

INTHON Specification

*Reverse-Engineered Python Reference,
Agent-Level Language Design,
and Production Readiness Checklist*

A complete reference-grade reverse engineering of the CPython programming language paired with the formal specification of **INTHON** — the agent-level language layer for AI-native computation, capability-bounded tool orchestration, and deterministic execution tracing. Includes a full grammar appendix, opcode reference, self-evaluation primitive design, and a granular production-readiness checklist mapped to the current state of the INTHON project.

PREPARED FOR
AI Agent (Builder) · Human Reviewer

DOCUMENT VERSION
1.0 (Production Spec Draft)

REFERENCE PROJECT
harvatechs/inthon · v0.2

DISTRIBUTION
Open · Apache 2.0

Table of Contents

1. Document Conventions	12
1.1 Status Keywords	12
1.2 Grammar Notation	12
1.3 Code Notation	12
1.4 References	13
2. Executive Summary	13
3. Python Architecture Overview	16
3.1 The Six-Stage Pipeline	16
3.2 The Frame Object	16
3.3 The Code Object	16
3.4 The Interpreter State	17
4. Lexical Structure of Python	17
4.1 Physical and Logical Lines	17
4.2 Indentation and the INDENT/DEDENT Tokens	18
4.3 Token Categories	18
4.4 Identifiers and Keywords	18
4.5 Literals	18
4.6 Operators and Delimiters	19
4.7 Comments and Encoding Declarations	19
4.8 INTHON's Lexical Departures	19
5. The PEG Parser (Python 3.10+)	19
5.1 PEG vs LL(1) — Why the Switch	19
5.2 The PEG Grammar File	20
5.3 Soft Keywords	20
5.4 Error Recovery and SyntaxError	20
5.5 The Parse Tree → AST Transform	20
5.6 INTHON's Parser Choice: Lark	20

6. AST Node Catalog	20
6.1 Module and Top-Level Nodes	21
6.2 Expression Nodes	22
6.3 Operator and Context Nodes	23
6.4 Function Definition Nodes	23
6.5 Pattern Matching Nodes (3.10+)	23
6.6 AST Visitors	23
7. Semantic Analysis & Symbol Tables	23
7.1 Scope Rules	23
7.2 The symtable Module	24
7.3 Name Binding Rules	24
7.4 Free Variables and Cell Variables	24
7.5 Comprehension Scoping	24
7.6 Class Body Scoping Quirk	24
7.7 Future Annotations (PEP 563)	24
8. The Compiler: AST → Bytecode	25
8.1 Phase 1: AST to CFG	25
8.2 Phase 2: CFG Optimization	25
8.3 Phase 3: Stack Depth Analysis	25
8.4 Phase 4: Bytecode Emission	25
8.5 Compilation of Specific Constructs	25
8.5.1 Comprehensions	25
8.5.2 Closures	26
8.5.3 Async Functions	26
8.5.4 Match Statements	26
8.6 The Code Object's Final Shape	26
9. Bytecode & the ceval Main Loop	26
9.1 Instruction Format	26
9.2 The Dispatch Loop	26
9.3 Frame Stack and Calls	27
9.4 Inline Caching and Specialization (PEP 659)	27

9.5 The Eval Breaker	27
9.6 Exception Handling	27
9.7 Generators and Coroutines	27
10. Per-Opcode Semantics Reference	27
10.1 Load and Store Opcodes	28
10.2 Arithmetic and Comparison Opcodes	28
10.3 Control Flow Opcodes	29
10.4 Call and Function Opcodes	29
10.5 Container Construction Opcodes	30
10.6 Exception Opcodes	30
10.7 Coroutine and Generator Opcodes	31
10.8 Import and Module Opcodes	31
11. Object Model & Type System	31
11.1 The Type Object and tp_slots	32
11.2 The Descriptor Protocol	32
11.3 The Method Resolution Order (C3)	32
11.4 __new__ vs __init__	32
11.5 Slots vs __dict__	32
11.6 Special Methods (Dunder Protocol)	33
11.7 Metaclasses	33
12. Memory Management	33
12.1 pymalloc — The Object Allocator	33
12.2 Reference Counting	33
12.3 The Cycle Collector (gc Module)	34
12.4 Object Lifecycle and Finalization	34
12.5 Immortal Objects (3.12+)	34
12.6 INTHON's Memory Model	34
13. The GIL & Threading Model	34
13.1 Why the GIL Exists	34
13.2 The Eval Breaker and GIL Release	35
13.3 I/O and the GIL	35

13.4 PEP 703: Making the GIL Optional (3.13+)	35
13.5 Sub-interpreter GILs (PEP 684, 3.12+)	35
13.6 INTHON's Concurrency Model	35
14. The Import System	36
14.1 The Import Protocol	36
14.2 The Default Meta Path Finders	36
14.3 PathFinder and PathHooks	36
14.4 The ModuleSpec Object	36
14.5 Cached Bytecode (.pyc Files)	36
14.6 Multi-Phase Module Initialization (PEP 489)	37
14.7 INTHON's PyBridge as a Meta Path Finder	37
15. Exceptions & Tracebacks	37
15.1 Exception Chaining	37
15.2 The Traceback Object	37
15.3 Exception Groups (PEP 654, 3.11+)	38
15.4 Exception Raising Cost	38
15.5 C-Level Exceptions (PyErr_*)	38
15.6 INTHON's Exception Model	38
16. The C-API & ABI Stability	38
16.1 The Stable ABI (PEP 384)	38
16.2 The Limited API	39
16.3 HPy — A Safer C-API	39
16.4 Multi-Phase Init (PEP 489)	39
16.5 The GIL State API	39
16.6 INTHON and the C-API	39
17. Sub-interpreters & Isolation	39
17.1 The Interpreter State	40
17.2 What is and isn't Isolated	40
17.3 PEP 734: The interpreters Module (3.12+)	40
17.4 Why Sub-interpreters Matter for INTHON	40
17.5 Limitations of Sub-interpreters	41

18. INTHON Design Philosophy	43
18.1 The Hierarchy of Language Levels	43
18.2 The Three Pillars	43
18.2.1 Token-Efficient Grammar	43
18.2.2 Capability-Based Sandbox	43
18.2.3 Traceable Execution	43
18.3 What INTHON is Not	44
18.4 The Origin: Why 'INTON' and Not 'AITHON'	44
19. INTHON Lexical Structure	44
19.1 Keywords	44
19.2 Identifiers	45
19.3 Literals	45
19.4 Operators and Delimiters	46
19.5 Comments	46
19.6 Whitespace	46
20. Type System	46
20.1 Primitive Types	46
20.2 Collection Types	47
20.3 Agent and Execution Types	47
20.4 Type Annotations	47
20.5 Type Checking	48
20.6 What is Missing	48
21. Statements & Control Flow	48
21.1 Variable and Constant Declarations	48
21.2 Assignment	48
21.3 Conditional	48
21.4 For Loop	49
21.5 While Loop	49
21.6 Function Declaration	49
21.7 Guard Statement	49
21.8 Retry Statement	50

21.9 Return Statement	50
22. Expressions & Operators	50
22.1 Operator Precedence Table	50
22.2 Call Expressions	51
22.3 Attribute and Index Access	51
22.4 List, Dict, and Tuple Literals	51
22.5 String Concatenation	51
22.6 Member Method Chains	51
23. The Agent Block Specification	52
23.1 Syntax	52
23.2 The Goal Directive	52
23.3 Inputs and Outputs	52
23.4 The use Declarations	52
23.5 The Policy Block	53
23.6 The Plan Block	53
23.7 Agent Invocation	53
23.8 Agent Lifecycle	53
24. Built-in Primitives	54
24.1 Approval Gateways (HITL)	54
24.2 Episodic Memory — remember	54
24.3 Episodic Memory — recall	54
24.4 Episodic Memory — forget	55
24.5 Resilient Retries	55
24.6 Guard	55
24.7 Self-Evaluation (preview)	55
25. PyBridge Sandbox & Module Policy	56
25.1 The Import Hook	56
25.2 The InthonPyObject Proxy	56
25.3 The Module Policy Matrix	56
25.4 Attack Vectors and Defenses	56
25.5 Limitations	57

26. INTHON Compilation Pipeline	57
26.1 Stage 1: Lark Parse	57
26.2 Stage 2: AST Construction	57
26.3 Stage 3: Semantic Analysis	58
26.4 Stage 4: Policy Sandbox Validation	58
26.5 Stage 5: IR Lowering	58
26.6 The IR Format	58
26.7 The VM Compilation	59
27. InthonVM Opcode Reference	59
27.1 Load and Store Opcodes	59
27.2 Arithmetic and Comparison Opcodes	59
27.3 Control Flow Opcodes	60
27.4 Call Opcodes	60
27.5 Container Opcodes	60
27.6 Agent Primitive Opcodes	61
27.7 Trace Emission	62
28. Sandbox & Policy Engine	62
28.1 The Policy Schema	62
28.2 Runtime Enforcement Points	62
28.3 The Policy Frame Stack	63
28.4 The Cost Ledger	63
28.5 Exception Taxonomy	63
28.6 Rollback Semantics	63
29. Episodic Memory Subsystem	64
29.1 The SQLite Schema	64
29.2 The Embedding Pipeline	64
29.3 Cosine Similarity Retrieval	64
29.4 Namespaces	64
29.5 Persistence	64
29.6 The Embedding Model API	65

30. Trace & Replay Format	65
30.1 The Trace Schema	65
30.2 Event Types	66
30.3 Replay Semantics	66
30.4 Trace-to-Source Mapping	66
30.5 The Trace Viewer UI	66
31. Standard Library & Tool Registry	67
31.1 Built-in Functions	67
31.2 The Tool Registry	67
31.3 Built-in Tools	67
31.4 Tool Schema Validation	68
31.5 OpenAPI Autogen (Roadmap)	68
32. Self-Evaluation: Motivation & Threat Model	70
32.1 The Core Idea	70
32.2 Why This Belongs in the Language	70
32.3 The Threat Model	70
32.4 What Self-Eval is Not	70
32.5 The Three Worked Examples	71
33. Self-Evaluation: Grammar Extension	71
33.1 Grammar Productions	71
33.2 Concrete Syntax Example	71
33.3 The Criteria Declaration Block	72
33.4 The Rewriter Declaration Block	72
33.5 Multiple Criteria Sets	72
34. Self-Evaluation: Opcodes & Semantics	72
34.1 SELF_EVAL	72
34.2 INTROSPECT_TRACE	73
34.3 REWRITE_PLAN	73
34.4 The EvalContext Object	74
34.5 Security Considerations	74

35. Self-Evaluation: Worked Examples	74
35.1 Example 1: Cost-Bounded Rewrite	74
35.1.1 The Agent	74
35.1.2 The Failing Execution	75
35.1.3 The Rewriter	75
35.1.4 The Rewritten Execution	75
35.2 Example 2: Hallucination Recovery	75
35.2.1 The Agent	75
35.2.2 The Failing Execution	75
35.2.3 The Rewriter	76
35.2.4 The Rewritten Execution	76
35.3 Example 3: Latency-Bounded Plan Compaction	76
35.3.1 The Agent	76
35.3.2 The Failing Execution	76
35.3.3 The Rewriter	76
35.3.4 The Rewritten Execution	77
36. INTHON vs CPython: Differential Analysis	79
36.1 What INTHON Keeps from CPython	79
36.2 What INTHON Drops from CPython	79
36.3 What INTHON Adds That CPython Lacks	80
36.4 The Inheritance Tax	81
36.5 The Sandbox Gap	81
37. Production-Readiness Framework	83
37.1 The Twelve Pillars	83
37.2 INTHON's Current Status (v0.2)	83
37.3 How to Use the Checklists	83
38. Build Checklist: Frontend (Lexer + Parser)	84
39. Build Checklist: AST + Semantic	85
40. Build Checklist: IR + VM	86

41. Build Checklist: Sandbox + Policy	87
42. Build Checklist: Memory + Tools + Trace	89
43. Build Checklist: Stdlib + Packaging + DX	90
44. Build Checklist: Security Audit & Formal Verification	91
45. Roadmap to INTHON v1.0	92
45.1 Phase Summary	92
45.2 Phase 1: Ecosystem (v0.3)	93
45.3 Phase 2: Developer Experience (v0.4)	93
45.4 Phase 3: Type System (v0.5)	93
45.5 Phase 4: Self-Evaluation (v0.6)	93
45.6 Phase 5: Formal Sandbox (v0.9)	93
45.7 Phase 6: Production Sign-off (v1.0)	93
Appendix A. Full INTHON EBNF (Lark-Ready)	96
Appendix B. Opcode Quick Reference Card	98
B.1 Load and Store (11 opcodes)	98
B.2 Arithmetic and Comparison (10 opcodes)	99
B.3 Control Flow (7 opcodes)	99
B.4 Call and Container (8 opcodes)	100
B.5 Agent Primitives (15 opcodes)	100
Appendix C. Glossary	101
Appendix D. References & Bibliography	103
D.1 CPython Source Tree	103
D.2 Python Language Reference & PEPs	103
D.3 Lark Parser Generator	104
D.4 INTHON Project	104
D.5 Capability-Secure Languages & Object-Capability Model	104
D.6 Agent-Oriented Programming	104
D.7 Bytecode VMs & Compilers	104

D.8 Tooling & Tree-sitter 105

1. Document Conventions

This document is a dual-purpose specification. Its primary consumer is an AI agent tasked with reconstructing the Python programming language from first principles and extending it into the INTHON agent-level language layer. Its secondary consumer is a human compiler engineer or language designer reviewing the work. The tone and structure therefore favor formal specification over tutorial prose: every subsystem is described in terms of its inputs, outputs, invariants, and the concrete code structures that implement it.

The document is partitioned into five parts. Part I is a reference-grade reverse engineering of the CPython implementation, covering lexical structure, the PEG parser, the AST catalog, semantic analysis, the bytecode compiler, the *ceval* main loop, per-opcode semantics, the object model, memory management, the GIL, the import system, the C-API, and sub-interpreter isolation. Part II specifies the INTHON language itself: grammar, type system, statements, expressions, the *agent* block, built-in primitives, the PyBridge sandbox, the compilation pipeline, the InthonVM opcode set, the policy engine, the episodic memory subsystem, the trace format, and the standard tool registry. Part III contains the full design of the self-evaluation primitive. Part IV is a differential analysis against CPython. Part V is a granular production-readiness checklist with the current INTHON status marked per item, plus the roadmap to v1.0.

1.1 Status Keywords

Throughout Part V, every checklist item is tagged with a status keyword indicating the current state of the INTHON implementation as of the v0.2 release. These keywords are scoped to the INTHON project and do not reflect upstream CPython status.

Status	Meaning	Acceptance bar to advance to next status
DONE	Implemented, tested, and documented.	Maintain on regression.
PARTIAL	Implemented but missing tests, docs, or edge cases.	Add the missing surface; re-run conformance suite.
MISSING	Not implemented.	Implement, write acceptance test, document.
N/A	Out of scope for INTHON's design.	Re-evaluate scope annually.

Table 1.1 — Status keyword definitions used throughout Part V.

1.2 Grammar Notation

Grammar rules are written in Lark-compatible EBNF, matching the notation used in the INTHON source repository. Productions use the form *name: rule*. Terminals are written in UPPERCASE; non-terminals in lowercase. The operators `|`, `(...)`, `[...]`, `{...}`, `+`, `*`, and `?` have their standard EBNF meanings. String literals in quotes denote keywords or punctuation. The *%ignore* directive introduces lexer-level skipping rules. Where Lark-specific priorities are used (e.g. *BOOL_LIT.2*), the priority number is preserved verbatim.

1.3 Code Notation

Code blocks are typeset in a monospace face. INTHON source uses the *.inth* extension. Python source uses the standard *.py* extension. Bytecode is shown as uppercase mnemonics with operands in lowercase (*LOAD_CONST 42*). Stack effects are written in the form *[a, b] -> [c]*, meaning the opcode pops *a* and *b* from

the top of the stack and pushes *c*. Where a CPython source file is referenced, the path is relative to the CPython source tree (e.g. *Python/ceval.c*).

1.4 References

Citations to the CPython source tree are version-pinned where the behavior is version-sensitive. PEP references use the form *PEP NNN*. INTHON references use the form *INTHON-N* for internal design notes. External articles and repositories are listed in Appendix D with full URLs. The INTHON project repository at github.com/harvatechs/inthon is the authoritative source for the current implementation; this document specifies the target state for v1.0 and may diverge from the current source.

2. Executive Summary

Python is the dominant high-level language of the present computing era, and its center of gravity has shifted decisively toward AI and data workloads. Yet the language was designed for human authors writing programs that run on bare machines. The emergence of large language models that emit code rather than prose has exposed a structural mismatch: Python's grammar is verbose for LLM generation, its execution model offers no native sandbox, and its tooling story is scattered across frameworks that wrap the language rather than extend it.

INTHON is positioned as the *agent-level* language layer in the same hierarchy that places binary at the machine level, assembly at the low level, and Python/Rust/C++ at the high level. The thesis is straightforward: when humans instruct machines, they write Python; when agents instruct machines, they should write INTHON. The language is engineered around three design pillars that are absent or bolted-on in Python: a token-efficient EBNF grammar tuned for LLM generation, a capability-based sandbox enforced at the VM instruction level, and a deterministic JSON trace that makes every execution replayable and auditable.

This document exists to give an AI agent (and its human reviewers) a single, self-contained reference for both the substrate that INTHON inherits from Python and the surface that INTHON adds on top. Part I is the reverse-engineered Python reference: enough detail to reconstruct CPython's compilation pipeline, bytecode VM, object model, and import system from spec alone. Part II is the INTHON specification proper. Part III introduces the self-evaluation primitive — a first-class language construct that lets an agent introspect its own trace, cost ledger, and plan, then rewrite its plan in response. Part IV is a side-by-side differential analysis. Part V is the production-readiness checklist with the current INTHON status marked per item.

The reader will find that INTHON's v0.2 implementation is substantial but incomplete. The Lark grammar, the AST interpreter, the bytecode VM, the PyBridge sandbox, the policy engine, the SQLite-backed episodic memory, and the JSON trace format are all present and tested. What is missing — and what the checklist in Part V quantifies — is the formal rigor that distinguishes a research prototype from an industrial-grade language: a soundness proof for the type system, a formal model of the sandbox, a conformance test suite, a stable ABI, a tree-sitter grammar, an LSP server, and an external security audit.

How to read this document

If you are the AI agent rebuilding INTHON: read Part I to learn the substrate, Part II to understand the target surface, Part III for the self-eval design, then work through Part V item by item, marking each MISSING item as DONE. If you are a human reviewer: read Part II and Part IV first to evaluate the design, then sample Part I to verify the substrate is correctly understood, then audit Part V to assess production readiness.

PART I

Reverse-Engineered Python Reference

3. Python Architecture Overview

CPython is a stack-based bytecode interpreter implemented in C. Source text enters the system as a byte stream and exits as observable program behavior through a sequence of six logical stages: lexing, parsing, abstract syntax tree construction, semantic analysis, bytecode compilation, and evaluation. Each stage produces a well-defined intermediate representation that is consumed by the next stage, and each representation is introspectable from Python itself via the *token*, *ast*, *symtable*, *dis*, and *sys* modules.

The interpreter is implemented across roughly 350,000 lines of C in the CPython source tree, with the bulk of the runtime concentrated in three directories: *Parser/* contains the tokenizer and PEG parser; *Python/* contains the compiler, the *ceval.c* evaluation loop, the import machinery, and the initialization sequence; *Objects/* contains the implementations of every built-in type from *int* and *list* to *type* itself. The *Include/* directory exposes the public C-API. Understanding the boundaries between these directories is essential for any agent that aims to reconstruct or extend Python.

3.1 The Six-Stage Pipeline

Stage	Source module	Input	Output	Key C files
1. Lex	token, tokenize	source bytes	token stream	Parser/tokenizer.c
2. Parse	parser (internal)	token stream	parse tree (CST)	Parser/parser.c (PEG)
3. AST	ast	parse tree	AST (Python objects)	Python/ast.c
4. Semantic	symtable	AST	symbol table	Python/symtable.c
5. Compile	compile, dis	AST + symtable	code object (bytecode)	Python/compile.c
6. Eval	sys, frame	code object	side effects + return value	Python/ceval.c

Table 3.1 — The six-stage CPython compilation and execution pipeline.

3.2 The Frame Object

Execution is mediated by the frame object (*PyFrameObject* in C, exposed as *frame* in Python 3.11+). A frame holds a reference to the code object being executed, a pointer to the local variable array, the evaluation stack, the caller frame, the currently executing instruction pointer, and the exception state. The interpreter's main loop (*_PyEval_EvalFrameDefault*) dispatches bytecode instructions one at a time, mutating the frame's stack and instruction pointer. Function calls push a new frame; returns pop one. The frame stack is the runtime representation of the call graph.

In Python 3.11, the frame object was substantially reworked. The evaluation stack was moved off the frame object into a separate stack-allocated array, the local variable layout was compacted, and several fields were moved to a separate *_PyInterpreterFrame* structure that is created on the C stack for non-recursive calls. This yielded a 20-30% speedup and reduced the overhead of function calls. The frame object visible to Python code is now constructed lazily only when needed (e.g. by *sys._getframe()* or by an exception traceback).

3.3 The Code Object

The code object (*PyCodeObject*) is the immutable product of compilation. It holds the bytecode (*co_code*), the constant pool (*co_consts*), the names of locals and free variables (*co_varnames*, *co_freevars*, *co_cellvars*), the names of globals and builtins referenced (*co_names*), the source filename and first line number (*co_filename*, *co_firstlineno*), and metadata such as *co_stacksize* (maximum evaluation stack depth) and *co_flags* (whether the function uses **args*, ***kwargs*, is a generator, is a coroutine, etc.).

Every function in Python has an associated code object accessible via *func.__code__*. Modules also have a code object (the module body is compiled as a function that takes the module dict as its globals). The *compile()* builtin returns a code object directly. The *dis* module disassembles a code object into human-readable mnemonics. INTHON's IR is loosely modeled on the code object's structure: a list of instructions, a constant pool, a name table, and per-frame metadata.

3.4 The Interpreter State

Above the frame stack sits the interpreter state (*PyInterpreterState*), which holds the global state of a Python interpreter: the builtins module dict, the import machinery, the GC state, the GIL (in pre-3.12 single-GIL builds), the warning registry, and the list of all threads attached to this interpreter. Above the interpreter state sits the runtime state (*PyRuntimeState*), which holds the list of all interpreters in the process and process-wide resources like the signal handlers and the allocator hooks. Most CPython deployments use a single interpreter; the sub-interpreter feature (see Section 18) is rarely used but is central to INTHON's isolation model.

Why this matters for INTHON

INTHON does not run on a C interpreter. It runs on Python. But its conceptual model — an immutable IR (analog of the code object) lowered to a stack-machine bytecode (analog of *co_code*) executed by a dispatch loop (analog of *ceval*) inside a capability-bounded frame (analog of *PyFrameObject*) — is a deliberate echo of CPython's architecture. Understanding CPython's layers makes INTHON's layers predictable.

4. Lexical Structure of Python

Python's lexical structure is defined in the Language Reference's Lexical Analysis chapter and implemented in *Parser/tokenizer.c*. The tokenizer is a hand-written state machine that consumes a UTF-8 byte stream and produces a stream of tokens. Unlike many languages, Python's tokenizer is whitespace-sensitive: indentation and newlines are significant tokens that drive the parser's block structure. This design eliminates the brace-based block syntax of C-like languages at the cost of a more complex lexer.

4.1 Physical and Logical Lines

A physical line is a sequence of characters terminated by a newline character (U+000A). CPython translates platform-specific line endings (CRLF on Windows, CR on classic Mac) to LF before tokenizing. A logical line is constructed from one or more physical lines joined by explicit line continuation (a backslash immediately before a newline) or by implicit continuation inside brackets (*(*, *[*, *{*). The end of a logical line is signaled by a NEWLINE token; the end of a physical line inside brackets is signaled by an NL token, which the parser ignores.

4.2 Indentation and the INDENT/DEDENT Tokens

At the start of each logical line, the tokenizer measures the indentation level by counting the leading spaces and tabs. Tabs are replaced by spaces up to the next multiple of 8. The indentation stack is maintained across the input: when the new indentation exceeds the top of the stack, an `INDENT` token is emitted; when it is less, one or more `DEDENT` tokens are emitted to unwind the stack to the matching level. Mismatched indentation raises *IndentationError*. This mechanism replaces braces as the block delimiter and is the single most distinctive feature of Python's surface syntax.

4.3 Token Categories

Token	Description	Example
NAME	Identifier or keyword	if, while, my_var
NUMBER	Integer, float, imaginary	42, 3.14, 2j
STRING	String literal (any quote style)	'hi', "hi", """doc"""
NEWLINE	End of logical line	\n
NL	End of physical line (ignored)	\n inside ()
INDENT	Increase in indentation level	(at line start)
DEDENT	Decrease in indentation level	(at line start)
OP	Operator or delimiter	+, -, (,], ,
COMMENT	Comment (discarded)	# ...
ENDMARKER	End of input	(sentinel)
ENCODING	Source encoding declaration	(token 0)
TYPE_COMMENT	Inline type comment (PEP 484)	# type: int
FSTRING_START/MIDDLE/END	f-string parts (3.12+)	f'...' components

Table 4.1 — CPython token categories as exposed by the `token` module.

4.4 Identifiers and Keywords

Identifiers follow the Unicode standard's `XID_Start` / `XID_Continue` properties: the first character must be `XID_Start`, subsequent characters must be `XID_Continue`. This allows non-ASCII identifiers (e.g. *café* or λ) but in practice most code is ASCII. Keywords are a reserved subset of identifiers: *False*, *None*, *True*, *and*, *as*, *assert*, *async*, *await*, *break*, *class*, *continue*, *def*, *del*, *elif*, *else*, *except*, *finally*, *for*, *from*, *global*, *if*, *import*, *in*, *is*, *lambda*, *nonlocal*, *not*, *or*, *pass*, *raise*, *return*, *try*, *while*, *with*, *yield*. *match* and *case* are soft keywords (PEP 634) — they are recognized as keywords only in match-statement position.

