PyKen x Plutuscope Documentation

PyKen

The PyKen project is a transpilation framework designed to translate Python-like smart contract definitions into Aiken, a domain-specific language for Cardano validators. The system allows developers to express validator logic in Python while automatically generating equivalent Aiken code.

Architecture Overview

The project consists of four main modules, each responsible for different stages of the transpilation pipeline:

1. py_parser.py – Extracts function metadata (name, args, return type, decorators) from Python AST.
2. py_validator.py – Wraps parsed functions into validator definitions and prepares Aiken-compatible code.
3. validator_emitter.py – Core transpiler: walks the AST and emits Aiken source code, handling constructs like imports, classes, functions, pipelines, pattern matching, asserts, try/raise, and more.
4. transpile.py – Entry script: reads a Python file, invokes the transpiler, prints the result, and writes it into a .ak file.

Module Details

py_parser.py

This module defines FunctionParser, an AST visitor that extracts essential metadata from Python functions. It maps Python type annotations into Aiken types using a type system dictionary. Each parsed function is represented with its name, argument list (with mapped types), return type, decorators, and the original AST node.

py_validator.py

This module defines ValidatorParser, which transforms parsed Python functions into Aiken validator definitions. It leverages ValidatorEmitter to translate the function body. Functions are wrapped inside 'validator { ... }' blocks. Additionally, a DataType base class is defined for lightweight representation of Aiken-like data types.

validator_emitter.py

This is the heart of the transpiler. It defines ValidatorEmitter, a visitor that walks Python AST nodes and translates them into equivalent Aiken constructs.

Key features include:
- Import translation: 'import a.b' → 'use a/b'.
- Class translation: emits either validator blocks or public type definitions.
- Pipeline recognition: converts chained variable reassignments into Aiken '|>' pipeline style.
- Control structures: if/else, pattern-matching with 'when', asserts, raises, and try-except handling.
- Expression serializer: converts literals, calls, attributes, lists, dicts, comprehensions, and operators into Aiken equivalents.

transpile.py

This script acts as the driver for the transpilation process. It reads a Python source file, passes its contents through emit_aiken_from_source, prints the transpiled output for debugging, and writes the result to a .ak file. It enables a straightforward one-command conversion workflow.

Quick Start

1. Ensure Python 3.10+ (Recommended)

2. Put your validator Python file (for example l6-time.py) in the repo (or edit transpile.py to point to your file).

3. Transpile:

```
PS C:\Users\USER\Desktop\PyKen Validators> python transpile.py "C:\Users\USER\Desktop\PyKen Validators\module 103\l0-mock-tx.py"
use cardano/assets.{PolicyId}
use cardano/transaction.{OutputReference, Transaction, placeholder}
use aiken/crypto.{Data}
use mocktail.{complete, mock_utxo_ref, mocktail_tx}
fn transaction_placeholder() -> Transaction {
  placeholder
}
validator always_succeed {
  mint(_redeemer: Data, _policy_id: PolicyId, _tx: Transaction) {
    True
  }
  spend(_datum: Option<Data>, _redeemer: Data, _input: OutputReference, _tx: Transaction) {
    True
  }
  else(_) {
    fail
  }
}
test m103_l0_aiken_builtin_placeholder {
  expect transaction_placeholder() == placeholder
}
test m103_l0_aiken_mocktail_tx {
  let tx = complete(mocktail_tx())
  expect placeholder == tx
}
test m103_l0_always_succeed_minting_policy {
  always_succeed.mint(None, "", placeholder)
}
test m103_l0_always_succeed_spending_validator {
  let OutputReference { .. } = input_ref
  always_succeed.spend(None, Void, input_ref, placeholder)
}
test m103_l0_mocktail_mock_utxo_ref {
  expect mock_utxo_ref(0, 0) == OutputReference { transaction_id: "cd82a190d3b4ef95a50bd791882959f541e308adb69b12d022d94a6d9f02bcf0", output_index: 0 }
}
Aiken code written to: C:\Users\USER\Desktop\PyKen Validators\module 103\l0-mock-tx.ak
PS C:\Users\USER\Desktop\PyKen Validators>
```

Workflow

The typical transpilation workflow is as follows:

