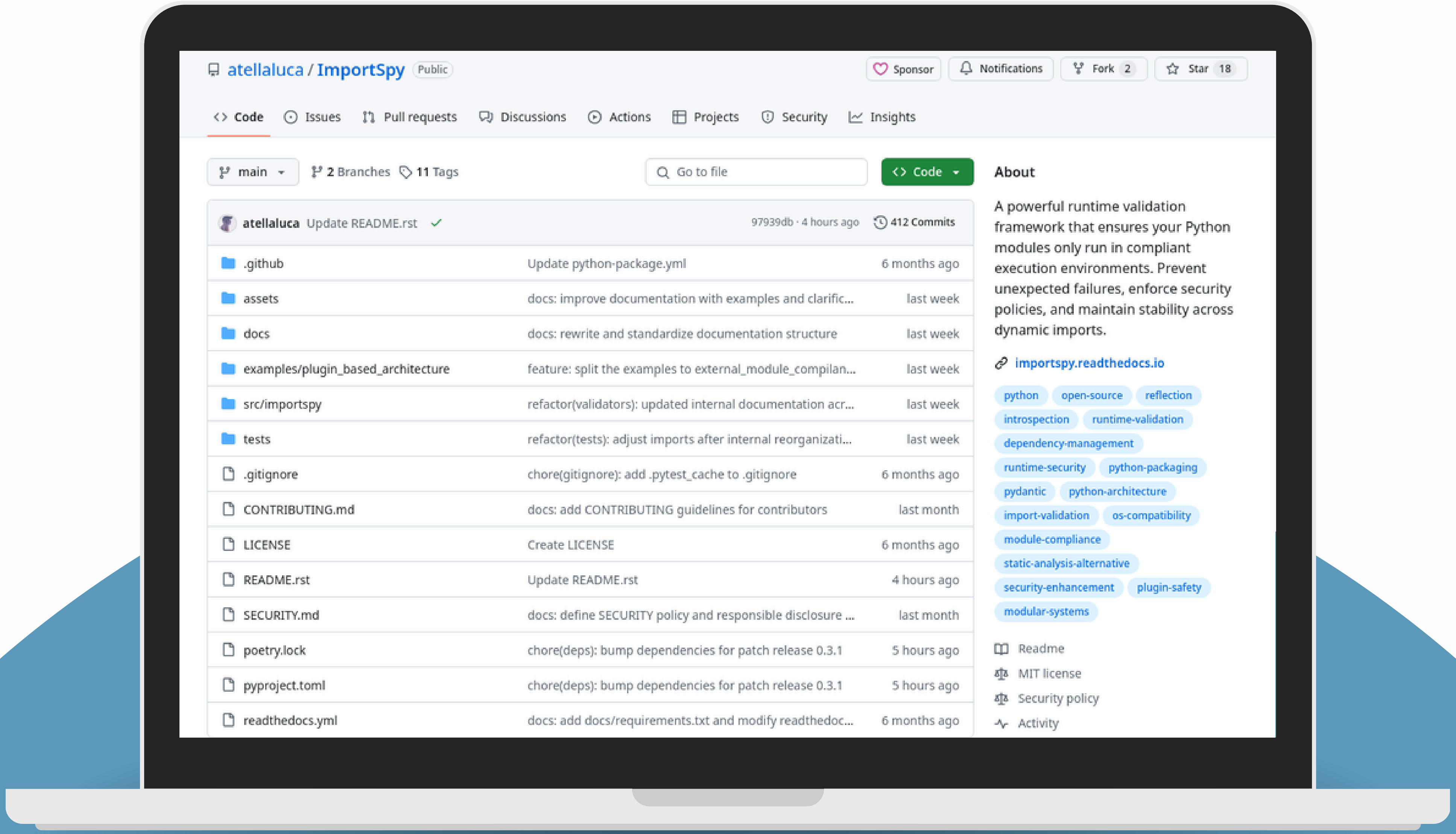
```
pip install importspy
```

ImportSpy

A Python library to enforce modular integrity and execution constraints, making the ecosystem safer, more predictable, and more secure.

Luca Atella



atellaluca / ImportSpy Public

Sponsor · Notifications · Fork 2 · Star 18

Code · Issues · Pull requests · Discussions · Actions · Projects · Security · Insights

main · 2 Branches · 11 Tags · Go to file · Code

atellaluca Update README.rst ✓ · 97939db · 4 hours ago · 412 Commits

.github	Update python-package.yml	6 months ago
assets	docs: improve documentation with examples and clarific...	last week
docs	docs: rewrite and standardize documentation structure	last week
examples/plugin_based_architecture	feature: split the examples to external_module_compilan...	last week
src/importspy	refactor(validators): updated internal documentation acr...	last week
tests	refactor(tests): adjust imports after internal reorganizati...	last week
.gitignore	chore(gitignore): add .pytest_cache to .gitignore	6 months ago
CONTRIBUTING.md	docs: add CONTRIBUTING guidelines for contributors	last month
LICENSE	Create LICENSE	6 months ago
README.rst	Update README.rst	4 hours ago
SECURITY.md	docs: define SECURITY policy and responsible disclosure ...	last month
poetry.lock	chore(deps): bump dependencies for patch release 0.3.1	5 hours ago
pyproject.toml	chore(deps): bump dependencies for patch release 0.3.1	5 hours ago
readthedocs.yml	docs: add docs/requirements.txt and modify readthedoc...	6 months ago