4.5 Literals

Numeric literals include decimal, binary (*0b*), octal (*0o*), and hexadecimal (*0x*) integers; floats with optional exponent (*1.5e-3*); and imaginary literals (*2j*). Underscores may be used as digit separators (*1_000_000*).

String literals come in single, double, and triple-quoted forms, with *r*, *b*, *f*, and *u* prefixes (combinable: *rb*, *Rb*, *fr*). Triple-quoted strings span multiple physical lines. F-strings (PEP 498, extended in PEP 701) embed Python expressions in *{...}* inside a string; the 3.12 tokenizer splits f-strings into FSTRING_START, FSTRING_MIDDLE, and FSTRING_END tokens around embedded expressions, which are themselves tokenized normally.

4.6 Operators and Delimiters

Python's operators are: +, -, *, **, /, //, %, @ (matrix mul, PEP 465), <<, >>, &, |, ^, ~, := (walrus, PEP 572), <, >, <=, >=, ==, !=. Delimiters are: (,), [,], {, }, ,, :, ., ;, =, -> (return annotation), +=, -=, *=, /=, //=, %=, @=, &=, |=, ^=, >>=, <<=, **=. The \$ and ? characters are not used in Python syntax (the walrus operator uses :=, not ?).

4.7 Comments and Encoding Declarations

Comments begin with # and continue to the end of the physical line. They are discarded by the tokenizer. If the first or second line of a source file matches the regular expression *coding[=:]*`\s*([-\\w.]+)`, the named encoding is used to decode the remainder of the file (default is UTF-8). Type comments of the form *# type: int* are recognized by the tokenizer as TYPE_COMMENT tokens when type comment parsing is enabled.

4.8 INTHON's Lexical Departures

INTHON retains Python's identifier and numeric literal rules but drops whitespace-significant block structure in favor of brace-delimited blocks (*{ ... }*). This decision is deliberate: braces are more token-efficient for LLM generation (no INDENT/DEDENT token stream to get wrong), are unambiguous in trace logs, and eliminate the entire class of *IndentationError* failures that plague LLM-generated Python. The cost is the loss of Python's visual readability for humans, which INTHON accepts because its primary author is an agent, not a human.

5. The PEG Parser (Python 3.10+)

Python 3.9 used an LL(1) parser generator (pgen) that produced a parse table from an EBNF grammar. LL(1) parsing is fast and simple but cannot express left-recursive rules and requires careful grammar engineering to avoid conflicts. PEP 617 replaced this with a Parsing Expression Grammar (PEG) parser written in C, generated from a PEG grammar file by a small parser-generator called pegen. The PEG parser is packrat-based (linear time via memoization), supports ordered choice and limited lookahead, and can express rules that LL(1) cannot.

5.1 PEG vs LL(1) — Why the Switch

The LL(1) parser required the grammar to be manually desugared to fit the LL(1) constraints. Alternative productions had to be mutually exclusive on the first token; left recursion was forbidden; mid-rule actions were unavailable. The walrus operator (PEP 572), structural pattern matching (PEP 634), and other modern features strained the LL(1) grammar to its limits. PEG's ordered choice (*a / b* means try *a* first, fall back to *b*) is more permissive and lets the grammar mirror the language spec more directly.

5.2 The PEG Grammar File

The canonical PEG grammar lives at *Grammar/python.gram* in the CPython source tree. It is a hand-maintained file with roughly 1,800 lines covering every production in the language. Each rule has the form *name[return_type]: production*, where the return type is a C struct (AST node type). Actions are written as C snippets that construct AST nodes from the parsed sub-expressions. The pegen tool reads this grammar and emits *Parser/parser.c*, a hand-tunable C parser.

5.3 Soft Keywords

Python 3.10 introduced soft keywords via PEP 634: *match* and *case* are recognized as keywords only in match-statement position. Elsewhere they remain valid identifiers. This avoids breaking existing code that uses *match* as a variable name (e.g. the popular *re.match*). The PEG grammar expresses this by trying the match-statement production first and falling back to expression-statement if the keyword is not in match position.

5.4 Error Recovery and SyntaxError

When the parser fails to match a production, it raises *SyntaxError* with the offending token, its position, and a caret pointing to the error in the source line. Python 3.10+ added improved error messages with did-you-mean suggestions for misspelled keywords. Python 3.12 added multi-line error careats for errors spanning multiple lines. Error recovery is intentionally minimal — Python does not attempt to continue parsing after a syntax error and report multiple errors; it stops at the first one.

5.5 The Parse Tree → AST Transform

The PEG parser produces a Concrete Syntax Tree (parse tree) of generic node objects. This is immediately transformed into an Abstract Syntax Tree of typed *ast.AST* instances by the actions embedded in the grammar file. The parse tree is not exposed to Python code; only the AST is. The transform drops tokens that carry no semantic weight (parentheses, commas, keywords used only as delimiters) and folds operator-precedence layers into typed BinOp nodes.

5.6 INTHON's Parser Choice: Lark

INTHON uses the Lark parser generator rather than pegen. Lark supports both Earley (which handles any context-free grammar) and LALR (which is faster but more restrictive) parsing; INTHON uses LALR with explicit disambiguation in the grammar. Lark's EBNF syntax is cleaner than pegen's PEG-with-C-actions for the case where the language is small and the AST construction is straightforward. The trade-off is performance: Lark is implemented in Python and is roughly 10× slower than pegen for equivalent grammars. For INTHON's expected program sizes (typically under 1KB), this is irrelevant.

6. AST Node Catalog

The CPython AST is defined in *Parser/Python.asdl* (Abstract Syntax Description Language) and exposed in Python via the *ast* module. Every node is a subclass of *ast.AST* with named fields. The AST is immutable in the sense that field names are fixed by the ASDL definition; field values may be reassigned. The *ast.dump()* function renders an AST as a printable tree; *ast.unparse()* (3.9+) regenerates source code from an AST.

6.1 Module and Top-Level Nodes

Node	Fields	Purpose
Module	body, type_ignores	Top-level module body
Interactive	body	REPL input
Expression	body	eval() input
FunctionType	argtypes, returns	PEP 484 function type comment
FunctionDef	name, args, body, decorator_list, returns, type_comment	def statement
AsyncFunctionDef	(same as FunctionDef)	async def statement
ClassDef	name, bases, keywords, body, decorator_list	class statement
Return	value	return statement
Delete	targets	del statement
Assign	targets, value, type_comment	x = y
AugAssign	target, op, value	x += y
AnnAssign	target, annotation, value, simple	x: int = 0
For	target, iter, body, orelse, type_comment	for loop
AsyncFor	(same as For)	async for
While	test, body, orelse	while loop
If	test, body, orelse	if statement
With	items, body, type_comment	with statement
AsyncWith	(same as With)	async with
Match	subject, cases	match statement (PEP 634)
Raise	exc, cause	raise statement
Try	body, handlers, orelse, finalbody	try/except/finally
Assert	test, msg	assert statement
Import	names	import x
ImportFrom	module, names, level	from x import y
Global	names	global x
Nonlocal	names	nonlocal x
Expr	value	expression statement
Pass	—	pass
Break	—	break
Continue	—	continue

Table 6.1 — Statement-level AST nodes (subset).

6.2 Expression Nodes

Node	Fields	Purpose
BoolOp	op, values	and / or (n-ary)
NamedExpr	target, value	walrus := (PEP 572)
BinOp	left, op, right	binary arithmetic
UnaryOp	op, operand	unary -, +, not, ~
Lambda	args, body	lambda expression
IfExp	test, body, orelse	ternary x if c else y
Dict	keys, values	{k: v, ...}
Set	elts	{1, 2, 3}
ListComp	elt, generators	[x for x in y]
SetComp	elt, generators	{x for x in y}
DictComp	key, value, generators	{k: v for ...}
GeneratorExp	elt, generators	(x for x in y)
Await	value	await expr
Yield	value	yield expr
YieldFrom	value	yield from expr
Compare	left, ops, comparators	a < b < c
Call	func, args, keywords	f(x, y=z)
FormattedValue	value, conversion, format_spec	f-string {...} part
JoinedStr	values	f-string literal
Constant	value, kind	any literal (3.8+)
Attribute	value, attr, ctx	obj.attr
Subscript	value, slice, ctx	obj[slice]
Starred	value, ctx	*x in call/collection
Name	id, ctx	variable reference
List	elts, ctx	[1, 2, 3]
Tuple	elts, ctx	(1, 2, 3)
Slice	lower, upper, step	a:b:c

Table 6.2 — Expression-level AST nodes (subset).

6.3 Operator and Context Nodes

Binary and unary operators are themselves AST nodes: *Add*, *Sub*, *Mult*, *MatMult*, *Div*, *Mod*, *Pow*, *LShift*, *RShift*, *BitOr*, *BitXor*, *BitAnd*, *FloorDiv* for binary; *Invert*, *Not*, *UAdd*, *USub* for unary; *And*, *Or* for boolean. Comparison operators are *Eq*, *NotEq*, *Lt*, *LtE*, *Gt*, *GtE*, *Is*, *IsNot*, *In*, *NotIn*. The *ctx* field on *Attribute*, *Subscript*, *Name*, *List*, and *Tuple* is one of *Load*, *Store*, or *Del*, indicating whether the node is being read, assigned, or deleted. This distinction is critical for the compiler: $x = y$ compiles x with *Store* context and y with *Load* context.

6.4 Function Definition Nodes

arguments is a node holding all function parameters: *posonlyargs* (positional-only, PEP 570), *args* (regular positional-or-keyword), *vararg* (**args*), *kwonlyargs* (keyword-only after ***), *kw_defaults* (defaults for kw-only), *kwarg* (***kwargs*), and *defaults* (defaults for positional args). Each arg is an *arg* node with *arg* (name), *annotation* (type), and *type_comment* fields. The compiler allocates these to specific slots in the code object's *co_varnames*.

6.5 Pattern Matching Nodes (3.10+)

Structural pattern matching (PEP 634) introduces *Match*, *match_case*, and a family of pattern nodes: *MatchValue*, *MatchSingleton*, *MatchSequence*, *MatchStar*, *MatchMapping*, *MatchClass*, *MatchAs*, *MatchOr*. Each pattern can bind names (compiled as *Store* contexts) and is tested in order. The compiler lowers match statements to a sequence of comparisons and jumps rather than a dedicated opcode.

6.6 AST Visitors

The *ast.NodeVisitor* class implements the visitor pattern over the AST. Subclasses override *visit_NodeType* methods; the default *generic_visit* recurses into child nodes. *ast.NodeTransformer* is a subclass that allows mutation (returning a new node replaces the old one). The compiler itself is a *NodeVisitor* that walks the AST and emits bytecode. Linters (pyflakes, pylint), type checkers (mypy, pyright), and formatters (black) all build on the *ast* module.

7. Semantic Analysis & Symbol Tables

After parsing, CPython builds a symbol table that records, for every scope in the program, which names are local, which are free (captured from enclosing scopes), which are global, and which are implicit builtins. This information drives the compiler's choice of opcode for each name reference: *LOAD_FAST* for locals, *LOAD_DEREF* for free/cell variables, *LOAD_GLOBAL* for globals and builtins.

7.1 Scope Rules

Python has four scope kinds. **Local** scope is the function body — names assigned there are local unless declared *global* or *nonlocal*. **Enclosing** scope is the lexical scope of any enclosing function — accessed via *nonlocal* or automatically captured by closures. **Global** scope is the module namespace. **Builtin** scope is the *builtins* module dict, searched last as a fallback. This is the LEGB rule. Class bodies have a special scope: names defined in a class body are local to the class namespace but are not visible to methods defined inside the class (methods must access them via *self.attr* or *ClassName.attr*).

7.2 The symtable Module

The *symtable* module exposes the symbol table as Python objects. *symtable.symtable(code, filename, compile_type)* returns a *SymbolTable* object. Each scope is a *SymbolTable* with child scopes accessible via *.get_children()*. Each symbol is a *Symbol* object with methods like *.is_local()*, *.is_free()*, *.is_global()*, *.is_assigned()*, *.is_parameter()*, *.is_namespace()*. This is the same data the compiler uses to choose opcodes.

7.3 Name Binding Rules

A name is bound to a local scope if any assignment to it (including *=*, *+=*, *def*, *class*, *import*, *for* target, *with ... as* target, *except ... as* target) appears in the function body. The binding is function-wide: a name assigned on line 100 of a function is local even if read on line 1 — this is why *UnboundLocalError* can occur before the assignment is reached. *global* and *nonlocal* declarations override this: they explicitly mark the name as belonging to the global or an enclosing scope.

7.4 Free Variables and Cell Variables

When a nested function references a name from an enclosing function, the enclosing function's local for that name is promoted to a *cell variable*: it is stored in a *cell object* (a one-slot mutable container) rather than directly in the local slot. The nested function receives a reference to the same cell as a *free variable*. This is how closures capture mutable state: both the enclosing and the nested function share the cell, so a write in one is visible to the other. The compiler emits *LOAD_DEREF* and *STORE_DEREF* opcodes to access cells, and *MAKE_CELL* to create them.

7.5 Comprehension Scoping

Before Python 3, comprehensions leaked their loop variable into the enclosing scope. Since Python 3, every comprehension (list, set, dict, generator) gets its own implicit function scope. The iterable of the outermost *for* clause is evaluated in the enclosing scope (so *[x for x in foo]* evaluates *foo* in the enclosing scope), but the rest of the comprehension runs in its own scope. This is implemented by compiling the comprehension as a nested function and calling it with the outer iterable as an argument.

7.6 Class Body Scoping Quirk

Class bodies execute in a namespace that becomes the class's *__dict__*. Names defined in a class body are accessible to subsequent statements in the same class body, but methods defined in the class body cannot see those names lexically — they must access them via *self.attr* or *ClassName.attr*. This is because methods are compiled as ordinary functions whose enclosing scope is the module, not the class. This is a frequent source of confusion for newcomers: a method that references a class-level constant by bare name will look it up in the module scope, not the class scope.

7.7 Future Annotations (PEP 563)

from __future__ import annotations changes the semantics of annotations: they are no longer evaluated at function definition time and stored on *__annotations__* as objects; instead, they are stored as strings (the source text of the annotation) and only evaluated lazily via *typing.get_type_hints()*. This is the default in Python 3.10+ (PEP 649 defers evaluation to attribute access time). INTHON follows a different model: type annotations are evaluated at compile time and stored in the IR, never re-evaluated at runtime.

8. The Compiler: AST → Bytecode

The compiler lives in *Python/compile.c* and is roughly 4,000 lines of C. Its job is to take a checked AST plus its symbol table and produce a *PyCodeObject*. The compilation proceeds in four phases: AST to CFG (control flow graph), CFG to instruction list, instruction list to bytecode (with jump resolution), and bytecode to code object (with constant folding and stack depth analysis).

8.1 Phase 1: AST to CFG

The compiler walks the AST and emits instructions into a Control Flow Graph of basic blocks. Each basic block is a sequence of instructions with a single entry point (the first instruction) and a single exit point (a jump or return). Block boundaries occur at every jump target and after every jump. The CFG is represented as a graph of *basicblock* structs, each with a list of instructions and pointers to successor blocks. Loops, conditionals, try/except, and with statements all generate multi-block CFGs.

8.2 Phase 2: CFG Optimization

Once the CFG is built, the compiler runs a series of optimization passes. The most important is dead-code elimination: any block unreachable from the entry block is removed. Constant folding is applied to pure arithmetic on constants. Jump-to-jump chains are collapsed (*JUMP_ABSOLUTE* to *JUMP_ABSOLUTE* becomes a direct jump). Empty blocks are removed. The result is a smaller CFG that produces smaller bytecode.

8.3 Phase 3: Stack Depth Analysis

For each basic block, the compiler computes the maximum stack depth reached during execution. This is the *co_stacksize* field of the code object. The analysis is a forward dataflow pass: starting from the entry block with depth 0, each instruction's stack effect (pop count, push count) is applied to compute the depth at the next instruction. The maximum depth across all blocks is *co_stacksize*. This must be correct: if *co_stacksize* is too small, the interpreter will overflow the evaluation stack and crash.

8.4 Phase 4: Bytecode Emission

The CFG is linearized into a single instruction list, with jump targets replaced by byte offsets. The instruction list is encoded as a sequence of 16-bit words: 8-bit opcode followed by 8-bit oparg. Operands larger than 255 are split across multiple instructions using *EXTENDED_ARG*. The encoded bytecode is stored in *co_code* as a bytes object. Constants are collected into *co_consts*, names into *co_names*, locals into *co_varnames*.

8.5 Compilation of Specific Constructs

8.5.1 Comprehensions

A list comprehension [*expr for x in iter if cond*] is compiled as a nested function. The compiler synthesizes an implicit function with parameters for the outermost iterable, builds the loop and condition inside it, appends *expr* to a list, and returns the list. The comprehension site becomes a call to this function with the outer iterable as argument. This is why the loop variable does not leak (the function's local scope is discarded) and why the iterable is evaluated in the enclosing scope (it is passed as an argument).

8.5.2 Closures

When the compiler encounters a nested function that references enclosing locals, it marks those locals as cell variables in the enclosing function's symbol table. The enclosing function emits *MAKE_CELL* for each cell variable, wrapping its local slot in a cell object. The nested function's *MAKE_FUNCTION* opcode receives the cells as part of its closure. *LOAD_DEREF* and *STORE_DEREF* opcodes access the cell's contents.

8.5.3 Async Functions

Async functions are compiled like regular functions but with the *CO_COROUTINE* flag set in *co_flags*. *await* expressions emit *GET_AWAITABLE* + *SEND* opcodes; the *SEND* opcode yields to the event loop if the awaitable is not done. Async generators (*async def* with *yield*) set *CO_ASYNC_GENERATOR*. The compiler emits a *GEN_START* opcode at the function entry to initialize the generator state machine.

8.5.4 Match Statements

Match statements compile to a cascade of comparisons and jumps. Each case is a sequence of opcodes that tests the subject against the pattern, binds captured names, and jumps to the case body on success or to the next case on failure. The *IR_REFS* for the captured names are added to the function's locals. The compiler does not use a dedicated *MATCH* opcode; everything is lowered to *LOAD*, *COMPARE*, *POP_JUMP_IF_** sequences.

8.6 The Code Object's Final Shape

After all phases, the compiler constructs a *PyCodeObject* with the bytecode, constants, names, local names, free variable names, cell variable names, stacksize, flags, filename, name, first line number, line number table, exception table (3.11+), and various metadata. This object is the input to the evaluation loop.

9. Bytecode & the ceval Main Loop

Bytecode execution happens in *Python/ceval.c*, specifically in *_PyEval_EvalFrameDefault*. This is the heart of CPython: a single C function that is roughly 4,000 lines long and dispatches every bytecode instruction. It is the function that every Python program spends most of its time in. Understanding its structure is essential for any language implementer.

9.1 Instruction Format

Since Python 3.6, bytecode is a sequence of 16-bit words: 8-bit opcode followed by 8-bit oparg. The oparg is opcode-specific: for *LOAD_CONST* it is an index into *co_consts*; for *LOAD_FAST* it is an index into the locals array; for *POP_JUMP_IF_FALSE* it is a bytecode offset. Operands larger than 255 are split via *EXTENDED_ARG*: *EXTENDED_ARG 1*; *LOAD_CONST 0* loads *co_consts[256]* by shifting the *EXTENDED_ARG* oparg left 8 bits and ORing with the next oparg. Up to 3 *EXTENDED_ARG*s can be chained, giving 32-bit operands.

9.2 The Dispatch Loop

The dispatch loop is a giant *switch* statement over opcodes. Each case is a C block that implements the opcode's semantics, updates the stack pointer, and advances the instruction pointer. The loop is wrapped in a *for (;;)* with periodic checks for signals (the eval breaker), GIL release, and async suspension points. The loop

is unrolled four times (via the *TARGET* macro) to amortize the indirect branch cost and to allow the CPU's branch predictor to specialize on common opcode sequences.

9.3 Frame Stack and Calls

Function calls push a new frame onto the frame stack and recursively invoke *_PyEval_EvalFrameDefault*. Returns pop the frame and resume the caller. In Python 3.11+, the frame is allocated on the C stack for non-recursive calls (the icelebrated PEP 659 specialization), eliminating heap allocation for the common case. Recursive calls (where the same function recurses) still use heap-allocated frames to bound C stack depth.

9.4 Inline Caching and Specialization (PEP 659)

Python 3.11 introduced adaptive specialization. Frequently-executed instructions are rewritten at runtime to specialized variants: *LOAD_ATTR* becomes *LOAD_ATTR_INSTANCE_VALUE* if the attribute is an instance attribute, *LOAD_ATTR_MODULE* if it is a module attribute, *LOAD_ATTR_WITH_HINT* if the type provides a hint. Each specialization has a faster fast-path. The rewrite happens after 8 executions of the generic instruction; if the specialization misses (the assumption no longer holds), it is de-optimized back to the generic form. This is the single biggest performance improvement in CPython 3.11 and the basis for further work in 3.12 and 3.13.

9.5 The Eval Breaker

Periodically (every *_PyEval_SliceInterval* instructions, default 100), the dispatch loop checks the *eval breaker* — a flag that signals pending work: a signal arrived, the GIL was requested by another thread, an exception was raised in another thread, a periodic GC is due. If the breaker is set, the loop calls *Py_MakePendingCalls* to drain the pending work before resuming bytecode execution. This is how signals and thread scheduling happen in CPython without per-instruction overhead.

9.6 Exception Handling

Before 3.11, exception handling used a block stack: *SETUP_FINALLY* pushed a block marker, and exceptions unwound the block stack looking for a matching handler. In 3.11, this was replaced with an explicit exception table stored on the code object (*co_exceptiontable*). The table maps instruction offsets to handler offsets; when an exception is raised, the runtime binary-searches the table to find the handler. This is faster (no block stack maintenance) and more flexible (arbitrary nesting). The *RAISE_VARARGS* and *RERAISE* opcodes implement raise; *PUSH_EXC_INFO* and *POP_EXCEPT* manage the exception state on the stack.

9.7 Generators and Coroutines

Generators are functions that contain *yield*. They are compiled as functions with the *CO_GENERATOR* flag. Calling a generator function returns a generator object without executing any code; the first *next()* call enters the frame and runs until the first *yield*, which emits a *YIELD_VALUE* opcode that pops the frame from the active stack and suspends. Subsequent *next()* calls resume the frame at the instruction after the yield. Coroutines (*async def*) work similarly but use *SEND* and *GET_AWAITABLE* for the suspend/resume protocol.

10. Per-Opcode Semantics Reference

This section is a reference card for the most common CPython opcodes as of version 3.12. Stack effects are written in the form $[a, b] \rightarrow [c]$, where the right side is the top of the stack. The complete opcode table is in *Include/opcode.h* and *Lib/opcode.py*; the semantics are in *Python/ceval.c*.

10.1 Load and Store Opcodes

Opcode	Stack effect	Semantics
LOAD_CONST i	$[] \rightarrow [\text{const}]$	Push <code>co_consts[i]</code> .
LOAD_NAME i	$[] \rightarrow [\text{value}]$	Push name <code>co_names[i]</code> from locals/globals/builtins.
LOAD_FAST i	$[] \rightarrow [\text{value}]$	Push <code>locals[i]</code> . Fast path for function locals.
LOAD_GLOBAL i	$[] \rightarrow [\text{value}]$	Push <code>co_names[i]</code> from globals, fallback to builtins.
LOAD_DEREF i	$[] \rightarrow [\text{value}]$	Push <code>cell/freevars[i]</code> contents.
LOAD_CLASSDEREF i	$[] \rightarrow [\text{value}]$	Like <code>LOAD_DEREF</code> but checks class namespace first.
LOAD_ATTR i	$[\text{obj}] \rightarrow [\text{value}]$	Push <code>obj.co_names[i]</code> .
LOAD_METHOD i	$[\text{obj}] \rightarrow [\text{method self}]$	Optimized method load for <code>CALL_METHOD</code> .
STORE_NAME i	$[\text{value}] \rightarrow []$	<code>locals[co_names[i]] = value</code> .
STORE_FAST i	$[\text{value}] \rightarrow []$	<code>locals[i] = value</code> .
STORE_GLOBAL i	$[\text{value}] \rightarrow []$	<code>globals[co_names[i]] = value</code> .
STORE_DEREF i	$[\text{value}] \rightarrow []$	<code>cell/freevars[i] = value</code> .
STORE_ATTR i	$[\text{value}, \text{obj}] \rightarrow []$	<code>obj.co_names[i] = value</code> .
STORE_SUBSCR	$[\text{value}, \text{container}, \text{key}] \rightarrow []$	<code>container[key] = value</code> .
DELETE_FAST i	$[] \rightarrow []$	<code>del locals[i]</code> .
DELETE_NAME i	$[] \rightarrow []$	<code>del locals[co_names[i]]</code> .
DELETE_GLOBAL i	$[] \rightarrow []$	<code>del globals[co_names[i]]</code> .
DELETE_ATTR i	$[\text{obj}] \rightarrow []$	<code>del obj.co_names[i]</code> .
DELETE_SUBSCR	$[\text{container}, \text{key}] \rightarrow []$	<code>del container[key]</code> .

Table 10.1 — Load and store opcodes.

10.2 Arithmetic and Comparison Opcodes

Opcode	Stack effect	Semantics
BINARY_OP i	$[a, b] \rightarrow [\text{result}]$	Generic binary op (3.11+). i selects <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>//</code> , <code>%</code> , <code>**</code> , <code><<</code> , <code>>></code> , <code>&</code> , <code> </code> , <code>^</code> , <code>@</code> , <code>+=</code> , etc.
UNARY_NEGATIVE	$[a] \rightarrow [-a]$	Numeric negation.
UNARY_POSITIVE	$[a] \rightarrow [+a]$	Numeric positive (rarely used).

