

PyMOP: A Runtime Verification Tool for Python — Appendix

0.1 Instrumentation Strategies Comparison

Strategy 1: Python-level Monkey Patching. Instrumentation using monkey patching involves dynamically modifying the behavior of a function or class method at runtime by replacing it with a wrapper that triggers events before and after calling the original. This is done by changing the function pointer to point to the wrapper, leaving the external interface unchanged.

Pros:

- Precisely targets specific functions without affecting unrelated ones.
- Fast and lightweight (only reassigns a function reference).
- Preserves the external API and return values of the function, reducing the chance of bugs.

Cons:

- Cannot patch built-in read-only types such as `list` and `dict` in CPython, which are implemented in C.
- Cannot instrument literals like `[]`, `{}`, `+`, `-`, `==`, `<`, etc.
- Cannot handle control-flow constructs like `for` loops or `if` statements.
- Instrumentation applies globally and cannot be toggled per module.

We found a partial workaround to the first limitation: by replacing references to read-only classes with references to a subclass, we can monkey-patch the subclass. However, this only affects newly created objects but not existing ones (e.g., from dependencies or the Python standard library). Literals like `[]` still point to the original built-in types and remain uninstrumented.

Strategy 2: Python-level Monkey Patching + C-level Monkey Patching via `forbiddenfruit` and `ctypes`. This strategy extends monkey patching using the `forbiddenfruit` library¹, which leverages CPython's `ctypes` API² to enable patching of built-in read-only types. It mitigates several limitations of strategy 1 by allowing direct manipulation of C-level type slots.

Pros:

- Enables instrumentation of built-in read-only types like `list`, `dict`, and `set`.
- Affects literals such as `[]`, `{}`, and `{"key": value}` directly.

Cons:

- Cannot instrument some CPython-optimized operations (e.g., `1 + 2`) or certain `__init__` methods that bypass Python-level dispatch due to C-level shortcuts.
- Lacks support for some methods like comparison dunder methods such as `__lt__`, `__eq__`, `__ne__`, although we suspect that such support is possible. we opened an issue to confirm that assumption with the maintainers³.
- Works only with CPython and is incompatible with other Python implementations.

Strategy 3: Monkey Patching + AST Rewriting. This strategy modifies source code at the AST (Abstract Syntax Tree) level before execution. Using `sys.meta_path`⁴, we intercept module imports and dynamically rewrite their AST just before the module is executed. We use this to replace literals, built-ins, or function calls with instrumented versions. Unlike offline instrumentation, runtime

¹<https://github.com/clarete/forbiddenfruit>

²<https://docs.python.org/3/library/ctypes.html>

³<https://github.com/clarete/forbiddenfruit/issues/80>

⁴https://docs.python.org/3/library/sys.html#sys.meta_path

AST rewriting avoids redundant disk I/O and ensures that only the actually imported code is transformed.

In our implementation, this is achieved by wrapping the existing path finders in `sys.meta_path` with new ones that delegate most of their functionality to the original ones, but override parts responsible for loading the modules. This ensures compatibility with Python’s standard module resolution and preserves import behavior. The wrappers return custom loaders that parse and transform the module’s AST, injecting instrumentation selectively. This mechanism is initialized early using `sitecustomize.py`⁵, ensuring that the transformation hooks are active before any user-defined code is imported.

Pros:

- Can instrument literals (`[]`, `{}`), CPython-optimized expressions like `1 + 2`, and implicit calls like `__init__`.
- Supports instrumentation of read-only built-in types.
- Enables instrumentation of dunder methods and control-flow constructs (`if`, `for`).
- Allows selective instrumentation of parts of the source code.

Cons:

- Slower due to the cost of AST parsing and transformation.
- May introduce bugs if AST transformations are not applied carefully and conservatively.