1. Write validator logic in Python (PyKen).
2. Run transpile.py with the target source file.
3. The transpiler parses the Python AST using py_parser.
4. py_validator and validator_emitter work together to generate equivalent Aiken code.
5. The output is written to a .ak file for further compilation and deployment on Cardano.

```
+--------------------+          +----------------------+         +-----------------+
| Your Python files  | --parse->| py_parser.py         | --uses->| type_sys.py     |
| (decorated funcs)  |          | (AST -> function)    |         +-----------------+
+--------------------+          +----------------------+
                                           |
                                           v
                                +----------------------+
                                | py_validator.py      |  (wraps body into validator blocks)
                                +----------------------+
                                           |
                                           v
                                +----------------------+
                                | validator_emitter.py |  (AST -> Aiken source)
                                +----------------------+
                                           |
                                           v
                                +----------------------+
                                | transpile.py         |  (prints & writes .ak)
                                +----------------------+

Support / test libs (not required to transpile):
- mocktail.py + cocktail.py  -> local transaction builder & helpers for unit tests
- aiken/*, cardano/*         -> type mirrors for writing Python validators that look like Aiken
```

Files & API Reference

1. transpile.py

- - **Functionality**: Driver script that reads SRC_FILE, invokes emit_aiken_from_source(src), and subsequently outputs .ak files.
- - **Edit**: Modify the SRC_FILE path to alter the input.

2. validator_emitter.py

- - **Functionality**: Core AST → Aiken emitter.
- - **Public API**: emit_aiken_from_source(src_text: str) -> str.
- - **Key Class**: ValidatorEmitter(ast.NodeVisitor) — traverses the AST and generates Aiken text.
- - **Heuristics Implemented** (essential to comprehend):
  - - **Imports**: import a.b → utilize a/b; from a.b import X → utilize a/b.{X}.
  - - **Classes**: If a class includes spend/mint/else_ methods, it is emitted as validator Name { ... }. Otherwise, it is emitted as pub type Name { ... } utilizing __init__ arguments (or a singular variant).
  - - **Pipelines**: Identifies sequences such as x = foo(...); x = bar(x,...); return x and emits base **|>** seg **|>** seg.
  - - **Pattern Matching**: Discerns isinstance chains and in (A, B) style branching and emits when <var> is { ... }.
  - - **Special Mappings**: Some, None, Datum, Data constructs, print → trace @"...".
  - - **Fallbacks**: Should the emitter be unable to generate a pipeline, it resorts to literal expression text or <expr> placeholder.
  - - **Important Helpers to Inspect**: _pipeline_segment_from_rhs, _try_emit_pipeline, _expr (serializes expressions).

3. py_parser.py

- - **Functionality**: Traverses the Python AST, extracting function definitions: name, args, ret_type, decorators, and retains the original AST node (thereby allowing the emitter to revisit the body).
- - **Mapping**: Maps annotation names to Aiken-like types utilizing type_sys.PY_TO_AIKEN (a straightforward mapping).
- - **Exported Object**: FunctionParser with a list of functions.

4. py_validator.py

- - **Functionality**: Encapsulates parsed function information into validators.
- - **Emission**: Employs ValidatorEmitter to emit function body statements and constructs a validator name(args) -> Bool { ... } block.
- - **Internal Use**: Additionally defines a DataType helper class for internal utilization.

5. type_sys.py

- - **Mapping**: PY_TO_AIKEN mapping (string-to-string) for Python annotation names → Aiken names.
- - **Functionality**: map_type function: given a string, returns the Aiken type or heuristic fallback.
- - **Type Logic**: Type enum + unify helper for rudimentary unification logic (utilized to reason about types; not extensively employed as of yet).

6. mocktail.py

- - **Functionality**: MocktailTx immutable-style builder and helpers (tx_in, tx_out, mint, complete, etc.) — designed to construct transaction test fixtures.
- - **Fallback Logic**: Resorts to lightweight dataclasses if Cardano libraries are absent.
- - **Serialization**: CBOR-datum serialization and blake2b fallback included.
- - **Testing**: Recommended test harness for unit-testing validators in a local context.

7. cocktail.py

- - **Functionality**: Utility functions to scrutinize transactions/inputs/outputs/value manipulations: inputs_at, outputs_with, value_geq, etc.
- - **Usage**: Utilized in Python tests to assert validator behavior.

8. **aiken/*, cardano/*, bytearray.py, string.py, int_utils.py, option_utils.py, aiken/crypto.py, etc.**

- - **Functionality**: Mirrors of Aiken standard library primitives and Cardano types. They facilitate the writing of Python that emulates Aiken and can be executed locally.
- - **Caution**: These modules are stubs (numerous functions are simplified or yield placeholder values), thus it is imperative to remember: these are intended for testing and transpilation convenience, not for production cryptographic accuracy.