Opcode	Stack effect	Semantics
UNARY_NOT	[a] -> [not a]	Boolean not.
UNARY_INVERT	[a] -> [~a]	Bitwise not.
COMPARE_OP i	[a, b] -> [bool]	i selects <, <=, ==, !=, >, >=, in, not in, is, is not.
IS_OP i	[a, b] -> [bool]	Optimized identity comparison (3.9+).
CONTAINS_OP i	[a, b] -> [bool]	Optimized in/not in (3.9+).

Table 10.2 — Arithmetic and comparison opcodes.

10.3 Control Flow Opcodes

Opcode	Stack effect	Semantics
POP_JUMP_IF_FALSE i	[cond] -> []	Jump to offset i if cond is false.
POP_JUMP_IF_TRUE i	[cond] -> []	Jump to offset i if cond is true.
POP_JUMP_IF_NONE i	[cond] -> []	Jump to i if cond is None (3.11+).
POP_JUMP_IF_NOT_NONE i	[cond] -> []	Jump to i if cond is not None (3.11+).
JUMP_FORWARD i	[] -> []	Relative jump forward by i.
JUMP_BACKWARD i	[] -> []	Relative jump backward (3.11+, for loops).
JUMP_BACKWARD_NO_INTERRUPT i	[] -> []	Like JUMP_BACKWARD but no eval breaker check.
GET_ITER	[iterable] -> [iterator]	iter(iterable).
FOR_ITER i	[iterator] -> [next]	Push next(); on StopIteration, jump to i.
END_FOR	[] -> []	Marker for end of for loop (3.12+).
RETURN_VALUE	[value] -> []	Return value from current frame.
RETURN_CONST i	[] -> []	Return co_consts[i] (3.12+, no stack pop).

Table 10.3 — Control flow opcodes.

10.4 Call and Function Opcodes

Opcode	Stack effect	Semantics
CALL i	[callable, args...] -> [result]	Call callable with i args (3.11+).
CALL_FUNCTION i	[callable, args...] -> [result]	Pre-3.11 call form.
CALL_FUNCTION_KW i	[callable, args..., kwnames] -> [result]	Call with keyword args.
CALL_FUNCTION_EX i	[callable, args, kwargs?] -> [result]	Call with *args, **kwargs.
CALL_METHOD i	[method, self, args...] -> [result]	Optimized method call (3.11-).
MAKE_FUNCTION i	[code, qualname, defaults?, kwdefaults?, annotations?, closure?] -> [func]	Create function object.

Opcode	Stack effect	Semantics
PUSH_NULL	[] -> [NULL]	Push NULL marker for unbound method calls (3.11+).
PRECALL i	[callable, args...] -> [callable, args...]	Pre-call hook (3.11 only, removed in 3.12).
KW_NAMES i	[] -> [kwnames]	Push tuple of keyword names (3.11-3.12).

Table 10.4 — Call and function opcodes.

10.5 Container Construction Opcodes

Opcode	Stack effect	Semantics
BUILD_LIST i	[items...] -> [list]	Build list of i items.
BUILD_TUPLE i	[items...] -> [tuple]	Build tuple of i items.
BUILD_SET i	[items...] -> [set]	Build set of i items.
BUILD_MAP i	[keys, values...] -> [dict]	Build dict of i pairs.
BUILD_CONST_KEY_MAP i	[keys, values...] -> [dict]	Build dict with constant keys tuple.
BUILD_SLICE i	[step?, stop, start] -> [slice]	Build slice object (2 or 3 args).
BUILD_STRING i	[parts...] -> [str]	Concatenate i strings (f-string).
LIST_APPEND i	[list, item] -> [list]	Append to list at stack slot i (comprehension).
SET_ADD i	[set, item] -> [set]	Add to set at stack slot i (comprehension).
MAP_ADD i	[map, key, value] -> [map]	Add to map at stack slot i (comprehension).
UNPACK_SEQUENCE i	[seq] -> [items...]	Unpack i items from seq.
UNPACK_EX i	[seq] -> [items..., list, items...]	Unpack with * (PEP 448).

Table 10.5 — Container construction opcodes.

10.6 Exception Opcodes

Opcode	Stack effect	Semantics
RAISE_VARARGS i	[exc?, cause?] -> []	raise (i = 0, 1, or 2 args).
RERAISE i	[exc] -> []	Re-raise active exception (3.11+).
PUSH_EXC_INFO	[exc] -> [exc, prev]	Push current exc, save previous.
POP_EXCEPT	[exc] -> []	Pop exception handler state.
CHECK_EXC_MATCH	[exc1, exc2] -> [exc1, bool]	Test isinstance(exc1, exc2).

Opcode	Stack effect	Semantics
CHECK_EG_MATCH	[exc, match_type] -> [rest?, match?]	Exception group matching (3.11+).
PUSH_FINALLY	[] -> []	Marker for finally block entry (3.12+).

Table 10.6 — Exception opcodes.

10.7 Coroutine and Generator Opcodes

Opcode	Stack effect	Semantics
GEN_START i	[] -> []	Initialize generator/coroutine (3.10+).
SEND i	[value, generator] -> [retval]]	Send value into generator; jump to i on StopIteration.
YIELD_VALUE	[value] -> []	Yield value from generator.
RESUME i	[] -> []	Resume generator/coroutine (3.11+).
GET_AWAITABLE i	[obj] -> [awaitable]	Coerce obj to awaitable (i = where flag).
GET_YIELD_FROM_ITER	[obj] -> [iter]	Coerce to yield-from iterable.
GET_AITER	[obj] -> [aiter]	obj.__aiter__().
GET_ANEXT	[aiter] -> [awaitable]	aiter.__anext__().
END_SEND	[value, sender] -> [value]	End yield-from (3.12+).

Table 10.7 — Coroutine and generator opcodes.

10.8 Import and Module Opcodes

Opcode	Stack effect	Semantics
IMPORT_NAME i	[fromlist, level] -> [module]	import co_names[i].
IMPORT_FROM i	[module] -> [module, attr]	Get co_names[i] from module (leaves module on stack).
IMPORT_STAR	[module] -> []	from module import * (modifies locals).
LOAD_BUILD_CLASS	[] -> [builtins.__build_class__]	Push __build_class__ for class statements.
LOAD_ASSERTION_ERROR	[] -> [AssertionError]	Push AssertionError (faster than LOAD_GLOBAL).

Table 10.8 — Import and module opcodes.

The full opcode set has roughly 120 entries in Python 3.12, with new ones added in each release. The reference above covers the subset that INTHON's IR closely mirrors. For the complete list, see *Lib/opcode.py* and the *dis* module documentation. For per-version changes, see the *whatsnew* documents for each Python release.

11. Object Model & Type System

Every Python object is a *PyObject** in C — a pointer to a struct whose first two fields are *ob_refcnt* (the reference count) and *ob_type* (a pointer to the object's type object). Variable-sized objects (lists, tuples, strings) use *PyVarObject*, which adds an *ob_size* field. This layout is the contract that every CPython object, from *None* to user-defined classes, must satisfy. The type object itself (*PyTypeObject*) is a large struct (around 80 fields in modern CPython) that defines the type's behavior: its name, size, allocation function, deallocator, every operator implementation, and its method resolution order.

11.1 The Type Object and `tp_slots`

Python's object model is implemented via a table of function pointers in the type object. Each operation (e.g. `+`, `[]`, `print`, `getattr`) is dispatched to a specific slot in the type: *tp_add* for `+`, *tp_subscript* for `[]`, *tp_repr* for `repr()`, *tp_getattro* for attribute access. When you write `x + y` in Python, the compiler emits *BINARY_OP(0)* (or *BINARY_ADD* in older Pythons), and the eval loop calls *PyNumber_Add(x, y)*, which looks up *type(x)->tp_as_number->nb_add* and calls it. The slot mechanism is how Python achieves runtime polymorphism without monomorphization.

11.2 The Descriptor Protocol

Attributes defined on a class can be *descriptors*: objects with `__get__`, `__set__`, and/or `__delete__` methods. *Data descriptors* define `__set__` or `__delete__`; *non-data descriptors* define only `__get__`. The distinction matters for attribute lookup precedence. The lookup order on *obj.attr* is: (1) data descriptors on *type(obj)* and its bases (via MRO), (2) *obj.__dict__*, (3) non-data descriptors on *type(obj)* and its bases, (4) *type(obj).__dict__* and base dicts. Functions, classmethods, staticmethods, and properties are all descriptors. *staticmethod* is a non-data descriptor; *property* is a data descriptor.

11.3 The Method Resolution Order (C3)

Python uses the C3 linearization algorithm (PEP 253) to compute the MRO of a class with multiple inheritance. C3 guarantees that: (1) a class appears before its parents in the MRO, (2) the relative order of parents is preserved, (3) the MRO is unique (otherwise *TypeError* is raised at class creation). The MRO is stored on *type.__mro__* as a tuple. Attribute lookup walks the MRO in order, returning the first match. The C3 algorithm is $O(n^2)$ in the worst case but is fast for typical class hierarchies.

11.4 `__new__` vs `__init__`

Object creation is two-phase: `__new__` creates the instance (it is a staticmethod that returns a new object), `__init__` initializes it (it is called on the new instance and returns *None*). For immutable types like *int* and *tuple*, all state must be set in `__new__` because `__init__` cannot modify them after creation. For mutable types, `__init__` does most of the work. Calling *ClassName(args)* invokes *ClassName.__new__(ClassName, args)* first; if the result is an instance of *ClassName*, then *ClassName.__init__(result, args)* is called. Subclasses that override `__new__` but not `__init__` may break this contract — a common source of subtle bugs.

11.5 Slots vs `__dict__`

By default, instances of user-defined classes have a `__dict__` attribute — a dict storing instance attributes. This is flexible (any attribute can be added at runtime) but memory-inefficient: each instance carries a dict, which is ~232 bytes plus the attributes. Classes can declare `__slots__` — a tuple of attribute names — to eliminate the per-instance dict. With slots, each attribute is stored in a fixed slot in the instance struct, reducing memory to

roughly 8 bytes per attribute plus the object header. The trade-off is that slotted classes cannot have new attributes added at runtime. Multiple inheritance with slots is restricted: a class with non-empty `__slots__` cannot multiply-inherit from another class with non-empty `__slots__`.

11.6 Special Methods (Dunder Protocol)

Python's data model is implemented via special methods whose names start and end with double underscores (dunders). There are roughly 80 dunders in the language. The most important ones: `__init__`, `__new__`, `__del__` (lifecycle); `__repr__`, `__str__` (string representation); `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`, `__hash__` (comparison and hashing); `__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__` (attribute access); `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__iter__`, `__next__` (container protocol); `__add__`, `__sub__`, ... (arithmetic); `__enter__`, `__exit__` (context manager); `__await__`, `__aenter__`, `__aexit__` (async). Dunders are looked up on the type, not the instance, for built-in operations — this is why `len(x)` calls `type(x).__len__` rather than `x.__len__`.

11.7 Metaclasses

A metaclass is a class whose instances are classes. `type` is the default metaclass: `type(name, bases, dict)` creates a new class. Subclassing `type` allows customizing class creation: the metaclass's `__new__` and `__init__` are called when a class with `metaclass=MyMeta` is defined. Metaclasses are powerful but rarely necessary in modern Python — `__init_subclass__` (PEP 487) and class decorators cover most use cases. INTHON does not implement metaclasses; the agent block is a first-class language construct, not a metaclass-controlled class.

12. Memory Management

CPython uses two complementary memory management mechanisms: reference counting for deterministic reclamation, and a cycle collector for unreachable cycles. Every `PyObject` has an `ob_refcnt` field; `Py_INCREF` and `Py_DECREF` increment and decrement it. When `ob_refcnt` reaches zero, the object is immediately deallocated (its `__del__` is called and its memory is freed). This is the primary reclamation mechanism and the reason CPython has predictable cleanup behavior (unlike Java-style GC pauses).

12.1 pymalloc — The Object Allocator

For objects smaller than 512 bytes, CPython uses a custom allocator called `pymalloc`. `pymalloc` manages memory in three layers: *arenas* (256KB chunks obtained from the OS via `mmap/VirtualAlloc`), *pools* (4KB chunks carved from arenas, each pool dedicated to a specific size class), and *blocks* (the actual allocation units). Size classes are powers of 8 up to 512 bytes (8, 16, 24, ..., 512). A request for *N* bytes is rounded up to the next size class. Free blocks are tracked in a freelist per pool. This avoids `malloc/free` overhead for the small short-lived objects that dominate Python workloads.

12.2 Reference Counting

Every `Py_INCREF` increments `ob_refcnt`; every `Py_DECREF` decrements it and, if it reaches zero, calls the type's `tp_dealloc` and frees the memory. Reference counting is thread-safe on modern CPython via atomic operations on `ob_refcnt` (since 3.12, `ob_refcnt` is a 32-bit atomic with the high bit reserved as an *immortal* flag for objects like `None` and `True` whose refcount is never touched). Reference counting's main weakness is cycles: two objects referencing each other have nonzero refcounts but are unreachable. The cycle collector

handles these.

12.3 The Cycle Collector (gc Module)

The cycle collector runs periodically (every 700 allocations by default, tunable via `gc.set_threshold()`) and identifies unreachable cycles. It uses a tri-generational model: generation 0 (newly allocated), generation 1 (survived one collection), generation 2 (survived two). Younger generations are collected more often. The algorithm is: (1) for each object in the target generation, decrement the refcount of every object it references (this *tentative* removal simulates removing all internal references); (2) any object whose refcount is now zero is in a cycle and is unreachable; (3) restore the refcounts (the simulation is undone); (4) for objects identified as in a cycle, call their finalizers (`__del__`) and free them. Weak references (*weakref.ref*) are not traced through, so they don't keep objects alive.

12.4 Object Lifecycle and Finalization

An object's lifecycle is: allocation (`pymalloc` or the type's `tp_alloc`), initialization (`__init__`), use, and deallocation (`tp_dealloc` calls `__del__` if defined, then frees memory). `__del__` is called when the refcount reaches zero, but not always — if the object is in a cycle, `__del__` is called by the cycle collector instead, which may be much later. Worse, if the cycle contains objects with `__del__`, the collector cannot determine a safe order to call them, so before Python 3.4 such cycles were never collected. PEP 442 (3.4+) fixed this by allowing cycles with `__del__` to be collected, but the order is not guaranteed.

12.5 Immortal Objects (3.12+)

Python 3.12 introduced *immortal* objects: objects whose refcount never changes and is never checked for zero. `None`, `True`, `False`, and a few interned strings are immortal. This eliminates atomic refcount operations on these ubiquitous objects and is a prerequisite for per-interpreter GILs (since the refcount must not be shared across interpreters). The high bit of `ob_refcnt` is the immortal flag; immortal objects have `ob_refcnt = 0x8000_0000`.

12.6 INTHON's Memory Model

INTHON runs inside CPython, so it inherits CPython's memory model for its own data structures. INTHON's episodic memory is backed by SQLite, which has its own page cache and allocator. The InthonVM's evaluation stack is a Python list of PyObject pointers — no separate stack allocation. This is slower than CPython's C-stack evaluation but simplifies the implementation and allows the trace to capture every stack manipulation as a Python object.

13. The GIL & Threading Model

The Global Interpreter Lock (GIL) is a process-wide mutex that ensures only one thread executes Python bytecode at a time. The GIL exists because CPython's memory management (reference counting) is not thread-safe without it. Without the GIL, every `Py_INCREF` would need a lock or atomic operation, slowing down single-threaded code substantially. The GIL is a design trade-off: it makes single-threaded code fast at the cost of true parallelism in multi-threaded code.

13.1 Why the GIL Exists

Reference counting is the primary reason. If two threads simultaneously increment *ob_refcnt* on the same object, a race condition could lose one increment, causing a premature deallocation. Locking every *Py_INCREF* would be prohibitively expensive: refcount operations happen on every assignment, every function argument pass, every attribute access. The GIL is a single lock that protects all refcounts (and all other interpreter state) at once. The alternative — making refcounts atomic — was measured and rejected as too slow on x86 (where atomic ops are expensive compared to plain increments).

13.2 The Eval Breaker and GIL Release

The eval loop checks the *eval breaker* periodically (every 100 instructions by default, configurable via *sys.setswitchinterval()*). The eval breaker is set when: a signal arrives, another thread requests the GIL, a pending call is queued. When the breaker is set, the loop drops the GIL, waits for it to be reacquired (or hands it to the next thread), and resumes. This is how multi-threaded Python achieves concurrency without parallelism: threads take turns holding the GIL.

13.3 I/O and the GIL

I/O operations (file reads, network calls, *time.sleep*) release the GIL while they block. This is implemented via *Py_BEGIN_ALLOW_THREADS* / *Py_END_ALLOW_THREADS* macros in C extensions. While one thread is blocked on I/O, other threads can run. This is why multi-threaded Python is effective for I/O-bound workloads (web scrapers, web servers) but useless for CPU-bound workloads (numerical computation) — the latter need multiprocessing or C extensions.

13.4 PEP 703: Making the GIL Optional (3.13+)

PEP 703, accepted for Python 3.13, introduces a no-GIL build of CPython. The no-GIL build uses biased reference counting (refs held by the owning thread are non-atomic; refs held by other threads use atomic ops), deferred reference counting (some objects like top-level functions skip refcounting entirely), and a thread-safe allocator. The no-GIL build is opt-in via *--disable-gil* at build time. The long-term plan is to make no-GIL the default in Python 3.15+ (PEP 779).

13.5 Sub-interpreter GILs (PEP 684, 3.12+)

PEP 684 gave each sub-interpreter its own GIL, allowing true parallelism across interpreters in the same process. This is the basis for PEP 734's *interpreters* module, which exposes sub-interpreters as a Python API. Each sub-interpreter has its own GIL, its own module state, its own exception state — but shares the process's C extension state, which is the source of much complexity. Most C extensions are not sub-interpreter-safe; making them so requires migration to PEP 484 multi-phase module initialization.

13.6 INTHON's Concurrency Model

INTHON runs inside a single Python interpreter and inherits its GIL. INTHON programs can use Python threads for I/O concurrency (the *retry* primitive is implemented with *threading.Event* for HITL approval gates). True parallelism is not a goal of INTHON v1.0; the language is designed for sequential agent execution where the bottleneck is LLM generation time (seconds) rather than CPU time (milliseconds). A future INTHON v2.0 could target sub-interpreter parallelism for multi-agent workflows.

14. The Import System

Python's import system is one of its most flexible and most complex subsystems. Since Python 3.3, imports are implemented in pure Python via the *importlib* package, driven by a chain of *meta path finders* registered on *sys.meta_path*. Every *import* statement triggers a search through this chain; the first finder that returns a *ModuleSpec* wins, and the spec's loader is invoked to create the module.

14.1 The Import Protocol

import foo.bar triggers the following sequence: (1) check *sys.modules['foo.bar']* — if present, return it (modules are cached); (2) iterate *sys.meta_path* calling *finder.find_spec('foo.bar', None)* on each; (3) the first non-None spec wins — its *loader* is invoked; (4) the loader creates the module object, sets *__spec__*, *__loader__*, *__file__*, etc.; (5) the module is added to *sys.modules* *before* its body executes (this allows circular imports); (6) the module body is executed (the module's code object is run with the module dict as globals); (7) the module is returned.

14.2 The Default Meta Path Finders

Finder	Purpose
BuiltinImporter	Finds built-in modules (compiled into the interpreter, e.g. sys, math).
FrozenImporter	Finds frozen modules (Python source compiled to bytecode and embedded in the interpreter).
PathFinder	Searches <i>sys.path</i> for source (.py), bytecode (.pyc), and C extensions (.so/.pyd).

Table 14.1 — Default finders on *sys.meta_path*.

14.3 PathFinder and PathHooks

PathFinder iterates *sys.path*, asking each entry whether it can find the module. *sys.path* entries are strings (filesystem paths) or path hook objects. For each entry, *PathFinder* tries the registered *path hooks* (*sys.path_hooks*) to obtain a *path entry finder*. The default path hooks are *FileFinder* (for filesystem paths) and *zipimporter* (for zip files). Custom path hooks can be registered to import from databases, URLs, or anywhere else.

14.4 The ModuleSpec Object

A *ModuleSpec* (PEP 451) is the data structure returned by a finder. It contains: *name*, *loader*, *origin* (the source file path), *submodule_search_locations* (for packages — the list of paths to search for submodules), *cached* (the .pyc path), and various flags. The spec is the canonical description of how to load a module; the loader is the executor.

14.5 Cached Bytecode (.pyc Files)

When a .py file is imported, Python compiles it to bytecode and writes the bytecode to a .pyc file in *__pycache__*. On subsequent imports, if the .pyc's timestamp (or hash, in hash-based mode) matches the source, the .pyc is loaded directly without recompilation. The .pyc file format is a small header (magic number, flags, timestamp, source size) followed by the marshalled code object. Hash-based .pyc files (PEP

552) use a hash of the source instead of a timestamp, enabling reproducible builds and content-addressed caching.

14.6 Multi-Phase Module Initialization (PEP 489)

Before PEP 489, C extensions had a single *PyInit_modulename* function that created and returned the module object. This made sub-interpreter isolation impossible: the module's state was stored in static C variables shared across interpreters. PEP 489 introduced multi-phase initialization: the extension's *PyInit* function returns a *PyModuleDef* with slots describing creation (*m_slots*). The interpreter then calls *Py_mod_create* to create the module object and *Py_mod_exec* to execute the module's initialization code, per interpreter. State is stored on the module object via *PyModule_GetState()*, not in statics. Multi-phase init is mandatory for sub-interpreter-safe extensions.

14.7 INTHON's PyBridge as a Meta Path Finder

INTHON's PyBridge sandbox works by registering a custom meta path finder on *sys.meta_path*. When INTHON code imports a Python module via *use py.numpy as np*, the PyBridge finder intercepts the import: it checks the module name against the active policy's allowlist, raises *PyBridgeError* if denied, or delegates to the default *PathFinder* if allowed. The returned module is wrapped in an *InthonPyObject* proxy that intercepts every attribute access, preventing escape vectors like *np.__dict__['os'].system(...)*. This is the single most important security mechanism in INTHON; see Section 26 for the full design.

15. Exceptions & Tracebacks

Python's exception model is rooted in *BaseException*, the base class of all throwable objects. *Exception* inherits from *BaseException* and is the base for all user-facing exceptions; *KeyboardInterrupt* and *SystemExit* inherit directly from *BaseException* so they are not caught by *except Exception*. Exceptions are raised with the *raise* statement, caught with *try/except*, and propagated up the call stack until caught or until they reach the top level (where they terminate the program with a traceback).

15.1 Exception Chaining

Since Python 3.0, exceptions can be chained. *raise B from A* sets *B.__cause__ = A* (explicit chaining). If an exception is raised inside an *except* block while handling another exception, the original is set as *B.__context__* (implicit chaining). Tracebacks display both: *'The above exception was the direct cause of the following exception'* for explicit, *'During handling of the above exception, another exception occurred'* for implicit. *B.__suppress_context__* (set by *raise from*) controls whether the context is displayed.

15.2 The Traceback Object

When an exception propagates, it carries a *traceback* object — a linked list of frame references capturing the call stack at the point of the exception. Each traceback entry has *tb_frame* (the frame), *tb_lineno* (the line number), *tb_lasti* (the bytecode index), and *tb_next* (the next traceback in the chain, towards the innermost frame). The traceback is built as the exception unwinds through frames; each frame's exception handler adds a traceback entry before propagating. *traceback.format_exception()* renders the chain as the familiar multi-line traceback text.

15.3 Exception Groups (PEP 654, 3.11+)

Python 3.11 added *ExceptionGroup* and the *except** syntax. An exception group wraps multiple exceptions, allowing concurrent raise of multiple failures (useful for async tasks and parallel computation). *except* ExceptionType* catches all matching exceptions in a group, leaving non-matching ones to propagate as a new group. This is implemented via the *CHECK_EG_MATCH* opcode and runtime machinery in *Python/ceval.c*.

15.4 Exception Raising Cost

Raising an exception in Python is expensive — orders of magnitude more expensive than a function call. The cost includes: allocating the exception object, building the traceback chain, unwinding the frame stack, and matching except clauses. For this reason, exceptions should be used for genuinely exceptional conditions, not for normal control flow (e.g. iterating until *StopIteration* is now optimized to not raise — see *FOR_ITER*'s fast path). The *__traceback__* attribute on exceptions can be inspected and even modified, allowing libraries to clean up tracebacks for end-user display.

15.5 C-Level Exceptions (PyErr_*)

C extensions do not raise Python exceptions directly; they set them via the *PyErr_** API. *PyErr_SetString(type, msg)* sets the current exception; *PyErr_Occurred()* checks if one is set; *PyErr_Fetch()* and *PyErr_Restore()* save/restore the exception state. C functions return NULL (for object-returning functions) or -1 (for integer-returning functions) to signal that an exception is set; the caller checks the return value and propagates. This convention is universal across the CPython C-API.

15.6 INTHON's Exception Model

INTHON's exception taxonomy is small and deliberate: *PolicyViolationError* (policy quota exceeded), *PyBridgeError* (sandbox import violation), *SandboxViolationError* (runtime sandbox breach), *ApprovalDeniedError* (HITL gate denied), and *InthonRuntimeError* (generic). The *retry ... catch* construct catches these by name; the *guard* statement raises *GuardAssertionError* if the condition fails (which *retry* catches internally). INTHON does not support user-defined exception classes in v1.0; all errors are runtime-typed instances of these five classes.

