The Journal of Open Source Software

Litrepl: Literate Paper Processor Promoting Transparency More Than Reproducibility

Sergei Mironov [1]

1 Independent Researcher, Armenia

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Matthew Feickert

Reviewers:

- @itepifanio
- @agoose77

Submitted: 05 January 2025
Published: unpublished

Summary

Litrepl is a lightweight text processing tool designed to recognize and evaluate code sections within Markdown or Latex documents. This functionality is useful for both batch document section evaluation and interactive coding within a text editor, provided a straightforward integration is established. Inspired by Project Jupyter, Litrepl aims to facilitate the creation of research documents. In the light of recent developments in software deployment, however, we have shifted our focus from informal reproducibility to enhancing transparency in communication with programming language interpreters, by either eliminating or clearly exposing mutable states within the communication process. The tool is designed to ensure easy code editor integration, and the project repository offers a reference Vim plugin.

Statement of need

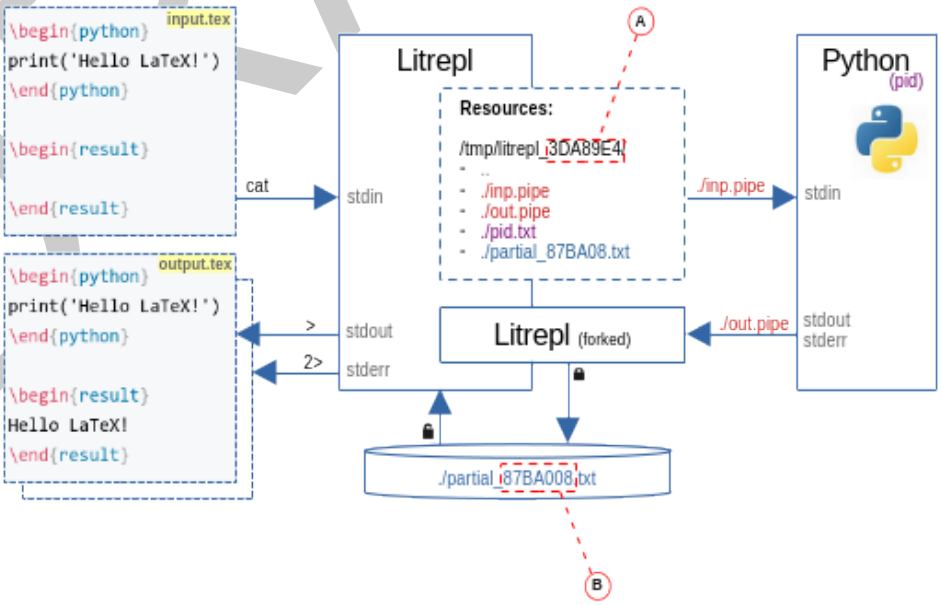

Figure 1: Litrepl resource allocation diagram. Hash **A** is computed based on the Litrepl working directory and the interpreter class. Hash **B** is computed based on the contents of the code section.

The Journal of Open Source Software

Literate Programming, formulated by Donald Knuth, shifts the focus from merely coding to explaining computational tasks to humans. This approach is seen in the WEB system (Knuth, 1984) and its successor tools, which use a document format that interleaves code sections with explanatory text. These systems can produce both readable documentation and executable code, and over time, this concept has evolved towards simplification (Ramsey, 1994).

The Read-Evaluate-Print Loop (REPL), a key concept in human-computer interaction, gained importance in the LISP and APL communities (Spence, 2020), (McCarthy, 1959), (Iverson, 1962). By combining a command-line interface with a language interpreter, REPL enables incremental and interactive programming, allowing users to modify the interpreter state directly. This approach is believed to enhance human thought processes by keeping users actively involved (Granger & Pérez, 2021).

A pivotal development in this field was the IPython interpreter (Perez & Granger, 2007), which led to the Jupyter Project and its Jupyter Notebook format (Kluyver et al., 2016). This format consists of logical sections like text and code that can interact with language interpreters, enabling REPL-like programming for well-structured, shareable documents. This is part of *Literate Computing* (Pérez & Granger, 2015), which aims to reach broad audiences, enhance reproducibility, and promote collaboration. Key technical aspects include bidirectional communication between the Jupyter Kernel and Notebook renderer, alongside client-server interactions between the web server and user browser.

We believe that reproducibility is crucial in the Literate Computing framework, enhancing communication among dispersed researchers. However, as illustrated by (Dolstra et al., 2010), we argue that this challenge exceeds the capacity of a single system, needing operating system-level solutions. Following (Vallet et al., 2022), we introduce *Litrepl*, which enables REPL-style editing by integrating with existing editors and formats, and reduces hidden state variables in a compact codebase. Litrepl uses bidirectional text streams for inter-process communication and supports Markdown and LaTeX formats with simplified parsers. It supports Python, Shell, and a custom large language model communication interpreter while leveraging POSIX (IEEE and The Open Group, 2024) system features.

The difference between Litrepl and other solutions, including Jupyter, is highlighted in the table below.