9. decorators.py

- - **Functionality**: @validator decorator: devoid of runtime effect (wraps function) but signifies functions with __pyken_validator__ = True, enabling parsers to reliably identify intended validators.

Plutuscope — Validator narration, simulation, and trace explorer

Plutuscope is a small CLI utility for working with Aiken .ak smart-contract projects. It helps you:

- scan a project for .ak files and collect runtime “trace” output (from aiken or synthesized instrumentation),

- parse and render those traces into a navigable tree,

- narrate validator functions and map tests that call them,

- optionally simulate simple execution trees for quick review.

1. Quick start

Prerequisites:

- Python 3.8+ (script uses pathlib, dataclasses, typing).

- Optional: aiken on your PATH to get real trace/test output. Without it, the tool falls back to file-based or synthesized traces.

- Install recommended Python packages for nicer output “rich -> (pip install rich)”.

The tool imports rich for console formatting.

Run examples

Scan project and inspect traces interactively:

python plutuscope.py --scan path/to/project

Narrate a single validator and optionally simulate:

python plutuscope.py --aiken contracts/my_validator.ak --simulate

Scan with instrumentation (inserts trace("enter ...") and trace("exit ...") in a temp copy):

python plutuscope.py --scan path/to/project --instrument

2. CLI / usage

usage: plutuscope.py [-h] [--aiken Aiken] [--tests Tests] [--simulate]
[--scan Scan] [--no-aiken] [--instrument]
[--verbose]

Key options:

- --scan <dir>: recursively find .ak files under <dir> and analyze them.

- --aiken <file>: analyze a single .ak validator file and print narration.

- --tests <log>: path to a test log (used by --aiken mode if provided).

- --simulate: include a small simulated execution tree in validator narration.

- --instrument: create a temporary instrumented copy of the project and synthesize traces from inserted trace(...) calls.

- --no-aiken: do not run aiken even if it exists; force mock/instrumented behavior.

- --verbose: show extra debug/logging information.

Interactive behavior (when scanning): - A summary table is printed showing discovered files and trace counts.

- Choose an index to view a rich.Tree rendering of parsed traces.

- Optionally display raw trace text with true/false colorized.

3. Major features & behavior

Traces collection

- If aiken is on PATH and prefer_aiken is True, the tool runs aiken check <project_dir> and:
  - attempts to parse JSON output for a tests key,
  - extracts trace-like lines from stdout/stderr (lines containing [TRACE], trace(, Entering function, Returning).
- If aiken is absent or disabled, the tool attempts fallbacks in this order:
  1. plutus.json (if present)
  2. tests.log (heuristic parsing into test names & statuses)
  3. If --instrument is set, copy the project into a temp dir and insert trace("enter <fn>")/trace("exit <fn>") into detected functions — then synthesize [TRACE] lines from those inserted calls.

4. As a last resort, mock_traces_from_file() scans the source file for fn and inline trace calls and emits a compact mock trace.

Trace parsing & rendering

- Trace parsing builds a tree of Node objects representing enter, return, check messages, and plain messages.
- Node fields: title, status (True/False/None), children, and src_loc (file, line).
- The console render shows green/red highlighting for true/false and prepends ✅/❌ to status nodes.

Tests & validator narration

- The tool parses test name(...) { ... } blocks across the project and extracts calls like validator.method(...) to map tests to validator methods.
- For a validator file, it extracts methods inside the validator { ... } block (e.g., spend(...) { ... }, mint(...) { ... }) or fn functions within the block.
- Heuristics narrate simple control flow: detects if/else, literal True/False, and fail occurrences and prints a readable summary for each method.

4. Internals (important functions & regexes)

Key functions (what they do)

- find_ak_files(root) — find .ak files recursively.
- find_project_root(start) — walks parents to locate aiken.toml.
- run_aiken_on_file(file_path, project_dir) — runs aiken check and captures stdout+stderr.
- run_aiken_and_collect(project_dir, instrument=False) — high-level collector combining aiken, fallback files, and instrumentation.
- _instrument_project_for_traces(project_dir) — copy+inject traces into .ak files and return temp path.
- _synthesize_traces_from_instrumented(tmp_project) — read inserted trace calls and turn into [TRACE] lines.
- mock_traces_from_file(file_path) — generate reasonable mock traces from a single file.
- parse_trace(log_text, file_hint) — parse [TRACE] lines into a Node tree.
- render_node_tree(node, parent) — render a Node tree using rich.Tree.
- pretty_print_validator(file, do_simulate) — narrate validator methods and find tests that call them.