About

A powerful runtime validation framework that ensures your Python modules only run in compliant execution environments. Prevent unexpected failures, enforce security policies, and maintain stability across dynamic imports.

importspy.readthedocs.io

python · open-source · reflection · introspection · runtime-validation · dependency-management · runtime-security · python-packaging · pydantic · python-architecture · import-validation · os-compatibility · module-compliance · static-analysis-alternative · security-enhancement · plugin-safety · modular-systems

Readme · MIT license · Security policy · Activity

Overview

The problem

Why Python needs import-level control?

Python is powerful and flexible — **but too permissive**

In most projects, any module can import any other, regardless of runtime conditions or structure. This creates problems:

- Modules may run in unsupported environments (wrong OS, version, CPU arch)
- Plugins may be loaded without matching the host's expectations
- Structural mismatches (missing classes or functions) cause late runtime errors
- There's no control over who is importing what, where, or how

Static tools like mypy or Bandit don't run at runtime. Even tests (pytest) don't enforce modular boundaries at import time. This is a blind spot — especially in sensitive domains like plugins, CI pipelines, regulated systems, and zero-trust architectures.

The solution

What ImportSpy is and how it works

ImportSpy introduces a new concept in Python: **runtime import contracts**

In most projects, any module can import any other, regardless of runtime conditions or structure. This creates problems:

- Modules may run in unsupported environments (wrong OS, version, CPU arch)
- Plugins may be loaded without matching the host's expectations
- Structural mismatches (missing classes or functions) cause late runtime errors
- There's no control over who is importing what, where, or how

Static tools like mypy or Bandit don't run at runtime. Even tests (pytest) don't enforce modular boundaries at import time. This is a blind spot — especially in sensitive domains like plugins, CI pipelines, regulated systems, and zero-trust architectures.

Architecture & Modes

How ImportSpy enforces contracts

ImportSpy uses a **structured and layered approach**

At its core, ImportSpy enforces contracts across two dimensions:

- **Context awareness**
  Where, when, and under which conditions code runs
- **Structure validation**
  What the importer exposes: classes, methods, attributes

This enforcement relies on a compact schema that defines runtime conditions and module structure. Contracts are enforced either at runtime (embedded mode) or during integration (via CLI mode).

The SpyModel

Semantic Contract Tree

At the heart of ImportSpy is SpyModel.
A hierarchical and recursive structure that defines both the execution context and the expected interface of a Python module.

```
Deployments
 └──── Runtimes (arch)
     └──── Systems (os, envs)
         └──── Pythons (version, interpreter)
             └──── Modules (filename, classes, functions...)
```

This model supports **multi-deployment logic**

Each deployment can define different rules based on a wide range of runtime and structural parameters — including operating system, CPU architecture, environment variables, Python version, interpreter type, module filename, module version, declared variables, functions, classes, methods, and attributes.

This makes contracts explicit, layered, and deeply context-aware, ensuring precise and flexible validation across diverse execution environments.

Writing Contracts

SpyModel as YAML

To simplify its usage, the SpyModel structure **is represented as a YAML file**

This makes contracts:

- Easy to read and write
- Declarative and versionable
- Compatible with both human and automated tools

Each YAML contract defines:

- Where a module can run
- Under which conditions
- And what the importer must provide

YAML contract & fully aligned code

The contract

```yaml
filename: analytics.py
deployments:
  - arch: x86_64
    systems:
      - os: linux
        pythons:
          - version: 3.11
            interpreter: CPython
            modules:
                classes:
                  - name: Logger
                    methods:
                      - name: log_event
                        arguments:
                          - name: self
                          - name: data
                            annotation: dict
                        return_annotation: bool
```

The code

The code runs successfully when executed in an environment matching the contract: x86_64, Linux, Python 3.11, CPython and filename analytics.py