How it works

The operation of Litrepl is best illustrated through the example below. Consider the document named `input.tex`:

```
$ cat input.tex
\begin{python}
import sys
print(f"I use {sys.platform.upper()} btw!")
\end{python}
\begin{result}
\end{result}
```

This document contains a Python code section and an empty result section marked with the corresponding LaTeX environment tags. To "execute" the code sections of the document, we pipe the whole document through the Litrepl processor as follows (only the last three relevant lines of the result are shown).

```
$ cat input.tex | litrepl eval-sections 1 | tail -n 3
\begin{result}
I use LINUX btw!
\end{result}
```

Name	Document format	Data format	Interpreter	Backend	Frontend
Jupyter	Jupyter[j1]	Rich	Many[j2]	Client-server via ZeroMQ[j3]	Web, Editor
Quarto	Markdown[q1]	Rich	Few	Modified Jupyter[q2]	Web, Editor
Codebraid	Markdown[c1]	Rich	Many[c2]	Same as Jupyter[c3]	Editor
R markdown	Markdown[r1]	Rich	Few[r2]	Linked libraries[r3]	Editor
Litrepl	Markdown, LaTeX	Text-only	Few	POSIX Pipes	Command line, Editor

[j1] Jupyter Notebook Format
[j2] Available kernels
[j3] Details on Jupyter backend communication link
[q1] Quarto Markdown extensions
[q2] Quarto uses the slightly modified Jupyter kernel protocol. See Jupyter kernel support and Quarto echo kernel
[c1] Codebraid Markdown extensions are described as code chunks formatting
[c2] Codebraid supports Jupyter kernels link
[c3] Codebraid supports interactive editing via Jupyter kernels.
[r1] The R Markdown language
[r2] In RMarkdown most languages do not share a session between code sections, the exceptions are R, Python, and Julia link
[r3] For Python, RMarkdown relies on Reticulate to run, which uses low-level Python features to organize the communication link.

The side effect of this execution is the starting of a session with the Python interpreter, which now runs in the background. We communicate with it by editing the document and re-running the above command.

Evaluation results are written back into the result sections, and the entire document is printed. At this stage, certain conditions can be optionally checked. First, adding `--pending-exitcode=INT` instructs Litrepl to report an error if a section takes longer than the timeout to evaluate. Second, setting `--exception-exitcode=INT` directs Litrepl to detect Python exceptions. Lastly, `--irreproducible-exitcode=INT` triggers an error if the evaluation result doesn't match the text initially present in the result section.

Interfacing Interpreters

Litrepl communicates with interpreters using POSIX uni-directional streams, one for writing input and another for reading outputs. During the communication, Litrepl makes the following general assumptions:

- The streams implement synchronous single-user mode.
- Command line prompts are disabled. Litrepl relies on the echo response, as described below, rather than on prompt detection.
- The presence of an echo command or equivalent. The interpreter must be able to echo an argument string provided by the user in response to such command.

The simplicity of this approach is a key advantage, but it also has drawbacks. There's no parallel evaluation at the communication level, locking the interpreter until each snippet is evaluated. This can be addressed by using interpreter-specific parallelism, such as Python's subprocess utilities or shell job control. The text-only data type limitation is more fundamental;

Litrepl overcomes this by supporting text-only document formats. Formats like LaTeX and Markdown handle non-text data via references or side channels (e.g., file systems), balancing benefits in version control with the need for explicit data transfer organization.

Session Management

Litrepl maintains interpreter sessions in the background to support a REPL environment. Resources are stored in an auxiliary directory specified by command-line arguments, environment variables, or automatically derived paths. This directory contains pipe files for I/O and a file with the interpreter's process ID for session management. When evaluating code, Litrepl creates a response file named by hashing the code to store interpreter output. A response reader process with a soft lock runs until the interpreter finishes and responds to a probe, introducing the system's only hidden state.

Conclusion

The tool is implemented in Python in under 2K lines of code according to the LOC metric, and has only two Python dependencies so far, at the cost of the dependency on the POSIX operating system interfaces. Needless to say, we used Litrepl to evaluate and verify the examples presented in this document.

References

Dolstra, E., Löh, A., & Pierron, N. (2010). NixOS: A purely functional linux distribution. *Journal of Functional Programming*, *20*(5–6), 577–615. https://doi.org/10.1017/S0956796810000195

Granger, B. E., & Pérez, F. (2021). Jupyter: Thinking and storytelling with code and data. *Computing in Science & Engineering*, *23*(2), 7–14. https://doi.org/10.1109/MCSE.2021.3059263

IEEE and The Open Group. (2024). *POSIX standard: Base specifications, issue 8* (Standard Issue 8; p. 4107). The Open Group. ISBN: 1-957866-40-6

Iverson, K. E. (1962). *A programming language*. John Wiley & Sons, Inc. ISBN: 0471430145

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C., & Team, J. D. (2016). Jupyter notebooks - a publishing format for reproducible computational workflows. *International Conference on Electronic Publishing*. https://api.semanticscholar.org/CorpusID:36928206

Knuth, D. E. (1984). Literate programming. *Computer/Law Journal*. https://api.semanticscholar.org/CorpusID:1200693

McCarthy, J. (1959). *Recursive functions of symbolic expressions and their computation by machine*. https://api.semanticscholar.org/CorpusID:267886520

Perez, F., & Granger, B. E. (2007). IPython: A system for interactive scientific computing. *Computing in Science and Engg.*, *9*(3), 21–29. https://doi.org/10.1109/MCSE.2007.53

Pérez, F., & Granger, B. E. (2015). *Project jupyter: Computational narratives as the engine of collaborative data science*. Jupyter blog. https://blog.jupyter.org/project-jupyter-computational-narratives-as-the-engine-of-collaborative-data-science-2b5fb94c3c58

Ramsey, N. (1994). Literate programming simplified. *IEEE Softw.*, *11*(5), 97–105. https://doi.org/10.1109/52.311070

Spence, R. (2020). APL demonstration 1975. In *Youtube video*. Imperial College London via Youtube. https://www.youtube.com/watch?v=_DTpQ4Kk2wA

Vallet, N., Michonneau, D., & Tournier, S. (2022). Toward practical transparent verifiable and long-term reproducible research using guix. *Scientific Data*, *9*(1), 597. https://doi.org/10.1038/s41597-022-01720-9