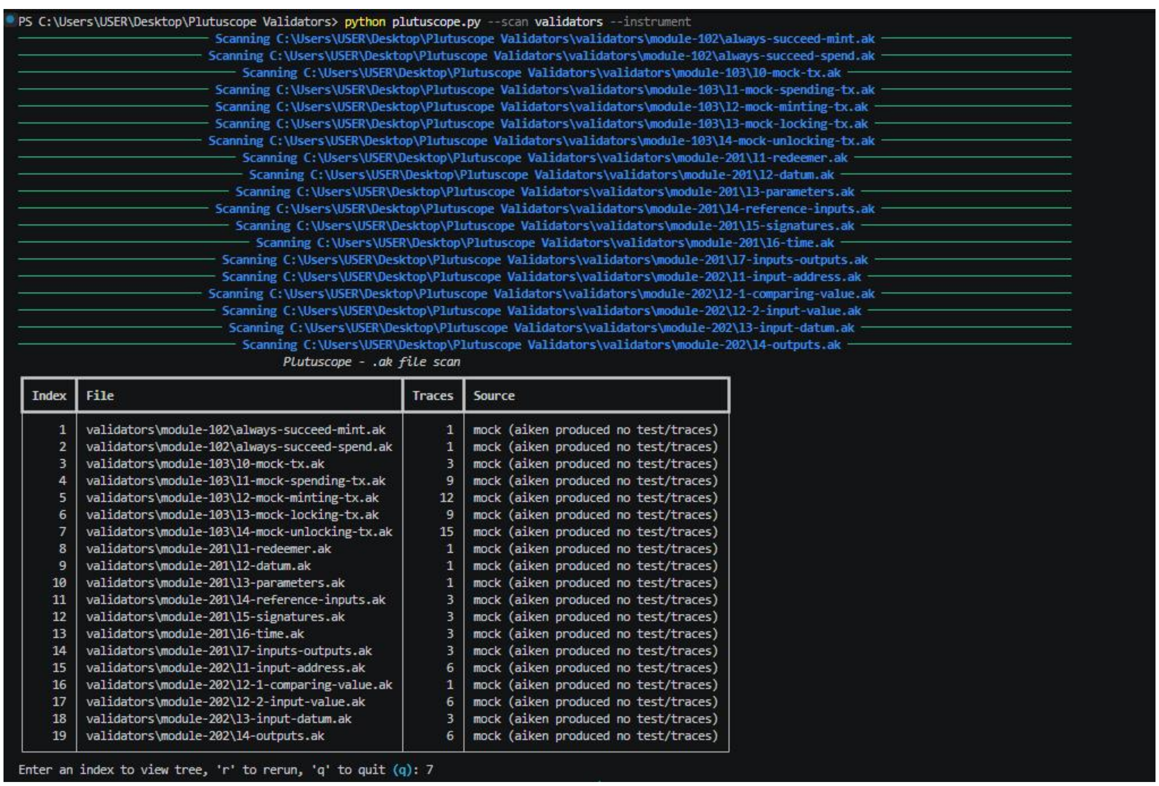
- parse_tests_in_text(text) — parse test ... {} blocks and calls inside them.

Important regexes

- TRACE_RE — recognizes lines starting with [TRACE] (case-insensitive).
- ENTER_RE / RETURN_RE — “Entering function ”, “Returning ”.
- CHECK_RE — matches check or matching expressions, capturing optional -> true|false tail.
- TEST_DECL_RE — finds test name(args) { ... } declarations.
- TEST_CALL_RE — finds validator.method(occurrences inside tests.
- TRACE_INLINE_RE — matches trace inline strings (currently matches double-quoted forms).