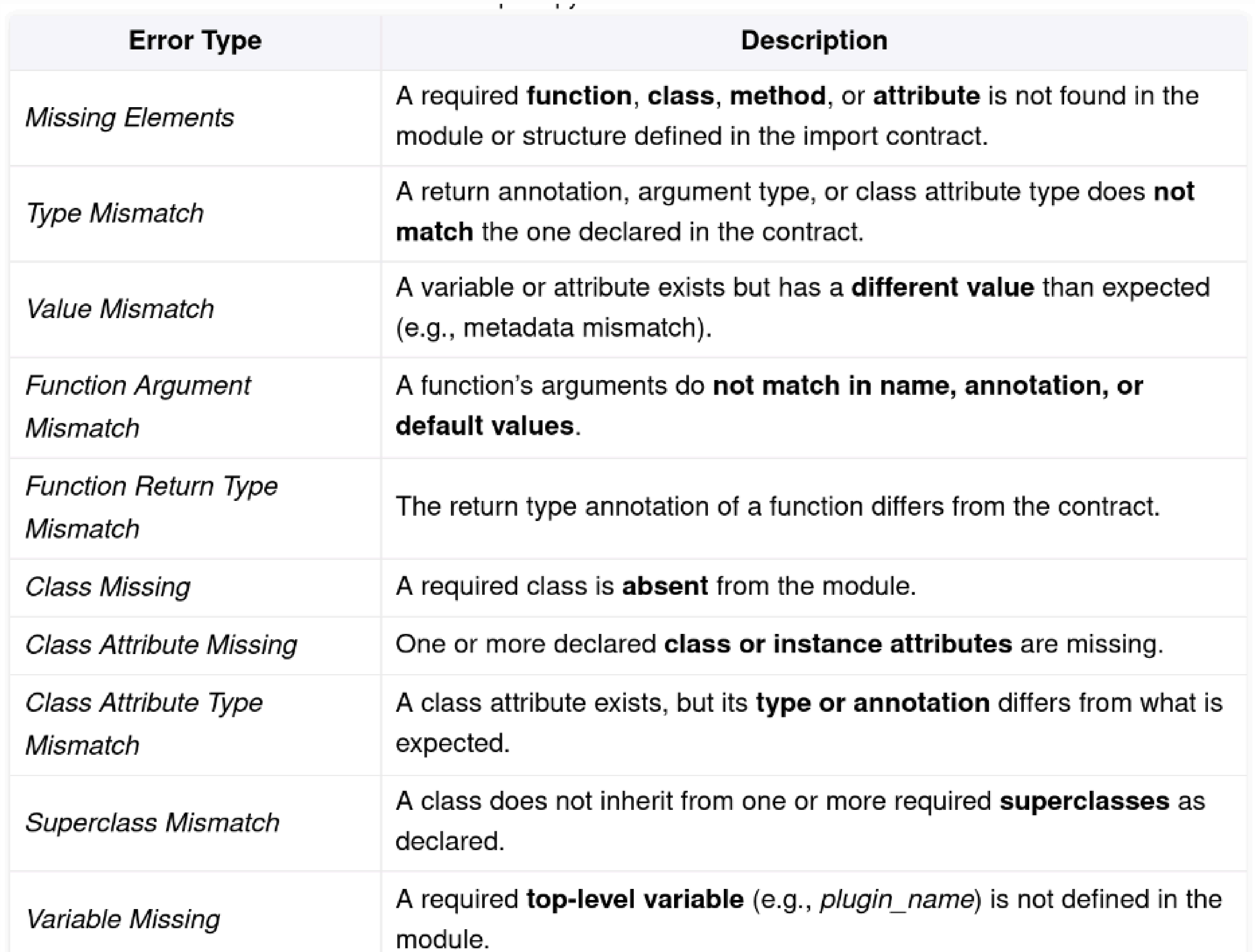
```python
class Logger:

    def log_event(self, data: dict) -> bool:
        """
        Logs an event to the internal logging system.

        Args:
            data (dict): Event data to log.

        Returns:
            bool: True if the log was successful, False otherwise.
        """
        try:
            # Example logic (this can be replaced with actual logging)
            print(f"[LOG] Event received: {data}")
            return True
        except Exception as e:
            print(f"[ERROR] Failed to log event: {e}")
            return False
```