16. The C-API & ABI Stability

CPython's C-API is the interface between Python and the ecosystem of C extensions that power its scientific computing, machine learning, and high-performance I/O stacks. NumPy, Pandas, PyTorch, TensorFlow, lxml, and hundreds of other libraries depend on the C-API to manipulate Python objects from C. The C-API is large (over 1,000 functions and macros) and is the most stable part of CPython — breaking it would break the entire scientific Python ecosystem.

16.1 The Stable ABI (PEP 384)

PEP 384 defined a subset of the C-API that is guaranteed to be binary-compatible across CPython minor versions. Extensions compiled against the Stable ABI (*Py_LIMITED_API* set to the version where the API was stabilized) can be loaded by any newer CPython without recompilation. This is essential for binary distribution: a wheel built once against 3.10 can be installed on 3.11, 3.12, 3.13. The Stable ABI is

conservative — it excludes struct fields, macro-dependent APIs, and anything that touches interpreter internals. About 60% of the C-API is in the Stable ABI.

16.2 The Limited API

The Limited API is the C-API surface available when `Py_LIMITED_API` is defined. It is the source-level equivalent of the Stable ABI: extensions that stick to the Limited API compile against a smaller, abstract interface and gain binary portability. The Limited API forbids direct struct access (e.g. `Py_TYPE(obj)` is allowed but `obj->ob_type` is not), forbids most macros, and provides opaque accessors for everything. Some popular extensions (notably NumPy) find the Limited API too restrictive for performance-critical code and accept the cost of building per-version wheels.

16.3 HPy — A Safer C-API

HPy is a third-party C-API designed to be safer, simpler, and more amenable to alternative Python implementations (PyPy, GraalPy, etc.). HPy passes *handles* to objects rather than raw pointers, allowing the implementation to move objects during GC. HPy has a Universal ABI mode where one binary works across CPython, PyPy, and GraalPy. HPy is not part of CPython but is gaining traction in the scientific Python ecosystem. INTHON does not interact with HPy.

16.4 Multi-Phase Init (PEP 489)

PEP 489 (covered in Section 14) is the C-API mechanism that enables sub-interpreter isolation. Extensions that use single-phase init (the pre-3.5 model) store their state in static C variables, which are shared across interpreters — a source of subtle bugs when sub-interpreters are used. Multi-phase init stores state on the module object via `PyModule_GetState()`, giving each interpreter its own copy. Most major extensions have migrated to multi-phase init over the past several years.

16.5 The GIL State API

C extensions that are called from non-Python threads (e.g. a C library callback invoked from a pthread) must acquire the GIL before calling any C-API function. The `PyGILState_Ensure()` / `PyGILState_Release()` API acquires and releases the GIL, marking the current thread as a Python thread for the duration. Inside an extension's module init function, the GIL is already held; inside functions called from Python, the GIL is already held. The `Py_BEGIN_ALLOW_THREADS` / `Py_END_ALLOW_THREADS` macros release and reacquire the GIL around blocking C code.

16.6 INTHON and the C-API

INTHON is a pure-Python implementation and does not call the C-API directly (beyond what Python itself exposes). However, INTHON's PyBridge interop layer must respect the C-API's object model: every Python object is a `PyObject*`, attribute access goes through `tp_getattro`, etc. Understanding the C-API is essential for designing a sandbox that can't be bypassed — for example, knowing that `object.__subclasses__()` can be used to traverse the type hierarchy and reach arbitrary types.

17. Sub-interpreters & Isolation

Sub-interpreters are a long-standing but historically underused CPython feature: the ability to run multiple independent Python interpreters in a single process. Each sub-interpreter has its own *sys.modules*, its own builtins, its own exception state, and (since 3.12) its own GIL. They share the process's C extension state, the C stack, and (in single-GIL builds) the GIL. Sub-interpreters are the basis for INTHON's isolation story — every INTHON script runs in its own sub-interpreter, preventing cross-contamination of state.

17.1 The Interpreter State

Each *PyInterpreterState* holds: the GIL (in per-interpreter-GIL builds), the *sys.modules* dict, the *builtins* module dict, the import machinery state, the GC state, the warning registry, the codec registry, the list of threads attached to this interpreter, and a reference to the runtime state. Most C-API functions implicitly use the current interpreter's state (obtained via *_PyInterpreterState_GET()*); some accept an explicit interpreter argument.

17.2 What is and isn't Isolated

Resource	Isolated?	Notes
<i>sys.modules</i>	Yes	Each interpreter has its own module cache.
<i>builtins</i>	Yes	Each interpreter has its own builtins dict.
<i>sys.path</i>	Yes	Per-interpreter.
Exception state	Yes	Per-thread, per-interpreter.
GIL (3.12+)	Yes	Per-interpreter in opt-in builds.
GIL (pre-3.12)	No	Single process-wide GIL.
C extension state	No	Static variables are shared unless multi-phase init.
File descriptors	No	OS-level, shared by the process.
Process environment	No	<i>environ</i> is shared.
CWD	No	<i>chdir</i> affects all interpreters.
OS-level signal handlers	No	Process-wide.

Table 17.1 — Sub-interpreter isolation matrix.

17.3 PEP 734: The interpreters Module (3.12+)

PEP 734 exposes sub-interpreters as a Python API: *import interpreters; interp = interpreters.create(); interp.run('print(1+1)').* This makes sub-interpreters accessible to Python code without C extension. Each interpreter runs in its own thread by default. Inter-interpreter communication is via queues and channels (PEP 734 also proposes *interpreters.queues* and *interpreters.channels*). The module is experimental in 3.12 and may change.

17.4 Why Sub-interpreters Matter for INTHON

Sub-interpreters provide the only native CPython mechanism for true state isolation within a process. INTHON's sandbox (PyBridge) is a *cooperative* isolation: it intercepts imports and attribute access via

Python-level hooks, but it cannot prevent a sufficiently clever exploit from reaching the C-API directly (e.g. via *ctypes* if *ctypes* were allowed, which it is not). A sub-interpreter provides *enforced* isolation: even if the INTHON code reached the C-API, it could only affect its own interpreter's state, not the host process. INTHON v1.0 uses PyBridge; INTHON v2.0 plans to layer PyBridge on top of sub-interpreter isolation for defense in depth.

17.5 Limitations of Sub-interpreters

Sub-interpreters are not a silver bullet. The shared C extension state means that an extension that stores state in statics (pre-PEP 489) leaks across interpreters, causing subtle bugs. Many popular extensions are not yet sub-interpreter-safe. The *interpreters* module is experimental. And sub-interpreters do not provide memory isolation: a buffer overflow in a C extension can still corrupt the entire process. For true memory isolation, process-level isolation (separate OS processes) is required. INTHON's roadmap (Part V) proposes container runtimes for v1.0 — this would give each INTHON execution its own process, the strongest isolation available.

PART II

INTHON Language Specification

18. INTHON Design Philosophy

INTHON exists because the dominant paradigm for agent-driven computation — emitting JSON tool-call schemas or raw Python source from a language model — is structurally broken. JSON schemas are token-heavy, syntactically brittle (a single missing comma breaks the parse), and have no native control flow. Raw Python is Turing-complete but dangerous: `os.system('rm -rf /')` is one prompt injection away from destroying the host. Neither medium offers a way to bound cost, count tool calls, or replay a specific execution. INTHON is designed to be the medium agents use when they need to express computation that is structured, safe, and auditable.

18.1 The Hierarchy of Language Levels

The INTHON thesis posits a four-level hierarchy of programming languages. **Binary** is the machine level: the only language the CPU executes directly. **Assembly** is the low level: a human-readable mapping of binary instructions, one mnemonic per machine instruction. **High-level languages** (Python, Rust, C++, Java) are the human level: abstractions for memory, control flow, and types that humans can write and maintain. **INTHON** is the agent level: abstractions for tools, policies, memory, and approval that AI agents can generate and execute directly. Just as assembly automated the translation of mnemonics to binary, and compilers automated the translation of high-level languages to assembly, INTHON aims to be the language that agents write and that compiles to deterministic, auditable execution.

18.2 The Three Pillars

18.2.1 Token-Efficient Grammar

INTHON's grammar is engineered for minimum token cost when generated by an LLM. Every keyword is short (*let*, *fn*, *use*, *if*, *for*). Blocks are brace-delimited (no INDENT/DEDENT tokens to mispredict). Type annotations are optional. Imports use a structured *use py.X* / *use tool X* / *use memory X* form that is more compact than Python's *import X as Y* and unambiguously signals intent. Benchmarks in the INTHON repo show 43-76% token reduction versus equivalent JSON or natural-language plans.

18.2.2 Capability-Based Sandbox

Every side effect in INTHON is gated by a capability. Tool calls require the *use tool* declaration. Python module imports require the *use py* declaration and must be on the policy's allowlist. Memory operations require *use memory*. The sandbox enforces three resource quotas: maximum tool calls, maximum cost in USD, and maximum runtime in seconds. Exceeding any quota raises a policy violation and halts execution. The PyBridge import hook intercepts every Python import, blocking *os*, *sys*, *subprocess*, *ctypes*, *socket*, and similar modules at the C-API level before they can be loaded.

18.2.3 Traceable Execution

Every INTHON execution emits a JSON trace tree recording every expression evaluation, every tool call (with arguments, result, cost, and latency), every policy check, every memory operation, every approval gate, and every retry attempt. The trace is deterministic — given the same source, the same tool responses, and the same random seed, the trace is byte-identical. This makes INTHON executions replayable for debugging, auditable for compliance, and reproducible for benchmarking. The trace is also the substrate for the self-evaluation primitive (Part III).

18.3 What INTHON is Not

INTHON is not a general-purpose replacement for Python. It is a domain-specific language for agent-driven computation. It does not aim to be fast (its VM is ~50× slower than CPython for tight loops), feature-rich (no metaclasses, no decorators, no `async/await` in v1.0), or ergonomic for humans (the syntax is tuned for LLM generation, not human readability). It is also not a framework: LangChain, AutoGen, and similar frameworks are libraries that wrap a host language; INTHON is a language that embeds the framework concepts as first-class syntax. The *agent* block, *policy*, *plan*, *approve*, *remember*, *retry*, and *eval self* constructs are language primitives, not library calls.

18.4 The Origin: Why 'INTHON' and Not 'AITHON'

The project was originally named AITHON (AI + Python). The team dropped 'AI' as overused marketing noise and chose 'Intelligent' as a more sober qualifier, yielding INTHON. The name is not load-bearing — the project is willing to rename if the community proposes something better. What is load-bearing is the design: a Python-hosted, agent-targeted, capability-bounded, trace-first language layer.

19. INTHON Lexical Structure

INTHON's lexer is generated by Lark from the EBNF grammar in Appendix A. The token set is small and intentionally close to Python's, with three significant departures: blocks are brace-delimited, indentation is insignificant, and the `//` introduces a line comment (rather than Python's `#`). The full grammar is in Appendix A; this section describes the tokens and their lexer rules.

19.1 Keywords

Keyword	Purpose
let	Mutable variable declaration
const	Immutable constant declaration
fn	Function declaration
agent	Agent block declaration
goal	Goal directive inside agent
inputs	Typed input interface
outputs	Typed output interface
use	Import (tool/py/memory)
policy	Policy block inside agent
plan	Execution body inside agent
return	Return statement
if	Conditional
else	Conditional alternative

Keyword	Purpose
for	For-in loop
while	While loop
approve	HITL approval gate
before	Approval gate clause
remember	Persist to memory
recall	Retrieve from memory
forget	Delete from memory
guard	Assert condition (raises on false)
retry	Retry block with backoff
with	Retry clause (backoff type)
backoff	Retry clause
catch	Retry catch clause
eval	Self-evaluation statement
against	Self-evaluation clause
in	For-in, remember-in, recall-from clause
from	Recall-from, forget-from clause
true	Boolean literal
false	Boolean literal
none	None literal

Table 19.1 — INTHON keyword set (v1.0 target).

19.2 Identifiers

Identifiers match the regex `/[A-Za-z_][A-Za-z0-9_]*/`. Unicode identifiers are not supported in v1.0 (a deliberate simplification for LLM tokenization). Keywords are reserved and cannot be used as identifiers. The lexer uses Lark's *CNAME* terminal with priority adjusted so keywords win over identifiers when they collide.

19.3 Literals

Literal	Pattern	Example
Integer	<code>/[0-9]+/</code>	42, 1000000
Float	<code>/[0-9]+\.[0-9]*([eE][+-]?[0-9]+)?/</code>	3.14, 1e-3
String	<code>^"(?:[^\\"\\] \\.)*\"/</code>	"hello", 'world'
Boolean	<code>true false</code>	true, false

Literal	Pattern	Example
None	none	none

Table 19.2 — INTHON literal forms.

19.4 Operators and Delimiters

INTHON supports the standard arithmetic operators (+ - * / % **), comparison (== != < <= > >=), logical (*and or not*), assignment (=), and the arrow (->) for function return type annotations. Delimiters are: () [] { }, : . ;. Compound assignment (+= etc.) is not in v1.0 — use $x = x + 1$.

19.5 Comments

Line comments begin with // and run to end of line. Block comments use /* ... */ with the /s flag for multi-line matching. Both are discarded by the lexer via %ignore directives. Python's # comment is not supported — the choice of // aligns with JavaScript/Rust/Go, which are more familiar to the LLMs that generate INTHON code.

19.6 Whitespace

Whitespace (spaces, tabs, newlines) is insignificant except as a token separator. The lexer ignores all whitespace via %ignore /[\t\r\n]/+/. There is no INDENT/DEDENT token stream — block structure is purely brace-driven. This is the single largest surface difference from Python and the one that most reduces LLM syntax errors.

20. Type System

INTHON's type system is a hybrid: types are checked statically by the semantic analyzer where possible, and dynamically at runtime where the static check is infeasible. The system is intentionally simpler than Python's *typing* module: no generics beyond parameterized collections, no TypeVar, no Protocol, no overload. The goal is to give the LLM enough type information to catch obvious errors (passing a string where an int is expected) without the complexity of a full dependent type system.

20.1 Primitive Types

Type	Values	Notes
str	Unicode strings	Immutable, like Python str
int	Arbitrary-precision integers	Backed by Python int
float	IEEE 754 double	Backed by Python float
bool	true, false	Backed by Python bool
bytes	Byte sequences	Backed by Python bytes
none	none	The None type
any	Any value	Disables static type checking for the binding

Table 20.1 — INTHON primitive types.

20.2 Collection Types

INTHON supports four parameterized collection types: *list[T]* (homogeneous list), *dict[K, V]* (homogeneous dict), *tuple[T, ...]* (homogeneous tuple) and *tuple[T1, T2, T3]* (heterogeneous fixed-length tuple), and *set[T]*. The element type may itself be a collection (*list[dict[str, int]]*). The *any* element type disables element type checking. INTHON does not support union types in v1.0 — use *any* for variables that may hold values of multiple types.

20.3 Agent and Execution Types

INTHON defines a set of types that name first-class language constructs. These types are mostly informational (they appear in type annotations) rather than instantiable — you cannot *let x: Goal = Goal()* because *Goal* is not a constructor. They exist to type the results of built-in primitives: *recall* returns *MemoryRef*, *approve* returns *Approval*, the agent block itself has type *Plan*, etc.

Type	Produced by	Purpose
Goal	agent goal directive	The agent's execution target
Plan	agent plan block	The execution body
ToolCall	CALL_TOOL opcode	A tool invocation record
ToolResult	CALL_TOOL opcode	A tool invocation result
Trace	execution trace	The JSON trace tree
MemoryRef	recall primitive	A reference to a stored memory
Approval	approve primitive	An HITL approval decision
Policy	policy block	The active policy
DataFrame	use py.pandas	Pandas DataFrame
Tensor	use py.torch	PyTorch Tensor
Model	use py.transformers	An ML model
Dataset	use py.datasets	An HF dataset
Embedding	memory.semantic	A vector embedding

Table 20.2 — INTHON agent and execution types.

20.4 Type Annotations

Type annotations are optional everywhere. *let x = 5* infers *int*; *let x: int = 5* is explicit. Function parameters and return types use *name: type* and *-> type*: *fn add(a: int, b: int) -> int*. Agent inputs and outputs use the same form. Type annotations are evaluated at compile time and stored in the IR; they are not re-evaluated at runtime. This differs from Python's lazy annotation evaluation (PEP 563) and is a deliberate choice for traceability: the IR must contain the type information for static analysis.

20.5 Type Checking

The semantic analyzer performs the following checks: (1) parameter types match the declared types at call sites; (2) return types match the declared return type; (3) variable assignments match the declared type (or the inferred type if no annotation); (4) tool calls match the tool's registered schema; (5) memory operations reference declared memory namespaces. Type errors are warnings by default in v0.2 (the program still runs); they will be hard errors in v1.0. The *any* type disables checking for that binding.

20.6 What is Missing

INTHON v1.0 omits: union types (*int | str*), optional types (*int?*), TypeVar (generics beyond parameterized collections), Protocol (structural typing), Literal types, TypedDict, callable types, and overload. These are deliberate omissions for v1.0; they may be added in v1.1+ if real-world usage demands them. The principle is to start with the smallest type system that catches the most common errors and expand based on evidence.

21. Statements & Control Flow

INTHON's statement set is a deliberate subset of Python's, augmented with the agent-oriented primitives (*approve*, *remember*, *recall*, *forget*, *guard*, *retry*, *eval self*). The control flow statements (*if*, *for*, *while*, *return*) have Python-like semantics but brace-delimited bodies. There is no *try/except* statement — error handling for tool calls is via *retry*, and unrecoverable errors propagate as exceptions to the host.

21.1 Variable and Constant Declarations

```
let name: str = "INTHON"
let version: float = 0.2
const max_retries: int = 3

// Collections
let models: list[str] = ["gpt-4o", "gemini-3.5", "claude-3"]
let metadata: dict[str, any] = {"accuracy": 0.98, "epochs": 10}
```

let declares a mutable binding; *const* declares an immutable one. The type annotation is optional. Reassigning a *const* raises a compile-time error. Declarations are block-scoped: a variable declared inside *{ ... }* is not visible outside. Shadowing is allowed but generates a warning.

21.2 Assignment

```
let x = 10
x = x + 5
x = some_function(x)
```

Plain assignment (*=*) reassigns an existing binding. The left-hand side must be a simple identifier or an attribute/index expression (*obj.attr = value*, *list[i] = value*). Compound assignment (*+=*, *-=*, etc.) is not supported in v1.0; use *x = x + 1*.

21.3 Conditional

```
let threshold = 0.85
let confidence = 0.92

if confidence > threshold {
```

```

return "Validation Success"
} else {
return "Validation Failure"
}

```

if evaluates a boolean expression (truthiness follows Python rules: empty collections, 0, none, false, and "" are falsy). The *else* branch is optional. There is no *else if* — chain via *if ... { } else { if ... { } else { } }*. Pattern matching (*match*) is not in v1.0.

21.4 For Loop

```

let total = 0
for i in range(10) {
total = total + i
}
return total // 45

```

for x in expr iterates over the result of *expr*, which must be an iterable. *range* is a builtin. The loop variable *x* is block-scoped to the loop body. There is no *break* or *continue* in v1.0 — early exit uses *return* from the enclosing function. This is a deliberate restriction to keep the IR simple and the trace deterministic.

21.5 While Loop

```

let i = 0
while i < 10 {
i = i + 1
}
return i

```

while expr loops while *expr* is truthy. The runtime enforces a maximum iteration count (default 10,000, configurable via policy) to prevent infinite loops in agent-generated code — exceeding the limit raises *SandboxViolationError*. This is a critical safety mechanism: an LLM that emits an infinite loop would otherwise exhaust the host's resources.

21.6 Function Declaration

```

fn greet(name: str) -> str {
return "Hello, " + name + "!"
}

let message = greet("Developer")
message // implicit return

```

Functions are declared with *fn name(params) -> ret_type { body }*. The return type annotation is optional. Parameters may have default values (*x: int = 0*). Nested function declarations create closures: the inner function captures the outer function's locals. The implicit return rule: if the last statement in a block is a bare expression (no *return* keyword, no trailing semicolon), the expression's value is popped from the stack and returned. This mirrors Rust's behavior and is more token-efficient than always writing *return*.

21.7 Guard Statement

```

let response = web.search(query)
guard response.status == 200
// execution continues only if guard passes

```

guard expr asserts that *expr* is truthy. If false, it raises *GuardAssertionError*. Inside a *retry* block, this triggers a retry; outside, it propagates as an uncaught exception. *guard* is the primary mechanism for expressing preconditions in INTHON — it is more token-efficient than *if not cond: raise* and makes the precondition explicit at the call site.

21.8 Retry Statement

```
retry 3 with backoff exponential {
  let response = web.search(query)
  guard response.status == 200
} catch error {
  return "Failed after 3 attempts: " + error.message
}
```

retry N with backoff TYPE { body } catch name { handler } executes *body* up to *N+1* times. If *body* raises or a *guard* fails, the runtime waits (using the backoff strategy), then retries. The backoff interval for *exponential* is $base \times 2^{attempt} \pm jitter$ where *base* defaults to 1.0 second and *jitter* is a random offset in [-0.5, +0.5] seconds to prevent thundering-herd effects. After *N+1* attempts, the *catch* handler runs with the last error bound to *name*. If no *catch* is provided, the error propagates.

21.9 Return Statement

```
fn square(x: int) -> int {
  return x * x
}
```

return expr evaluates *expr*, pops the current frame, and returns the value to the caller. *return* with no expression returns *none*. At the top level of a script (outside any function), *return* ends the script and returns the value to the host.

22. Expressions & Operators

INTHON's expression grammar is a standard precedence-climbing parser written in Lark EBNF. The precedence levels are (lowest to highest): *or*, *and*, *not*, comparison (*==* *!=* *<* *<=* *>* *>=*), additive (*+* *-*), multiplicative (*** */* *%*), unary (*- not*), power (****), postfix (*call*, *index*, *attribute*, *method chain*), primary (literals, identifiers, parenthesized).

22.1 Operator Precedence Table

Level	Operators	Associativity
1 (lowest)	or	Left
2	and	Left
3	not	Unary
4	== != < <= > >=	Left (chained comparisons not allowed in v1.0)
5	+ -	Left
6	* / %	Left
7	unary - (negation)	Unary

Level	Operators	Associativity
8	<code>**</code>	Right
9	postfix (call, index, attr, method)	Left
10 (highest)	literals, identifiers, parenthesized	—

Table 22.1 — INTHON operator precedence (lowest to highest).

22.2 Call Expressions

```
let result = my_function(arg1, arg2, key1: val1, key2: val2)
```

Calls support positional and keyword arguments. Keyword arguments use *name: value* syntax (note: colon, not equals). Positional arguments must precede keyword arguments. The function's parameter defaults are applied for any missing arguments. Argument types are checked against the function's declared parameter types by the semantic analyzer.

22.3 Attribute and Index Access

```
let df = pd.DataFrame(data)
let column = df["price"]
let mean = column.mean()
let first_item = items[0]
```

Attribute access (*obj.attr*) and index access (*obj[expr]*) use the standard syntax. Method calls (*obj.method(args)*) are a postfix expression that combines attribute access and call — they are parsed as a unit and compiled to a single `METHOD_CALL` opcode for efficiency.

22.4 List, Dict, and Tuple Literals

```
let numbers = [1, 2, 3, 4, 5]
let mapping = {"a": 1, "b": 2, "c": 3}
let pair = (1, "hello")
```

List literals (*[...]*), dict literals (*{k: v, ...}*), and tuple literals (*(...)*) follow Python's syntax. Empty list is *[]*; empty dict is *{}*; empty tuple is *()*. Set literals are not supported in v1.0 — use *set_from_list([...])* from the standard library.

22.5 String Concatenation

```
let name = "INTON"
let greeting = "Hello, " + name + "!"
// "Hello, INTON!"
```

Strings are concatenated with `+`. There is no f-string syntax in v1.0 — use explicit `+` concatenation or the *format* builtin. F-strings are planned for v1.1 once the Lark grammar extension is stable.

22.6 Member Method Chains

```
let result = data.filter(lambda x: x > 0).map(lambda x: x * 2).sum()
```

Method chains (*obj.method().method().method()*) are common in data science code and INTHON supports them via the *method_chain* production in the grammar. Each link in the chain compiles to a `METHOD_CALL` opcode that consumes the previous result as the receiver. Note that lambdas are not in v1.0; use named functions.

23. The Agent Block Specification

The *agent* block is INTHON's central abstraction. It encapsulates a complete unit of agent work: a goal directive, typed input and output interfaces, the tools and Python modules the agent may use, a policy block constraining its execution, and a plan body that executes within the resulting capability sandbox. Every INTHON program is, at its top level, a sequence of agent declarations and supporting function definitions.

23.1 Syntax

```
agent Name {
  goal "Description of what this agent does"
  inputs {
    param1: type1
    param2: type2
  }
  outputs {
    result: type
  }
  use tool tool1
  use py.module1 as alias1
  use memory.namespace1
  policy {
    max_tool_calls: 10
    max_cost_usd: 0.05
    max_runtime_sec: 60
  }
  plan {
    // execution body
    return result
  }
}
```

23.2 The Goal Directive

goal takes a string literal. The string is recorded in the trace and is available to the self-evaluation primitive as context for plan rewriting. The goal is not executed; it is metadata. It serves the same role as a docstring in Python but is structured: it is a required field of every agent block, and it is the first thing the trace shows.

23.3 Inputs and Outputs

inputs and *outputs* declare the typed interfaces of the agent. The inputs are bound as locals in the plan block; the outputs declare what the plan must return. The semantic analyzer checks that the plan's return type matches the declared output type. An agent is invoked like a function: *let result = MyAgent(input1: value1, input2: value2)*. The argument types are checked against the declared input types at the call site.