5. Example outputs

Summary table (--scan)

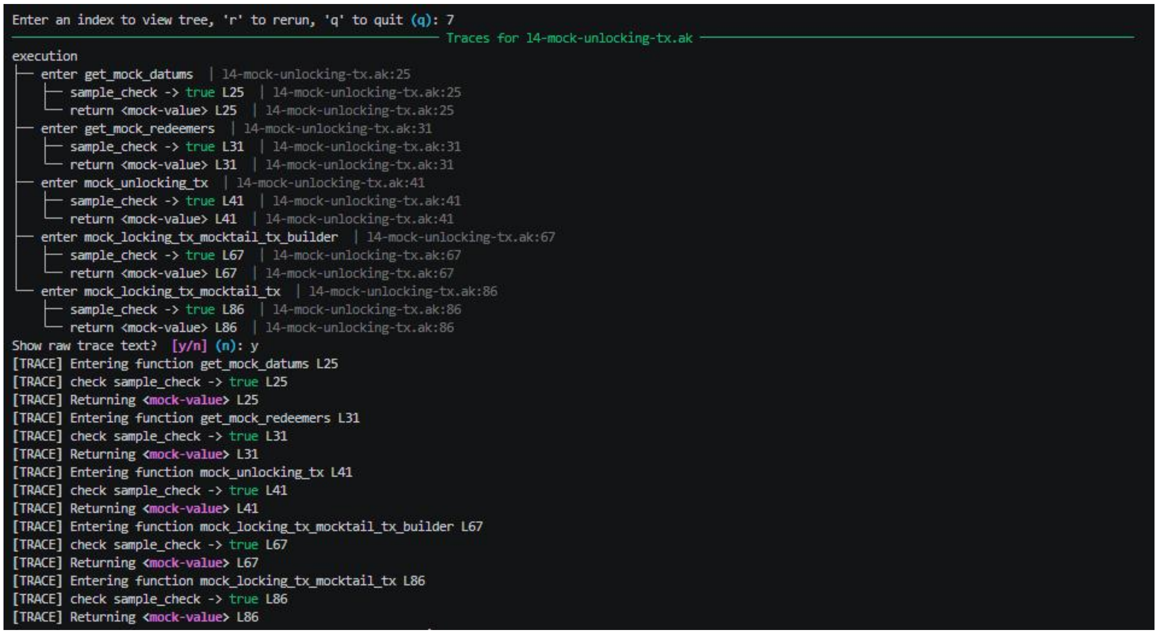


```
PS C:\Users\USER\Desktop\Plutuscope Validators> python plutuscope.py --scan validators --instrument
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-102\always-succeed-mint.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-102\always-succeed-spend.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-103\10-mock-tx.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-103\11-mock-spending-tx.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-103\12-mock-minting-tx.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-103\13-mock-locking-tx.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-103\14-mock-unlocking-tx.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-201\11-redeemer.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-201\12-datum.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-201\13-parameters.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-201\14-reference-inputs.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-201\15-signatures.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-201\16-time.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-201\17-inputs-outputs.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-202\11-input-address.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-202\12-1-comparing-value.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-202\12-2-input-value.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-202\13-input-datum.ak
Scanning C:\Users\USER\Desktop\Plutuscope Validators\validators\module-202\14-outputs.ak
                         Plutuscope - .ak file scan
```

Index	File	Traces	Source
1	validators\module-102\always-succeed-mint.ak	1	mock (aiken produced no test/traces)
2	validators\module-102\always-succeed-spend.ak	1	mock (aiken produced no test/traces)
3	validators\module-103\10-mock-tx.ak	3	mock (aiken produced no test/traces)
4	validators\module-103\11-mock-spending-tx.ak	9	mock (aiken produced no test/traces)
5	validators\module-103\12-mock-minting-tx.ak	12	mock (aiken produced no test/traces)
6	validators\module-103\13-mock-locking-tx.ak	9	mock (aiken produced no test/traces)
7	validators\module-103\14-mock-unlocking-tx.ak	15	mock (aiken produced no test/traces)
8	validators\module-201\11-redeemer.ak	1	mock (aiken produced no test/traces)
9	validators\module-201\12-datum.ak	1	mock (aiken produced no test/traces)
10	validators\module-201\13-parameters.ak	1	mock (aiken produced no test/traces)
11	validators\module-201\14-reference-inputs.ak	3	mock (aiken produced no test/traces)
12	validators\module-201\15-signatures.ak	3	mock (aiken produced no test/traces)
13	validators\module-201\16-time.ak	3	mock (aiken produced no test/traces)
14	validators\module-201\17-inputs-outputs.ak	3	mock (aiken produced no test/traces)
15	validators\module-202\11-input-address.ak	6	mock (aiken produced no test/traces)
16	validators\module-202\12-1-comparing-value.ak	1	mock (aiken produced no test/traces)
17	validators\module-202\12-2-input-value.ak	6	mock (aiken produced no test/traces)
18	validators\module-202\13-input-datum.ak	3	mock (aiken produced no test/traces)
19	validators\module-202\14-outputs.ak	6	mock (aiken produced no test/traces)