Writing Contracts

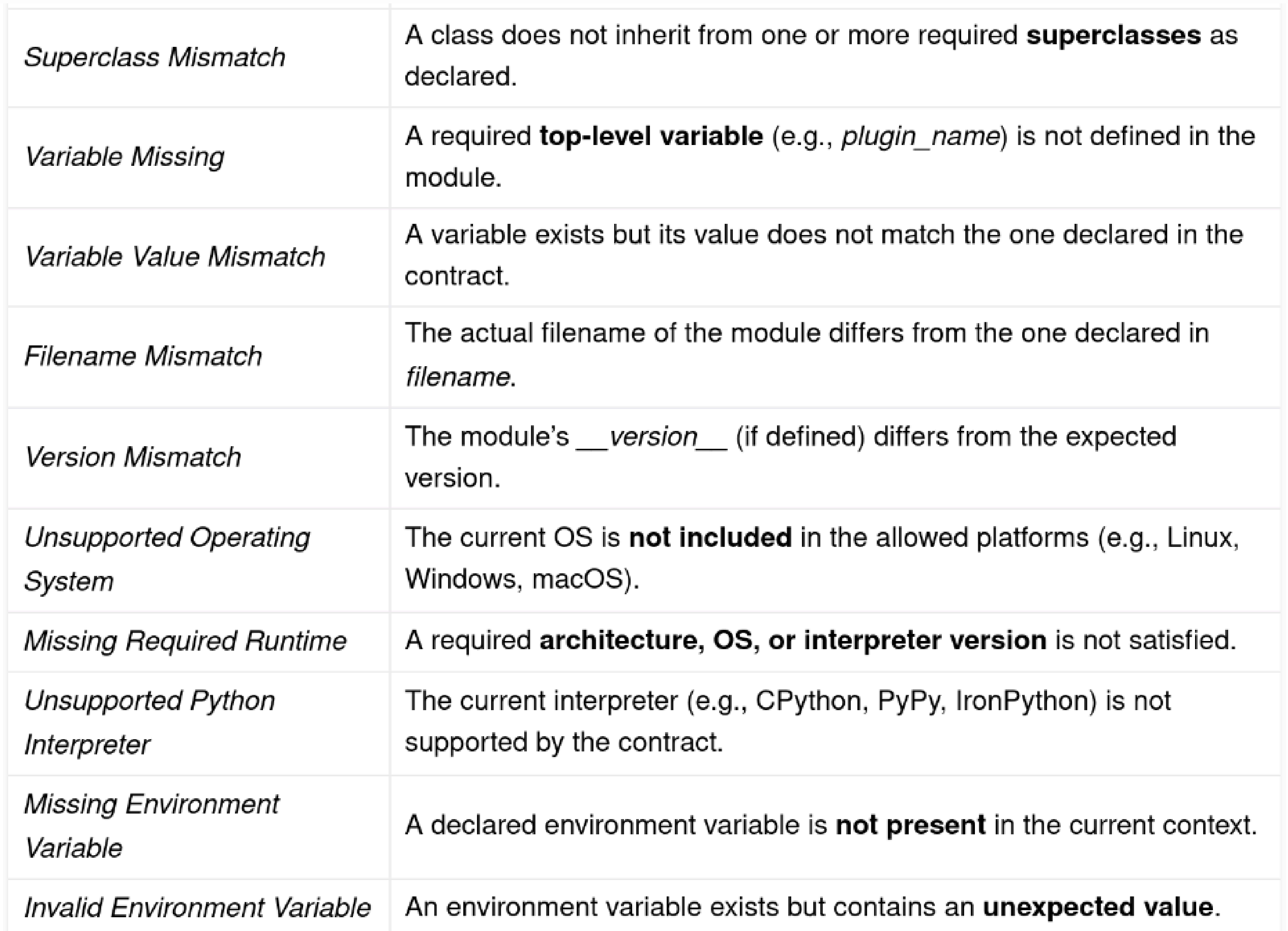
What happens if the contract fails?

When the importing module does not comply with the declared contract,

ImportSpy blocks execution and raises a clear, specific error



Error Type	Description
Missing Elements	A required **function**, **class**, **method**, or **attribute** is not found in the module or structure defined in the import contract.
Type Mismatch	A return annotation, argument type, or class attribute type does **not match** the one declared in the contract.
Value Mismatch	A variable or attribute exists but has a **different value** than expected (e.g., metadata mismatch).
Function Argument Mismatch	A function's arguments do **not match in name, annotation, or default values**.
Function Return Type Mismatch	The return type annotation of a function differs from the contract.
Class Missing	A required class is **absent** from the module.
Class Attribute Missing	One or more declared **class or instance attributes** are missing.
Class Attribute Type Mismatch	A class attribute exists, but its **type or annotation** differs from what is expected.
Superclass Mismatch	A class does not inherit from one or more required **superclasses** as declared.
Variable Missing	A required **top-level variable** (e.g., *plugin_name*) is not defined in the module.



Superclass Mismatch	A class does not inherit from one or more required **superclasses** as declared.
Variable Missing	A required **top-level variable** (e.g., *plugin_name*) is not defined in the module.
Variable Value Mismatch	A variable exists but its value does not match the one declared in the contract.
Filename Mismatch	The actual filename of the module differs from the one declared in *filename*.
Version Mismatch	The module's *__version__* (if defined) differs from the expected version.
Unsupported Operating System	The current OS is **not included** in the allowed platforms (e.g., Linux, Windows, macOS).
Missing Required Runtime	A required **architecture, OS, or interpreter version** is not satisfied.
Unsupported Python Interpreter	The current interpreter (e.g., CPython, PyPy, IronPython) is not supported by the contract.
Missing Environment Variable	A declared environment variable is **not present** in the current context.
Invalid Environment Variable	An environment variable exists but contains an **unexpected value**.

