

Type Enforced: A Python type enforcer for type

- ₂ annotations
- 3 Connor Makowski [□] ^{1¶}, Willem Guter [□] ¹, and Timothy Russell [□] ¹
- 4 1 Massachusetts Institute of Technology Cambridge, United States ROR ¶ Corresponding author

DOI: 10.xxxxx/draft

Software

- Review 🗗
- Repository 🗗
- Archive ♂

Editor: Open Journals ♂ Reviewers:

@openjournals

Submitted: 01 January 1970 Published: unpublished

License

Authors of papers retain copyright and release the work under a ¹⁵ Creative Commons Attribution 4.Q International License (CC BY 4.0),

Summary

type_enforced is a pure Python package designed to enforce type annotations at runtime without the need for a special compiler. It provides an intuitive decorator-based interface that allows developers to enforce explicit typing constraints on function and method inputs, return types, dataclasses, and class instances. The package supports a comprehensive set of Python's built-in types, typing module constructs (such as List, Dict, Union, Optional, and Literal), nested data structures, and custom constraints. By offering runtime validation of type annotations and constraints, type_enforced enhances code reliability, readability, and maintainability.

Statement of Need

Python's dynamic typing system offers flexibility but can lead to runtime errors that are difficult to diagnose in web applications, complex scientific software, and research applications. Static type checking tools such as Mypy provide valuable compile-time validation; however, they do not prevent runtime type errors. Existing runtime enforcement libraries often require extensive boilerplate code or lack support for advanced typing features and nested structures.

The type_enforced package addresses these limitations by providing robust runtime enforcement of Python type annotations with minimal overhead. It supports advanced typing features including nested iterables, union types, dataclasses, inheritance-based validation, uninitialized class type checks, and custom constraints (Constraint, GenericConstraint). This functionality is particularly important for research software development, where correctness of data types is critical for reproducibility and reliability.

In scientific computing, type_enforced helps ensure the correctness of numerical simulations, validates complex machine learning data pipelines, and reduces subtle runtime errors that could compromise reproducibility. It has proven valuable in collaborative environments, such as transportation modeling and logistics optimization, where contributors with varying Python expertise develop models that must integrate seamlessly through APIs or shared workflows. By catching type mismatches early, type_enforced improves the robustness and trustworthiness of research software.

33 Functionality and Features

37

- ³⁴ Key features provided by the package include:
 - **Decorator-based enforcement**: Easily apply enforcement to functions, methods, classes, static methods, class methods, and dataclasses.
 - Comprehensive typing support: Supports built-in Python types (int, str, list, dict, etc.), typing module constructs (List, Dict, Union, Optional, Literal, Any), union



- types (int | float), nested structures (dict[str, dict[str, int]]), and deeply
 nested iterables (list[set[str]]).
 - **Custom constraints**: Validate input values with built-in constraint classes (e.g., numerical bounds) or user-defined generic constraints (e.g., membership in a predefined set).
 - Inheritance-aware validation: Validate instances against class hierarchies.
 - Flexible enable/disable mechanism: Enable or disable enforcement selectively at the function or class level to accommodate debugging versus production environments.

46 Related Work

41

42

43

44

56

57

58

60 61

63

64

65

66

68

75

76

77

78

79

80

81

83

- Python's ecosystem for type checking and data validation is rich and rapidly evolving, reflecting
- the growing need for both static and runtime type safety in scientific and production code.
- The landscape can be broadly divided into static type checkers, runtime type checkers, and
- $_{50}$ project-based frameworks. Recent empirical studies, such as (?), have analyzed the adoption
- and semantics of Python's type systems in real-world codebases, highlighting both the promise
- and the challenges of practical type enforcement.

