The Journal of Open Source Software

Type Enforced: A Python type enforcer for type annotations

Connor Makowski [1¶], **Willem Guter** [1], **and Timothy Russell** [1]

1 Massachusetts Institute of Technology Cambridge, United States ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

TODO

- [T] Check your name
- ☐ Who else are we going to include on this?
- ☐ I read this as Connor needs to be the corresponding author
- [T] Check the list of tags

JOSS Software Requirements

- ☒ Be stored in a repository that can be cloned without registration.
- ☒ Be stored in a repository that is browsable online without registration.
- ☐ Have an issue tracker that is readable without registration.
- ☐ Permit individuals to create issues/file tickets against your repository.

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 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 (`WithSubclasses`), and custom constraints (`Constraint`, `GenericConstraint`). This makes it particularly suitable for research software development where correctness of data types is critical for reproducibility and reliability.

Functionality and Features

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`), union types (`int | float`), nested structures (`dict[dict[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 using the provided utility class (`WithSubclasses`).
- **Flexible enable/disable mechanism**: Enable or disable enforcement selectively at the function or class level to accommodate debugging versus production environments.

TODO - [ ] Check if what I wrote makes any sense

Research Applications

The functionality provided by `type_enforced` is particularly beneficial in scientific computing contexts where strict data validation is crucial. Potential research applications include:

- Ensuring correctness of numerical simulations by enforcing precise data types.
- Validating complex data pipelines in machine learning workflows.
- Enhancing reproducibility in computational experiments by preventing subtle runtime type errors.
- Improving the robustness of research software for transportation modeling and logistics optimization, particularly in collaborative environments where contributors have diverse levels of Python expertise. For example, type enforcement has proven valuable when domain experts develop models and their outputs are integrated through APIs for interactive applications, ensuring reliability and consistency across the workflow.

Related Work

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 project-based frameworks. Recent empirical studies, such as Rak-amnouykit et al. (2020), 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.

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** (Jukka Lehtosalo 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 Rak-amnouykit et al. (2020), 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 Rak-amnouykit et al. (2020) 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** (Samuel Colvin 2017): Pydantic is a widely used library for runtime data validation, 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** (Alex Grönholm 2016): Typeguard offers 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** (Russell Keith-Magee 2016): Provides basic runtime enforcement but does not support advanced typing features such as deeply nested structures or constraint-based validations.
- **Marshmallow**: (Steven 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.

Project-Based Type Checkers

These tools integrate type checking with specific frameworks, providing tailored solutions for popular Python ecosystems.

- **django-stubs** and **typeddjango**: Extend static type checking to Django projects, enabling type-safe development in large web applications.
- **flask-pydantic** and **flask-marshmallow**: Provide seamless integration of runtime validation into Flask applications, leveraging Pydantic and Marshmallow models, respectively.
- **FastAPI**: Built on top of Pydantic, FastAPI uses type hints for both static analysis and runtime validation of HTTP request and response bodies, ensuring robust API contracts.

Discussion

The diversity of tools reflects the dual nature of Python's type system—supporting both static and dynamic paradigms. As Rak-amnouykit et al. (2020) 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.

TODO

- ☐ Figure out how to reference something in the paper.bib file
- ☐ Do we need to have the project-based type checkers listed or marshmallo?
- ☐ Should I get rid of the last sentence?
- ☐ Can someone take a look at these and see if what Tim wrote makes sense?
- ☐ Are there other better known examples of Python type checkers?

Usage Example

A simple example demonstrating basic usage:

```python
import type_enforced

@type_enforced.Enforcer()
def calculate_area(width: float, height: float) -> 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:

```python
from type_enforced import Enforcer
from type_enforced.utils import Constraint

@Enforcer()
def positive_number(value: [int, Constraint(ge=0)]) -> int:
    return value

positive_number(10)   # Passes
positive_number(-5)   # Raises TypeError due to constraint violation
```

Acknowledgements

Development of this software was supported by the MIT Center for Transportation & Logistics (CTL).

TODO

- ☐ Do we want to say anything here? Really this is a Connor question

---

References