Capability	Strategy 1	Strategy 2	Strategy 3
Instrument regular Python functions	✓	✓	✓
Instrument built-in read-only types	✗	✓	✓
Instrument literals (<code>[]</code> , <code>1+2</code>)	✗	✓*	✓
Instrument comparison dunders (<code>__eq__</code> , etc.)	✗	✗	✓
Instrument control flow (<code>for</code> , <code>if</code>)	✗	✗	✓
Selective instrumentation per module	✗	✗	✓
Cross-implementation support (e.g., PyPy)	✓	✗	✓
Ease of implementation	✓	Medium	Medium
Performance overhead	✓	✓	✗

Table 1. Comparison of instrumentation strategies in PyMOP.

* C-level Monkey Patching via `forbiddenfruit` can handle some literals like list literals `[]` but not others like integer literal addition `1+2`.

Comparison. Different instrumentation strategies are suitable for different scenarios. Strategy 1 (pure monkey patching) is ideal for instrumenting regular Python functions and classes in a lightweight and portable way. Strategy 2 (monkey patching with `ctypes`) is useful when monitoring built-in types and their literals is required. Finally, Strategy 3 (monkey patching with AST rewriting) offers the most comprehensive coverage, including literals, control flow, and selective instrumentation, and is the appropriate choice when deep instrumentation is needed to fully monitor program behavior. We used Strategy 3 for the evaluation in this paper.

Discussion. Implementing instrumentation across the Python stack introduced several challenges. Monkey patching is lightweight but cannot intercept operations on literals or built-in types. C-level patching extends coverage, though it relies on CPython internals and remains incomplete in areas

⁵<https://docs.python.org/3/library/site.html#module-sitecustomize>

like comparison dunderers. AST rewriting enables deeper instrumentation, including control flow and implicit operations, but comes with transformation complexity and runtime overhead.

We also experimented with implementing instrumentation directly in C as a Python extension. While this approach worked reliably, we ultimately used the `forbiddenfruit` library, which achieves similar effects via CPython’s `ctypes` API, offering more flexibility without requiring compilation or deployment of native extensions.

In parallel, Python provides built-in instrumentation hooks such as `sys.setprofile`⁶, which triggers callbacks on every function call, return, or exception. While useful for general tracing, it does not capture low-level operations on literals or built-in types. Similarly, Runtime Audit Hooks⁷ allow inspection of various runtime events, but lack the selectivity and context awareness needed to monitor specific program behaviors without substantial post-processing.

⁶<https://docs.python.org/3/library/sys.html>

⁷<https://peps.python.org/pep-0578/>

0.2 Documentation Keywords and Extraction Patterns

We first search API docs for hints on API usage constraints captured in sentences with "should", "only", "must", "note", etc. (Our artifacts contain the complete list of keywords.)

To identify API usage constraints from natural-language docs, we search for sentences sentences that contain restrictive cues. In particular, we first filter sentences in API docs using a set of keywords that frequently signal API usage requirements or recommendations. The complete list of keywords used are as follows:

- note
- warning
- prohibited
- only once
- must
- imperative
- only
- not support
- permissible
- should

In addition to keyword filtering, we also applied a set of regular-expression patterns to extract ordering, dependency, and obligation constraints expressed in API docs. The complete list of patterns used are as follows:

(1) X before Y

```
/(.+)s+(?:before|prior to|preceding|earlier than|ahead of|in advance of)
\s+(.+)/i
```

(2) X after Y

```
/(.+)s+(?:after|subsequent to|later than|afterward)\s+(.+)/i
```

(3) Call/Invoke/Execute/Run/Perform X before Y

```
/(?:\bcall(?:s|ed|ing)?\b|\binvoke(?:s|d|ing)?\b|\bexecute(?:s|d|ing)?\b|
\b(?:run|start|perform|buse|initialize|load|open|connect)(?:s|ed|ing)?\b)
\s+(.+)s+(?:before|prior to|preceding|earlier than|ahead of|in advance of)
\s+(.+)/i
```

