Latexify-py

Alexander D. Cai
Harvard College
alexcai@college.harvard.edu

Abstract

We present the latexify-py Python package for converting Python source code to LaTeX pseudocode. latexify-py is intended for authors of technical texts who wish to provide a readable LaTeX pseudocode description of an algorithm alongside a Python reference implementation. latexify-py is implemented using the Python ast package. It extends a previous implementation at https://github.com/google/latexify_py by focusing on modularity and improved user experience. The source code is available at https://github.com/adzcai/latexify-py.

1 Introduction

Programming courses often use a textbook as a reference. Such textbooks often describe algorithms using **pseudocode**, which ensures that the algorithm is understandable to readers regardless of their programming language background. Using LaTeX to write this pseudocode also enables the use of mathematical notation, which often provides a much more concise and familiar presentation that is implementation-agnostic. Another opposing paradigm is the **literate programming** paradigm, as represented by the use of Jupyter notebooks or R Markdown files, in which the algorithms are implemented using a specific programming language. These have the benefit of being *executable*, but force the author to choose language-specific or, often, framework-specific implementation details which may be irrelevant to the algorithm's functionality. latexify-py supports translation from a low-level Python implementation of an algorithm to a high-level pseudocode description of that algorithm. (Note that this has the opposite direction as most compilers, which take in high-level code and compile it down to low-level machine instructions.) While Python, as a high-level programming language, is already highly readable, supporting translation to pseudocode still offers the benefits outlined above. This allows authors of pedagogical programming texts the benefits of both worlds. The most relevant previous work in this direction is the project google/latexify_py on GitHub, which has 7.3k stars at the time of writing. This project builds on top of google/latexify_py with a focus on extensibility and user experience. Our main contributions are as follows:

1. **Plugin system** 2.2. We rewrite

2 Methods

In this section, we discuss the key features of latexify-py and its improvements upon google/latexify_py.

2.1 Overall procedure

Python exposes the builtin ast package for enabling programmers to parse and transform the abstract syntax tree (AST) corresponding to a given string of Python source code. It provides the ast.parse command for parsing a Python code string into its AST representation. It also provides the two classes ast.NodeTransformer and ast.NodeVisitor for recursively traversing the AST. latexify-py consists mostly of subclasses of these two classes. The translation from Python to LaTeX takes place in two stages:

1. **AST transformation**. The first stage transforms the AST itself using subclasses of Python's `ast.NodeTransformer`. For example, the `AugAssignReplacer` class of `google/latexify_py` replaces Python's **augmented assignment operations** (e.g. `x += 1`) with the explicit assignment `x = x + 1`.
2. **AST translation**. The second stage recursively translates the AST into LaTeX code using subclasses of Python's `ast.NodeVisitor`. For example, our `IdentifierConverter` class converts Python identifiers into the corresponding LaTeX strings.

2.2 Plugin system

Our main contribution was rewriting the package in terms of an extensible **plugin system**. The original package consisted of a fixed set of program transformations. Users could opt in or out of specific features by providing keyword arguments to the public API. For example, the `use_math_symbols` keyword argument could be set to true to replace a variable name corresponding to a Greek letter with the corresponding Greek letter:

```
def greek_demo(alpha, beta, Omega):
    return alpha * beta + Omega
with_symbols = latexify.function(greek_demo, use_math_symbols=True)
without_symbols = latexify.function(greek_demo, use_math_symbols=True)
```

Here, `with_symbols` results in

$$\mathrm{greek\_demo}(\alpha, \beta, \Omega) = \alpha\beta + \Omega$$

while `without_symbols` results in

$$\mathrm{greek\_demo}(\mathrm{alpha}, \mathrm{beta}, \mathrm{Omega}) = \mathrm{alpha} \cdot \mathrm{beta} + \mathrm{Omega}.$$

This system does not enable the user to customize handling of specific AST nodes. This violates the principle that frameworks should have **escape hatches** (e.g. Rust's `unsafe` or React's `useRef`) that enable the user to circumvent the framework's design choices.

We implement an extensible **plugin system** to support user customization. The implementation is simple and modular. We create a `PluginList` class inheriting from `ast.NodeVisitor`. Its constructor takes in a list of `ast.NodeVisitor` instances. Its `visit` method, which takes in an AST node and outputs a LaTeX string, simply iterates through the list of `ast.NodeVisitors` and calls the first one that supports translation of the AST node.

The list of `ast.NodeVisitors` must in particular be subclasses of our `Plugin` class, which replaces all calls to `visit` with calls to the `visit` function of the stack that it is part of. This ensures that, for example, if a user wishes to override some `latexify-py` functionality by providing their plugin, the translation will always check the user's plugin before falling back to the library implementation.

We also refactor the functionality of `google/latexify_py` in terms of the new plugin system. In particular, the `ExpressionCodegen` class of `google/latexify_py` had grown to include LaTeX translations for various `numpy` commands, which seemed odd to include in the base package since these depended on `numpy` while the actual expressions would not. As a result, `google/latexify_py` would convert expressions such as `np.linalg.inv(np.array([[1, 1], [-1, 1]]))` purely based on the identifier names, even if `numpy` was not installed in the environment.

2.3 Identifiers

The original package only allowed for replacing identifiers with other valid Python identifiers (see `transformers.identifier_replacer.py`). That is, it was impossible to associate a given Python identifier with an arbitrary LaTeX expression. This was because the `IdentifierReplacer` class was implemented as a subclass of `ast.NodeTransformer`, which only supports replacing an AST node with another valid AST node. It also did not support replacement of nested identifiers such as `np.linalg.eigvals` with a single LaTeX expression. Doing so in the original implementation would require adding `np.linalg` to a list of prefixes that would be entirely removed in the transformation process. In our implementation,

A Appendix / supplemental material

Optionally include supplemental material (complete proofs, additional experiments and plots) in appendix. All such materials **SHOULD be included in the main submission.**