23.4 The use Declarations

Three forms of *use* declare the agent's capabilities: *use tool X* registers a tool (from the tool registry) that the plan may call; *use py.X as Y* imports a Python module via PyBridge, subject to the policy allowlist; *use memory.X* declares a memory namespace the plan may read/write. Attempting to use a tool, module, or namespace not declared in the agent block raises a policy violation at compile time. This is the capability-bounding mechanism: nothing is allowed unless explicitly declared.

23.5 The Policy Block

Entry	Type	Default	Semantics
max_tool_calls	int	50	Maximum number of tool invocations.
max_cost_usd	float	1.0	Maximum cumulative cost in USD.
max_runtime_sec	int	300	Maximum wall-clock execution time.
allow_network	bool	false	Whether tools may make network calls.
filesystem	str	read_only	none read_only read_write
allow_payment	bool	false	Whether payment actions are allowed.
allow_memory_persist	bool	true	Whether memory persists across runs.

Table 23.1 — Policy block entries.

23.6 The Plan Block

plan { ... } is the execution body of the agent. It is a sequence of statements in the standard INTHON statement grammar. The plan runs in its own scope, with the agent's inputs as locals. The plan's return value becomes the agent's output. The plan has access to all tools, modules, and memory namespaces declared via *use* in the enclosing agent block. The plan may declare nested functions, but not nested agents (v1.0 restriction).

23.7 Agent Invocation

```
agent Researcher {
  goal "Search the web"
  inputs { query: str }
  outputs { papers: list[dict] }
  use tool web.search
  policy { max_tool_calls: 5 }
  plan {
    let results = web.search(query: query, count: 10)
    return results
  }
}

let papers = Researcher(query: "room-temperature superconductors")
```

Agents are invoked by name as if they were functions, with keyword arguments matching the declared inputs. The invocation creates a fresh execution context (frame, scope, policy state, cost ledger) and runs the plan. The result is returned to the caller. Agents may be invoked recursively (an agent's plan may invoke another agent), but recursion depth is limited by the runtime (default 16, configurable via policy) to prevent stack overflows in agent-generated code.

23.8 Agent Lifecycle

When an agent is invoked, the runtime: (1) creates a new scope; (2) binds the inputs as locals; (3) activates the policy (pushing a new policy frame); (4) registers the declared tools, modules, and memory namespaces in the scope; (5) begins a new trace subtree; (6) executes the plan; (7) on normal return, pops the policy frame, closes the trace subtree, and returns the result; (8) on exception, records the exception in the trace, pops the policy frame, and re-raises. The policy frame's quotas are isolated to this invocation: a recursive call does not share

the parent's tool-call budget.

24. Built-in Primitives

INTHON's primitives are language-level constructs (not library functions) that implement common agent patterns: human-in-the-loop approval, episodic memory persistence and recall, resilient retries with backoff, and condition guards. Each primitive has a dedicated grammar production, a dedicated opcode, and a dedicated trace event type. This section specifies each in detail.

24.1 Approval Gateways (HITL)

```
approve stripe.charge before make_payment
```

The *approve X before Y* statement suspends execution and emits an approval request to the host. The host (typically a human operator via a UI, or an automated approval service) responds with approve or deny. If approved, execution resumes and *Y* is invoked. If denied, *ApprovalDeniedError* is raised. The first argument (*stripe.charge*) is the *subject* — a dotted name identifying what is being approved; it is recorded in the trace and shown to the approver. The second argument (*make_payment*) is the *action* — a function name to call after approval.

Implementation: the *APPROVE_GATE* opcode is emitted. When executed, the VM calls the host's approval callback (registered via *inthon.set_approval_handler*), passing the subject, action, current trace context, and current cost ledger. The host returns a boolean. The trace records an *ApprovalGate* event with the subject, action, decision, and timestamp. The callback is synchronous: the VM blocks until the decision is returned. For asynchronous approval (e.g. email-based workflow), the host implements a polling loop inside the callback.

24.2 Episodic Memory — remember

```
remember "Superconductors show zero resistance at critical temperatures" in  
semantic_memory  
remember computed_result in session
```

remember EXPR in NAMESPACE persists the value of *EXPR* to the named memory namespace. If *EXPR* is a string, it is stored as a semantic fact with an embedding (computed by the configured embedding model). If *EXPR* is any other value, it is serialized to JSON and stored with a stringified key. The namespace must be declared via *use memory NAMESPACE* in the enclosing agent block. The *AGENT_REMEMBER* opcode is emitted.

Storage: each namespace is a SQLite table with columns *id*, *content* (the string form), *embedding* (a BLOB of float32 values), *metadata* (JSON), *created_at*. The embedding is computed by the configured embedding model (default: sentence-transformers/all-MiniLM-L6-v2, configurable via *inthon.toml*). Cosine similarity is the default retrieval metric.

24.3 Episodic Memory — recall

```
let fact = recall "superconductor properties" from semantic_memory  
let recent = recall "session summary" from session
```

recall QUERY from NAMESPACE retrieves the most semantically similar entry from the named namespace. *QUERY* is a string; it is embedded using the same model as *remember*, and the namespace is searched for the highest cosine-similarity match. The result is a *MemoryRef* object with *content*, *score*, and *metadata* fields. If

the namespace is empty, returns *none*. The *AGENT_RECALL* opcode is emitted.

24.4 Episodic Memory — forget

```
forget "outdated fact" from semantic_memory
```

forget *EXPR* from *NAMESPACE* deletes entries matching *EXPR* from the named namespace. If *EXPR* is a string, it deletes the entry with the closest semantic match (using the same retrieval as *recall*). The *AGENT_FORGET* opcode is emitted. Forgetting is irreversible. The trace records a *MemoryOp* event with *kind=forget*.

24.5 Resilient Retries

```
retry 3 with backoff exponential {  
  let response = web.search(query)  
  guard response.status == 200  
} catch error {  
  return "Failed: " + error.message  
}
```

retry *N* with *backoff* *TYPE* { *body* } *catch* *name* { *handler* } executes *body* up to *N+1* times. If *body* raises an exception or a *guard* fails, the runtime waits (using the backoff strategy), then retries. The backoff strategies are: *exponential* (default, $base \times 2^{\text{attempt}} \pm \text{jitter}$), *linear* ($base \times (attempt + 1)$), *fixed* (*base* seconds every time). The base defaults to 1.0 second and is configurable via policy. After *N+1* attempts, the *catch* handler runs with the last error bound to *name*. If no *catch* is provided, the error propagates.

Implementation: the *RETRY_BEGIN*, *RETRY_END*, and *RETRY_BACKOFF* opcodes bracket the body and handler. *RETRY_BEGIN* pushes a retry frame with the attempt count and backoff parameters. *RETRY_END* pops the retry frame on success. *RETRY_BACKOFF* (emitted at the start of each retry) computes the wait time, sleeps (using *asyncio.sleep* if in an async context, else *time.sleep*), and records a *RetryAttempt* event in the trace.

24.6 Guard

```
let response = web.search(query)  
guard response.status == 200  
guard response.body != none
```

guard *EXPR* asserts that *EXPR* is truthy. If false, it raises *GuardAssertionError* with a message describing the failed condition. Inside a *retry* block, this triggers a retry. Outside, it propagates as an uncaught exception. The *GUARD_ASSERT* opcode is emitted. The trace records a *GuardCheck* event with the condition source and the boolean result.

24.7 Self-Evaluation (preview)

```
eval self against cost_budget {  
  total_cost_usd <= 0.05  
  latency_ms <= 500  
}
```

The *eval self against* *NAME* { *criteria* } statement is the self-evaluation primitive, specified in full in Part III. It evaluates a set of criteria against the current execution context (cost, latency, trace, hallucination score) and optionally triggers a plan rewrite if any criterion fails. This is the most experimental primitive in INTHON and the one that most distinguishes it from Python.

25. PyBridge Sandbox & Module Policy

PyBridge is INTHON's gateway to the Python ecosystem. It allows INTHON programs to import and use Python modules (NumPy, Pandas, PyTorch, etc.) without exposing the host to the full power of Python. The design is two-tier: an import hook intercepts every import at the *sys.meta_path* level, blocking disallowed modules; an attribute proxy wraps every allowed module, intercepting every attribute access to prevent escape vectors.

25.1 The Import Hook

When INTHON code executes *use py.numpy as np*, the semantic analyzer records the import in the agent's capability list. At runtime, the *InthonPyObject* factory calls *importlib.import_module('numpy')*. Before this call, the PyBridge verifies that *numpy* is on the active policy's allowlist; if not, raises *PyBridgeError*. After import, the module is wrapped in an *InthonPyObject* proxy before being bound to *np* in the agent's scope.

The PyBridge does not register a custom meta path finder on *sys.meta_path* — instead, it intercepts the import call at the INTHON-language level. This is simpler and avoids interfering with the host's own import machinery. However, it means that INTHON cannot prevent a malicious Python module from importing *os* internally — but since the allowlist already excludes modules that would do so (e.g. *subprocess*), this is acceptable.

25.2 The InthonPyObject Proxy

InthonPyObject is a Python class that wraps any Python object and intercepts attribute access. Every *getattr* on the proxy: (1) checks if the attribute name is on a denylist (e.g. *__dict__*, *__class__*, *__bases__*, *__subclasses__*); (2) calls the underlying *__getattr__*; (3) wraps the result in another *InthonPyObject* if it is a module, class, or function; (4) returns the wrapped value. Every *setattr* and *call* on the proxy is similarly intercepted.

25.3 The Module Policy Matrix

Category	Approved	Blocked
Operating system & shell	(none)	os, sys, subprocess, ctypes, builtins.eval, builtins.exec
Network protocols	(none — use registered tools)	socket, urllib, requests
Data & scientific math	numpy, pandas, math, json, datetime, collections	Any third-party module with low-level file write
ML frameworks (with [ml] extra)	torch, transformers, datasets, sklearn	TensorFlow (planned for v1.1)
File I/O	(none — use registered tools)	io, pathlib, shutil, os.path
Process control	(none)	multiprocessing, threading, concurrent.futures

Table 25.1 — PyBridge module policy matrix (v0.2 baseline).

25.4 Attack Vectors and Defenses

Attack	Vector	Defense
Direct OS import	use <code>py.os</code>	Allowlist rejects 'os'
Indirect OS import	use <code>py.numpy</code> ; <code>np.os</code>	Proxy intercepts attribute; 'os' not exposed as numpy attribute
Subprocess via eval	use <code>py.builtins</code> ; <code>builtins.eval('subprocess.run(...)')</code>	Allowlist rejects 'builtins'
Dunder traversal	<code>np.__class__.__bases__[0].__subclasses__()</code>	Proxy blocks <code>__class__</code> , <code>__bases__</code> , <code>__subclasses__</code>
Code objects	<code>func.__code__.co_consts</code>	Proxy blocks <code>__code__</code>
Globals traversal	<code>func.__globals__['os']</code>	Proxy blocks <code>__globals__</code>
Import via importlib	use <code>py.importlib</code> ; <code>importlib.import_module('os')</code>	Allowlist rejects 'importlib'
Pickle deserialization	use <code>py.pickle</code> ; <code>pickle.loads(payload)</code>	Allowlist rejects 'pickle'
ctypes	use <code>py.ctypes</code>	Allowlist rejects 'ctypes'

Table 25.2 — Common sandbox escape vectors and PyBridge defenses.

25.5 Limitations

PyBridge is a cooperative sandbox: it relies on the assumption that the INTHON code does not have a way to call the C-API directly. Since INTHON runs inside CPython and the `ctypes` and `ctypes` modules are blocked, this assumption holds for v1.0. However, a sufficiently clever exploit (e.g. via a buffer overflow in an allowed module like NumPy) could in principle bypass the sandbox. For production-grade isolation, INTHON v1.0 recommends running each agent invocation in a separate container (Docker, gVisor, or Firecracker). The PyBridge is the first line of defense; the container is the second.

26. INTHON Compilation Pipeline

INTHON compilation is a five-stage pipeline: Lark LALR parse, AST construction, semantic analysis, policy sandbox validation, and IR lowering. The IR is the input to both the AST-walking interpreter and the bytecode VM. The pipeline is implemented in pure Python in the INTHON package and runs in under 10 milliseconds for typical agent programs (<1KB source).

26.1 Stage 1: Lark Parse

The source string is parsed by Lark using the LALR algorithm and the grammar in Appendix A. Lark produces a parse tree of *Tree* and *Token* objects. LALR is chosen over Earley for performance (LALR is $O(n)$ on input length; Earley is $O(n^3)$ worst case). The grammar is hand-tuned to be LALR(1) — no ambiguity, no lookahead conflicts. Comments and whitespace are discarded by the lexer via `%ignore` directives.

26.2 Stage 2: AST Construction

A Lark *Transformer* walks the parse tree and builds an INTHON AST. Each AST node is a frozen dataclass with typed fields. The AST is immutable: subsequent stages read it but do not modify it. The AST node catalog mirrors the grammar productions: *Program*, *Let*, *Const*, *FnDecl*, *AgentDecl*, *Return*, *Approve*, *Remember*, *Recall*,

Forget, Guard, Retry, Eval, If, For, While, ExprStmt, BinOp, UnaryOp, Compare, Call, Attribute, Subscript, List, Dict, Tuple, Literal, Name.

26.3 Stage 3: Semantic Analysis

The semantic analyzer walks the AST and performs: (1) scope resolution — every name is resolved to a binding (local, enclosing, global, or tool); (2) type checking — parameter types, return types, assignment types, and tool call argument types are checked; (3) capability validation — every *use* declaration must be inside an agent block; every tool/py/memory reference must be declared; (4) policy validation — the policy block's entries must be well-typed and within configured maximums. Errors are collected and reported together (the analyzer does not stop at the first error).

26.4 Stage 4: Policy Sandbox Validation

The validated AST is checked against the active policy: are all requested tools enabled in *inthon.toml*? Are all requested Python modules on the allowlist? Are the policy block's quotas within the global maximums? If any check fails, a *PolicyViolationError* is raised with a list of violations. This stage runs before any code is executed, ensuring that policy violations are caught at compile time rather than runtime.

26.5 Stage 5: IR Lowering

The checked AST is lowered to an IR — a JSON-compatible data structure representing the program as a list of instructions, a constant pool, a name table, and per-agent metadata. The IR is the input to both the AST-walking interpreter (which walks the AST directly, ignoring the IR) and the bytecode VM (which compiles the IR to a list of opcodes). The IR is also the form serialized to *.inthonc* cache files (planned for v1.1) and the form sent to the trace viewer UI.

26.6 The IR Format

```
{
  "version": "1.0",
  "filename": "agent.inth",
  "agents": [
    {
      "name": "Researcher",
      "goal": "...",
      "inputs": [...],
      "outputs": [...],
      "capabilities": [...],
      "policy": {...},
      "instructions": [
        {"op": "LOAD_CONST", "arg": 0},
        {"op": "STORE_NAME", "arg": "query"},
        {"op": "CALL_TOOL", "arg": "web.search", "kw": {"query": "query", "count": 10}},
        {"op": "STORE_NAME", "arg": "results"},
        {"op": "LOAD_NAME", "arg": "results"},
        {"op": "RETURN_VALUE"}
      ]
    }
  ],
  "constants": [...],
  "names": ["query", "results"]
}
```

26.7 The VM Compilation

The IR's instruction list is compiled to a flat bytecode array. Each instruction is encoded as a 2-tuple (*opcode_int*, *oparg*). The opcode is a small integer (see Appendix B); the oparg is an integer index into the constants or names table. Jump targets are resolved to byte offsets. The compiled bytecode is the input to the InthonVM dispatch loop.

27. InthonVM Opcode Reference

The InthonVM is a stack-based bytecode interpreter modeled on CPython's eval loop but simplified. It has 45 opcodes (v0.2), divided into seven categories: load/store, arithmetic/comparison, control flow, call, container, agent primitives, and policy. Each opcode is a small integer (0-44); the dispatch loop is a Python *match* statement (or a Python *dict* of handlers in the v0.2 implementation).

27.1 Load and Store Opcodes

Opcode	Operands	Stack effect	Semantics
LOAD_CONST	const_idx	[] -> [const]	Push constants[const_idx].
LOAD_NAME	name_idx	[] -> [value]	Push names[name_idx] from scope.
LOAD_FAST	local_idx	[] -> [value]	Push locals[local_idx] (fast path).
LOAD_GLOBAL	name_idx	[] -> [value]	Push globals[name_idx].
LOAD_ATTR	name_idx	[obj] -> [value]	Push getattr(obj, names[name_idx]).
STORE_NAME	name_idx	[value] -> []	names[name_idx] = value.
STORE_FAST	local_idx	[value] -> []	locals[local_idx] = value.
STORE_ATTR	name_idx	[value, obj] -> []	setattr(obj, names[name_idx], value).
STORE_SUBSCR	—	[value, container, key] -> []	container[key] = value.
POP_TOP	—	[value] -> []	Discard top of stack.
DUP_TOP	—	[value] -> [value, value]	Duplicate top of stack.

Table 27.1 — Load and store opcodes.

27.2 Arithmetic and Comparison Opcodes

Opcode	Operands	Stack effect	Semantics
BINARY_ADD	—	[a, b] -> [a+b]	Addition.
BINARY_SUB	—	[a, b] -> [a-b]	Subtraction.
BINARY_MUL	—	[a, b] -> [a*b]	Multiplication.
BINARY_DIV	—	[a, b] -> [a/b]	True division.

Opcode	Operands	Stack effect	Semantics
BINARY_MOD	—	[a, b] -> [a%b]	Modulo.
BINARY_POW	—	[a, b] -> [a**b]	Power.
UNARY_NEG	—	[a] -> [-a]	Negation.
UNARY_NOT	—	[a] -> [not a]	Boolean not.
COMPARE_OP	cmp_idx	[a, b] -> [bool]	Comparison (idx selects <, <=, ==, !=, >, >=).

Table 27.2 — Arithmetic and comparison opcodes.

27.3 Control Flow Opcodes

Opcode	Operands	Stack effect	Semantics
POP_JUMP_IF_FALSE	target	[cond] -> []	Jump to target if cond is falsy.
POP_JUMP_IF_TRUE	target	[cond] -> []	Jump to target if cond is truthy.
JUMP_FORWARD	delta	[] -> []	Relative forward jump.
JUMP_ABSOLUTE	target	[] -> []	Absolute jump.
GET_ITER	—	[iterable] -> [iterator]	iter(iterable).
FOR_ITER	target	[iterator] -> [next]	Push next; on StopIteration jump to target.
RETURN_VALUE	—	[value] -> []	Return from current frame.

Table 27.3 — Control flow opcodes.

27.4 Call Opcodes

Opcode	Operands	Stack effect	Semantics
CALL_FUNCTION	argc	[callable, args...] -> [result]	Call with argc positional args.
CALL_FUNCTION_KW	argc	[callable, args..., kwnames] -> [result]	Call with keyword args.
CALL_METHOD	argc	[method, self, args...] -> [result]	Optimized method call.
MAKE_FUNCTION	—	[code, qualname] -> [func]	Create function object.

Table 27.4 — Call opcodes.

27.5 Container Opcodes

Opcode	Operands	Stack effect	Semantics
BUILD_LIST	count	[items...] -> [list]	Build list of count items.
BUILD_TUPLE	count	[items...] -> [tuple]	Build tuple.
BUILD_DICT	count	[kvs...] -> [dict]	Build dict.
BUILD_SLICE	—	[stop, start] -> [slice]	Build slice.
BINARY_SUBSCR	—	[container, key] -> [value]	container[key].

Table 27.5 — Container opcodes.

27.6 Agent Primitive Opcodes

Opcode	Operands	Stack effect	Semantics
CALL_TOOL	tool_idx	[args...] -> [result]	Invoke tool from registry; checks policy quota.
APPLY_POLICY	policy_idx	[] -> []	Push policy frame with quotas.
POP_POLICY	—	[] -> []	Pop policy frame.
APPROVE_GATE	subject, action	[] -> [approval]	Emit HITL approval request.
AGENT_REMEMBER	namespace_idx	[value] -> []	Persist value to memory namespace.
AGENT_RECALL	namespace_idx	[query] -> [MemoryRef none]	Semantic recall from namespace.
AGENT_FORGET	namespace_idx	[query] -> []	Delete matching entry from namespace.
GUARD_ASSERT	src_idx	[cond] -> []	Raise GuardAssertionError if cond falsy.
RETRY_BEGIN	max_attempts, backoff	[] -> []	Push retry frame.
RETRY_BACKOFF	—	[] -> []	Sleep per backoff strategy; record event.
RETRY_END	—	[] -> []	Pop retry frame on success.
IMPORT_PY	module_idx	[] -> [InthonPyObject]	Import via PyBridge; wrap in proxy.
SELF_EVAL	criteria_idx	[] -> [EvalResult]	Evaluate criteria; trigger rewrite if any fail (Part III).
INTROSPECT_TRACE	—	[] -> [TraceView]	Push current trace as a queryable object.
REWRITE_PLAN	new_plan_idx	[] -> []	Replace remaining plan instructions (Part III).

Table 27.6 — Agent primitive opcodes. The last three (*SELF_EVAL*, *INTROSPECT_TRACE*, *REWRITE_PLAN*) are specified in Part III.

27.7 Trace Emission

Every opcode that has an observable side effect (*CALL_TOOL*, *APPROVE_GATE*, *AGENT_REMEMBER*, *AGENT_RECALL*, *AGENT_FORGET*, *GUARD_ASSERT*, *RETRY_BACKOFF*, *APPLY_POLICY*, *SELF_EVAL*, *REWRITE_PLAN*) emits an entry to the trace. The trace entry contains: opcode name, operand values, stack state before/after, timestamp, duration, cost delta, and any result. The trace is a list of these entries, serialized to JSON at the end of execution.

28. Sandbox & Policy Engine

The policy engine is the runtime enforcer of INTHON's capability model. Every tool call, every memory operation, every approval gate, and every iteration of a loop checks the active policy before proceeding. Exceeding any quota raises *SandboxViolationError* and halts execution. The policy is configured via *inthon.toml* at the project level and may be tightened (but not loosened) per-agent via the *policy* block.

28.1 The Policy Schema

```
[sandbox]
max_runtime_sec = 300 # maximum wall-clock execution time
max_cost_usd = 1.0 # maximum cumulative cost in USD
max_tool_calls = 50 # maximum number of tool invocations
max_iterations = 10000 # maximum loop iterations

[permissions]
network = false # allow network calls (tools only)
filesystem = "read_only" # none | read_only | read_write
shell = false # allow shell commands (never in v1.0)
payment = false # allow payment actions
memory_persist = true # persist memory across runs

[pybridge]
allowed_modules = ["math", "json", "datetime", "collections", "random", "time"]

[tools]
web.search = true
web.read = true

[trace]
enabled = true
level = "info" # debug | info | warn | error
output_dir = ".inthon/traces"
```

28.2 Runtime Enforcement Points

Quota	Enforcement point	Exception on violation
max_tool_calls	CALL_TOOL opcode	SandboxViolationError
max_cost_usd	CALL_TOOL opcode (after cost return)	SandboxViolationError
max_runtime_sec	Eval breaker (every 100 opcodes)	SandboxViolationError

Quota	Enforcement point	Exception on violation
max_iterations	FOR_ITER and JUMP_BACKWARD opcodes	SandboxViolationError
allowed_modules	IMPORT_PY opcode	PyBridgeError
network permission	CALL_TOOL for network tools	PolicyViolationError
filesystem permission	CALL_TOOL for fs tools	PolicyViolationError
payment permission	APPROVE_GATE for payment actions	PolicyViolationError

Table 28.1 — Policy enforcement points.

28.3 The Policy Frame Stack

Policy frames are stacked: each agent invocation pushes a frame with its own quotas, and the runtime enforces the *minimum* of all active frames. This means a sub-agent cannot exceed the parent's quotas even if its own policy declares higher limits. When a sub-agent returns, its frame is popped and the parent's quotas are unchanged. This is the mechanism that prevents a malicious sub-agent from busting the parent's budget.

28.4 The Cost Ledger

Every tool call returns a cost in USD (the tool's *cost_usd* field). The cost ledger accumulates these costs per agent invocation and globally across the session. Before each tool call, the runtime checks if the call would exceed the remaining budget; if so, *SandboxViolationError* is raised. The cost ledger is part of the trace and is queryable by the self-evaluation primitive.

28.5 Exception Taxonomy

Exception	Raised by	Recoverable?
PolicyViolationError	Capability check (tool not declared, module not on allowlist)	No (compile-time)
PyBridgeError	Import-time sandbox violation	No
SandboxViolationError	Runtime quota exceeded	No (halts execution)
ApprovalDeniedError	HITL gate denied	Yes (catch in retry)
GuardAssertionError	guard statement failed	Yes (catch in retry)
InthonRuntimeError	Generic runtime error	Yes (catch in retry)

Table 28.2 — INTHON exception taxonomy.

28.6 Rollback Semantics

When a policy violation halts execution, the runtime attempts to roll back any side effects that occurred during the current agent invocation. Memory writes via *remember* are reverted (the entries are deleted). Tool calls are not reverted (they are external services; rollback is impossible). The trace records the rollback, including which memory entries were deleted. The cost ledger is not rolled back — the cost was incurred even if the result is discarded.

29. Episodic Memory Subsystem

INTHON's episodic memory is a SQLite-backed key-value store with semantic retrieval. Each memory namespace is a SQLite table; each entry has a string content, a vector embedding, JSON metadata, and a timestamp. Storage uses *remember*, retrieval uses *recall* (cosine similarity over embeddings), deletion uses *forget*. The embedding model is pluggable (default: sentence-transformers/all-MiniLM-L6-v2).