Embedded mode

Runtime self-validation

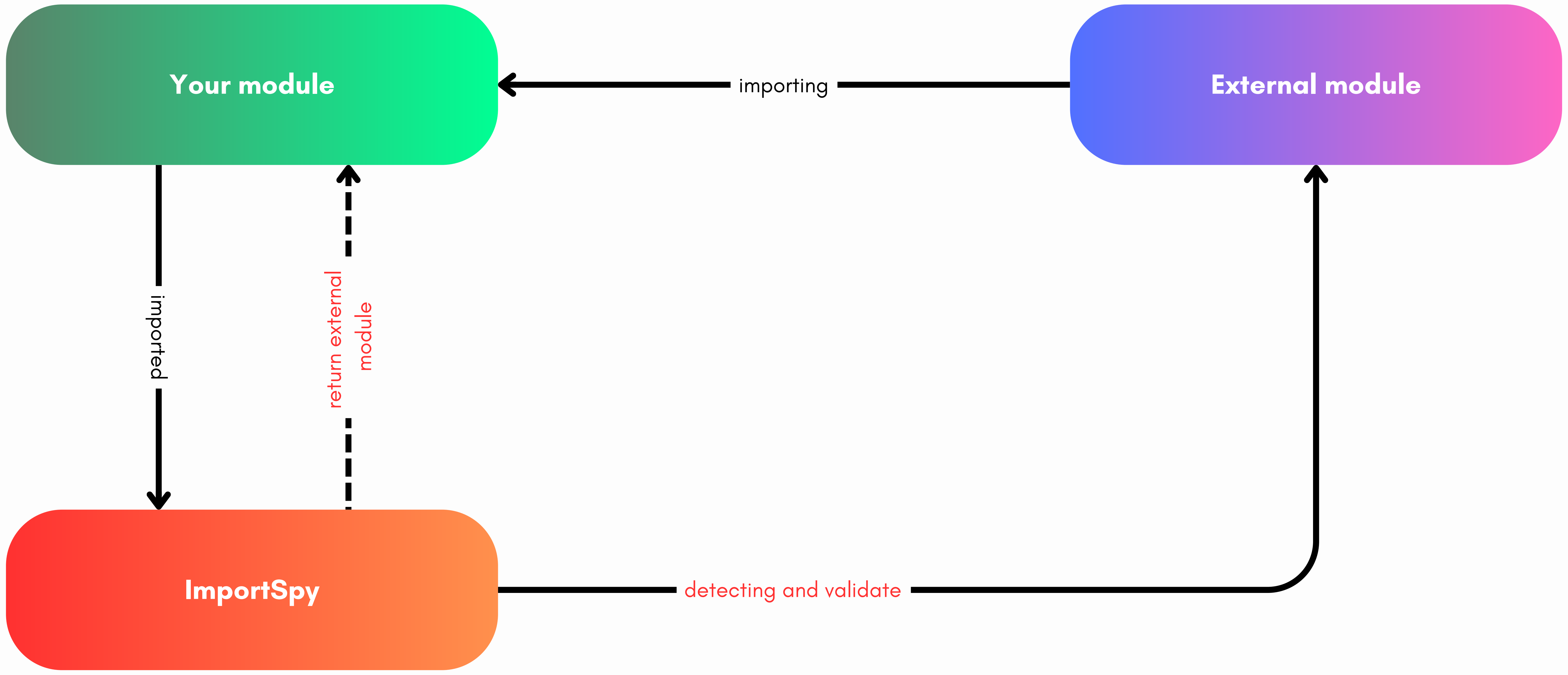
Allows a Python module to validate the context and the structure **of the module that is importing it**, directly at runtime.

```python
from importspy import Spy

Spy().importspy(filepath="contract.yml")
```

The module checks that:

- The execution environment matches the declared constraints (OS, architecture, interpreter, Python version, env vars...)
- The external module complies with the required structure (e.g. it defines specific classes, functions, attributes...)

If any condition fails, ImportSpy blocks execution immediately by raising a detailed ValueError that clearly reports the reason.



✅ When to use Embedded Mode:

- In plugins that must only run inside compatible or trusted hosts
- In modular architectures where the host must provide a defined interface
- To make a module self-aware and execution-aware, preventing misuse by unknown contexts