(4) Call/Invoke/Execute/Run/Perform X after Y

```
/(?:\bcall(?:s|ed|ing)?\b|\binvoke(?:s|d|ing)?\b|\bexecute(?:s|d|ing)?\b|
\b(?:run|start|perform|buse|initialize|load|open|connect)(?:s|ed|ing)?\b)
\s+(.+)s+(?:after|subsequent to|later than|afterward)\s+(.+)/i
```

(5) X depends on Y

```
/(.+)s+(?:depends on|relies on|contingent upon|requires|necessitates)
\s+(.+)/i
```

(6) Cannot call/invoke/execute/run/perform X until Y has been called/invoked

```
/(?:\bcannot\b|\bcan't\b|\bdo not\b|\bdon't\b|\bshould not\b|\bshouldn't\b|
\bunable to\b).*?(?:\bcall(?:s|ed|ing)?\b|\binvoke(?:s|d|ing)?\b|
\bexecute(?:s|d|ing)?\b|\brun(?:s|ning)?\b|\bperform(?:s|ed|ing)?\b)
\s+(.+) \s+(?:until|before|after|when|once|unless) \s+(.+) \s+has.*?
(?:\bcalled\b|\binvoked\b|\bexecuted\b|\brun\b|\bperformed\b)/i
```

(7) Must call/invoke/execute/run/perform X before Y

```
/(?:\bmust\b|\bneeds to\b|\bneed to\b|\bshould\b|\bhave to\b|\bought to\b|
\brequired to\b|\bessential to\b|\bcrucial to\b)
\s+(?:call|invoke|execute|run|perform) \s+(.+) \s+
(?:before|prior to|preceding|earlier than|ahead of|in advance of)
\s+(.+)/i
```

(8) Should call/invoke/execute/run/perform X after Y

```
/\bshould.*?\b(?:call|calls|called|calling|invoke|invokes|invoked|invoking|
execute|executes|executed|executing|run|runs|ran|running|
perform|performs|performed|performing).*?
\b(?:[w]+\(.+?\)).*?\b(?:[w]+\(.+?\))/i
```

(9) First X, followed by Y

```
/(?:\bfirst\b|\binitially\b|\bto start\b|\bbeginning with\b|\bat first\b)
\s+(.+) ,? \s+(?:followed by|then|next|afterward|subsequently|and then)
\s+(.+)/i
```

(10) Once X is called/invoked, proceed with Y

```
/(?:\bonce\b) \s+(.+) \s+(?:is|has been)
\s+(?:called|invoked|executed|run|performed),? \s+
(?:proceed with|proceed to|continue to|go ahead and|move on to|
be sure to|ensure that you) \s+
(?:call|invoke|execute|run|perform) \s+(.+)/i
```

(11) X should always precede Y

```
/(.+) \s+(?:should|must|needs to)? \s*(?:always)? \s*(?:precede|come before)
\s+(.+)/i
```

(12) X cannot be called until Y

```
/(.+) \s+(?:cannot|can't|should not|shouldn't|unable to) \s+be\s+
(?:called|invoked|executed|run|performed)
\s+(?:until|before|after|when|once|unless) \s+(.+)/i
```

(13) Failure to call/invoke/execute/run/perform X before Y will cause

```
/(?:\bfailure to\b|\bneglecting to\b|\bomission of\b|\bnot\b|\bwithout\b)
\s+(?:call|invoke|execute|run|perform) \s+(.+) \s+
(?:before|prior to|preceding|earlier than|ahead of|in advance of) \s+(.+)
.*?(?:will cause|results in|leads to|triggers|causes|may result in)/i
```

(14) X is required before Y

```
/(.?)\s+(?:is|are)\s+(?:required|essential|crucial)
\s+(?:before|prior to|preceding|earlier than|ahead of|in advance of)
\s+(.+)/i
```

(15) Ensure/Make sure to call/invoke/execute/run/perform X before Y

```
/(?:\bensure\b|\bmake sure\b)\s+(?:to\s+)?(?:call|invoke|execute|run|perform)
\s+(.?)\s+(?:before|prior to|preceding|earlier than|ahead of|in advance of)
\s+(.+)/i
```