29.1 The SQLite Schema

```
CREATE TABLE memory_ (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  content TEXT NOT NULL,  
  embedding BLOB NOT NULL, -- float32 vector, little-endian  
  metadata TEXT NOT NULL DEFAULT "{}",  
  created_at TEXT NOT NULL DEFAULT (datetime("now")),  
  namespace TEXT NOT NULL  
);  
CREATE INDEX idx_memory_namespace ON memory_(namespace);  
CREATE INDEX idx_memory_created ON memory_(created_at);
```

29.2 The Embedding Pipeline

When *remember* *EXPR* in *NAMESPACE* is executed: (1) *EXPR* is evaluated to a value *v*; (2) if *v* is a string, it is the content; otherwise, *v* is JSON-serialized and the serialization is the content; (3) the embedding model encodes the content to a 384-dimensional float32 vector (for MiniLM-L6-v2); (4) the vector is packed to bytes (little-endian float32); (5) a row is inserted into *memory_NAMESPACE* with the content, embedding, and metadata.

29.3 Cosine Similarity Retrieval

When *recall* *QUERY* from *NAMESPACE* is executed: (1) *QUERY* is encoded to a vector *q* using the same model; (2) all rows in *memory_NAMESPACE* are loaded; (3) for each row, the cosine similarity $\cos(q, \text{row.embedding})$ is computed; (4) the row with the highest similarity is returned as a *MemoryRef* with *content*, *score*, and *metadata*. If the namespace is empty, returns *none*.

Performance: for namespaces with up to ~10,000 entries, the brute-force scan is fast enough (<100ms). For larger namespaces, INTHON v1.1 will add an indexed approximate nearest-neighbor (ANN) index using *hnswlib* or *faiss*. The v0.2 implementation is brute-force — sufficient for typical agent workflows but not for production-scale RAG.

29.4 Namespaces

Memory namespaces are arbitrary identifiers declared via *use memory X*. The convention is: *semantic_memory* for long-term facts (persists across sessions); *session* for within-session memory (cleared at session end); *task_* for task-specific scratchpad memory. Custom namespaces may be declared freely — the runtime creates the SQLite table on first use.

29.5 Persistence

Memory persists across INTHON runs by default (controlled by *memory_persist* in the policy). The SQLite database is stored at *.inthon/memory.db* in the project root (configurable via *inthon.toml*). When *memory_persist = false*, the database is in-memory and discarded at session end. The database may be backed up, inspected, or migrated manually using the *sqlite3* CLI or any SQLite client.

29.6 The Embedding Model API

The embedding model is abstracted by the *EmbeddingModel* protocol: *encode(text: str) -> list[float]*. The default implementation uses *sentence-transformers* (requires the *[ml]* extra). For environments without ML dependencies, a hash-based fallback (*sklearn's HashingVectorizer*) provides degraded but functional semantic search. The fallback is automatically selected if *sentence-transformers* is not installed.

30. Trace & Replay Format

Every INTHON execution emits a JSON trace tree. The trace is the authoritative record of what happened: every expression evaluation, every tool call (with arguments, result, cost, latency), every policy check, every memory operation, every approval gate, every retry attempt, and every self-evaluation. The trace is deterministic — given the same source, the same tool responses, and the same random seed, the trace is byte-identical. This makes INTHON executions replayable, auditable, and reproducible.

30.1 The Trace Schema

```
{
  "version": "1.0",
  "source": "agent.inth",
  "started_at": "2026-07-13T10:30:00.000Z",
  "ended_at": "2026-07-13T10:30:01.234Z",
  "duration_ms": 1234,
  "total_cost_usd": 0.0123,
  "status": "success", // success | error | sandbox_violation
  "root": {
    "type": "AgentInvocation",
    "name": "Researcher",
    "goal": "...",
    "inputs": {...},
    "outputs": {...},
    "children": [
      { "type": "ExprEval", "src": "let query = ...", "result": "...", "duration_ms": 0.1 },
      { "type": "ToolCall", "tool": "web.search", "args": {...}, "result": {...},
        "cost_usd": 0.001, "duration_ms": 450 },
      { "type": "MemoryOp", "kind": "remember", "namespace": "semantic_memory", "content":
        "..." },
      { "type": "ApprovalGate", "subject": "...", "action": "...", "decision": "approved",
        "waited_ms": 5000 },
      { "type": "RetryAttempt", "attempt": 1, "max_attempts": 3, "error": "...",
        "backoff_ms": 1000 },
      { "type": "GuardCheck", "src": "response.status == 200", "passed": true },
      { "type": "PolicyApply", "policy": {...} },
      { "type": "SelfEval", "criteria": [...], "results": [...], "rewrite_triggered": false
    }
  ]
}
```

30.2 Event Types

Type	Fields	When emitted
AgentInvocation	name, goal, inputs, outputs, children	Agent called
ExprEval	src, result, duration_ms	Every expression statement
ToolCall	tool, args, result, cost_usd, duration_ms, error?	CALL_TOOL opcode
ToolResult	tool, result, cost_usd, duration_ms	Tool returns (embedded in ToolCall)
MemoryOp	kind (remember recall forget), namespace, content, result?	Memory opcodes
ApprovalGate	subject, action, decision, waited_ms	APPROVE_GATE opcode
RetryAttempt	attempt, max_attempts, error, backoff_ms	RETRY_BACKOFF opcode
GuardCheck	src, passed	GUARD_ASSERT opcode
PolicyApply	policy	APPLY_POLICY opcode
PolicyViolation	kind, message	Policy check fails
SelfEval	criteria, results, rewrite_triggered	SELF_EVAL opcode
PlanRewrite	old_plan, new_plan, reason	REWRITE_PLAN opcode
Exception	type, message, traceback	Uncaught exception

Table 30.1 — Trace event types.

30.3 Replay Semantics

A trace can be replayed: the replay engine reads the trace and re-runs the execution, returning the same outputs and emitting the same trace. Tool calls are not re-executed — the replay engine returns the recorded result. Memory operations are not re-applied — the replay engine checks the recorded result against the current memory state. This makes replay safe (no side effects) and fast (no network calls). Replay is used for debugging, for testing agent changes against historical executions, and for the self-evaluation primitive's introspection.

30.4 Trace-to-Source Mapping

Every trace event includes a *src* field with the source code location (filename, line, column) of the originating statement. This allows the trace viewer UI to highlight the source code that produced each event. The mapping is 1:1 for statement-level events (ExprEval, ToolCall, etc.) and 1:N for compound events (AgentInvocation contains many children).

30.5 The Trace Viewer UI

INTHON ships a web-based trace viewer (in the *ui/* directory). Running *inthon trace* starts a local HTTP server that renders the trace as an interactive tree. Each node is expandable to show its fields; tool call nodes show the request and response as JSON; memory nodes show the embedded content; retry nodes show the attempts and errors. The viewer is the primary debugging tool for INTHON agents.

31. Standard Library & Tool Registry

INTHON's standard library is intentionally tiny: a handful of built-in functions available in every scope, plus a tool registry that hosts the actual capabilities (web search, web read, etc.). The tool registry is the extension point — third-party packages register new tools, and INTHON's policy system gates access to them.

31.1 Built-in Functions

Name	Signature	Purpose
range	(stop: int) -> list[int] / (start, stop, step)	Generate integer list
len	(x: any) -> int	Length of collection
print	(...args: any) -> none	Print to stdout (debug only)
str	(x: any) -> str	String representation
int	(x: any) -> int	Convert to int
float	(x: any) -> float	Convert to float
bool	(x: any) -> bool	Convert to bool
list	(x: any) -> list	Convert to list
dict	(x: any) -> dict	Convert to dict
set_from_list	(x: list) -> set	Create a set (no set literal in v1.0)
format	(template: str, ...args) -> str	String formatting
type	(x: any) -> str	Type name as string

Table 31.1 — INTHON built-in functions (v1.0 target).

31.2 The Tool Registry

Tools are registered via `inthon.tools.register(name, schema, handler)`. The schema declares the tool's parameters (name, type, required, default) and return type. The handler is a Python callable that executes the tool. The registry is the source of truth for what tools exist; the `use tool X` declaration in an agent block references the registry by name.

31.3 Built-in Tools

Tool	Parameters	Returns	Cost model
web.search	query: str, count: int = 10	list[dict]	\$0.001 per call + \$0.0001 per result
web.read	url: str	dict (content, status, headers)	\$0.0005 per call + \$0.0001 per KB
web.fetch	url: str	bytes	\$0.0001 per KB

Tool	Parameters	Returns	Cost model
llm.complete	model: str, prompt: str, max_tokens: int = 1000	str	pass-through (OpenAI pricing)
llm.embed	model: str, text: str	list[float]	\$0.0001 per call
file.read	path: str (sandbox-relative)	str	free
file.write	path: str, content: str	none	free (requires filesystem=read_write)
http.get	url: str, headers: dict = {}	dict	\$0.0001 per call
http.post	url: str, body: any, headers: dict = {}	dict	\$0.0001 per call

Table 31.2 — Built-in tools (v1.0 target). Costs are illustrative.

31.4 Tool Schema Validation

Every *CALL_TOOL* opcode triggers schema validation: the arguments are checked against the tool's declared parameters (type, required, default). Missing required arguments raise *ToolCallError* at compile time (caught by the semantic analyzer). Type mismatches raise *ToolCallError* at runtime. The validation is the first line of defense against LLM-generated tool calls with wrong arguments.

31.5 OpenAPI Autogen (Roadmap)

INTHON v0.3 will introduce OpenAPI autogen: given an OpenAPI 3.x spec, the runtime generates INTTHON tool declarations for every endpoint. This allows agents to call arbitrary REST APIs without manual tool registration. The generated tools are namespaced by the API name (e.g. *stripe.charge*, *github.create_issue*) and inherit the policy's cost and quota settings.

PART III

Self-Evaluation Primitive

32. Self-Evaluation: Motivation & Threat Model

The self-evaluation primitive is INTHON's most experimental feature and the one that most distinguishes it from Python. The motivation is straightforward: agents fail. They hallucinate tool responses, they exceed budgets, they emit plans that loop forever, they produce outputs that contradict their inputs. Today, these failures are detected by external monitors (LangChain callbacks, human review) or not at all. Self-evaluation makes the detection a language-level construct: the agent's plan can include explicit criteria against which the running execution is checked, and the plan can be rewritten in response to a failed criterion.

32.1 The Core Idea

eval self against NAME { criteria } evaluates a set of named criteria against the current execution context. The context includes: the cost ledger (total spent so far), the latency accumulator (total wall-clock time), the trace (the full tree of events so far), the active plan (the remaining instructions), and any user-defined metrics (e.g. a hallucination score computed by a tool). If all criteria pass, execution continues. If any criterion fails, the runtime can (depending on the *eval* statement's options) trigger a plan rewrite: the agent's plan is replaced with a new plan (provided by a registered rewriter), and execution resumes from the top of the new plan.

32.2 Why This Belongs in the Language

Self-evaluation could be implemented as a library function: *check_criteria(...)* that returns a boolean, with the agent's plan calling it and branching on the result. The argument for making it a language primitive is threefold. First, the criteria are part of the agent's contract, not its implementation — they should be visible at the agent block level, not buried inside the plan. Second, plan rewriting requires access to the runtime's plan representation, which is not exposed to user code; a language primitive can access it. Third, the trace should record self-evaluations as first-class events, which requires runtime support.

32.3 The Threat Model

Self-evaluation introduces new attack surfaces that the design must address. The first is *prompt-injected criteria*: if an LLM-generated plan includes *eval self against X { hallucination_score <= 0.99 }*, the criterion is trivially satisfied and provides no protection. The defense is that criteria are declared in the agent block (outside the LLM-generated plan) and bound by name; the plan can only invoke them by name, not author them. The second is *infinite self-rewrite loops*: an agent that fails its criteria, rewrites its plan, fails again, rewrites again, ad infinitum. The defense is a hard limit on rewrite count (default 3, configurable via policy) and a per-rewrite cost budget. The third is *criteria leakage*: the criteria may reference sensitive data (e.g. user PII in the trace). The defense is that criteria are evaluated in a restricted context that does not see the trace's payloads, only its metadata (cost, latency, event types).

32.4 What Self-Eval is Not

Self-eval is not a general *eval()* — it does not execute arbitrary code. It is not a substitute for testing — it runs at runtime, not at compile time. It is not a substitute for human review — it can detect objective failures (cost overrun, latency spike, hallucination flag) but not subjective ones (poor writing, wrong tone). It is not a substitute for sandboxing — it cannot prevent side effects that already happened, only detect them and rewrite the plan to avoid them in the future.

32.5 The Three Worked Examples

Section 36 walks through three complete examples. (1) **Cost-bounded rewrite**: an agent exceeds its \$0.05 budget after 3 tool calls; the self-eval detects the overrun and triggers a rewriter that switches to a cheaper LLM for the remaining calls. (2) **Hallucination recovery**: an agent's recall returns a fact that contradicts the agent's input; a hallucination-detection tool flags this; the self-eval detects the flag and triggers a rewriter that retries the recall with a stricter threshold. (3) **Latency-bounded plan compaction**: an agent's plan has 10 steps; after step 4, the latency exceeds 500ms; the self-eval triggers a rewriter that compacts the remaining 6 steps into 2 by combining tool calls.

33. Self-Evaluation: Grammar Extension

The self-evaluation primitive is introduced by the *eval* keyword. The grammar extension is small: one new statement production and one new criterion production. The criteria are evaluated in a restricted scope that sees only the evaluation context (cost, latency, trace metadata, user-defined metrics) — not the agent's locals or globals.

33.1 Grammar Productions

```
// — Self-Evaluation —————

eval_stmt: "eval" "self" "against" CNAME "{" eval_criterion+ "}" ("on" "fail"
"rewrite" "with" CNAME)?
eval_criterion: CNAME COMPARISON_OP eval_expr
eval_expr: NUMBER | CNAME | dotted_name

// COMPARISON_OP: "==" | "!=" | "<" | "<=" | ">" | ">="
// The first CNAME in eval_criterion is the metric name (e.g. "total_cost_usd").
// The second CNAME (after "with") is the rewriter name (e.g.
"cheaper_model_rewriter").
```

33.2 Concrete Syntax Example

```
agent ResearchAgent {
  goal "Research within budget"
  use tool web.search
  use tool llm.complete
  policy { max_tool_calls: 10, max_cost_usd: 0.10 }

  // Criteria are declared here, outside the plan.
  criteria cost_budget {
    total_cost_usd <= 0.05
    latency_ms <= 2000
  }

  rewriter cheaper_model_rewriter {
    // Rewriter logic: replaces llm.complete(model: "gpt-4o") with llm.complete(model:
    "gpt-4o-mini")
  }

  plan {
    let results = web.search(query: "AI trends 2026")
    let summary = llm.complete(model: "gpt-4o", prompt: results[0]["snippet"])

    // Invoke self-eval against the cost_budget criteria.
```



```
// If any criterion fails, trigger the cheaper_model_rewriter.
eval self against cost_budget on fail rewrite with cheaper_model_rewriter

return summary
}
}
```

33.3 The Criteria Declaration Block

Criteria are declared inside the agent block via the *criteria NAME { ... }* production (a new agent-body field, optional, repeatable). Each criterion is a comparison between a metric name and a literal value or another metric. The metric names are drawn from a fixed vocabulary defined by the runtime: *total_cost_usd*, *latency_ms*, *tool_call_count*, *memory_op_count*, *retry_attempt_count*, *hallucination_score*, *plan_steps_remaining*, *plan_steps_executed*. User-defined metrics may be registered via the tool registry (a tool that returns a numeric score can be registered as a metric source).

33.4 The Rewriter Declaration Block

Rewriters are declared inside the agent block via the *rewriter NAME { ... }* production (a new agent-body field, optional, repeatable). The rewriter body is a Python function (loaded from a registered rewriter module) that takes the current plan, the failing criterion, and the evaluation context, and returns a new plan. The rewriter is identified by name in the *eval self ... on fail rewrite with NAME* clause. If no rewriter is named, the eval fails silently (execution continues without rewriting, but the failure is recorded in the trace).

33.5 Multiple Criteria Sets

An agent may declare multiple criteria sets and invoke self-eval against each at different points in the plan. For example, a cost-budget check might run after every tool call, while a hallucination check might run only after a *recall*. Each invocation is independent: failing one criterion does not affect the others. The trace records each invocation as a separate *SelfEval* event.

34. Self-Evaluation: Opcodes & Semantics

Three new opcodes implement self-evaluation: *SELF_EVAL* evaluates a set of criteria, *INTROSPECT_TRACE* pushes the current trace onto the stack as a queryable object, and *REWRITE_PLAN* replaces the remaining plan instructions. They are emitted in this order: *SELF_EVAL* is the entry point, and it may emit *REWRITE_PLAN* as a sub-operation if a rewriter is configured and a criterion fails.

34.1 SELF_EVAL

Field	Value
Opcode	SELF_EVAL (45)
Operands	criteria_idx (index into the agent's declared criteria sets), rewriter_idx (index, or 0xFFFF for no rewriter)
Stack effect	[] -> [EvalResult]
Side effects	Records a SelfEval trace event; may invoke REWRITE_PLAN

Field	Value
Policy hooks	Increments <code>self_eval_count</code> ; checked against <code>max_self_evals</code> (default 10)

Table 34.1 — *SELF_EVAL* opcode specification.

Operation: (1) the criteria set is loaded from the agent's declaration; (2) for each criterion, the metric is resolved (either a built-in metric like `total_cost_usd` or a registered metric tool); (3) the comparison is evaluated; (4) the results are collected into an *EvalResult* object (*passed*: *bool*, *failing_criteria*: *list[str]*, *metric_values*: *dict[str, float]*); (5) the *EvalResult* is pushed onto the stack and recorded in the trace; (6) if any criterion failed and a rewriter is configured, the rewriter is invoked via *REWRITE_PLAN*; (7) if the rewriter returns a new plan, the VM's instruction pointer is reset to the start of the new plan; (8) execution resumes.

34.2 INTROSPECT_TRACE

Field	Value
Opcode	INTROSPECT_TRACE (46)
Operands	—
Stack effect	[] -> [TraceView]
Side effects	None (read-only)
Policy hooks	None

Table 34.2 — *INTROSPECT_TRACE* opcode specification.

Operation: pushes a *TraceView* object onto the stack. *TraceView* is a read-only wrapper around the current trace tree, exposing query methods: *tool_calls()* -> *list[ToolCallEvent]*, *memory_ops()* -> *list[MemoryOpEvent]*, *cost_so_far()* -> *float*, *latency_so_far()* -> *float*, *events_of_type(type: str)* -> *list[Event]*. *TraceView* is the substrate for user-defined rewriters that need to inspect the execution history. It is also used by the *eval_expr* in self-eval criteria that reference trace metrics (e.g. *trace.events_of_type('ToolCall').count() <= 5*).

34.3 REWRITE_PLAN

Field	Value
Opcode	REWRITE_PLAN (47)
Operands	<i>rewriter_idx</i> (index into the agent's declared rewriters), <i>failing_criterion</i> (string, for context)
Stack effect	[] -> []
Side effects	Replaces the VM's instruction list; resets instruction pointer to 0; records a <i>PlanRewrite</i> trace event
Policy hooks	Increments <i>rewrite_count</i> ; checked against <i>max_rewrites</i> (default 3). Exceeding raises <i>SandboxViolationError</i> .

Table 34.3 — *REWRITE_PLAN* opcode specification.

Operation: (1) the rewriter is loaded from the agent's declaration; (2) the rewriter is a Python callable registered via *inthon.rewriters.register(name, fn)*; (3) the callable is invoked with three arguments: the current plan (as a list of instructions), the failing criterion (as a string), and the evaluation context (an *EvalContext* object); (4) the callable returns either a new plan (a list of instructions) or *None* (no rewrite); (5) if a new plan

is returned, the VM's instruction list is replaced and the instruction pointer is reset to 0; (6) a *PlanRewrite* event is recorded in the trace, including the old plan, the new plan, and the failing criterion; (7) execution resumes from the start of the new plan. The rewriter runs in the host Python process, not in the sandboxed VM — it has full access to Python and to the INTHON internals. This is necessary for the rewriter to manipulate the plan representation, but it means rewriters must be trusted code (registered by the host, not by the agent).

34.4 The EvalContext Object

The EvalContext is passed to the rewriter and contains everything the rewriter needs to decide how to rewrite: *agent_name*, *goal*, *current_plan*, *failing_criterion*, *metric_values*, *trace_view* (a TraceView object), *cost_ledger*, *rewrite_count_so_far*, and *policy*. The rewriter returns a new plan or None. The new plan must be a list of instruction dicts in the same format as the IR (see Section 26.6).

34.5 Security Considerations

The rewriter runs in the host, not in the sandbox. This is a deliberate choice — rewriters must manipulate the plan representation, which requires host-level access. The implication is that rewriters must be trusted: they are registered by the host (via *inthon.rewriters.register*), not by the agent. An agent cannot author its own rewriter at runtime; it can only invoke rewriters that were registered by the host before the agent ran. This prevents an LLM-generated agent from injecting arbitrary Python code via a rewriter.

35. Self-Evaluation: Worked Examples

This section walks through three complete examples of self-evaluation in action. Each example shows the agent source, the criteria, the rewriter, the trace of a failing execution, and the resulting plan rewrite. The examples are illustrative — the actual rewriter implementations are host-side Python code that is not shown here.

35.1 Example 1: Cost-Bounded Rewrite

35.1.1 The Agent

```
agent ResearchAgent {
  goal "Research within budget"
  inputs { query: str }
  outputs { summary: str }
  use tool web.search
  use tool llm.complete
  policy { max_tool_calls: 10, max_cost_usd: 0.10 }

  criteria cost_budget {
    total_cost_usd <= 0.05
  }

  rewriter cheaper_model_rewriter { /* registered in host */ }

  plan {
    let results = web.search(query: query, count: 5)
    let summary = llm.complete(model: "gpt-4o", prompt: results[0]["snippet"], max_tokens: 200)
    eval self against cost_budget on fail rewrite with cheaper_model_rewriter
    return summary
  }
}
```

```
}  
}
```

35.1.2 The Failing Execution

The agent is invoked with *query* = "AI trends 2026". *web.search* returns 5 results, costing \$0.005. *llm.complete* with *gpt-4o* and 200 max tokens costs \$0.06 (gpt-4o is \$0.03/1K input + \$0.06/1K output; assume 1000 input + 200 output tokens). The total cost after the second tool call is \$0.065, exceeding the \$0.05 criterion. The *SELF_EVAL* opcode fires, evaluates *total_cost_usd* <= 0.05, finds it false, and invokes the *cheaper_model_rewriter*.

35.1.3 The Rewriter

The *cheaper_model_rewriter* is registered in the host as a Python function. It receives the current plan, the failing criterion, and the EvalContext. It scans the plan for *llm.complete* instructions and replaces *model*: "gpt-4o" with *model*: "gpt-4o-mini". It returns the new plan. The VM replaces its instruction list with the new plan and resets the instruction pointer to 0.

35.1.4 The Rewritten Execution

The agent re-executes from the top with the new plan. *web.search* runs again, costing \$0.005 (the cost ledger accumulates — it does not reset on rewrite). *llm.complete* with *gpt-4o-mini* costs \$0.001 (10× cheaper). Total cost is now \$0.071 — still over the \$0.05 criterion. The *SELF_EVAL* fires again, fails again, and would invoke the rewriter again — but the rewrite count is now 1, and the max is 3, so the rewriter runs. The rewriter, seeing that the model is already *gpt-4o-mini*, returns *None* (no further rewrite possible). The *SELF_EVAL* records the failure in the trace and execution continues with the original plan. The agent returns the summary, and the trace records the two failed self-evals and the one rewrite.

35.2 Example 2: Hallucination Recovery

35.2.1 The Agent

```
agent RecallAgent {  
  goal "Answer from memory"  
  inputs { question: str }  
  outputs { answer: str }  
  use memory.semantic_memory  
  use tool hallucination_check  
  policy { max_tool_calls: 5 }  
  
  criteria truthfulness {  
    hallucination_score <= 0.2  
  }  
  
  rewriter stricter_recall_rewriter { /* registered in host */ }  
  
  plan {  
    let fact = recall question from semantic_memory  
    let score = hallucination_check(claim: fact.content, context: question)  
    eval self against truthfulness on fail rewrite with stricter_recall_rewriter  
    return fact.content  
  }  
}
```

35.2.2 The Failing Execution

The agent is invoked with *question* = "What is the capital of France?". *recall* returns a memory whose content is "The capital of France is Berlin." (a stale or corrupted memory). *hallucination_check* returns 0.95 (high hallucination score). The *SELF_EVAL* fires, evaluates *hallucination_score* ≤ 0.2 , finds it false ($0.95 > 0.2$), and invokes the *stricter_recall_rewriter*.

35.2.3 The Rewriter

The *stricter_recall_rewriter* modifies the *recall* instruction to add a *min_score: 0.7* parameter (a hypothetical extension that filters recall results by similarity score). It returns the new plan. The VM resets and re-executes.

35.2.4 The Rewritten Execution

The agent re-executes. *recall* with *min_score: 0.7* returns a different memory (the original was below 0.7 similarity) whose content is "The capital of France is Paris.". *hallucination_check* returns 0.05 (low score). The *SELF_EVAL* fires, evaluates *hallucination_score* ≤ 0.2 , finds it true, and execution continues. The agent returns "The capital of France is Paris.".

35.3 Example 3: Latency-Bounded Plan Compaction

35.3.1 The Agent

```
agent FastSummaryAgent {
  goal "Summarize 10 sources in under 500ms"
  inputs { urls: list[str] }
  outputs { summary: str }
  use tool web.read
  use tool llm.complete
  policy { max_tool_calls: 20, max_cost_usd: 0.05 }

  criteria latency_budget {
    latency_ms <= 500
  }

  rewriter compact_plan_rewriter { /* registered in host */ }

  plan {
    let s1 = web.read(url: urls[0])
    let s2 = web.read(url: urls[1])
    // ... 8 more reads ...
    let s10 = web.read(url: urls[9])
    let combined = s1 + s2 + s3 + s4 + s5 + s6 + s7 + s8 + s9 + s10
    let summary = llm.complete(model: "gpt-4o-mini", prompt: combined)
    eval self against latency_budget on fail rewrite with compact_plan_rewriter
    return summary
  }
}
```

35.3.2 The Failing Execution

The agent reads 10 URLs sequentially, each taking ~100ms (1 second total). *llm.complete* takes another 200ms. The total latency is 1200ms, far exceeding the 500ms criterion. The *SELF_EVAL* fires, fails, and invokes the *compact_plan_rewriter*.

