals

Path	Role
`src/pynescript/compiler/numba_builtins.py`	Kernels
`src/pynescript/compiler/compiler.py`	`_emit_numeric_mode`, call lowering
`src/pynescript/compiler/engine.py`	Requires `_HAS_NUMBA` when not object_mode
`tests/test_compiler_numba.py`	Correctness

Expanding coverage is primarily **new kernels + visitor call mapping**, not changes to the bar-loop skeleton.

Invariants & edge cases

1. **No Python objects in numeric mode.** Strings, UDTs, maps, drawings force object mode before njit is applied.
2. **Warm-up cost.** First `compile_script` invokes a dummy run; production servers should cache `CompiledScript`.
3. **EMA/RSI definitions** are explicit in source—document any intentional deviation when comparing to interpret `ta.*` for edge seeds.
4. **`cache=False` on entry.** Do not expect Numba's on-disk cache to key generated functions across processes without further work.
5. **Anaconda static libpython.** Packaging notes in AGENTS/build docs apply when shipping JIT-heavy binaries.

Worked examples

Benchmark mental model

```
interpret:  O(bars × ast_nodes × python_dispatch)
numeric:    O(bars × lowered_ops) in machine code after JIT
```

For simple SMA scripts, internal notes cite order-of-magnitude speedups versus list-based interpretation on long series (see `docs/COMPILER_PLAN.md` qualitative claims).

Using from Pro API

```
Runtime(...).run(source, ohlcv, mode="compile")
```

Converts bar dicts → arrays, compiles, reshapes plots into the standard envelope, flags `object_mode` when applicable.

Failure modes

Symptom	Cause
TypingError from Numba	Unsupported construct leaked into numeric emit
All-`nan` plots	Period longer than series; or history index bugs
ImportError numba	Environment missing dependency
Divergent RSI vs interpret	Seed / average method—file fixture and compare both paths
Object mode unexpectedly	Visitor detected drawing/UDT/map—inspect `visitor.object_mode`

Performance notes

- Prefer **one compile, many runs** with different OHLCV.
- Keep scripts in the numeric subset for realtime ticks.
- Object mode still wins over AST walking for UDT-heavy scripts but will not match peak njit throughput.

See also

- Compiler overview

- Technical builtins (interpret)
- Series & history

STRATEGY BROKER

title: "Compile strategy broker" description: "CompileStrategyBroker: pending orders, OHLC fills, OCA, commission, and event export."

COMPILE STRATEGY BROKER

Abstract

When `CompilerVisitor` detects strategy usage, object-mode emission instantiates **`CompileStrategyBroker`**—a lightweight broker aligned with the interpreter's fill-before-script cadence, but designed for generated loops (no AST, no full `StrategyState` metric surface). It supports market entry/close and pending limit/stop/stop-limit orders with per-bar OHLC triggers, OCA cancel/reduce, commission, and slippage.

Conceptual model

```
sequenceDiagram
  participant Loop as Generated for __bar_idx
  participant B as CompileStrategyBroker
  Loop->>B: set_bar(idx, OHLC)
  Loop->>B: process_pending_orders(OHLC)
  Loop->>B: entry/close/order… (script body)
  Note over B: events[], pending_orders, position_*
  Loop->>B: to_events() at return
```

Interface surface

Construction (from `strategy()` kwargs when lowered)

```
CompileStrategyBroker(
    initial_capital=100_000.0,
    commission_value=0.0,
    commission_type="percent",  # percent | cash_per_order | cash_per_contract
    slippage_ticks=0,
    mintick=0.01,
)
```

Bar context

```
broker.set_bar(bar_index, bar_time, mark, open_=…, high=…, low=…, close=…)
broker.process_pending_orders(open_=…, high=…, low=…, close=…)
```

Orders

Method	Role
`entry(...)`	Market or pending entry; reverse flattens opposite
`close` / `close_all`	Reduce/flat with optional price
Pending book	`PendingOrder` with type, direction, qty, limit/stop, OCA, `max_fill_per_bar`, `is_entry`

Fill logic (`_trigger_price`)

Type	Long trigger	Short trigger	Fill price idea
market	—	—	close (immediate path)
limit	`low <= limit`	`high >= limit`	gap-aware vs open
stop	`high >= stop`	`low <= stop`	gap-aware vs open
stop-limit	stop traded and limit available	symmetric	limit price

Partial fills: `max_fill_per_bar` caps quantity per bar; residual stays pending.

OCA

After a fill, siblings in the same `oca_name`:

- **`cancel`** — remove sibling, emit `cancel`
- **`reduce`** — decrease sibling qty by fill size
- **`none`** — no interaction

Economics

- **Slippage:** `± slippage_ticks * mintick` by direction on fills.
- **Commission:** percent of notional, cash per order, or cash per contract.
- **Position:** signed `position_size`, `position_avg_price`, `netprofit`, `equity` helpers for return payload.

Events

`_emit` appends dicts with keys compatible with strategy event serialization (`kind`, `id`, `direction`, `qty`, `order_type`, `limit`, `stop`, `oca_name`, `comment`, `bar_index`, `bar_time`, `ohlc`). Object-mode return includes `'__events': __strategy.to_events()` plus position/netprofit/equity snapshots.

Internals

Path	Role
`src/pynescript/compiler/strategy_broker.py`	Broker implementation
`src/pynescript/compiler/compiler.py`	Emits `__strategy` wiring in object mode
`src/pynescript/ast/evaluator/builtins/strategy.py`	Interpreter counterpart (richer)
`tests/test_compiler_strategy.py`	Compile strategy coverage

NA helpers treat `None`, blank/`na` strings, and float NaN as missing prices when classifying order types.

Invariants & edge cases

1. **Same order as interpreter:** `set_bar` → `process_pending_orders` → script body.
2. **Not a full metric engine.** Prefer interpret mode for `strategy.closedtrades.profit(i)` style analytics until parity is complete.
3. **Direction normalization** accepts `strategy.long`, `long`, `1`, `buy`, etc.
4. **Reverse on opposite entry** via `close_all(comment="reverse")` then open.
5. **Event dicts are mutable lists**—fine for run output; not frozen dataclasses.

Worked examples

Generated prologue (illustrative)

```
__strategy = CompileStrategyBroker(initial_capital=100000.0)
for __bar_idx in range(n_bars):
    __strategy.set_bar(__bar_idx, 0, float(close_arr[__bar_idx]),
        open_=float(open_arr[__bar_idx]), high=float(high_arr[__bar_idx]),
        low=float(low_arr[__bar_idx]), close=float(close_arr[__bar_idx]))
    __strategy.process_pending_orders(...)
    # lowered strategy.entry / close calls
return {..., '__events': __strategy.to_events(),
        '__position_size': __strategy.position_size,
        '__netprofit': __strategy.netprofit,
        '__equity': __strategy.equity}
```

Limit buy pending

```
strategy.entry("L", strategy.long, limit=low - syminfo.mintick)
```

Object-mode lowers to a pending limit; subsequent bars fill when `low` trades through.

Failure modes

Symptom	Cause
No fills	Triggers never met; or market path not invoked
OCA sibling remains	`oca_type` none / name mismatch
PnL vs interpret drift	Commission/slippage/mintick defaults differ
Missing events in API envelope	Host only maps plots—not `__events`

See also

- Strategy builtins (interpret)
- Events
- Compiler overview
- Numba path