53 Static Type Checkers

- Static type checkers analyze code before execution, using type hints to catch potential errors and improve code reliability without incurring runtime overhead.
 - Mypy (Lehtosalo & Mypy contributors, 2012): Mypy is the most widely adopted static type checker for Python, implementing a conventional static type system based on PEP 484. It enforces fixed variable types and reports errors when type annotations are violated. As detailed by (?), Mypy represents the canonical approach to static type checking in Python, and its semantics have become a baseline for evaluating new type inference tools.
 - **Pyright**: A fast type checker developed by Microsoft, offering real-time feedback in editors.
 - PyType: Developed by Google, PyType also provides static analysis and type inference for Python code, but with a distinct approach. Unlike Mypy, PyType maintains separate type environments for different branches in control flow and can infer more precise union types for variables that take on multiple types. The comparative study by (?) shows that PyType and Mypy differ in their handling of type joins, attribute typing, and error reporting, reflecting broader trade-offs in static analysis for dynamic languages.

Runtime Type Checkers and Data Validation

Runtime type checkers enforce type constraints as the program executes, which is particularly valuable when handling external data or integrating with user-facing APIs.

- Pydantic (Colvin et al., 2017): Pydantic is a widely used library for runtime data validation targed at dataclass like objects, leveraging type hints to enforce data schemas and automatically cast input values. It is central to frameworks like FastAPI and is particularly effective for validating input from untrusted sources.
- Typeguard (Grönholm, 2016): Typeguard offers single type level runtime enforcement of function type annotations, raising errors when arguments or return values violate declared types. It is lightweight and integrates easily into existing codebases.
- Enforce (Keith-Magee, 2016): Provides basic runtime enforcement but does not support advanced typing features such as deeply nested structures or constraint-based validations.
- Marshmallow: (Loria, 2013): Marshmallow provides serialization, deserialization, and validation of complex data structures, with support for custom validation logic. It is commonly used in web frameworks for API data validation.



type_enforced: In contrast to the above, type_enforced offers decorator-based runtime enforcement of Python type annotations, including support for nested structures, custom constraints, and inheritance-aware validation. Its focus is on minimal boilerplate and compatibility with modern Python typing constructs, making it suitable for research and collaborative environments where correctness and ease of use are paramount.

90 Discussion

The diversity of tools reflects the dual nature of Python's type system—supporting both static and dynamic paradigms. As (?) demonstrate, the adoption of type annotations is increasing, but real-world usage patterns remain heterogeneous, and the semantics of type checking tools can differ in subtle but important ways. Packages like type_enforced complement this landscape by providing runtime guarantees that static checkers cannot, especially in collaborative or data-driven research settings. Compared to these tools, type_enforced uniquely combines comprehensive type annotation enforcement with powerful constraint validation capabilities and inheritance-aware checks.

Usage Example

100 A simple example demonstrating basic usage:

```
import type_enforced
@type enforced.Enforcer()
def calculate_area(width: int | float, height: int | float) -> int | float:
    return width * height
calculate_area(3.0, 4.5)
                            # Passes
calculate_area('3', 4.5)
                           # Raises TypeError at runtime
An example demonstrating constraint validation:
import type enforced
from type_enforced.utils import Constraint
@type enforced.Enforcer()
def positive_integer(value: int | Constraint(ge=0)) -> int:
    return value
positive_integer(10)
                       # Passes
positive_integer(-5)
                       # Raises TypeError due to constraint violation
```

Acknowledgments

- Development of this software was supported by:
 - MIT Center for Transportation & Logistics (CTL)
 - MIT Computational Analytics, Visualization & Education Lab (CAVE)

References

105

```
Colvin, S., Jolibois, E., Ramezani, H., Garcia Badaracco, A., Dorsey, T., Montague, D., Matveenko, S., Trylesinski, M., Runkle, S., Hewitt, D., Hall, A., & Plot, Victorien. (2017).

Pydantic (v2.11.7). Zenodo. https://doi.org/10.5281/zenodo.15662245
```



Grönholm, A. (2016). Typeguard (Version 4.4.4). https://github.com/agronholm/typeguard

Keith-Magee, R. (2016). Enforce (Version 0.3.4). https://github.com/RussBaz/enforce

Lehtosalo, J., & Mypy contributors, the. (2012). *Mypy* (Version 1.16.1). https://github.com/python/mypy

Loria, S. (2013). *Marshmallow* (Version 4.0.0). https://github.com/marshmallow-code/marshmallow