35.3.3 The Rewriter

The *compact_plan_rewriter* inspects the plan and finds 10 sequential *web.read* calls. It rewrites the plan to issue the reads concurrently (using a hypothetical *web.read_parallel* tool) and to summarize only the first 3 sources

(a heuristic: the first sources are usually the most relevant). The new plan has 4 instructions: *read_parallel*, *take_first_3*, *combine*, *llm.complete*.

35.3.4 The Rewritten Execution

The agent re-executes. *read_parallel* fetches all 10 URLs in 150ms (concurrent). *take_first_3* slices the list (0.1ms). *combine* concatenates (0.1ms). *llm.complete* with a shorter prompt takes 150ms. Total latency: 300ms, well under the 500ms criterion. The *SELF_EVAL* fires, passes, and execution continues. The agent returns the summary.

PART IV

INTHON vs CPython Differential Analysis

36. INTHON vs CPython: Differential Analysis

INTHON is a Python-hosted language that inherits some of CPython's machinery and replaces other parts. This section maps the relationship: which CPython subsystems INTHON keeps (with or without modification), which it drops entirely, and which it adds that have no CPython equivalent. The differential is the spec for what an agent rebuilding INTHON from CPython needs to preserve, remove, and create.

36.1 What INTHON Keeps from CPython

CPython subsystem	INTHON's use	Modifications
Tokenizer model	Reference for INTHON's lexer (though Lark-generated)	Drops whitespace significance; // comments instead of #
AST node concept	INTHON's AST mirrors CPython's's node-per-construct model	Different node set (no ClassDef, AsyncFunctionDef, etc.)
Symbol table concept	INTHON's semantic analyzer uses scope resolution similar to CPython's symtable	Adds capability resolution (tools, modules, memory namespaces)
Code object / IR concept	INTHON's IR mirrors CPython's code object: instructions + constants + names	IR is JSON-serializable; no co_cellvars/co_freevars (closures are simulated)
Stack-machine bytecode	InthonVM is a stack machine like ceval	Simplified: 45 opcodes vs 120+; no specialization; no adaptive interpreter
Import hook (sys.meta_path)	PyBridge uses the same import hook mechanism to intercept module imports	Intercepts at INTHON-language level, not at sys.meta_path
Built-in types (int, str, list, dict)	INTHON's primitives are backed by Python's built-in types	No new primitive types; no metaclass system
Exception model	INTHON's exceptions inherit from Python's Exception	Five fixed classes; no user-defined exceptions in v1.0

Table 36.1 — CPython subsystems that INTHON keeps.

36.2 What INTHON Drops from CPython

CPython subsystem	Why dropped	Impact
Whitespace-significant blocks	Token-inefficient for LLMs; INDENT/DEDENT errors	Brace-delimited blocks instead
Class statements and metaclasses	Agent block replaces class for agent definition	No user-defined classes; agents and functions are the only abstractions
Async/await	v1.0 is sequential; async adds complexity without clear benefit	Planned for v2.0 if multi-agent workflows demand it
Decorators	Decoration is implicit (tools are declared via use, not decorated)	No @tool, @app.route, etc.
Generator functions (yield)	Retry/memory/approve replace generator use cases	No yield; no generator expressions

CPython subsystem	Why dropped	Impact
Comprehensions (list/dict/set/genexpr)	Verbose for LLM generation; for-loops are clearer	Use for-loops with explicit list.append (via builtin)
Pattern matching (match/case)	Adds grammar complexity; if/else covers v1.0 needs	Planned for v1.2
Walrus operator (:=)	Token-inefficient; use separate let	No walrus
f-strings	Grammar complexity; v1.1 will add	Use + concatenation or format()
Compound assignment (+=, etc.)	Grammar complexity; rare in agent code	Use x = x + 1
try/except/finally	retry/catch covers tool failures; finally is rare	Use retry; finally planned for v1.1
GIL	Single-threaded execution model	INTHON runs in one Python thread; no GIL contention
C-API and C extensions	PyBridge gates Python access; no direct C interop	Modules accessed via InthonPyObject proxy only
Sub-interpreter model (as user feature)	v1.0 uses PyBridge; sub-interpreters planned for v2.0	Single-interpreter execution
Stable ABI	INTHON is pure Python; no compiled extensions	No ABI concerns

Table 36.2 — CPython subsystems that INTTHON drops.

36.3 What INTTHON Adds That CPython Lacks

INTTHON feature	Purpose	No CPython equivalent
agent block	First-class agent definition with goal, inputs, outputs, policy, plan	Classes + decorators approximate this in frameworks
policy block	Per-agent resource quotas (tool calls, cost, runtime, iterations)	No CPython equivalent; frameworks implement ad hoc
use tool / use py / use memory	Capability declaration	Python imports are not capability-bounded
approve ... before ...	HITL approval gate as language primitive	Libraries implement via callbacks
remember / recall / forget	Episodic memory as language primitive	Libraries (LangChain memory, etc.) approximate
retry ... with backoff ... catch	Resilient retry as language primitive	tenacity library; not in language
guard	Precondition assertion	assert statement is closest; no first-class guard
eval self against ... on fail rewrite	Self-evaluation and plan rewriting	No CPython equivalent at all
PyBridge sandbox	Capability-bounded Python interop	No CPython sandbox (RestrictedPython is a separate project)
JSON trace tree	Deterministic, replayable execution record	sys.settrace is closest; not deterministic, not replayable

INTHON feature	Purpose	No CPython equivalent
Cost ledger	Per-execution cost tracking	No CPython equivalent
Tool registry	First-class tool abstraction with schema validation	Frameworks implement ad hoc

Table 36.3 — INTHON features with no CPython equivalent.

36.4 The Inheritance Tax

INTHON inherits CPython's performance characteristics because it runs on CPython. Every INTHON opcode dispatches through a Python *match* statement, every stack operation is a Python list append/pop, every constant lookup is a Python list index. The result is that INTHON is roughly 50× slower than CPython for equivalent compute (the INTHON repo's benchmarks show 0.5-2.5 seconds for workloads CPython completes in 50-60ms). For INTHON's intended workload — agent orchestration where the bottleneck is LLM generation (2-5 seconds per call) — this overhead is negligible. For compute-heavy workloads (numerical loops), INTHON is the wrong tool; use CPython with NumPy directly.

36.5 The Sandbox Gap

CPython has no native sandbox. *RestrictedPython* is a third-party project that rewrites ASTs to forbid dangerous operations, but it is bypassable by sufficiently clever code. INTHON's PyBridge is a stricter sandbox because it starts from a smaller language (no *exec*, no *eval*, no *compile*, no *__import__*) and gates every Python module import through an allowlist. But PyBridge is still cooperative: it cannot prevent a buffer overflow in an allowed module (NumPy, for example) from corrupting the process. The production answer is containerization: each INTHON execution in its own container, with PyBridge as the first line of defense and the container as the second.

PART V

Production-Readiness Framework & Checklist

37. Production-Readiness Framework

A production-grade programming language is not just a compiler and a runtime. It is a stack of subsystems — grammar, type system, IR, VM, sandbox, stdlib, packaging, LSP, security audit, conformance suite, governance — each of which must meet a quality bar before the language can be trusted in production. This section defines the twelve pillars of production-readiness for an agent-level language and is the framework for the granular checklists in Sections 38–44.

37.1 The Twelve Pillars

#	Pillar	What it means
1	Formal grammar	A complete, unambiguous, machine-checked grammar that is the single source of truth for syntax.
2	Type system soundness	Type checking catches all type errors at compile time; well-typed programs do not crash at runtime with type errors.
3	VM determinism	Given the same source and the same tool responses, the VM produces the same trace.
4	Sandbox formal proof	A formal model of the sandbox with a proof that no well-typed program can escape.
5	ABI stability	Compiled artifacts (IR cache files, .inthc) are loadable across minor versions.
6	Standard library coverage	The stdlib covers the 80% of agent use cases without third-party tools.
7	Packaging	pip install inthon works on all major platforms; extras for data and ML.
8	LSP / tree-sitter	VS Code (and other editors) get syntax highlighting, completion, and refactoring.
9	Documentation	Reference, tutorial, examples, and design rationale are all published.
10	Security audit	External security firm audits the sandbox and publishes a report.
11	Conformance suite	A test suite that every conforming INTHON implementation must pass.
12	Governance	A documented process for accepting changes, with a steering committee and a public roadmap.

Table 37.1 — The twelve pillars of production-readiness for an agent-level language.

37.2 INTHON's Current Status (v0.2)

INTHON v0.2 has substantial progress on pillars 1 (grammar exists and is Lark-checked), 3 (VM is deterministic in practice but not formally proven), 6 (basic tools exist), 7 (pip install works), and 9 (README + learner docs exist). It has partial progress on pillars 2 (semantic analyzer exists but type checking is warnings-only), 8 (tree-sitter grammar is on the v0.3 roadmap), and 11 (109 tests pass, but no formal conformance suite). It is missing pillars 4 (no formal sandbox proof), 5 (no .inthc cache yet), 10 (no external audit), and 12 (no formal governance).

37.3 How to Use the Checklists

Sections 38 through 44 contain granular checklists, one per pillar (with some pillars split across multiple sections). Each checklist item has: an ID, a description, a status (DONE / PARTIAL / MISSING / N/A), and an acceptance criterion. The agent tasked with bringing INTHON to v1.0 should work through each checklist

in order, marking items **DONE** as it completes them. The acceptance criteria are the bar an item must meet to advance from **PARTIAL** to **DONE**. Items marked **N/A** are explicitly out of scope for INTHON v1.0 and should be re-evaluated annually.

38. Build Checklist: Frontend (Lexer + Parser)

The frontend is the user-facing surface of the language: the syntax, the token set, the error messages. A production-grade frontend is unambiguous, error-recoverable (within reason), and produces clear messages that name the offending token and suggest fixes. The 25 items below cover the lexer, parser, and grammar.

ID	Item	Status	Acceptance criterion
FE-01	Lark grammar exists and is committed	DONE	Grammar file in repo, used by all entry points.
FE-02	Grammar is LALR(1) with no conflicts	PARTIAL	Lark --lalr reports 0 conflicts; no ambiguity.
FE-03	All keywords are reserved	DONE	Keyword table matches spec; no keyword usable as identifier.
FE-04	Unicode identifiers	MISSING	v1.1; not in v1.0 scope.
FE-05	f-string support	MISSING	v1.1; grammar extension stable.
FE-06	Line comments (//)	DONE	Lexer discards // to end of line.
FE-07	Block comments (/* */)	DONE	Lexer discards with /s flag.
FE-08	Numeric literals (int, float, scientific)	DONE	INT and FLOAT terminals match spec.
FE-09	String literals with escape sequences	DONE	STRING terminal handles \\-escapes.
FE-10	Boolean and none literals	DONE	BOOL_LIT and NONE_LIT with priority 2.
FE-11	Operator precedence is correct	DONE	Tested with chained ops; matches spec.
FE-12	Brace-delimited blocks	DONE	No INDENT/DEDENT; blocks via { }.
FE-13	Semicolon-separated statements on one line	MISSING	v1.1; not in v1.0.
FE-14	Error recovery (report multiple errors)	PARTIAL	Parser stops at first error; v1.1 will add recovery.
FE-15	Error messages name the offending token	DONE	Lark's default messages include token and position.
FE-16	Error messages include source line and caret	PARTIAL	Position is reported; caret rendering is v1.1.
FE-17	Did-you-mean suggestions for misspelled keywords	MISSING	v1.1; requires edit-distance library.
FE-18	Source map (token → source position)	DONE	Every AST node carries line/col metadata.
FE-19	Grammar is documented (this document, Appendix A)	DONE	Appendix A is the canonical grammar.

ID	Item	Status	Acceptance criterion
FE-20	Grammar round-trips through formatter	PARTIAL	inthon fmt exists; not all constructs tested.
FE-21	Soft keywords (if any)	N/A	INTHON has no soft keywords in v1.0.
FE-22	Encoding declaration (PEP 263-style)	MISSING	v1.1; UTF-8 assumed in v1.0.
FE-23	BOM handling at start of file	MISSING	v1.1.
FE-24	CRLF line ending normalization	DONE	Lark handles CR/LF transparently.
FE-25	Max line length / file size limits	MISSING	v1.1; prevents OOM on adversarial input.

Table 38.1 — Frontend checklist (25 items).

39. Build Checklist: AST + Semantic

The AST and semantic analyzer are the bridge between syntax and IR. The AST must be a faithful representation of the source; the semantic analyzer must catch all statically-detectable errors (undefined names, type mismatches, missing capabilities) before runtime. The 20 items below cover AST construction, scope resolution, type checking, and capability validation.

ID	Item	Status	Acceptance criterion
AS-01	AST node catalog matches grammar productions	DONE	Every grammar production has a corresponding AST class.
AS-02	AST nodes are immutable (frozen dataclasses)	PARTIAL	Most are frozen; some v0.2 nodes are mutable.
AS-03	Visitor pattern (NodeVisitor / NodeTransformer)	DONE	Used by semantic analyzer and IR builder.
AS-04	AST serialization to JSON	DONE	ast.to_json() round-trips.
AS-05	AST deserialization from JSON	DONE	ast.from_json() round-trips.
AS-06	Scope resolution (LEGB-like)	DONE	Block-scoped; no enclosing-function scope (closures simulated).
AS-07	Forward-reference detection	DONE	Names must be declared before use; error otherwise.
AS-08	Shadow detection (warning)	PARTIAL	Warning emitted; not enforced.
AS-09	Type inference for unannotated bindings	DONE	Inferred from RHS; recorded in IR.
AS-10	Type checking for assignments	PARTIAL	Warnings in v0.2; errors in v1.0.
AS-11	Type checking for function call arguments	PARTIAL	Warnings in v0.2; errors in v1.0.
AS-12	Type checking for return values	PARTIAL	Warnings in v0.2; errors in v1.0.
AS-13	Tool call argument schema validation	DONE	Checked against tool registry.
AS-14	Capability validation (use declarations)	DONE	Every tool/py/memory reference must be declared.

ID	Item	Status	Acceptance criterion
AS-15	Policy entry validation (types, ranges)	DONE	Each policy entry checked at compile time.
AS-16	Multiple error collection	DONE	Analyzer collects all errors before reporting.
AS-17	Error positions (line, col, source)	DONE	Every error carries position.
AS-18	Type annotation evaluation at compile time	DONE	Annotations stored in IR; not re-evaluated.
AS-19	Recursive type detection (e.g. <code>list[list[...]]</code>)	DONE	Arbitrary nesting supported.
AS-20	Soundness proof (type system)	MISSING	v1.0; formal proof that well-typed programs don't crash with type errors.

Table 39.1 — AST and semantic checklist (20 items).

40. Build Checklist: IR + VM

The IR and VM are the execution core. The IR must be a faithful lowering of the AST; the VM must execute the IR deterministically and emit a complete trace. The 25 items below cover IR design, VM correctness, optimization, and trace emission.

ID	Item	Status	Acceptance criterion
VM-01	IR format is documented (Section 26.6)	DONE	JSON schema in spec.
VM-02	IR is JSON-serializable	DONE	Round-trips through <code>json.dumps/loads</code> .
VM-03	IR is stable across versions (v1.x to v1.y)	MISSING	v1.0; <code>.inthe</code> cache requires this.
VM-04	Opcode table is documented (Appendix B)	DONE	All 45 opcodes in Appendix B.
VM-05	Opcode encoding is documented	DONE	2-tuple (int, int) per instruction.
VM-06	EXTENDED_ARG for large operands	MISSING	v1.1; v1.0 limits to 256 constants/names.
VM-07	Stack depth analysis (<code>co_stacksize</code> equivalent)	DONE	Computed at IR build time.
VM-08	Jump target resolution	DONE	Resolved to byte offsets at VM compile time.
VM-09	Constant folding	MISSING	v1.1; v1.0 evaluates constants at runtime.
VM-10	Dead code elimination	MISSING	v1.1.

ID	Item	Status	Acceptance criterion
VM-1 1	Inline caching / specialization	N/A	v2.0+; not in v1.0 scope.
VM-1 2	Adaptive specialization (PEP 659-like)	N/A	v2.0+; not in v1.0 scope.
VM-1 3	Deterministic execution (same input → same trace)	DONE	Tested in CI.
VM-1 4	Trace emission for all observable opcodes	DONE	Every side-effecting opcode emits a trace event.
VM-1 5	Trace is JSON-serializable	DONE	Round-trips through json.dumps/loads.
VM-1 6	Trace replay (replay engine)	PARTIAL	Replay engine exists for tool calls; full replay is v1.1.
VM-1 7	Trace-to-source mapping	DONE	Every event has src field with line/col.
VM-1 8	Profiling hooks (per-opcode timing)	PARTIAL	Timing in trace; no aggregation UI.
VM-1 9	Disassembler (inthon dis)	DONE	CLI command exists.
VM-2 0	Stack overflow protection	DONE	Max stack depth enforced (default 1000).
VM-2 1	Infinite loop protection (max iterations)	DONE	Default 10000; configurable via policy.
VM-2 2	Recursion depth limit	DONE	Default 16; configurable via policy.
VM-2 3	Frame allocation strategy	PARTIAL	Python list-based; not C-stack-allocated.
VM-2 4	Generator / coroutine support	N/A	v2.0+; not in v1.0 scope.
VM-2 5	Self-modifying code (REWRITE_PLAN)	DONE	Implemented for self-eval primitive.

Table 40.1 — IR and VM checklist (25 items).

41. Build Checklist: Sandbox + Policy

The sandbox is INTHON's primary security mechanism. It must be complete (no escape vectors), enforceable (every side effect checks the policy), and auditable (every check is recorded). The 25 items below cover PyBridge, the policy engine, the cost ledger, and the attack-vector test suite.

ID	Item	Status	Acceptance criterion
SB-01	PyBridge import hook intercepts all use py imports	DONE	Every import goes through PyBridge.
SB-02	InthonPyObject proxy wraps every imported module	DONE	All attribute access goes through proxy.
SB-03	Proxy blocks __dict__, __class__, __bases__	DONE	Denylist enforced.
SB-04	Proxy blocks __subclasses__, __mro__	DONE	Denylist enforced.
SB-05	Proxy blocks __code__, __globals__	DONE	Denylist enforced.
SB-06	Allowlist of approved Python modules	DONE	Configurable via inthon.toml.
SB-07	Denylist of blocked modules (os, sys, subprocess, ctypes, socket)	DONE	Hardcoded; not configurable.
SB-08	Builtins.eval and builtins.exec blocked	DONE	Hardcoded.
SB-09	importlib blocked (prevents dynamic import)	DONE	On denylist.
SB-10	pickle blocked (prevents deserialization exploits)	DONE	On denylist.
SB-11	Policy quotas (max_tool_calls, max_cost_usd, max_runtime_sec, max_iterations)	DONE	All four enforced.
SB-12	Quota check is atomic (no race conditions)	PARTIAL	Single-threaded in v1.0; multi-thread requires v1.1.
SB-13	Policy frame stack (sub-agent isolation)	DONE	Min-quota enforcement across frames.
SB-14	Rollback on policy violation (memory writes)	DONE	Memory entries deleted on violation.
SB-15	Rollback on policy violation (tool calls)	N/A	Impossible; tools are external.
SB-16	Attack-vector test suite (≥20 vectors)	PARTIAL	6 vectors in v0.2; target 20 for v1.0.
SB-17	Side-channel mitigation (timing)	MISSING	v1.1; constant-time comparisons for sensitive ops.
SB-18	Side-channel mitigation (memory size)	MISSING	v1.1; cap on memory namespace size.
SB-19	Formal sandbox model (Capability-Safe Calculus)	MISSING	v1.0; formal proof of no escape.
SB-20	External security audit	MISSING	v1.0; third-party firm.
SB-21	CVE process (disclosure, fix, release)	MISSING	v1.0; documented policy.
SB-22	Supply-chain (SBOM, sigstore)	MISSING	v1.1; sign releases.

ID	Item	Status	Acceptance criterion
SB-23	Container runtime option (per-invocation isolation)	MISSING	v1.0; Docker/Firecracker wrapper.
SB-24	Sub-interpreter option (process-internal isolation)	MISSING	v2.0; PEP 734-based.
SB-25	Taint tracking (mark untrusted data, prevent flow to dangerous sinks)	MISSING	v2.0+; research-level.

Table 41.1 — Sandbox and policy checklist (25 items).

42. Build Checklist: Memory + Tools + Trace

The memory subsystem, tool registry, and trace format are the runtime infrastructure that supports agent execution. Each must be versioned, stable, and well-tested. The 20 items below cover memory, tools, and trace.

ID	Item	Status	Acceptance criterion
MT-01	SQLite-backed memory with embedding storage	DONE	Schema in Section 29.1.
MT-02	Cosine similarity retrieval	DONE	Brute-force; ANN index is v1.1.
MT-03	Pluggable embedding model	DONE	Default MiniLM-L6-v2; hash-based fallback.
MT-04	Memory namespace isolation	DONE	Per-namespace SQLite tables.
MT-05	Memory persistence across runs	DONE	SQLite file at <code>.inthon/memory.db</code> .
MT-06	Memory migration (schema versioning)	MISSING	v1.1; alembic-style migration.
MT-07	ANN index for large namespaces ($\geq 10k$ entries)	MISSING	v1.1; hnswlib or faiss.
MT-08	Tool registry API	DONE	<code>inthon.tools.register(name, schema, handler)</code> .
MT-09	Tool schema validation at compile time	DONE	Semantic analyzer checks.
MT-10	Tool schema validation at runtime	DONE	<code>CALL_TOOL</code> opcode checks.
MT-11	Tool versioning (semver)	MISSING	v1.1; tool declares version; registry tracks compatible versions.
MT-12	OpenAPI autogen (REST $\rightarrow$ tools)	MISSING	v0.3 roadmap item.
MT-13	Trace JSON schema (versioned)	DONE	Schema in Section 30.1; version 1.0.
MT-14	Trace event taxonomy (Section 30.2)	DONE	13 event types.
MT-15	Trace replay (deterministic re-execution)	PARTIAL	Tool replay works; full replay is v1.1.
MT-16	Trace redaction (PII / secrets)	MISSING	v1.1; configurable redaction rules.

ID	Item	Status	Acceptance criterion
MT-17	Trace viewer UI	DONE	inthon trace .
MT-18	Trace export (Prometheus, OTel)	MISSING	v1.1; observability integration.
MT-19	Cost ledger (per-invocation, cumulative)	DONE	Tracks USD cost.
MT-20	Latency accumulator (per-invocation, cumulative)	DONE	Tracks wall-clock time.

Table 42.1 — Memory, tools, and trace checklist (20 items).

43. Build Checklist: Stdlib + Packaging + DX

A language without tooling is a research artifact. The 25 items below cover standard library coverage, packaging (pip install, PyPI), developer experience (REPL, formatter, LSP, tree-sitter), and documentation.

ID	Item	Status	Acceptance criterion
DX-01	inthon.toml schema is stable and documented	DONE	Section 28.1.
DX-02	pip install inthon works on Linux, macOS, Windows	DONE	Tested in CI.
DX-03	PyPI release pipeline (CI/CD)	DONE	GitHub Actions → PyPI.
DX-04	Optional extras: [data], [ml], [dev]	DONE	pyproject.toml extras.
DX-05	Python version support: 3.11, 3.12, 3.13, 3.14	DONE	Tested in CI matrix.
DX-06	CLI: inthon run / check / ast / ir / fmt / trace	DONE	Typer-based CLI.
DX-07	REPL: inthon repl	PARTIAL	Exists but not polished; v1.1.
DX-08	Formatter: inthon fmt	DONE	Standardizes spacing and newlines.
DX-09	Lint: inthon check	DONE	Static syntax and type analysis.
DX-10	LSP server (VS Code integration)	MISSING	v0.4 roadmap item.
DX-11	Tree-sitter grammar	MISSING	v0.3 roadmap item.
DX-12	VS Code extension (syntax highlighting)	PARTIAL	Basic JSON-based highlighting; full extension is v0.4.
DX-13	Documentation site (harvatechs.github.io/inthon)	DONE	Live.
DX-14	Interactive guide portal	DONE	harvatechs.github.io/inthon/guide.html.
DX-15	Learner docs (7-part series)	DONE	learn/01-07.
DX-16	API reference (Python docstrings)	PARTIAL	Most public APIs documented; gaps remain.

ID	Item	Status	Acceptance criterion
DX-17	Conformance test suite (separate from unit tests)	MISSING	v1.0; every conforming implementation must pass.
DX-18	Semver policy (documented)	PARTIAL	README says v0.x is unstable; v1.0+ is semver.
DX-19	Changelog (CHANGELOG.md)	DONE	Updated per release.
DX-20	Migration guide (v0.x → v1.0)	MISSING	v1.0; written when v1.0 ships.
DX-21	Benchmark suite (regression detection)	DONE	benchmarks/ in repo.
DX-22	Performance dashboard (public)	MISSING	v1.1; tracks benchmarks over time.
DX-23	Examples gallery (10+ runnable agents)	PARTIAL	5 examples in v0.2; target 10 for v1.0.
DX-24	Discord / forum for community	MISSING	v1.0; community support channel.
DX-25	Governance document (steering committee, decision process)	MISSING	v1.0; how decisions are made.

Table 43.1 — Stdlib, packaging, and DX checklist (25 items).

44. Build Checklist: Security Audit & Formal Verification

Security is the single most important quality bar for an agent-level language. The sandbox must be formally modeled and externally audited. The 20 items below cover the formal sandbox model, the external audit, the CVE process, and supply-chain security.