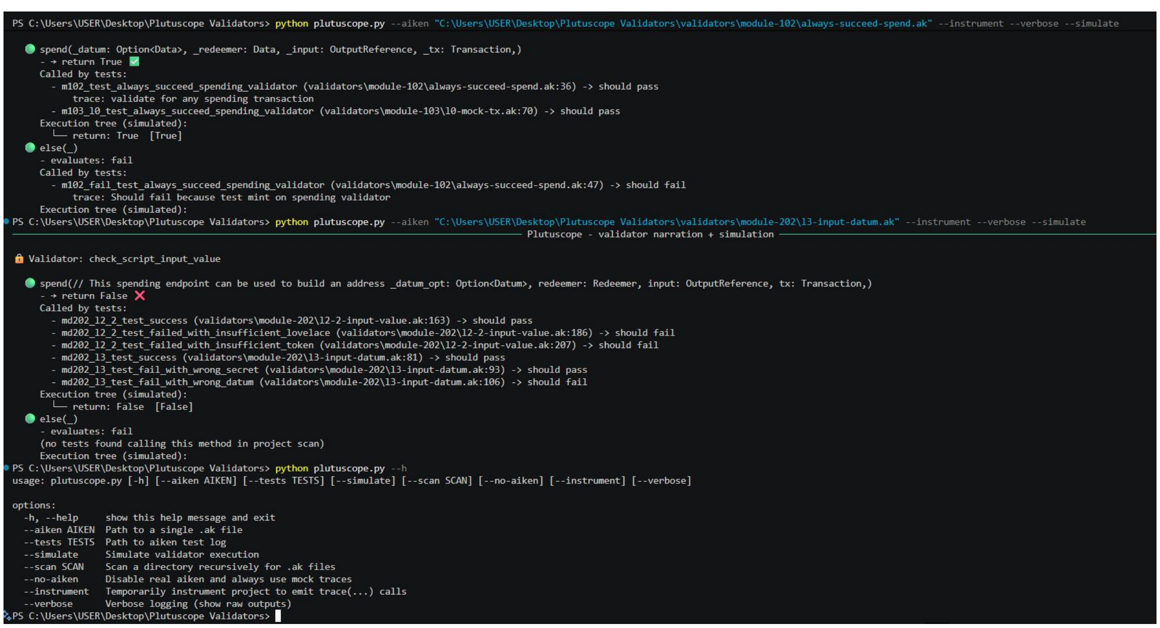
```
Enter an index to view tree, 'r' to rerun, 'q' to quit (q): 7
```

Rendered trace tree



```
Enter an index to view tree, 'r' to rerun, 'q' to quit (q): 7
──────────────────────────────────────── Traces for 14-mock-unlocking-tx.ak ────────────────────────────────────────
execution
├── enter get_mock_datums  │ 14-mock-unlocking-tx.ak:25
│   ├── sample_check -> true L25  │ 14-mock-unlocking-tx.ak:25
│   └── return <mock-value> L25  │ 14-mock-unlocking-tx.ak:25
├── enter get_mock_redeemers  │ 14-mock-unlocking-tx.ak:31
│   ├── sample_check -> true L31  │ 14-mock-unlocking-tx.ak:31
│   └── return <mock-value> L31  │ 14-mock-unlocking-tx.ak:31
├── enter mock_unlocking_tx  │ 14-mock-unlocking-tx.ak:41
│   ├── sample_check -> true L41  │ 14-mock-unlocking-tx.ak:41
│   └── return <mock-value> L41  │ 14-mock-unlocking-tx.ak:41
├── enter mock_locking_tx_mocktail_tx_builder  │ 14-mock-unlocking-tx.ak:67
│   ├── sample_check -> true L67  │ 14-mock-unlocking-tx.ak:67
│   └── return <mock-value> L67  │ 14-mock-unlocking-tx.ak:67
└── enter mock_locking_tx_mocktail_tx  │ 14-mock-unlocking-tx.ak:86
    ├── sample_check -> true L86  │ 14-mock-unlocking-tx.ak:86
    └── return <mock-value> L86  │ 14-mock-unlocking-tx.ak:86
Show raw trace text? [y/n] (n): y
[TRACE] Entering function get_mock_datums L25
[TRACE] check sample_check -> true L25
[TRACE] Returning <mock-value> L25
[TRACE] Entering function get_mock_redeemers L31
[TRACE] check sample_check -> true L31
[TRACE] Returning <mock-value> L31
[TRACE] Entering function mock_unlocking_tx L41
[TRACE] check sample_check -> true L41
[TRACE] Returning <mock-value> L41
[TRACE] Entering function mock_locking_tx_mocktail_tx_builder L67
[TRACE] check sample_check -> true L67
[TRACE] Returning <mock-value> L67
[TRACE] Entering function mock_locking_tx_mocktail_tx L86
[TRACE] check sample_check -> true L86
[TRACE] Returning <mock-value> L86
```