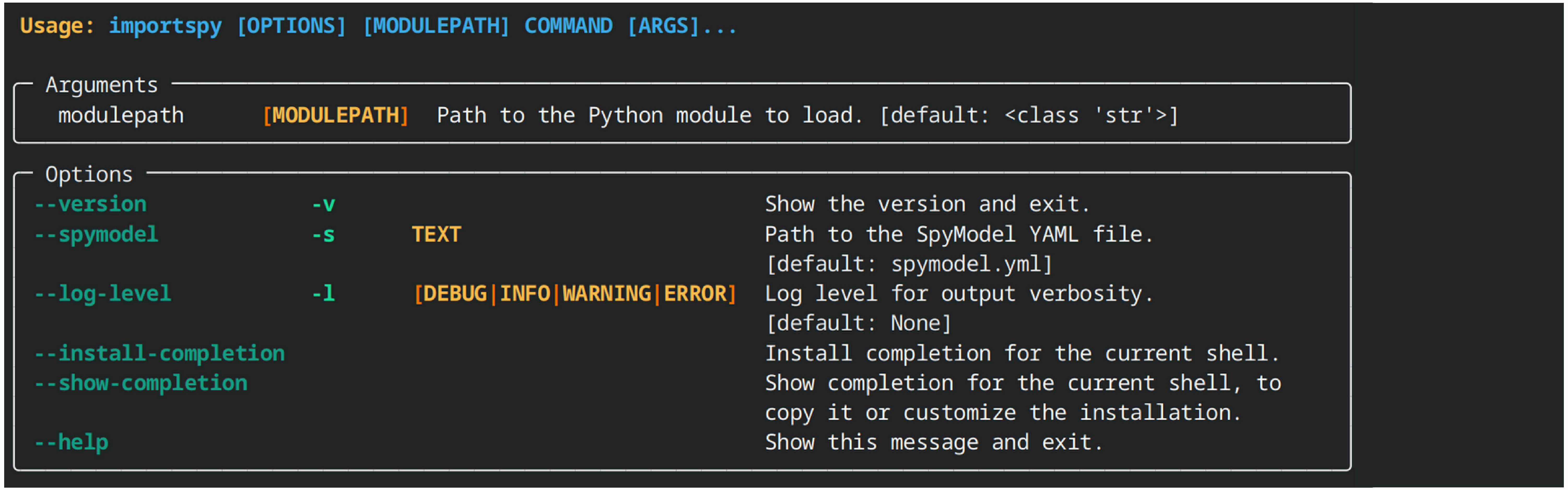
CLI mode

Validate modules in CI/CD & Dev workflow

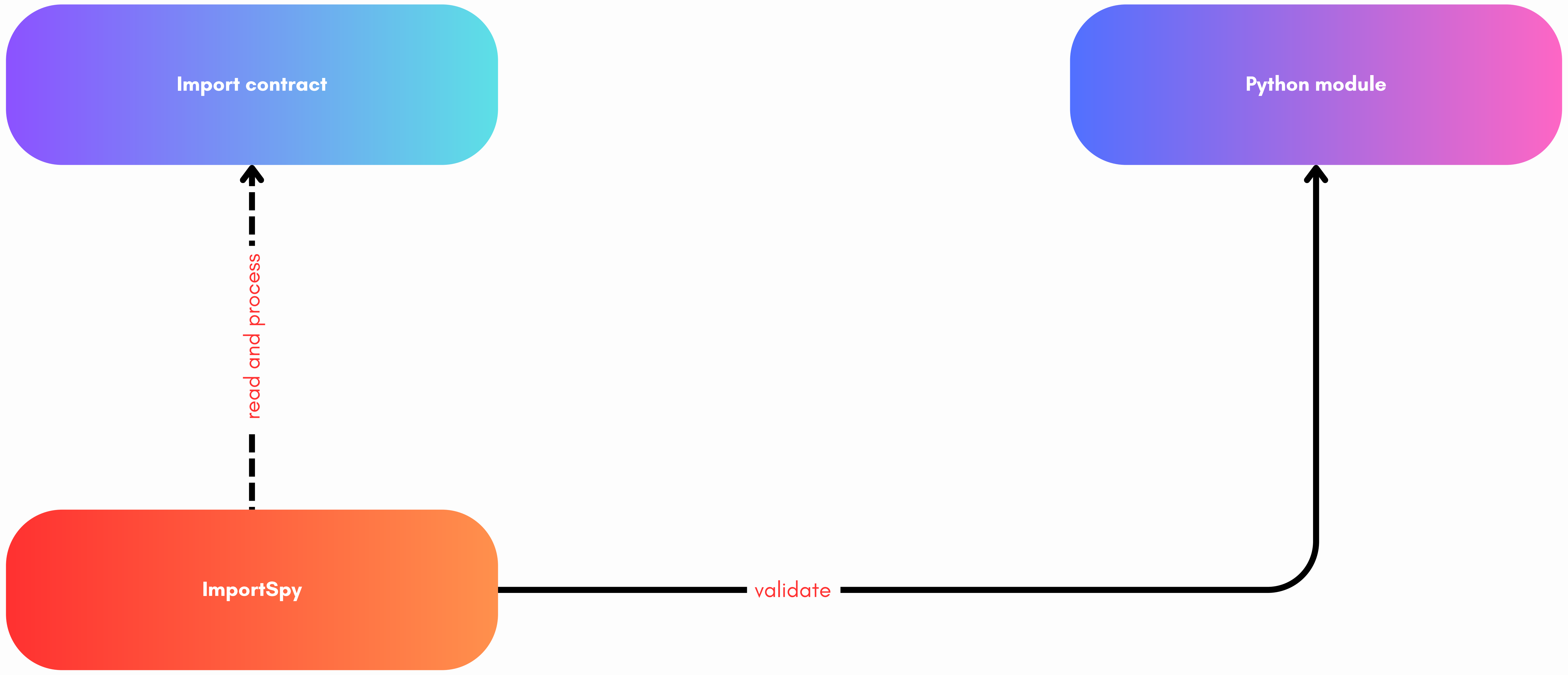
Lets you **validate Python modules externally**, without modifying their code.

It's ideal for:

- Enforcing contracts in CI/CD pipelines
- Verifying third-party modules before integration
- Running bulk or automated checks on multiple modules



```
Usage: importspy [OPTIONS] [MODULEPATH] COMMAND [ARGS]...

╭─ Arguments ──────────────────────────────────────────────────────────────────────────────╮
│  modulepath      [MODULEPATH]  Path to the Python module to load. [default: <class 'str'>] │
╰──────────────────────────────────────────────────────────────────────────────────────────╯
╭─ Options ────────────────────────────────────────────────────────────────────────────────╮
│ --version             -v                                Show the version and exit.        │
│ --spymodel            -s      TEXT                      Path to the SpyModel YAML file.   │
│                                                         [default: spymodel.yml]           │
│ --log-level           -l      [DEBUG|INFO|WARNING|ERROR] Log level for output verbosity.  │
│                                                         [default: None]                   │
│ --install-completion                                    Install completion for the current shell. │
│ --show-completion                                       Show completion for the current shell, to │
│                                                         copy it or customize the installation.    │
│ --help                                                  Show this message and exit.       │
╰──────────────────────────────────────────────────────────────────────────────────────────╯
```