(16) Do/Perform X before Y

```
/(?:\bdo\b|\bperform\b)\s+(.?)\s+
(?:before|prior to|preceding|earlier than|ahead of|in advance of)
\s+(.+)/i
```

(17) You need to perform X first, followed by Y

```
/(?:\byou\b|\bwe\b|\bthey\b|\bone must\b)\s+
(?:need to|must|should)\s+
(?:perform|call|invoke|execute|run)\s+(.?)\s+
(?:first|initially|to start|beginning with|at first),?\s+
(?:followed by|then|next|afterward|subsequently|and then)\s+(.+)/i
```

(18) X must be called prior to invoking Y

```
/(.?)\s+(?:must|needs to|need to|should|have to|ought to|required to)\s+be\s+
(?:called|invoked|executed|run|performed)\s+
(?:prior to|before|preceding|ahead of|earlier than|in advance of)\s+
(?:calling|invoking|executing|running|performing)\s+(.+)/i
```

(19) Before calling

```
/(?:before|prior to|preceding|ahead of|earlier than|in advance of)\s+
(?:calling|invoking|executing|running|performing)\s+(.?),\s+
(?:must|need to|needs to|should|have to|ought to|required to)\s+
(?:call|invoke|execute|run|perform)\s+(.+)/i
```

(20) After completing

```
/(?:after|following|subsequent to|later than|afterward)\s+
(?:completing|invoking|calling)\s+(.?),\s+
(?:proceed with|proceed to|continue to|go ahead and|move on to|
be sure to|ensure that you)\s+
(?:call|invoke|execute|run|perform)\s+(.+)/i
```

(21) Should be called prior to

```
/(.?)\s+(?:should|must|needs to|need to|have to|ought to)\s+be\s+
(?:called|invoked|executed|run|performed)\s+
(?:prior to|before|preceding|ahead of|earlier than|in advance of)\s+
(?:invoking|calling|executing|running|performing)\s+(.+)/i
```

(22) Do not forget/miss to perform/invoke X

```
/(?:do not|don't|never|should not|must not)\s+
(?:forget|miss|neglect|fail)\s+(?:to\s+)?(?:perform|invoke|call|free|close|execute)
\s+(.+)/i
```

(23) Remember to perform/invoke X

```
/(?:remember|reminder|be sure|make sure|ensure|always)\s+
(?:to\s+)?(?:perform|invoke|call|free|close|execute)\s+(.+)/i
```

(24) To avoid X, do Y

```
/(?:to avoid|to prevent|in order to avoid|in order to prevent)\s+
(.+),?\s*(?:do|call|invoke|execute|run|perform)\s+(.+)/i
```

(25) If X, Y is unnecessary/redundant

```
/(?:if|when|in case)\s+(.+),?\s*(?:doing\s+)?(.+)\s*(?:is)?\s*
(?:unnecessary|redundant|superfluous|pointless|ineffective)/i
```

(26) Keep in mind to do X

```
/(?:keep in mind|remember|note|be aware|ensure|make sure)
\s+(?:that\s+)?(.+)/i
```

(27) Recommended to do X

```
/(?:not\s+)?(?:recommended|advised|suggested|encouraged|discouraged|
inadvisable|unadvisable)\s+(?:to\s+)?(.+)/i
```

(28) Need to explicitly overwrite X

```
/(?:need to|must|should|have to|ought to)\s+
(?:explicitly|manually)?\s*(?:overwrite|set|define|specify|provide)
\s+(.+)/i
```

(29) Must do X after Y

```
/(?:must|need to|should|have to|ought to|required to)\s+
(?:call|invoke|execute|run|perform|do)\s+(.+)\s+
(?:after|once|when|afterward)\s+(.+)/i
```

(30) Must do X at the end/eventually

```
/(?:must|need to|should|have to|ought to|required to)\s+
(?:call|invoke|execute|run|perform|do)\s+(.+)\s+
(?:at the end|eventually|finally|after all operations)/i
```