Validator narration and "--help"



```
PS C:\Users\USER\Desktop\Plutuscope Validators> python plutuscope.py --aiken "C:\Users\USER\Desktop\Plutuscope Validators\validators\module-102\always-succeed-spend.ak" --instrument --verbose --simulate

  🟢 spend(_datum: Option<Data>, _redeemer: Data, _input: OutputReference, _tx: Transaction,)
     - → return True ✅
     Called by tests:
       - m102_test_always_succeed_spending_validator (validators\module-102\always-succeed-spend.ak:36) -> should pass
           trace: validate for any spending transaction
       - m103_10_test_always_succeed_spending_validator (validators\module-103\10-mock-tx.ak:70) -> should pass
     Execution tree (simulated):
       └── return: True  [True]
  🟢 else(_)
     - evaluates: fail
     Called by tests:
       - m102_fail_test_always_succeed_spending_validator (validators\module-102\always-succeed-spend.ak:47) -> should fail
           trace: Should fail because test mint on spending validator
     Execution tree (simulated):
PS C:\Users\USER\Desktop\Plutuscope Validators> python plutuscope.py --aiken "C:\Users\USER\Desktop\Plutuscope Validators\validators\module-202\13-input-datum.ak" --instrument --verbose --simulate
──────────────────────────────────────── Plutuscope - validator narration + simulation ────────────────────────────────────────

🔒 Validator: check_script_input_value

  🟢 spend(// This spending endpoint can be used to build an address _datum_opt: Option<Datum>, redeemer: Redeemer, input: OutputReference, tx: Transaction,)
     - → return False ❌
     Called by tests:
       - md202_12_2_test_success (validators\module-202\12-2-input-value.ak:163) -> should pass
       - md202_12_2_test_failed_with_insufficient_lovelace (validators\module-202\12-2-input-value.ak:186) -> should fail
       - md202_12_2_test_failed_with_insufficient_token (validators\module-202\12-2-input-value.ak:207) -> should fail
       - md202_13_test_success (validators\module-202\13-input-datum.ak:81) -> should pass
       - md202_13_test_fail_with_wrong_secret (validators\module-202\13-input-datum.ak:93) -> should pass
       - md202_13_test_fail_with_wrong_datum (validators\module-202\13-input-datum.ak:106) -> should fail
     Execution tree (simulated):
       └── return: False  [False]
  🟢 else(_)
     - evaluates: fail
     (no tests found calling this method in project scan)
     Execution tree (simulated):
PS C:\Users\USER\Desktop\Plutuscope Validators> python plutuscope.py --h
usage: plutuscope.py [-h] [--aiken AIKEN] [--tests TESTS] [--simulate] [--scan SCAN] [--no-aiken] [--instrument] [--verbose]

options:
  -h, --help     show this help message and exit
  --aiken AIKEN  Path to a single .ak file
  --tests TESTS  Path to aiken test log
  --simulate     Simulate validator execution
  --scan SCAN    Scan a directory recursively for .ak files
  --no-aiken     Disable real aiken and always use mock traces
  --instrument   Temporarily instrument project to emit trace(...) calls
  --verbose      Verbose logging (show raw outputs)
PS C:\Users\USER\Desktop\Plutuscope Validators>
```

Appendix — quick glossary

- aiken — the Aiken toolchain (assumed to be available on PATH for best results).
- trace("...") — inline tracing function used by Aiken or injected by the instrumentation helper.
- plutus.json / tests.log — typical fallback files that may contain test or trace results.