✅ When to use CLI Mode:

- **To enforce contracts in CI/CD pipelines before deployment, ensuring modules run only in authorized conditions**
- **To validate third-party or external modules without modifying their code**
- **To batch-check multiple modules during integration, release preparation, or security audits**

Key use cases

Where ImportSpy adds value

1. Plugin systems and modular architectures

In ecosystems where modules are loaded dynamically — such as plugins, extensions, or integration points — it is crucial to ensure that imported components respect a defined structure and interface.
ImportSpy allows plugin authors to enforce who can load them, under what conditions, and with which structural guarantees. This ensures that plugins only operate in safe, compatible host environments, reducing runtime errors, undefined behaviors, or silent failures.

Perfect for complex platforms, automation systems, CMSs, and any plugin-based architecture that integrates third-party code.

Key use cases

Where ImportSpy adds value

2. DevSecOps and secure software supply chain

Modern pipelines automate testing, deployment, and promotion of code across environments — but rarely validate structural contracts at the integration level.
ImportSpy brings enforcement into CI/CD by ensuring that modules can only run in approved environments, with validated importing contexts.

This creates a new security layer in the software supply chain: even if code passes unit tests, it won't deploy if it violates declared usage contracts. Ideal for DevSecOps practices, infrastructure-as-code, and continuous integration systems like GitHub Actions, GitLab CI, or Jenkins.

Key use cases

Where ImportSpy adds value

3. Regulated domains: Finance & Healthcare

Sectors like finance and healthcare demand predictability, auditability, and compliance in every part of the stack. Even a small plugin or misconfigured script can lead to massive risk.ImportSpy allows developers to define strict execution contracts, ensuring that modules are only used in authorized conditions — for example, in production, on certified environments, under specific interpreters or platform versions.

By enforcing both environment and structure, ImportSpy contributes to regulatory compliance and reduces the risk of unauthorized code execution. It's a lightweight but powerful tool in any compliance-by-design workflow.

Key use cases

Where ImportSpy adds value

4. IoT, edge and embedded systems

In distributed systems — such as IoT devices or edge compute nodes — code often runs in isolated environments with limited oversight.ImportSpy ensures that modules will only execute on the right device, interpreter, OS, or configuration. It can block accidental or malicious imports on unauthorized platforms, enforce compatibility across constrained hardware, and help developers manage versions and module boundaries.

In environments where modules are updated or deployed remotely, ImportSpy acts as a runtime gatekeeper, preventing misuse and runtime fragmentation.

Key use cases

Where ImportSpy adds value

5. Open-Source libraries and governance

Open-source maintainers often struggle with integrations: users may import their modules incorrectly, in unsupported versions, or in unintended ways.
With ImportSpy, maintainers can declare runtime usage contracts — defining the required structure and assumptions for their code to function correctly.
This reduces support requests, avoids misunderstandings, and improves the resilience of shared libraries. It also aligns with principles of self-documenting and self-validating APIs, helping other developers integrate correctly from day one.

Key use cases

Where ImportSpy adds value

6. Education & Training

Concepts like modular design, separation of concerns, and runtime validation are hard to teach in practice.
ImportSpy provides a clear, declarative format (YAML) to express what a module needs from its environment and from its importer. Students and developers can learn how to define expectations explicitly and enforce them automatically.
It also teaches the principle of contract-driven development, which is valuable in system design, interface work, and collaborative team development.

Roadmap & features

What we'll build with the grant

Over 12 months, we'll deliver:

1. Hash-based integrity checking: bind contract to module hash (e.g. SHA-256)
2. Improved support for dynamic importers: importlib, setuptools entrypoints, etc.
3. Contract validator CLI tool with dry-run mode
4. Automatic YAML generator from module structure
5. Expanded docs and tutorials: real-world examples, video guides
6. Community-building: GitHub Discussions, webinars, blog series
7. CI/CD integration templates: GitHub Actions, GitLab, etc.
8. Version 1.0 stable release, PyPI and ReadTheDocs ready

Technical challenges

Why this project matters

Building ImportSpy is not a matter of utility code — it's about defining a new execution layer in Python: **one that enables self-protecting modules and import-aware architectures**.

To achieve this, the project must overcome several complex and critical challenges

Technical challenges

Why this project matters

1. **Secure code hashing**

Ensuring module integrity in Python is complex due to its highly dynamic nature. Decorators, runtime code generation, and packaging quirks can alter bytecode or structure between environments, making traditional hashing unreliable. ImportSpy must design a selective and stable hashing mechanism that allows developers to bind contracts to specific components of the code, like function signatures or class bodies, without being affected by superficial changes or tooling artifacts. This challenge is crucial to enable tamper detection and runtime trust enforcement across distributed systems.