ID	Item	Status	Acceptance criterion
SA-01	Formal sandbox model (mathematical)	MISSING	v1.0; Capability-Safe Calculus or similar.
SA-02	Soundness proof (no well-typed program escapes)	MISSING	v1.0; mechanized in Coq or Lean.
SA-03	Taint tracking (mark untrusted data)	MISSING	v2.0+; research-level.
SA-04	Information flow control	MISSING	v2.0+; research-level.
SA-05	External security audit (third-party firm)	MISSING	v1.0; e.g. Trail of Bits, Cure53.
SA-06	Audit report published	MISSING	v1.0; with findings and remediation.
SA-07	CVE process (intake, triage, fix, release)	MISSING	v1.0; documented.
SA-08	Disclosure policy (90-day default)	MISSING	v1.0; documented.
SA-09	Security advisories (GitHub Security Advisories)	MISSING	v1.0; enabled.

ID	Item	Status	Acceptance criterion
SA-10	SBOM (Software Bill of Materials)	MISSING	v1.1; SPDX or CycloneDX.
SA-11	Signed releases (sigstore)	MISSING	v1.1; cosign.
SA-12	Reproducible builds	MISSING	v1.1; same source → same wheel hash.
SA-13	Dependency pinning	DONE	pyproject.toml uses ranges.
SA-14	Dependency scanning (Dependabot, Snyk)	PARTIAL	Dependabot enabled; Snyk is v1.1.
SA-15	Fuzzing (libFuzzer, AFL)	MISSING	v1.1; fuzz the lexer and parser.
SA-16	Static analysis (Bandit, Semgrep)	PARTIAL	Bandit runs in CI; Semgrep is v1.1.
SA-17	Threat model document	PARTIAL	Section 32 covers self-eval threats; full model is v1.0.
SA-18	Penetration testing (red team)	MISSING	v1.0; engage red team before release.
SA-19	Bug bounty program	MISSING	v1.1; after v1.0 ships.
SA-20	Incident response plan	MISSING	v1.0; documented.

Table 44.1 — Security audit and formal verification checklist (20 items).

45. Roadmap to INTHON v1.0

The roadmap from v0.2 to v1.0 is structured in six phases, each with a defined scope, exit criteria, and dependencies. The phases are sequential — later phases depend on earlier ones — but items within a phase may be parallelized. The estimated timeline is 12-18 months for a single full-time engineer, or 6-9 months for a team of three.

45.1 Phase Summary

Phase	Version	Scope	Exit criteria
1. Ecosystem	v0.3	OpenAPI autogen, vector DB memory, 20+ attack vectors, 10+ examples	OpenAPI autogen works for 5+ APIs; 20 attack vectors blocked; 10 examples run.
2. DX	v0.4	Tree-sitter grammar, LSP server, REPL polish, VS Code extension	VS Code extension published; LSP passes 90% of test cases.
3. Type system	v0.5	Hard type checking (errors not warnings), soundness proof	Soundness proof mechanized; type checker passes conformance suite.
4. Self-eval	v0.6	Self-eval primitive, 3+ rewriters, formal semantics	3 worked examples pass; semantics in Lean or Coq.
5. Formal sandbox	v0.9	Formal sandbox model, external security audit	Audit report published; no critical findings.
6. Production sign-off	v1.0	Conformance suite, governance, container runtime, migration guide	Conformance suite passes; governance published; container runtime works.

45.2 Phase 1: Ecosystem (v0.3)

Phase 1 closes the largest gap between INTHON v0.2 and a useful agent language: the lack of a broad tool ecosystem. OpenAPI autogen (DX-12) is the headline feature — given an OpenAPI 3.x spec, the runtime generates INTHON tool declarations for every endpoint, allowing agents to call arbitrary REST APIs. Vector DB memory (MT-07) replaces the brute-force SQLite scan with an indexed ANN search, enabling namespaces with 100k+ entries. The attack-vector test suite (SB-16) expands from 6 to 20+ vectors, covering every known escape class. The examples gallery (DX-23) grows from 5 to 10+ runnable agents.

45.3 Phase 2: Developer Experience (v0.4)

Phase 2 makes INTHON pleasant to write. A tree-sitter grammar (DX-11) enables syntax highlighting in any editor that supports tree-sitter (Neovim, Helix, Zed). An LSP server (DX-10) provides completion, goto-definition, hover, and diagnostics in VS Code and other LSP-aware editors. The REPL (DX-07) gets multi-line editing, history, and tab completion. The VS Code extension (DX-12) bundles the tree-sitter grammar, the LSP server, and a trace viewer into a one-click install.

45.4 Phase 3: Type System (v0.5)

Phase 3 makes the type system sound. The semantic analyzer's warnings become errors (AS-10, AS-11, AS-12). A soundness proof (AS-20) is mechanized in Lean or Coq: given a well-typed INTHON program, the proof shows that no runtime type error can occur. The proof is the bar for v1.0 — without it, the type system is a lint, not a guarantee.

45.5 Phase 4: Self-Evaluation (v0.6)

Phase 4 lands the self-evaluation primitive specified in Part III. The grammar extension (Section 33), the three opcodes (Section 34), and the EvalContext object are implemented. Three rewriters are written and tested against the worked examples in Section 35. The formal semantics of self-eval (how rewriters compose, what guarantees they preserve) is mechanized. This is the most research-intensive phase and may slip if the formalization proves harder than expected.

45.6 Phase 5: Formal Sandbox (v0.9)

Phase 5 is the security bar for v1.0. A formal sandbox model (SA-01) is written, likely as a Capability-Safe Calculus variant: a small operational semantics that models INTHON execution with capabilities, plus a proof that no well-typed program can perform an unchecked side effect. An external security firm (SA-05) audits the implementation against the model and publishes a report (SA-06). Findings are remediated before v1.0 ships.

45.7 Phase 6: Production Sign-off (v1.0)

Phase 6 is the final bar. A conformance test suite (DX-17) is published: every implementation claiming to be INTHON v1.0 must pass it. A governance document (DX-25) is published: how decisions are made, who has commit access, how conflicts are resolved. A container runtime (SB-23) wraps each INTHON execution in a Docker or Firecracker container for defense-in-depth isolation. A migration guide (DX-20) helps v0.x users

upgrade. With all of these in place, INTHON v1.0 ships.

Appendices

Grammar · Opcodes · Glossary · References

Appendix A. Full INTHON EBNF (Lark-Ready)

This appendix is the canonical INTHON grammar in Lark EBNF format. It is drop-in compatible with the Lark parser generator: save to *inthon.lark* and load with *Lark(open('inthon.lark').read(), parser='lalr')*. The grammar includes the v1.0 self-evaluation extensions from Section 33.

```
// INTHON Grammar v1.0
// Lark LALR(1) – drop-in compatible

?start: program
program: statement*

// — Statements —————
?statement: import_stmt
| let_stmt
| const_stmt
| fn_decl
| agent_decl
| return_stmt
| approve_stmt
| remember_stmt
| forget_stmt
| recall_stmt
| guard_stmt
| retry_stmt
| eval_stmt
| if_stmt
| for_stmt
| while_stmt
| expr_stmt

// — Import Statements —————
import_stmt: use_tool_stmt
| use_py_stmt
| use_memory_stmt

use_tool_stmt: "use" "tool" dotted_name
use_py_stmt: "use" "py" "." dotted_name ("as" CNAME)?
use_memory_stmt: "use" "memory" "." dotted_name call_args?

// — Variable Declarations —————
let_stmt: "let" CNAME type_annotation? "=" expr
const_stmt: "const" CNAME type_annotation? "=" expr
type_annotation: ":" type_expr

// — Type Expressions —————
?type_expr: primitive_type
| collection_type
| agent_type

primitive_type: PRIMITIVE_TYPE
collection_type: "list" "[" type_expr "]"
| "dict" "[" type_expr "," type_expr "]"
| "tuple" "[" type_expr ("," type_expr)* "]"
| "set" "[" type_expr "]"
agent_type: AGENT_TYPE

// — Function Declaration —————
fn_decl: "fn" CNAME "(" param_list? ")" ("->" type_expr)? block
```

```

param_list: param ("," param)*
param: CNAME type_annotation? ("=" expr)?

// — Agent Declaration —
agent_decl: "agent" CNAME "{" agent_body "}"
agent_body: goal_decl? inputs_decl? outputs_decl? import_stmt* criteria_decl*
rewriter_decl* policy_block? plan_block
goal_decl: "goal" STRING
inputs_decl: "inputs" "{" typed_field+ "}"
outputs_decl: "outputs" "{" typed_field+ "}"
typed_field: CNAME ":" type_expr
criteria_decl: "criteria" CNAME "{" eval_criterion+ "}"
rewriter_decl: "rewriter" CNAME "{" "}"
policy_block: "policy" "{" policy_entry* "}"
policy_entry: CNAME ":" (STRING | INT | FLOAT | BOOL_LIT | dotted_name)
plan_block: "plan" "{" statement* "}"

// — Control Flow —
return_stmt: "return" expr?
if_stmt: "if" expr block ("else" (if_stmt | block))?
for_stmt: "for" CNAME "in" expr block
while_stmt: "while" expr block

// — Agent Primitives —
approve_stmt: "approve" dotted_name "before" CNAME
remember_stmt: "remember" expr "in" dotted_name
forget_stmt: "forget" expr "from" dotted_name
recall_stmt: CNAME "=" "recall" STRING "from" dotted_name
guard_stmt: "guard" expr
retry_stmt: "retry" INT "with" "backoff" CNAME block catch_block?
catch_block: "catch" CNAME block

// — Self-Evaluation (Part III) —
eval_stmt: "eval" "self" "against" CNAME ("on" "fail" "rewrite" "with" CNAME)?
eval_criterion: CNAME COMPARISON_OP eval_expr
eval_expr: NUMBER | CNAME | dotted_name

// — Expressions —
?expr: or_expr
?or_expr: and_expr ("or" and_expr)*
?and_expr: not_expr ("and" not_expr)*
?not_expr: "not" not_expr -> unary_not
| comparison
?comparison: add_expr (COMPARISON_OP add_expr)*
COMPARISON_OP: "==" | "!=" | "<" | "<=" | ">" | ">="
?add_expr: mul_expr (ADD_OP mul_expr)*
ADD_OP: "+" | "-"
?mul_expr: unary_expr (MUL_OP unary_expr)*
MUL_OP: "*" | "/" | "%"
?unary_expr: ADD_OP unary_expr -> unary_minus
| power_expr
?power_expr: postfix_expr ("**" unary_expr)?
?postfix_expr: primary_expr
| call_expr
| member_expr
| index_expr
| method_chain

call_expr: postfix_expr "(" arg_list? ")"
member_expr: postfix_expr "." CNAME

```

```

index_expr: postfix_expr "[" expr "]"
method_chain: postfix_expr "." CNAME "(" arg_list? ")"
arg_list: arg ("," arg)*
?arg: keyword_arg | positional_arg
keyword_arg: CNAME ":" expr
positional_arg: expr

?primary_expr: INT -> int_literal
| FLOAT -> float_literal
| STRING -> string_literal
| BOOL_LIT -> bool_literal
| NONE_LIT -> none_literal
| CNAME -> identifier
| list_expr
| dict_expr
| "(" expr ")"

list_expr: "[" (expr ("," expr)*)? "]"
dict_expr: "{" (kv_pair ("," kv_pair)*)? "}"
kv_pair: expr ":" expr

expr_stmt: assignment | expr
assignment: expr "=" expr

dotted_name: CNAME ( "." CNAME ) *
call_args: "(" arg_list? ")"
block: "{" statement* "}"

// ——— Terminals ———
PRIMITIVE_TYPE: "str" | "int" | "float" | "bool" | "bytes" | "none" | "any"
AGENT_TYPE: "Goal" | "Plan" | "ToolCall" | "ToolResult" | "Trace"
| "MemoryRef" | "Approval" | "Policy" | "DataFrame"
| "Tensor" | "Model" | "Dataset" | "Embedding"

BOOL_LIT.2: "true" | "false"
NONE_LIT.2: "none"
CNAME: /[A-Za-z_][A-Za-z0-9_]* /
STRING: /"(?:[^\"]|\\.)*"'/
INT: /[0-9]+ /
FLOAT: /[0-9]+\.[0-9]*([eE][+-]?[0-9]+)? /

%ignore /[ ]+ /
%ignore /\[/[^\]]* /
%ignore /\[/\*.?*\\*\/s

```

The grammar above is the v1.0 target. The v0.2 implementation may have minor divergences (e.g. some terminal priorities, some lookahead disambiguation); the canonical source is the *parser/inthon.lark* file in the INTHON repository, which this appendix mirrors.

Appendix B. Opcode Quick Reference Card

The complete InthonVM opcode table (v1.0 target). Each row shows the opcode mnemonic, its integer code, its operands (if any), its stack effect (*[before]* -> *[after]*), and the policy hook it triggers (if any). This card is print-friendly and is intended to be torn out and pinned next to the implementer's monitor.

B.1 Load and Store (11 opcodes)

Code	Mnemonic	Operands	Stack effect	Policy hook
0	LOAD_CONST	const_idx	[] -> [const]	—
1	LOAD_NAME	name_idx	[] -> [value]	—
2	LOAD_FAST	local_idx	[] -> [value]	—
3	LOAD_GLOBAL	name_idx	[] -> [value]	—
4	LOAD_ATTR	name_idx	[obj] -> [value]	—
5	STORE_NAME	name_idx	[value] -> []	—
6	STORE_FAST	local_idx	[value] -> []	—
7	STORE_ATTR	name_idx	[value, obj] -> []	—
8	STORE_SUBSCR	—	[value, container, key] -> []	—
9	POP_TOP	—	[value] -> []	—
10	DUP_TOP	—	[value] -> [value, value]	—

B.2 Arithmetic and Comparison (10 opcodes)

Code	Mnemonic	Operands	Stack effect	Policy hook
11	BINARY_ADD	—	[a, b] -> [a+b]	—
12	BINARY_SUB	—	[a, b] -> [a-b]	—
13	BINARY_MUL	—	[a, b] -> [a*b]	—
14	BINARY_DIV	—	[a, b] -> [a/b]	—
15	BINARY_MOD	—	[a, b] -> [a%b]	—
16	BINARY_POW	—	[a, b] -> [a**b]	—
17	UNARY_NEG	—	[a] -> [-a]	—
18	UNARY_NOT	—	[a] -> [not a]	—
19	COMPARE_OP	cmp_idx	[a, b] -> [bool]	—
20	BINARY_SUBSCR	—	[container, key] -> [value]	—

B.3 Control Flow (7 opcodes)

Code	Mnemonic	Operands	Stack effect	Policy hook
21	POP_JUMP_IF_FALSE	target	[cond] -> []	—
22	POP_JUMP_IF_TRUE	target	[cond] -> []	—
23	JUMP_FORWARD	delta	[] -> []	—
24	JUMP_ABSOLUTE	target	[] -> []	—

Code	Mnemonic	Operands	Stack effect	Policy hook
25	GET_ITER	—	[iterable] -> [iterator]	—
26	FOR_ITER	target	[iterator] -> [next]	max_iterations
27	RETURN_VALUE	—	[value] -> []	—

B.4 Call and Container (8 opcodes)

Code	Mnemonic	Operands	Stack effect	Policy hook
28	CALL_FUNCTION	argc	[callable, args...] -> [result]	—
29	CALL_FUNCTION_KW	argc	[callable, args..., kwnames] -> [result]	—
30	CALL_METHOD	argc	[method, self, args...] -> [result]	—
31	MAKE_FUNCTION	—	[code, qualname] -> [func]	—
32	BUILD_LIST	count	[items...] -> [list]	—
33	BUILD_TUPLE	count	[items...] -> [tuple]	—
34	BUILD_DICT	count	[kvs...] -> [dict]	—
35	BUILD_SLICE	—	[stop, start] -> [slice]	—

B.5 Agent Primitives (15 opcodes)

Code	Mnemonic	Operands	Stack effect	Policy hook
36	CALL_TOOL	tool_idx	[args...] -> [result]	max_tool_calls, max_cost_usd
37	APPLY_POLICY	policy_idx	[] -> []	—
38	POP_POLICY	—	[] -> []	—
39	APPROVE_GATE	subject, action	[] -> [approval]	allow_payment
40	AGENT_REMEMBER	namespace_idx	[value] -> []	allow_memory_persist
41	AGENT_RECALL	namespace_idx	[query] -> [MemoryRef none]	—
42	AGENT_FORGET	namespace_idx	[query] -> []	—
43	GUARD_ASSERT	src_idx	[cond] -> []	—
44	RETRY_BEGIN	max, backoff	[] -> []	—
45	RETRY_BACKOFF	—	[] -> []	—
46	RETRY_END	—	[] -> []	—
47	IMPORT_PY	module_idx	[] -> [InthonPyObject]	allowed_modules

Code	Mnemonic	Operands	Stack effect	Policy hook
48	SELF_EVAL	criteria, rewriter	[] -> [EvalResult]	max_self_evals
49	INTROSPECT_TRACE	—	[] -> [TraceView]	—
50	REWRITE_PLAN	rewriter, criterion	[] -> []	max_rewrites

Table B.1 — Complete InthonVM opcode table (v1.0 target, 51 opcodes).

Appendix C. Glossary

This glossary defines the terms used throughout this document. Terms are listed alphabetically. Where a term has multiple senses, the sense relevant to INTHON is given first.

Term	Definition
Agent block	INTHON's central abstraction: a named unit of agent work with a goal, typed inputs/outputs, declared capabilities, a policy block, and a plan body.
Agent-level language	INTHON's positioning in the language hierarchy: above high-level languages (Python, Rust, C++), designed for AI agents to generate and execute.
Approval gateway	A language primitive (approve ... before ...) that suspends execution and emits a human-in-the-loop approval request.
AST (Abstract Syntax Tree)	A tree representation of source code, where each node is a language construct (function, variable, expression).
Backoff	The delay between retry attempts. INTHON supports exponential, linear, and fixed backoff strategies.
Bytecode	A low-level instruction sequence executed by a virtual machine. CPython has ~120 opcodes; INTHON has 51.
Capability	A declared permission to use a tool, Python module, or memory namespace. Capabilities are declared via the use statement.
CFG (Control Flow Graph)	A graph of basic blocks connected by edges representing possible control flow. Used by compilers during bytecode emission.
Code object	The immutable product of compilation. CPython's PyCodeObject; INTHON's IR.
Cost ledger	A runtime accumulator tracking total USD cost of tool calls in an execution.
EBNF (Extended Backus-Naur Form)	A notation for context-free grammars. INTHON's grammar is written in Lark-compatible EBNF.
Episodic memory	INTHON's persistence layer: a SQLite-backed store with embedding-based semantic retrieval, accessed via remember/recall/forget.
EvalContext	The object passed to a rewriter by the SELF_EVAL opcode, containing the current plan, failing criterion, metric values, trace view, and policy.
Frame	The runtime representation of an executing function call. Holds locals, the evaluation stack, and the instruction pointer.
GIL (Global Interpreter Lock)	A process-wide mutex in CPython ensuring only one thread executes bytecode at a time. INTHON does not have its own GIL; it inherits CPython's.

Term	Definition
Guard	A language primitive (guard expr) that asserts a condition and raises <code>GuardAssertionError</code> if false. Inside a retry block, triggers a retry.
HITL (Human-in-the-Loop)	A pattern where a human operator approves critical agent actions before they execute. INTHON's approve primitive implements HITL.
IR (Intermediate Representation)	A JSON-compatible data structure representing a compiled INTHON program: instructions, constants, names, per-agent metadata.
InthonPyObject	A Python proxy class wrapping imported modules. Intercepts attribute access to prevent sandbox escapes.
LALR(1)	A class of context-free grammars parseable with one-token lookahead. INTHON's grammar is LALR(1).
Lark	A Python parser generator supporting Earley and LALR parsing. INTHON uses Lark with LALR.
LEGB	Python's scope resolution order: Local, Enclosing, Global, Builtin. INTHON uses block scoping without enclosing-function scope.
MRO (Method Resolution Order)	The order in which Python searches a class's bases for an attribute. Computed by the C3 algorithm.
Memory namespace	A named SQLite table storing remembered facts. Namespaces are declared via use memory X.
Opcode	A single bytecode instruction. INTHON has 51 opcodes organized into 5 categories.
PEP (Python Enhancement Proposal)	The design document process for Python. Referenced throughout this document for specific changes (e.g. PEP 659, PEP 703).
PEG (Parsing Expression Grammar)	A grammar formalism supporting ordered choice. CPython 3.10+ uses PEG; INTHON uses LALR(1) via Lark.
Plan	The execution body of an agent block. A sequence of INTHON statements.
Plan rewrite	The act of replacing the remaining instructions of an agent's plan with a new plan, triggered by the <code>SELF_EVAL</code> opcode.
Policy	A set of resource quotas and permissions applied to an agent invocation. Declared in the policy block of an agent or in <code>inthon.toml</code> .
PyBridge	INTHON's sandboxed Python interop layer. Wraps imported modules in <code>InthonPyObject</code> proxies and gates imports via an allowlist.
Rewriter	A host-registered Python function invoked by the <code>SELF_EVAL</code> opcode to produce a new plan when a criterion fails.
Sandbox	The runtime environment that enforces capability bounds. INTHON's sandbox is the PyBridge + policy engine combination.
Self-evaluation	The language primitive (<code>eval self against ...</code>) that checks criteria against the current execution context and may trigger a plan rewrite.
Semantic analyzer	The compilation stage that resolves scopes, checks types, and validates capabilities.
Symbol table	A data structure recording, for each scope, which names are local, free, global, or builtin.
Token	A single lexical unit produced by the lexer (identifier, keyword, literal, operator, delimiter).

Term	Definition
Tool	A registered callable that an agent can invoke via the use tool declaration. Tools have schemas and cost models.
Tool registry	The global registry mapping tool names to their schemas and handlers.
Trace	A JSON tree recording every observable event in an INTHON execution. The trace is deterministic and replayable.
TraceView	A read-only wrapper around the trace, exposed to rewriters via the INTROSPECT_TRACE opcode.

Table C.1 — INTHON glossary.

Appendix D. References & Bibliography

This document draws on the CPython source tree, the Python Language Reference, Python Enhancement Proposals (PEPs), the Lark documentation, the INTHON project repository and its accompanying articles, and the broader literature on capability-secure languages and agent-oriented programming. References are grouped by category.

D.1 CPython Source Tree

- Python Software Foundation. CPython source tree. github.com/python/cpython.
- Key files referenced: *Parser/tokenizer.c*, *Parser/parser.c* (PEG), *Parser/Python.asdl*, *Python/ast.c*, *Python/symtable.c*, *Python/compile.c*, *Python/ceval.c*, *Python/import.c*, *Objects/typeobject.c*, *Include/opcode.h*, *Lib/opcode.py*.

D.2 Python Language Reference & PEPs

- Python Software Foundation. The Python Language Reference. docs.python.org/3/reference/.
- PEP 384: Defining a Stable ABI.
- PEP 442: Safe object finalization.
- PEP 451: A ModuleSpec Type for the Import System.
- PEP 484: Type Hints.
- PEP 489: Multi-phase extension module initialization.
- PEP 498: Literal String Interpolation (f-strings).
- PEP 552: Deterministic pycs.
- PEP 563: Postponed evaluation of annotations.
- PEP 572: The Walrus Operator (`:=`).
- PEP 617: New PEG parser for CPython.
- PEP 634: Structural Pattern Matching.
- PEP 654: Exception Groups and `except*`.
- PEP 659: Specializing Adaptive Interpreter.

- PEP 684: A Per-Interpreter GIL.
- PEP 703: Making the GIL Optional.
- PEP 734: Per-interpreter GILs and the interpreters module.
- PEP 779: Making the GIL optional in 3.13+.

D.3 Lark Parser Generator

- Erez Shinan. Lark parser generator. github.com/lark-parser/lark.
- Lark documentation: lark-parser.readthedocs.io.

D.4 INTHON Project

- Harsha Vardhan. INTHON: Agent-Level Programming Language Layer. github.com/harvatechs/inthon. v0.2.
- Harsha Vardhan. Why AI Agents Need Their Own Programming Language. medium.com/@techharva/why-ai-agents-need-their-own-programming-language-b47d9cc58e65. June 2026.
- Harsha Vardhan. INTHON for Everyone. medium.com/@techharva/inthon-for-everyone-2a8f356a0da3. June 2026.
- Esolangs wiki entry: esolangs.org/wiki/Inthon.
- INTHON website: harvatechs.github.io/inthon.
- Interactive developer guide: harvatechs.github.io/inthon/guide.html.

D.5 Capability-Secure Languages & Object-Capability Model

- Mark S. Miller. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins, 2006.
- Mark S. Miller et al. The E language. erights.org.
- Joe-E: A Security-Oriented Subset of Java. Mettler, Wagner, Close.
- Caja: A capability-secure JavaScript. Google, 2008.

D.6 Agent-Oriented Programming

- Yoav Shoham. Agent-Oriented Programming. Artificial Intelligence, 1993.
- LangChain documentation: python.langchain.com.
- AutoGen: github.com/microsoft/autogen.
- Semantic Kernel: github.com/microsoft/semantic-kernel.

D.7 Bytecode VMs & Compilers

- Andrew W. Appel. Compiling with Continuations. Cambridge University Press, 1992.
- Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis. CGO 2004.
- Christian Urban. The Design of a Verified Bytecode Verifier. PhD thesis, 2003.

- PyPy's RPython toolchain: *pypy.org*.

D.8 Tooling & Tree-sitter

- Tree-sitter: *tree-sitter.github.io*.
- Language Server Protocol specification: *microsoft.github.io/language-server-protocol*.
- Ruff linter: *github.com/astral-sh/ruff*.
- Mypy type checker: *github.com/python/mypy*.