Technical challenges

Why this project matters

2. Automatic contract generation

Writing structural contracts manually, even in YAML, can be tedious and error-prone — especially in large codebases or legacy systems. Many developers would benefit from being able to auto-generate a contract based on the actual structure of a Python module: its classes, methods, attributes, and signatures.

Delivering automatic contract generation would significantly lower the barrier to entry, speed up adoption, and make ImportSpy usable even for non-expert teams. It would turn the contract system into a self-documenting layer — and provide a foundation for future integrations, such as IDE plugins or live documentation tools.

Technical challenges

Why this project matters

3. Multi module and dependency-aware validation

In real-world applications, Python modules rarely operate in isolation — they interact with packages, submodules, or dynamically loaded components. A single-module contract is powerful, but often not enough.
ImportSpy aims to extend validation to multi-module setups, allowing contracts to specify expected relationships across modules, packages, or plugin collections. This includes validating that a module importing another also satisfies its dependencies' contracts. Implementing this requires graph traversal, dependency resolution, and scoped contract application — effectively enabling a form of semantic contract enforcement across dependency chains. It's a major step toward structural integrity in large Python systems.

Technical challenges

Why this project matters

4. Contract schema evolution

As ImportSpy grows, its YAML-based contract language must evolve to support more expressive features — such as integrity checks, conditional rules, or interface inheritance. This evolution must be managed without breaking compatibility for existing users, requiring schema versioning, formal validation tools, and migration guides. Creating a flexible yet durable contract model ensures that teams can adopt ImportSpy at scale, maintain it over time, and confidently extend its use without fear of regressions.

Technical challenges

Why this project matters

5. Runtime performance

Because ImportSpy validates contracts during the module import process, any latency introduced can degrade the developer experience and application startup time. The challenge lies in optimizing introspection, reducing redundant checks, and leveraging caching mechanisms to keep runtime overhead negligible. This requires a deep understanding of Python's import internals and performance profiling. Achieving near-zero latency will make ImportSpy viable for real-time systems and high-frequency execution environments.

Technical challenges

Why this project matters

6. Integration with Asynchronous Module Imports

Modern Python applications increasingly utilize asynchronous programming paradigms to improve performance and responsiveness. However, integrating import validation mechanisms like ImportSpy with asynchronous module imports presents unique challenges. Ensuring that contracts are enforced correctly without introducing performance bottlenecks or disrupting the asynchronous event loop requires careful design. Addressing this challenge would enable ImportSpy to seamlessly support asynchronous applications, thereby broadening its applicability in contemporary Python development.

This replacement highlights a forward-looking challenge that aligns with current trends in Python development and underscores ImportSpy's commitment to evolving alongside modern programming practices.

Technical challenges

Why this project matters

7. Security audit and trustworthiness

Since ImportSpy acts as a gatekeeper to module execution, any internal vulnerability could compromise the safety of the entire system. Reflection, contract parsing, and runtime evaluation all involve sensitive operations. To build trust, the project must undergo an external security audit focused on injection risks, execution bypasses, and safe defaults. Establishing this level of scrutiny will make ImportSpy suitable for use in high-integrity contexts such as finance, healthcare, public infrastructure, and open-source compliance pipelines.

Sustainability & growth plan

How ImportSpy lives beyond the grant

ImportSpy is and will remain open source under the MIT license

- We'll offer consulting services for teams who want to adopt contract-based architecture
- We'll develop a contract generator to help new users define YAML contracts quickly
- We'll explore SaaS extensions (e.g. hosted contract validation for CI pipelines)
- We'll activate GitHub Sponsors and donation-based support
- We'll offer live and recorded workshops on contract-driven development

About the author

Luca Atella

From Southern Italy to open-source impact

Luca Atella is a 25-year-old software developer and open-source contributor based in Basilicata, Southern Italy. With a strong focus on backend development, modular architectures, and DevSecOps practices, he brings a pragmatic yet forward-thinking approach to Python software design.

ImportSpy was born from Luca's direct experience with integration fragility in plugin-heavy systems. He created it as a response to the need for self-validating, structurally aware modules in modern software environments. The project reflects his vision of Python as a language that should empower developers to write safer, more predictable, and interoperable code — without giving up flexibility.

About the author

Luca Atella

From Southern Italy to open-source impact

Luca is currently completing his Computer Science degree at the University of Basilicata while dedicating his full-time effort to growing ImportSpy into a sustainable and impactful open-source initiative.

He plans to launch a developer-focused activity in Southern Italy, combining contract-driven tooling with education and compliance-oriented services.

He shares his work on GitHub at github.com/atellaluca and connects with the community via linkedin.com/in/luca-atella.

Want to explore ImportSpy?

Define, enforce, trust your imports.

ImportSpy is fully open-source, well-documented, and ready to try.
You'll find:

- Detailed documentation with real-world examples
- A clean, typed codebase on GitHub
- A growing ecosystem of usage patterns and contracts

📘 Docs: importspy.readthedocs.io
🧪 Source: github.com/atellaluca/ImportSpy
🟢 MIT licensed
🛠️ Python 3.10+